COMPUTER SCIENCE

TOWARD PERFORMANCE & PORTABILITY & PRODUCTIVITY IN PARALLEL PROGRAMMING

A Holistic Code *Generation*, *Optimization*, and *Execution* Approach
for Data-Parallel Computations Targeting Modern Parallel Architectures

Inaugural Dissertation
for the Award of a Doctoral Degree
Dr. rer. nat.
in the Field of Mathematics and Computer Science
from the Faculty of Mathematics und Natural Sciences
of the University of Münster, Germany

submitted by

ARI RASCH

born in Essen, Germany
– 2024 –

DEAN:	Prof. Dr. Arthur Bartels
FIRST ASSESSOR:	Prof. Dr. Sergei Gorlatch
SECOND ASSESSOR:	Prof. Dr. Herbert Kuchen
THIRD ASSESSOR:	Prof. Dr. Albert Cohen
DATE OF ORAL EXAMINATION:	February 13, 2025
DATE OF GRADUATION:	February 13, 2025

Dedicated to the loving memory of my father *Hemin Rasch*.
1950 – 2016

ABSTRACT

Achieving *performance*, *portability*, and *productivity* for data-parallel computations, such as time-intensive linear algebra routines (e.g., matrix multiplication) and stencil computations (such as convolution), both heavily used in the field of machine learning, has emerged as a major research challenge. The complex hardware design of contemporary parallel architectures, including Graphics Processing Units (GPUs) and multi-core CPUs, requires advanced program optimizations to fully exploit the performance potential of architectures which nowadays are characterized by deep and complex memory and core hierarchies. Furthermore, due to the diverse hardware landscape, it has proven challenging to achieve a consistently high level of performance for the same source code on different kinds of architectures, a.k.a. (performance) portability: different architectures require different kinds of optimizations, thereby often posing challenging, often even contradicting requirements on code optimization. Also, to make programming modern architectures amenable to common application developers, the complexity of achieving performance and portability must be hidden behind a user-productive programming interface.

This thesis introduces a holistic approach to code *generation*, *optimization*, and *execution* for data-parallel computations targeting modern parallel architectures. The ultimate goal of our approach is to simultaneously achieve *performance*, *portability*, and *productivity*, in one combined approach, which is identified as a major research challenge.

The first part of this thesis introduces the algebraic formalism of *Multi-Dimensional Homomorphisms (MDH)* – a novel approach to *generating* code that can be fully automatically optimized (auto-tuned) for a particular target architecture and characteristics of the input and output data (such as size and memory layout); our code generation approach is hidden behind a productive user interface that expresses a wide range of data-parallel computations, agnostic of hardware and optimization details.

The second part of this thesis introduces the *Auto-Tuning Framework (ATF)* for automatically *optimizing* parameterized program code (as generated by our MDH approach). In contrast to existing auto-tuners, ATF targets modern parallel implementations for state-of-the-art architectures – such implementations have so-called *interdependencies* among their performance-critical parameters, which are efficiently handled by ATF.

The third part of this thesis introduces our approach *Host Code Abstraction (HCA)* for *executing* program code in state-of-practice parallel programming approaches (as generated via MDH and optimized using ATF), such as NVIDIA CUDA for GPU programming and OpenCL for multiple kinds of architectures. HCA simplifies programming so-called *host code*, by offering a high-level user interface that hides host code optimizations and boilerplate program code.

PUBLICATIONS

This thesis is based on the ideas and methodologies presented in the following publications:

Ranking by CORE (https://www.core.edu.au/) according to CORE2020 (conference papers) or the year of submission (articles), respectively.

[1] **Ari Rasch**. "(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms." In: *ACM Trans. Program. Lang. Syst.* (2024). 73 pages. ISSN: 0164-0925. DOI: 10.1145/3665643. **Rank A***.

[2] **Ari Rasch**. *Full Version: (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms*. 2024. arXiv: 2405.05118 [cs.PL]. 131 pages.

[3] **Ari Rasch**, Richard Schulze, Denys Shabalin, Anne Elster, Sergei Gorlatch, and Mary Hall. "(De/Re)-Compositions Expressed Systematically via MDH-Based Schedules." In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, 61–72. ISBN: 9798400700880. DOI: 10.1145/3578360.3580269. **Rank A**.

[4] Richard Schulze, **Ari Rasch**, and Sergei Gorlatch. "Code Generation and Optimization for Deep-Learning Computations on GPUs via Multi-Dimensional Homomorphisms *(Best Poster Finalist)*." In: *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, 2 pages. URL: https://sc21.supercomputing.org/proceedings/tech_poster/poster_files/rpost163s2-file3.pdf.

[5] **Ari Rasch**, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)." In: *ACM Trans. Archit. Code Optim.* 18.1 (Jan. 2021), 26 pages. ISSN: 1544-3566. DOI: 10.1145/3427093. **Rank A**.

[6] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "md_poly: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms." In: *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. 2020, 4 pages.

[7] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "md_poly: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms." In: *ACM SRC Grand Finals Candidates, 2019 - 2020*. 2020, 5 pages.

[8] **Ari Rasch**, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. "dOCAL: high-level distributed programming with OpenCL and CUDA." In: *The Journal of Supercomputing* 76.7 (2020), pp. 5117–5138. DOI: 10.1007/s11227-019-02829-2. **Rank B**.

[9] **Ari Rasch** and Sergei Gorlatch. "ATF: A generic directive-based auto-tuning framework." In: *Concurrency and Computation: Practice and Experience* 31.5 (2019). e4423 cpe.4423, 14 pages. DOI: https://doi.org/10.1002/cpe.4423. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4423. **Rank A**.

[10] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms." In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2019, pp. 354–369. DOI: 10.1109/PACT.2019.00035. **Rank A**.

[11] **Ari Rasch**, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. "High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms." In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 526–533. ISBN: 9781450359337. DOI: 10.1145/3297280.3297330. **Rank B**.

[12] **Ari Rasch** and Sergei Gorlatch. "Multi-dimensional Homomorphisms and Their Implementation in OpenCL." In: *International Journal of Parallel Programming* 46.1 (2018), pp. 101–119. ISSN: 1573-7640. DOI: 10.1007/s10766-017-0508-z. **Rank A**.

[13] **Ari Rasch**, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. "OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA." In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 2018, pp. 408–416. DOI: 10.1109/PADSW.2018.8644541. **Rank B**.

[14] Mona Riemenschneider, Alexander Herbst, **Ari Rasch**, Sergei Gorlatch, and Dominik Heider. "eccCL: parallelized GPU implementation of Ensemble Classifier Chains." In: *BMC Bioinformatics* 18.1 (2017), p. 371. DOI: 10.1186/s12859-017-1783-9. **Rank A**.

[15] **Ari Rasch**, M. Haidl, and S. Gorlatch. "ATF: A Generic Auto-Tuning Framework." In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2017, pp. 64–71. DOI: 10.1109/HPCC-SmartCity-DSS.2017.9. **Rank B**.

ACKNOWLEDGMENTS

A long journey has come to an end. Many people have accompanied me on my way and I would like to thank them in the following.

First of all, I would like to thank my doctoral supervisor Prof. Dr. Sergei Gorlatch. It was his trust and his permission to let me work freely on a topic whose success was initially unforeseeable that made this thesis possible. I would also like to express my gratitude for Sergei's endless willingness to let me attend various conferences and events around the world and meet fascinating people, many of whom I can now call not only colleagues but also friends.

Very special thanks goes to Richard Schulze. Without his trust in my vision at a time when this was by no means a matter of course, and his unending willingness to work on this project, this work would not exist today. I am very happy to call you a good colleague, but above all a good friend, with whom I have experienced many ups and downs in my life – thank you!

I would like to thank also all of my other group members (who I prefer to call friends rather than colleagues): Bastian Hagedorn, Bastian Köpcke, Jens Gutsfeld (who somehow always belonged more to our group than to his own group), Johannes Lenfers, Michael Haidl, Michel Steuwer, Tim Humernbrum, and Waldemar Gorus. Special thanks particularly also goes to our secretary Julia Kaiser-Mariani for always taking care of our demanding paperwork and fighting with us against university bureaucracy.

A big "thank you" also goes to Vinod Grover and his team at NVIDIA. They believed in my research and let me apply it to their projects during my internship. It was a great pleasure.

I would also like to thank each of the 17 Bachelor's and 11 Master's students that I co-supervised over the years, as well as all attendees of the 15 project seminars I co-supervised. All these theses and seminars were initiated in close connection with my research and thus contributed significantly to the success of this work.

Last but not least, I would like to thank my parents and my sister for their support, as well as all my friends and also training partners from Team Angelo Gym in the Netherlands.

CONTENTS

Part I

INTRODUCTION & BACKGROUND

1 INTRODUCTION

1.1 PARALLEL PROGRAMMING IN TODAY'S WORLD

The end of *Dennard Scaling*[1] and *Moore's Law*[2] have changed the landscape of computer systems significantly [87]. While traditional systems work inherently sequential, state-of-the-art computer systems rely on massively parallel processor architectures which can achieve tremendous performance. This new performance potential has significantly widened the scope of computer systems, from complex scientific simulations to artificial intelligence. For example, due to the enormous computational power of state-of-the-art processors, deep learning [187] – an emerging class of artificial intelligence algorithms – is able to significantly outperform traditional approaches, e.g, to speech recognition and image classification.

While the advances in computing power open up incredible opportunities, they also pose major challenges on programming: computer programs have to be specifically optimized for each particular combination of a parallel architecture, application class, and characteristics of the input and output data (such as their size and memory layout). Such optimization is challenging, because it requires expert knowledge from the programmer about both hardware details as well as low-level programming techniques, which usually goes beyond the expertise of the common domain scientist, such as physicists and artificial intelligence researchers. Even for performance experts, optimization is time consuming, challenging, and error prone due to the ever increasing complexity of hardware and the diversity of applications. Consequently, the practicability of modern systems is severely hindered.

Three major goals turned out to be essential in programming state-of-the-art computer systems, but also challenging to achieve: *performance*, *portability*, and *productivity*. We outline each of the three goals in the following.

[1] Dennard Scaling: "As transistors get smaller, their power density stays constant." [267]

[2] Moore's Law: "The number of transistors in an integrated circuit doubles every two years." [295]

1.2 CHALLENGES: PERFORMANCE & PORTABILITY & PRODUCTIVITY

The ultimate goal of modern parallel programming is to achieve high *performance* that is *portable* across different kinds of processor architectures and data characteristics, in a *productive* and thus programmer-friendly manner. Achieving these three goals is challenging – major computer science conferences and workshops [329–331, 333–336, 340, 358, 359, 365, 366] identified simultaneously achieving performance, portability, and productivity for programming modern computer systems as an ongoing, major research challenge. This is because for high performance, programmers have to optimize program code for the complex hardware of modern architectures, e.g., NVIDIA many-core GPU or Intel multi-core CPU, which are characterized by deep and complex memory and core hierarchies. For portable performance over such architectures, the programmer has to consider that architectures usually differ significantly in their characteristics: depth of their memory and core hierarchies, how they manage fast memory resources (a.k.a. *caches*) – automatically managed caches (as in CPUs) vs. manually managed caches (GPUs) – etc. Even among devices of the same architecture, characteristics such as the sizes of fast memory resources and the numbers of cores tend to differ considerably, making the hardware landscape even more diverse and complex. Achieving performance portability also over different data characteristics, rather than over architectures only, is often even more challenging: for example, a high-performance implementation of matrix multiplication on big power-of-two input sizes, as often used in the traditional field of numerical computations, is programmed fundamentally differently as compared to matrix multiplication on small, irregularly shaped input matrices, e.g., as currently occurring in deep learning [155]. Furthermore, modern architectures are usually programmed on a low abstraction level (e.g., in CUDA for GPUs [350], or OpenCL [342] which is a popular standard for uniformly programming different kinds of architectures: GPU, CPU, etc), making programming such architectures tedious and cumbersome, e.g., because of complex index computations and synchronization, thereby severely limiting programmer's productivity. Moreover, the state-of-the-art parallel programming approaches (such as CUDA and OpenCL) require so-called *host code* for their execution, which is cumbersome and error-prone to program: the programmer has to explicitly perform data transfers between host and device memory, exploit asynchronous computation efficiency for high performance (e.g., overlapping data transfers and/or device computations), etc.

In the following, we illustrate the importance of each challenge – performance, portability, and productivity – using the popular example of matrix multiplication.

1.2.1 *Performance Challenge*

Modern architectures offer enormous performance potential. However, this potential can only be achieved by efficiently targeting the memory and core hierarchies of architectures via advanced program optimizations (which are thoroughly discussed later in this thesis).

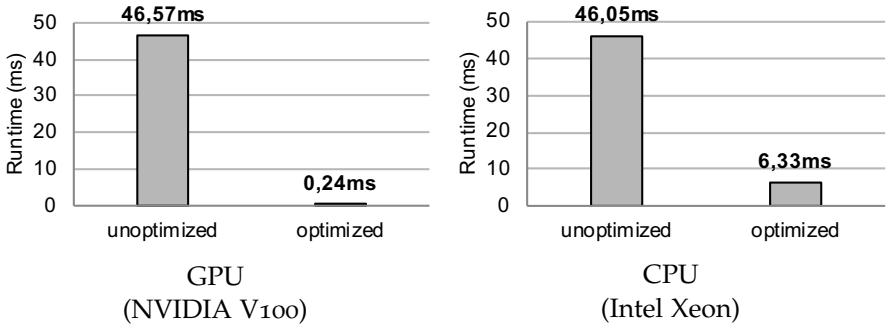


Figure 1: Runtime (lower is better) of unoptimized vs optimized matrix multiplication on GPU (left) and CPU (right).

Figure 1 illustrates the importance of optimization for performance. On GPU (left part of the figure), an optimized implementation of matrix multiplication (taken from [115] and discussed later in Listing 2) achieves substantially higher performance, by $> 202\times$, as compared to a semantically equal but unoptimized variant (from [356] and discussed later in Listing 1); on CPU (right part of the figure), optimization achieves a speedup of $> 7\times$.

CPUs are less sensitive to optimization than GPUs, because the CPU hardware is more focused on automatically conducting optimization (e.g., so-called *out-of-order execution* and *pipelining*), instead of offering additional cores or further fast memory resources, as GPUs do, in order to achieve higher peak performance: the optimized GPU implementation in Figure 1 performs better than the optimized CPU version by $> 27\times$.

We will thoroughly discuss the optimizations required to achieve such performance potentials as in Figure 1 in this thesis.

1.2.2 *Portability Challenge*

Apart from the traditional goal of functional code portability, performance portability becomes more and more important[3]. However, state-of-practice programming models (such as CUDA and OpenCL) provide either no portability at all (CUDA), or they are designed toward functional portability only (OpenCL) such that algorithms require semantically equal but differently optimized implementations to achieve high performance on different architectures. This is due to the fundamental differences in the hardware design of architectures (which we discuss in Chapter 2 of this thesis).

[3] *Functional Portability* means the same program code is executable on different kinds of architectures, whereas *Performance Portability* additionally requires from the program code to achieve the same high performance potentials across architectures.

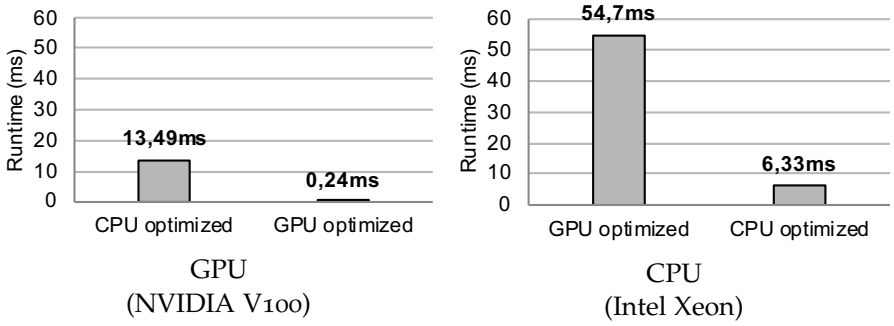


Figure 2: Runtime (lower is better) of GPU/CPU-optimized matrix multiplication on GPU (left) and CPU (right).

Figure 2 illustrates the effect of architecture-specific optimizations, as required for performance portability, using the example of matrix multiplication in OpenCL. We observe that on GPU (left part of the figure), the CPU-optimized implementation performs worse, by > 58×, as compared to the GPU-optimized variant. Similarly, on CPU (right part), the GPU-optimized variant achieves lower performance, by > 8×, as compared to the CPU-optimized implementation. As also discussed for Figure 1, CPU is less sensitive to code optimizations than GPU, because CPU performs several optimizations automatically in hardware.

This thesis will introduce generalized optimizations to achieve high performance on various kinds of parallel architectures, including GPUs and CPUs.

1.2.3 *Productivity Challenge*

It is crucial for the practicality of architectures to provide users with a productive interface for programming them. However, the current state-of-practice programming models (such as CUDA and OpenCL) rely on low abstraction levels, which makes programming complex, error prone, and cumbersome for the user.

```
1  __kernel void MatMul( __global const float A[M][K] ,
2                        __global const float B[K][N] ,
3                        __global       float C[M][N] )
4  {
5    int i = get_global_id(0);
6    int j = get_global_id(1);
7
8    for( int k=0 ; k<K ; ++k )
9      C[i][j] += A[i][k] * B[k][j];
10 }
```

Listing 1: Naive OpenCL implementation of matrix multiplication.

Listing 1 shows a naive OpenCL implementation of matrix multiplication, taken from [356]: each thread (lines 5 and 6) straightforwardly computes one element of the output matrix (lines 8 and 9). While the naive implementation in Listing 1 is arguably easy to program, it achieves only poor performance (see Figure 1).

```
1     __kernel void MatMul( /* ... */ )
2     {
3       const size_t i_wg_l_1 = get_group_id(2);
4     // ... 5 lines skipped
5
6       __private TYPE_TS res_p[/*...*/][/*...*/];
7       {
8     // ... 7 lines skipped
9               for (size_t p_iteration_l_1 = 0; p_iteration_l_1 < (2);
                    ++p_iteration_l_1) {
10                for (size_t p_iteration_l_2 = 0; p_iteration_l_2 < (1)
                      ; ++p_iteration_l_2) {
11                  size_t p_iteration_r_1 = 0;
12                  res_p[p_step_l_1][(((p_iteration_l_1) * 1 + 0))][(0)][
                        p_step_l_2][((p_iteration_l_2) * 1 + 0)] = f(
13                      a[(((l_step_l_1 x* (32 / 1) + (((p_step_l_1 *
                            (2) + (((p_iteration_l_1) * 1 + 0)) / 1) * 1
                             + i_wi_l_1 * 1 + ((((p_iteration_l_1) * 1 +
                             0)) % 1))) / 1) * (64 * 1) + i_wg_l_1 * 1 +
                             ((((p_step_l_1 * (2) + (((p_iteration_l_1)
                            * 1 + 0)) / 1) * 1 + i_wi_l_1 * 1 + ((((
                            p_iteration_l_1) * 1 + 0)) % 1))) % 1))) *
                            1024 + (((l_step_r_1 * (2 / 1) + (((
                            p_step_r_1 * (1) + (((p_iteration_r_1) * 1 +
                             0)) / 1) * 1 + i_wi_r_1 * 1 + ((((
                            p_iteration_r_1) * 1 + 0)) % 1))) / 1) * (2
                            * 1) + i_wg_r_1 * 1 + ((((p_step_r_1 * (1) +
                             (((p_iteration_r_1) * 1 + 0)) / 1) * 1 +
                            i_wi_r_1 * 1 + ((((p_iteration_r_1) * 1 + 0)
                            ) % 1))) % 1)))],
14    // ... 107 lines skipped
15    }
```

Listing 2: Optimized OpenCL implementation of matrix multiplication.

Listing 2 shows for comparison an excerpt of an OpenCL matrix multiplication (taken from [115]) that performs the same matrix multiplication as in Figure 1, but is specifically optimized for NVIDIA GPU. The GPU-optimized implementation (Listing 2) is significantly more complex than the naive implementation (Listing 1): the user is in charge of managing parallelization (Listing 2, line 3) and different memory regions (line 6), as well as performing complex index computations (line 13), etc.

We discuss the design and implementation of optimized implementations in detail in this thesis, and we introduce easy-to-use high-level programming constructs for the programmer from which we generate such optimized implementations fully automatically.

1.3 CONTRIBUTIONS OF THIS THESIS

To address the three main challenges in modern parallel programming – performance, portability, and productivity – this thesis introduces a novel, holistic approach to *generating*, *optimizing*, and *executing* code for state-of-the-art computer architectures (illustrated in Figure 3).

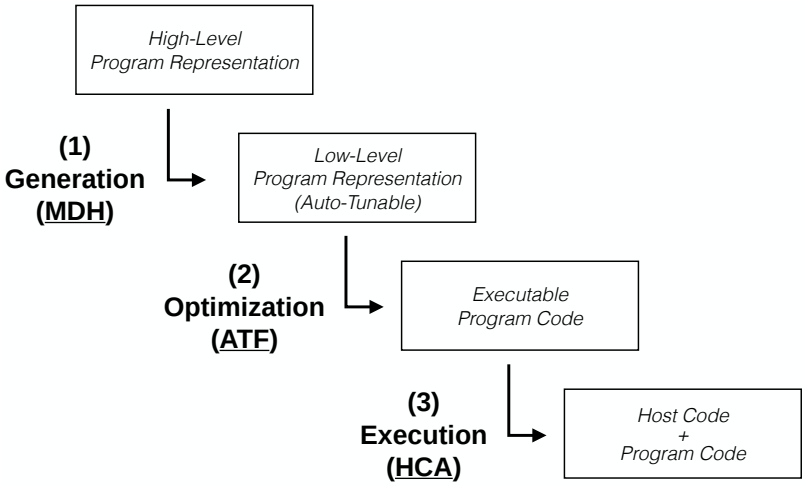


Figure 3: Basic overview of the code *generation*, *optimization*, and *execution* approach introduced in this thesis.

This thesis makes the following three major contributions:

(1) CODE GENERATION We introduce *Multi-Dimensional Homomorphisms (MDH)* – an algebraic class of functions that formally define and express *data-parallel computations* (such as linear algebra routines and stencil computations) which are the time-intensive building blocks in many important application areas, e.g., scientific simulations and artificial intelligence. For MDHs, we introduce two kinds of representations: 1) a *high-level functional representation* to uniformly express various kinds of computations – ranging from linear algebra routines and stencil computations to data mining algorithms and quantum chemistry computations – in the same formalism, and in a convenient, hardware- and optimization-agnostic manner; 2) a *low-level functional representation* that enables formally expressing and reasoning about optimizations for the memory and core hierarchies of state-of-the-art parallel architectures, and that is designed as straightforwardly transformable to executable program code (e.g., in CUDA or OpenCL). We develop a systematic process for lowering an instance in our high-level representation to an instance in our low-level representation, in a formally sound manner, based on performance-critical parameters (such as sizes of tiles and numbers of threads). The parameters of our lowering process enable fully automatically optimizing our low-level instances for the particular target architecture and characteristics of the input and output data, by determining well-performing values of formally specified parameters.

(2) CODE OPTIMIZATION We introduce the *Auto-Tuning Framework (ATF)* to automatically determine well-performing values of performance-critical parameters. In contrast to related approaches, ATF has a particular focus on parameters that have *interdependencies* among them (e.g., the value of one parameter has to be smaller than the value of another parameter), because interdependent parameters are ubiquitous in modern parallel programming (as we discuss thoroughly in this thesis). For interdependent parameters, ATF introduces novel processes for each of the three main phases in auto-tuning: *generating*, *storing*, and *exploring* the parameters' search space. ATF is designed to target programs written in arbitrary programming languages, and it can be used for arbitrary tuning objectives (e.g., high runtime performance and/or low energy consumption). An additional feature of ATF is its extensible, easy-to-use user interface which is based on simple code annotations, thereby making auto-tuning appealing also to common application developers. ATF also allows to be conveniently used for programs written in mainstream programming languages (e.g., C++ and Python), by being implemented additionally as a programming library in these languages.

(3) CODE EXECUTION We introduce the approach *Host Code Abstraction (HCA)* – currently implemented as the C++ library *dOCAL (distributed OpenCL/CUDA Abstraction Layer)* for OpenCL and CUDA – to conveniently execute parallel programs on their target devices; the devices may be part of distributed multi-device systems in HCA. Our HCA approach provides an easy-to-use high-level user interface for programming *host code* which is required in approaches such as OpenCL and CUDA to perform data transfers between devices (belonging to potentially different nodes), manage synchronization among devices, etc. The user interface of HCA hides such low-level host code details from the programmer, including managing boilerplate code for creating, managing, and destroying low-level host code objects, e.g., OpenCL's so-called *command queues* and CUDA's *streams* objects. Furthermore, HCA automatically performs host code optimizations for the user, e.g., overlapping data transfers between host and device with computations, as well as recognizing and avoiding unnecessary data transfers between devices; such optimizations are essential for achieving high host code performance, as we demonstrate experimentally in this thesis.

1.4 OUTLINE OF THIS THESIS

This thesis is structured as follows.

Part I – Introduction & Background

The remainder of this part discusses background knowledge required for understanding the basic ideas presented in this thesis.

CHAPTER 2 recapitulates the design of modern high-performance computer systems as well as of the state-of-practice programming models used for programming these systems.

Part II – A Novel, Holistic Approach to Code Generation & Optimization & Execution

This part presents the three main contributions of this thesis (summarized in Section 1.3). For high generality, each contribution is addressed individually and thus independently from the other contributions (including their experimental evaluation and comparison to related work). We conclude this part by demonstrating that our three contributions – although developed independently – complement each other to a holistic code generation, optimization, and execution approach.

CHAPTER 3 starts with an overview of our novel approach to code *generation*, *optimization*, and *execution*. In this approach, we *generate* code using *MDH* (Contribution 1 of this thesis), *optimize* the MDH-generated code via *ATF* (Contribution 2), and we *execute* the MDH-generated and ATF-optimized code using *HCA* (Contribution 3).

CHAPTER 4 discusses our *code generation* approach, which is based on *Multi-Dimensional Homomorphisms (MDH)*. We formally introduce MDHs, as well as their formal high- and low-level representations, and we show how we systematically lower an instance in the MDH's high-level representation to a device- and data-optimized instance in its low-level representation, in a formally sound and auto-tunable manner. Our high-level MDH representation is capable of expressing various kinds of data-parallel computations, and our low-level representation is powerful enough to express, in the same formalism, the memory optimization and parallelization strategies of fundamentally different state-of-practice code generation approaches. Furthermore, our experiments confirm that, in combination with auto-tuning, our MDH approach achieves higher performance than the current state-of-practice approaches (including hand-optimized libraries provided by vendors), on both GPU and CPU, on real-world data sets for computations that are popular in five important application areas: 1) linear algebra, 2) stencil computations, 3) quantum chemistry 4) data mining, and 5) deep learning.

CHAPTER 5 discusses our *code optimization* approach, which is based on *Auto-Tuning Framework (ATF)*. We introduce ATF's novel processes to generating, storing, and exploring the search spaces of performance-critical parameters – the three main phases in auto-tuning – for programs whose parameters may have interdependencies among them. Moreover, we design and thoroughly discuss ATF's user interface which has a particular focus on usability and extensibility, thereby making auto-tuning attractive also for common application developers. Our experiments confirm that compared to state-of-the-art auto-tuning approaches, ATF substantially improves each phase of the auto-tuning process for programs with interdependent parameters, and ATF still achieves the same good auto-tuning results for programs that rely on traditional parameters (which have no interdependencies among them).

CHAPTER 6 discusses our *code execution* approach, which is based on the *Host-Code Abstraction (HCA)*, currently implemented as the *distributed OpenCL/CUDA Abstraction Layer (dOCAL)* for OpenCL and CUDA. We introduce HCA's high-level user interface for conveniently implementing host code, agnostic of low-level details and free from boilerplate code. Moreover, we discuss HCA's automatic host code optimizations which are conducted transparently for the user. Based on well-proven code metrics, we confirm that HCA significantly simplifies host code programming while still achieving performance competitive to hand-optimized host code.

CHAPTER 7 discusses two types of interfaces for using our approach holistically, i.e., when combining all of its three steps – generation, optimization, and execution: 1) a DSL-based interface, that relies on *Domain-Specific Language (DSL)* for data-parallel computations, and 2) a C-based interface that offers auto-parallelization of sequential programs in the popular C programming language, based on easy-to-use code annotations.

Part III – Future Work

We see this work as a starting point for many future directions. As such, we dedicate one part of this thesis to presenting our work-in-progress results and ideas for potential future research and development.

Part IV – Conclusion

This part summarizes the insights presented in this thesis.

2 BACKGROUND

This work introduces our novel approach to generating, optimizing, and executing program code for modern computer systems. Such systems are characterized by a collection of devices, often belonging to different compute nodes, and each device provides its own, usually deep and complex memory and core hierarchies. Devices are the basic building blocks of systems; carefully programming them, via modern programming approaches (such as CUDA and OpenCL), is key to achieve the full performance potential of these systems.

In the following, we discuss in Chapter 2.1 the structure of modern computer systems, with a particular focus on their memory and core hierarchies which are both of fundamental interest for this thesis. Afterward, we present the state-of-practice programming models used for programming such systems in Chapter 2.2 (low-level models) and Chapter 2.3 (high-level models).

2.1 MODERN PARALLEL COMPUTER SYSTEMS

Modern computer systems consist of processors (a.k.a. *devices*), such as GPUs and CPUs, as their basic building blocks. In Chapter 2.1.1, we discuss the basic architectural design of recent device architectures; for this, we use the examples of GPU and CPU, because these are currently the most commonly used architectures in high-performance computing systems [364], and because both architectures differ significantly in major characteristics. Afterward, we discuss in Chapter 2.1.2 multi-device and multi-node systems which consist of multiple devices that may belong to different compute nodes.

2.1.1 *Recent Device Architectures*

Figure 4 shows the basic architectural structure of GPU (left part of the figure) and CPU (right part). Common GPUs provide the programmer with 3-layered memory and core hierarchies; for high performance, computations have to be efficiently de-composed and assigned to these memory and core layers, and the computed intermediate results need to be efficiently re-composed, as we thoroughly discuss in this thesis. A GPU's top memory layer is called *Device Memory (`DM`)* which is the largest but slowest memory region, and the bottom layer is the so-called *Register Memory (`RM`)* which is the fastest but smallest region; the middle layer is called *Shared Memory (`SM`)*. The core hierarchy of a GPU consists on the top layer of a *Device (`DEV`)* which has access to device memory `DM` (indicated by a dashed arrow in Figure 4), and the device consists of *Streaming Multiprocessors (`SMX`)* (sometimes also called *Compute Units*). Each `SMX` has access to its own

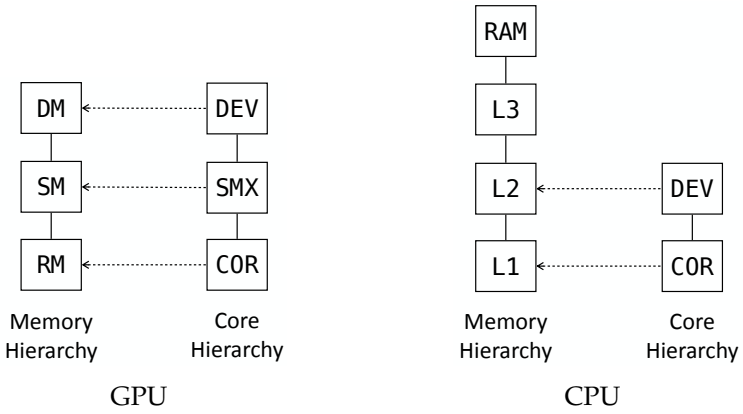


Figure 4: Basic architectural structure of GPU (left) and CPU (right): GPUs rely on 3 memory layers (`DM`, `SM`, `RM`) and 3 core layers (`DEV`, `SMX`, `COR`); CPUs provide 4 memory layers (`RAM`, `L3`, `L2`, `L1`) and 2 core layers (`DEV`, `COR`). Dashed arrows indicate access rights.

shared memory region `SM` (as well as to all memory regions above – in this case, device memory `DM`), and an `SMX` consists of *Cores (`COR`)* which have access to register memory `RM`. In contrast, CPUs usually rely on only 2 core layers (`DEV` and `COR`), but provide 4 memory layers: main memory (`RAM`) and 3 fast memory regions (`L3`, `L2`, `L1`).

Figure 5 shows extended variants of GPU and CPU architectures. The extended GPU architecture (left part of the figure) provides a further core layer, between layers `SMX` and `COR`, the *Warp (`WRP`)* [345] layer (also known as *Wavefront* [361]); the extended CPU architecture (right part of Figure 5) relies on the additional core layer *Single Instruction Multiple Data (`SIMD`)* below layer `COR`.

The relation between memory and core layers is often limited by the hardware (indicated by dashed arrows in Figures 4 and 5). For example, while a GPU's cores (`COR`) can access register memory (`RM`) and all memory regions above (`SM` and `DM`), GPU's multiprocessors (`SMX`) can access shared memory (`SM`) and device memory (`DM`) only.

In addition to architectural differences, GPUs and CPUs usually differ significantly also in further important characteristics: number of cores, sizes of fast memory regions, memory latencies, clock frequencies, automatically managed memory (as in CPU) vs manually managed memory (GPU), etc. For example, while modern Intel CPU's usually rely on 18 cores or less, NVIDIA GPUs consist often of more than 100 multiprocessors per GPU, and each individual multiprocessor again consists of multiple cores.

We show in Chapter 4 that even architectures differ significantly in major characteristics, we can unify optimization strategies for them via a generalized de-composition and re-composition approach for computations.

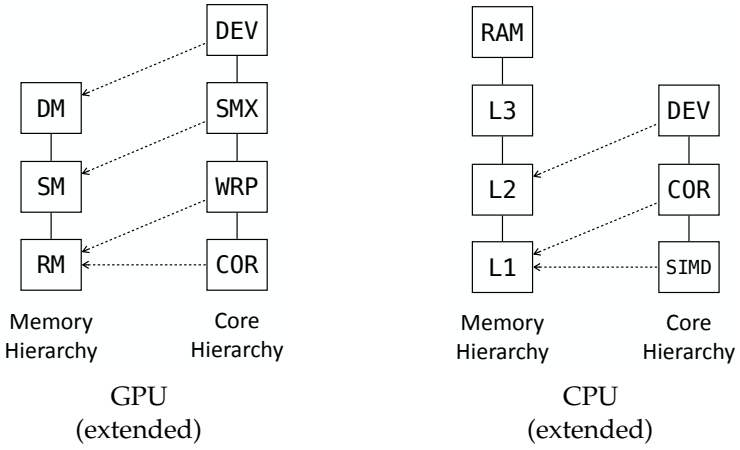


Figure 5: Extended architecture variants of GPU (left) and CPU (right): the extended GPU architecture relies on the additional core layer `WRP` (Warp) and the extended CPU variant provides core layer `SIMD` (Single Instruction Multiple Data). Dashed arrows indicate access rights.

We refrain from discussing hardware features that are out of programmer's control, because such features are not relevant for this programming-oriented work. For example, CPUs rely on complex hardware logics for, e.g., *Instruction-Level Parallelism (ILP)* or *Out-of-Order Execution*, which cannot be explicitly controlled and are not intended by the hardware architecture to be influenced by the programmer. In contrast, even though CPUs provide automatically managed fast memory resources (caches), the behavior of these memory resources can be positively influenced by special optimization techniques (e.g., *memory tiling* which we introduce later) and thus are addressed in this thesis. We also do not discuss very recent, domain-specific hardware features, such as NVIDIA's *Tensor Cores* [148] for computing machine-learning-specific workloads, as we consider targeting such features as future work (outlined in Chapter 8).

2.1.2 *Multi-Device/Multi-Node Systems*

Figure 6 shows examples of a multi-device and multi-node system[1], which extend the memory and core hierarchies of devices by further layers (Figure 6 extends the example devices shown in Figure 5). In the case of the multi-device system (left part of Figure 6), device's memory hierarchy (i.e., of the GPU in Figure 5) is extended by layer `NM` *(Node Memory)* which represents the memory region of the compute node containing the multiple devices, and the device is also extended by core layer `NOD` *(Node)*. The multi-node system (right part of Figure 6) extends a multi-device system (left part of Figure 6) further, by memory layer `CM` *(Cluster Memory)* and core layer `CLT` *(Cluster)*. Note that for illustration, this particular example multi-node system in Figure 6 extends a multi-device system that consists of CPUs (and not of GPUs as shown in the left part of Figure 6).

[1] Multi-node systems are also known as *Distributed Systems* in the literature.

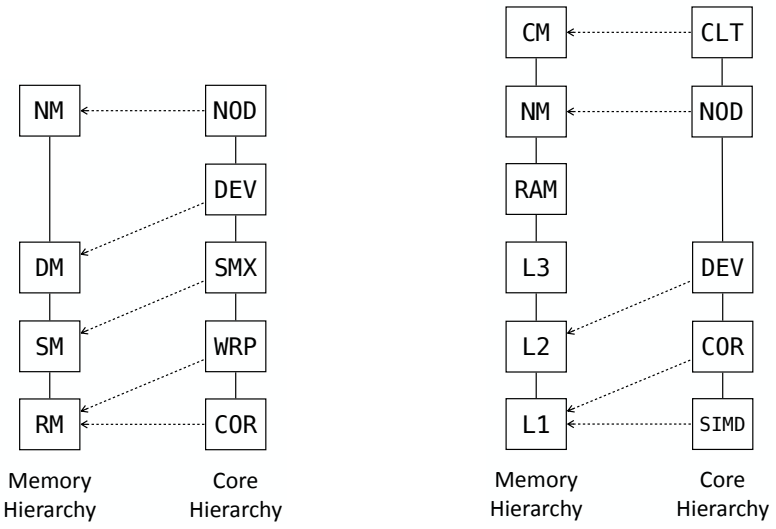


Figure 6: Example of Multi-Device System (left) and Multi-Node System (right): the multi-device system introduces memory layer `NM` (Node Memory) and core layer `NOD` (Node), and the multi-node system layers `CM` (Cluster Memory) and `CLT` (Cluster). Dashed arrows indicate access rights.

We currently assume multi-device and multi-node systems to be *homogeneous*, i.e., systems contain identical devices only and different nodes are assumed to contain an equal number of devices. This assumption of homogeneity allows us to use an equal work distribution across devices. In Chapter 8, we present our work-in-progress results for *heterogeneous* multi-device and multi-node systems which require advanced work distribution strategies across devices to achieve their full performance potential.

2.2 LOW-LEVEL PARALLEL PROGRAMMING MODELS

We discuss the state-of-practice low-level parallel programming models currently used for programming modern computer systems (discussed in Chapter 2.1): *Single-Device Models* (in Chapter 2.2.1), *Multi-Device Models* (Chapter 2.2.2), and *Multi-Node Models* (Chapter 2.2.3).

2.2.1 *Single-Device Models*

OpenCL [342] and CUDA [350] are two popular state-of-practice parallel programming models that target single devices and operate on a low abstraction level. Both models extend the C programming language, e.g., by constructs for managing memory regions and threads on different memory and core layers. For example, in Listing 1, we see in lines 1-3 the OpenCL-specific memory qualifier `__global` which indicates that data are located in OpenCL's top memory region – OpenCL introduces three memory regions, namely: `__global` (top layer), `__local` (mid layer), and `__private` (bottom layer) – and in

lines 5 and 6 of the listing, we acquire thread ids. In contrast to CUDA which can only be used for programming NVIDIA GPUs, OpenCL targets a wide range of architectures, including Intel and AMD CPUs, ARM mobile devices, and FPGAs.

The CUDA programming model is very similar to the OpenCL model, but CUDA uses a slightly different syntax, and it provides some convenience constructs (such as *templates* and *lambda functions* [346]) which are not available in OpenCL. Moreover, since CUDA is limited to NVIDIA GPUs, GPU-specific assembly intrinsics are often used in CUDA code to target special GPU hardware features – the intrinsics often achieve high performance, but are not part of the standard CUDA specification. For example, the latest NVIDIA GPUs contain so-called *Tensor Cores* [148] which compute small matrix multiplications in hardware – tensor cores need to be programmed via intrinsics (located in NVIDIA's so-called `nvcuda::wmma` namespace [8]) that are not part of the CUDA standard specification. *Warps* are a further popular hardware feature of NVIDIA GPUs (discussed in Chapter 2.1.1). These extend the core hierarchy of GPUs, and they have to be exploited for high performance using non-standard CUDA programming techniques [148]: for example, while `SMX` and `COR` are part of the standard CUDA specification, and thread ids on these two layers can be naturally acquired in CUDA code, via calls `blockIdx` (`SMX` layer) and `threadIdx` (`COR` layer), thread ids on the `WRP` layer have to be computed manually in CUDA, as `warp_id = threadIdx.x / 32`. We show later in this thesis that such hidden hardware features, such as warps, can be naturally targeted with our approach, in a systematic way.

2.2.2 *Multi-Device Models*

Multi-device systems are programmed via so-called *host code* which scatters data to the system's devices, starts and synchronizes computations among devices, etc. In approaches OpenCL and CUDA, host code programming is defined in the OpenCL and CUDA specifications, in the form of C/C++ programming library interfaces.

Listing 3 illustrates an example CUDA host code (shortened via ellipsis for brevity) taken from [348]. After some initialization (line 3), the so-called CUDA streams are created for each GPU (line 8), which queue device operations (like data transfers and synchronization), and memory is allocated on the devices (lines 10-11) and host (lines 12-13) via CUDA's host code functions `cudaMalloc` and `cudaMallocHost`. Afterward, data are copied from the host to the devices (lines 24), device computations are started (line 26), and the computed data are copied from the devices to the host (line 29). Finally, the devices are synchronized (line 38), the GPUs' computed intermediate results combined to the final result (line 40), and the CUDA's low-level object are freed (lines 43-46).

Host code can become significantly more complex than in Listing 3; for example, when data has to be transferred among devices, or different device computations are started consecutively. We present in Chapter 6 our high-level abstractions for considerably simplifying host code programming.

```
int main(int argc, char **argv) {

  // ...

  // Create streams for issuing GPU command asynchronously and
      allocate memory
  for (i = 0; i < GPU_N; i++) {
      checkCudaErrors(cudaSetDevice(i));
      checkCudaErrors(cudaStreamCreate(&plan[i].stream));
      // Allocate memory
      checkCudaErrors(cudaMalloc(/*...*/));
      checkCudaErrors(cudaMalloc(/*...*/));
      checkCudaErrors(cudaMallocHost(/*...*/));
      checkCudaErrors(cudaMallocHost(/*...*/));
      // ...
  }

  // ...

  // Copy data to GPU, launch the kernel and copy data back.
  for (i = 0; i < GPU_N; i++) {
      // Set device
      checkCudaErrors(cudaSetDevice(i));
      // Copy input data from CPU
      checkCudaErrors(cudaMemcpyAsync(/*...*/));
      // Perform GPU computations
      reduceKernel<<<BLOCK_N, THREAD_N, 0, plan[i].stream>>>(/*...*/);
      getLastCudaError("reduceKernel() execution failed.\n");
      // Read back GPU results
      checkCudaErrors(cudaMemcpyAsync(/*...*/));
  }

  // Process GPU results
  for (i = 0; i < GPU_N; i++) {
      float sum;
      // Set device
      checkCudaErrors(cudaSetDevice(i));
      // Wait for all operations to finish
      cudaStreamSynchronize(plan[i].stream);

      // ...

      // Shut down this GPU
      checkCudaErrors(cudaFreeHost(plan[i].h_Sum_from_device));
      checkCudaErrors(cudaFree(plan[i].d_Sum));
      checkCudaErrors(cudaFree(plan[i].d_Data));
      checkCudaErrors(cudaStreamDestroy(plan[i].stream));
  }

  // ...
}
```

Listing 3: CUDA host code for multi-device execution.

2.2.3 *Multi-Node Models*

Modern computer systems often consist of multiple compute nodes, with each node representing a multi-device system. For programming multi-node systems, usually multiple programming models are combined in host code programming, such as *Message Passing Interface (MPI)* [344] for distributing computations to the system's multiple nodes with a multi-device model (e.g., OpenCL or CUDA, as in Listing 3) for programming the individual nodes.

Listing 4 shows a straightforward MPI host code (slightly modified and shortened for brevity) taken from [347]. After initialization (line 5), data are scattered to the nodes (line 18) and computations are started on the nodes via function `computeOnNode` (line 26). The function executes a multi-device program (the program is not shown in the Listing for brevity) which is analogous to the program in Listing 3. Finally, the intermediate results computed by the different nodes are combined to the final result (line 33).

Our host code programming abstraction that we present later in Chapter 6 targets multi-node systems. For this, our abstraction internally uses the *Boost.Asio* [332] low-level networking library, rather than a higher-level approach such as MPI. We use Boost.Asio similarly as MPI (Listing 4: scattering data to nodes, starting node computations, combining nodes' intermediate results), but we use the low-level interface of Boost.Asio to exactly control the interactions between nodes, e.g., to exactly determine when data are transferred between nodes or when computations on them are started on the nodes. Thereby, our abstraction is in full control of the node, instead of delegating interactions to, e.g., an opaque MPI implementation.

```
int main(int argc, char *argv[]) {
  // ...

  // Initialize MPI state
  MPI_CHECK(MPI_Init(&argc, &argv));

  // Get our MPI node number and node count
  int commSize, commRank;
  MPI_CHECK(MPI_Comm_size(MPI_COMM_WORLD, &commSize));
  MPI_CHECK(MPI_Comm_rank(MPI_COMM_WORLD, &commRank));

  // ...

  // Allocate a buffer on each node
  float *dataNode = new float[dataSizePerNode];

  // Dispatch a portion of the input data to each node
  MPI_CHECK(MPI_Scatter(dataRoot, dataSizePerNode, MPI_FLOAT, dataNode
      , dataSizePerNode, MPI_FLOAT, 0, MPI_COMM_WORLD));

  if (commRank == 0) {
      // No need for root data any more
      delete[] dataRoot;
  }

  // On each node, run multi-device computation
  computeOnNode(dataNode, blockSize, gridSize);

  // Reduction to the root node, computing the sum of output elements
  float sumNode = sum(dataNode, dataSizePerNode);
  float sumRoot;

  MPI_CHECK(
      MPI_Reduce(&sumNode, &sumRoot, 1, MPI_FLOAT, MPI_SUM, 0,
          MPI_COMM_WORLD));

  if (commRank == 0) {
      float average = sumRoot / dataSizeTotal;
      cout << "Average of square roots is: " << average << endl;
  }

  // Cleanup
  delete[] dataNode;
  MPI_CHECK(MPI_Finalize());

  // ...
}
```

Listing 4: MPI host code for multi-node execution.

2.3 HIGH-LEVEL PARALLEL PROGRAMMING MODELS

We discuss two kinds of high-level parallel programming models, namely: 1) *domain-specific models* which are specifically designed and optimized for a particular application domain, such as vendor libraries NVIDIA cuBLAS [43] and Intel oneMKL [28] for computing linear algebra routines; 2) *general-purpose models* which aim to target multiple application domains, e.g., OpenMP [357] and OpenACC [355], as well as fully automatic approaches such as polyhedral compilers PPCG [218] and Pluto [260]. Domain-specific models usually achieve both high performance and user productivity, but are limited to computations within their target domain. Moreover, the current domain-specific approaches are often limited also to certain architectures, e.g., only GPU (as cuBLAS) or only CPU (as oneMKL). In contrast, general-purpose models target a broad range of application domains, but usually at the cost of performance and/or productivity. We discuss both kinds of models in the following, with a particular focus on their user interfaces. We present experimental results for both kinds of approaches in Section 7.2.5 of this thesis.

2.3.1 *Domain-Specific Models*

A plethora of programming models are specifically designed and optimized toward particular application domains. In the following, we focus on some illustrative example models, for the popular domains of linear algebra routines and deep-learning computations.

NVIDIA cuBLAS [43] and Intel oneMKL [28] are two domain-specific models targeting linear algebra routines. Both approaches are highly optimized by hand, at the assembly level, to efficiently target NVIDIA GPU (cuBLAS) or Intel CPU (oneMKL), correspondingly, based on easy-to-use, domain-specific user interfaces.

Listings 5 and 6 show how NVIDIA cuBLAS and Intel oneMKL are used for expressing matrix multiplication; the implementations are taken from [352] and [339]. The user conveniently sets the memory layouts of matrices (line 3 in both listings) and their sizes (line 4); the matrices are straightforwardly passed to cuBLAS and oneMKL (e.g., in lines 6,7,9 of Listing 5).

```
1  cublasStatus_t cublasSgemm(
2          cublasHandle_t handle,
3          cublasOperation_t transa, cublasOperation_t transb,
4          int m, int n, int k,
5          const float* alpha,
6          const float* A, int lda,
7          const float* B, int ldb,
8          const float* beta,
9          float *C, int ldc)
10 }
```

Listing 5: Matrix Multiplication in NVIDIA cuBLAS.

```
namespace oneapi::mkl::blas::column_major {
    void gemm(sycl::queue &queue,
              onemkl::transpose transa, onemkl::transpose transb,
              std::int64_t m, std::int64_t n, std::int64_t k,
              Ts alpha, sycl::buffer<Ta,1> &a, std::int64_t lda,
                        sycl::buffer<Tb,1> &b, std::int64_t ldb,
              Ts beta , sycl::buffer<Tc,1> &c, std::int64_t ldc)
}
```

Listing 6: Matrix Multiplication in Intel oneMKL.

Approaches such as Apache TVM [120] and Facebook's Tensor-Comprehensions (TC) [110] target the entire domain of deep-learning computations and thus are more general than cuBLAS and oneMKL which work for linear algebra routines only. The user interfaces of TVM and TC rely on domain-specific languages for expressing deep-learning computations, agnostic of hardware and optimization details.

Listings 7 and 8 show example programs in TVM and TC, taken from [120] and [110]. The TVM program (Listing 7) computes transposed matrix multiplication, and the TC program (Listing 8) computes *Max-Pool*; both computations are heavily used in deep learning [360, 363] and used as running examples by the TVM and TC developers. We observe from the two listing that TVM and TC are slightly more complex to use than cuBLAS and oneMKL (illustrated in Listings 5 and 6). However, the additional complexity allows TVM and TC to be used for expressing arbitrary deep-learning computations, whereas cuBLAS and oneMKL are limited to the domain of linear algebra routines only (which is a sub-domain of the deep learning computation domain).

```
    m, n, h = t.var('m'), t.var('n'), t.var('h')
    A = t.placeholder((m, h), name='A')
    B = t.placeholder((n, h), name='B')
    k = t.reduce_axis((0, h), name='k')
    C = t.compute((m, n), lambda y, x: t.sum(A[k, y] * B[k, x], axis=k
        ))
```

Listing 7: Transposed Matrix Multiplication in TVM.

```
    def maxpool2x2(float (B,C,H,W) in)->(out) {
        out(b,c,i,j) max=! in(b,c, 2 * i + kh, 2 * j + kw) where kh in
             0:2, kw in 0:2
    }
```

Listing 8: Max-Pool in Tensor Comprehensions (TC).

2.3.2 *General-Purpose Models*

General-purpose programming models can be categorized into two classes: 1) auto-parallelization approaches, such as approaches OpenMP [357] and OpenACC [355], as well as polyhedral compilers PPCG [218] and Pluto [260]; 2) DSL-based models, e.g., the Lift approach [192]. Approaches relying on auto-parallelization usually take as input sequential program code (such as PPCG and Pluto), e.g., written in the C programming language, and the approaches then automatically analyze and parallelize the code, sometimes guided by easy-to-use, user-defined code annotations (as in OpenMP and OpenACC). Such automatic approaches usually achieve high user productivity, but they often struggle to achieve the full performance potential of state-of-the-art computer systems, which we confirm experimentally in Sections 4.5 and 7.2.5. This is because semantic information is required for efficient optimization and parallelization, which are often too challenging to extract automatically from sequential code (Rice's theorem [274]), as we discuss in detail later in this thesis. In contrast, DSL-based approaches rely on high-level program representations (e.g., based on parallel patterns, such as `map` and `reduce` [242]), thereby implicitly requesting from the programmer the semantic information required for generating well-performing code from DSL programs. However, DSL programming is sometimes still experienced as less productive by users with extensive practice in mainstream programming languages, e.g. the C language.

We discuss both kinds of general-purpose programming models in the following.

Auto-Parallelization Approaches

Listing 9 shows a typical implementation of matrix multiplication in the C programming language: we iterate over all matrix dimensions (lines 4, 5, 7), initialize the output buffer (line 6), and perform the additions and multiplications required for computing matrix multiplication (line 8).

```
1  void matmul(int M, int N, int K,
2              float A[M][K], float B[K][N], float C[M][N])
3  {
4      for (int i = 0; i < M; i++)
5          for (int j = 0; j < N; j++) {
6              C[i][j] = 0;
7              for (int k = 0; k < K; k++)
8                  C[i][j] = C[i][j] + A[i][k] * B[k][j];
9          }
10 }
```

Listing 9: Matrix Multiplication in the C programming language, which can be automatically parallelized by PPCG for GPU and Pluto for CPU.

The sequential C code in the listing can be automatically optimized and parallelized by general-purpose approaches, such as PCCG for GPU [218] or Pluto for CPU [260].

Approaches such as OpenMP [357] and OpenACC [355] also rely on automatic parallelization, but they need code annotations (a.k.a. *directives*) from the user.

Listing 10 shows how OpenMP directives are used for parallelizing the average computation of array elements for CPU; the code is taken from [328]. The sequential C code is straightforwardly annotated with a so-called *C pragma* (line 2). The pragma enables OpenMP to automatically parallelize the program and to combine the intermediate results of threads via operator + (addition) – a semantic information that is explicitly provided by the user via the pragma. Auto-parallelization approaches such as PCCG and Pluto often struggle with parallelizing the computation in Listing 10, because it is challenging for them to identity the addition operator automatically: semantic code analysis is required, which is complex to be done automatically [168, 178, 308] and often even impossible for many applications (Rice's theorem [274]).

```
1 double ave=0.0, A[MAX]; int i;
2 #pragma omp parallel for reduction (+:ave)
3 for (i=0;i< MAX; i++) {
4   ave += A[i];
5 }
6 ave = ave/MAX;
```

Listing 10: Average Computation via OpenMP-annotated C Code.

OpenACC is a further, annotation-based approach, which works very similarly to OpenMP, but is focussed on GPU architectures.

Listing 11 shows an OpenACC-annotated program for computing *SAXPY (Single-precision A times X Plus Y)*, taken from [354]. The sequential C code for SAXPY is annotated with `#pragma acc kernels` (line 3) which enables OpenACC to automatically optimize and parallelize the sequential code for GPU.

```
1 void saxpy_parallel(int n, float a, float *x, float *restrict y)
2 {
3   #pragma acc kernels
4     for (int i = 0; i < n; ++i)
5       y[i] = a*x[i] + y[i];
6 }
```

Listing 11: SAXPY Computation via OpenACC-annotated C Code.

We introduce in this thesis a user interface for our approach to automatically optimize and parallelize sequential C code (discussed later in Section 7.2). We will see that our interface aims to combine the advantages of both *annotation-free approaches* (such as PPCG and Pluto) and *annotation-based approaches* (e.g., OpenMP and OpenACC): for high productivity, our approach works without annotations (as PPCG and Pluto) and for high performance, annotations (as in OpenMP and OpenACC) are optionally allowed.

DSL-Based Approaches

In contrast to auto-parallelization approaches, DSL-based approaches rely on Domain-Specific Languages (DSLs) for their input programs, rather than (annotated) sequential program code in mainstream programming languages, such as C.

Listings 12 and 13 show example DSL programs in the Lift approach [192] for computing *Matrix Multiplication* (Listings 12, taken from [169]) and *Jacobi Stencil* (Listings 13, from [124]); the programs are used by the Lift compiler to generate OpenCL code for them, currently mainly targeted to GPUs. The Lift DSL relies on a set of functional basic building blocks (a.k.a. *parallel patterns* or *skeletons* [242]), such as `reduce` (Listing 12, line 2) and `map` (line 8), which are used to express more complex computations, e.g., matrix multiplication (Listing 12) or Jacobi stencil (Listing 13). The pattern set of Lift often needs to be extended by the Lift developers to enable targeting new classes of applications, e.g., by patterns `pad` and `slide` for stencil computations [124] (Listing 13, lines 6 and 7), which both are not part of the original Lift specification [192].

```
1 val dotProduct = fun( (a, b) =>
2     reduce(add, 0.0f, map(mult, zip(a, b))) )
3
4 val mm = fun( Array(Array(Float, M), K),
5                Array(Array(Float, K), N),
6     (A, B) =>
7     map(fun(aRow =>
8         map(fun(bCol =>
9         dotProduct(aRow,bCol)), transpose(B))), A))
```

Listing 12: Matrix Multiplication expressed in Lift.

```
1 val sumNbh = fun(nbh => reduce(add, 0.0f, nbh))
2
3 val stencil =
4     fun( A: Array(Float ,N) =>
5     map(sumNbh ,
6         slide(3, 1,
7         pad(1, 1, clamp , A))))
```

Listing 13: Jacobi Stencil expressed in Lift.

The approach we introduce in this thesis also relies on functional building blocks, but in contrast to Lift, it relies on exactly three general patterns which uniformly express a broad range of different computations, as we thoroughly discuss in this Chapters 4 and Section 7.1 of this thesis.

Listing 14 shows a DSL program for computing *Gaussian Blur* in the high-level approach Tiramisu [83] (taken from [83]) which relies on an imperative-style DSL language, rather than a functional language as Lift. In contrast to Lift which is mainly used for single-GPU systems, Tiramisu targets also multi-node computer systems that may consist of multiple GPUs as well as CPUs.

```
// Declare the iterators i, j and c.
Var i(0, N-2), j(0, M-2), c(0, 3);

Computation bx(i, j, c), by(i, j, c);

// Algorithm.
bx(i,j,c)=(in(i,j,c)+in(i,j+1,c)+in(i,j+2,c))/3;
by(i,j,c)=(bx(i,j,c)+bx(i+1,j,c)+bx(i+2,j,c))/3;
```

Listing 14: Gaussian Blur in Tiramisu.

The current DSL-based approaches, such as Lift and Tiramisu, often achieve good performance. However, such approaches often require from the user explicitly guiding the optimization process, which is complex and thus hinders productivity.

```
1  val appliedReduce = isApp(isApp(isApp(isReduce)))
2  val blocking = ( baseline ';'
3      tile(32,32)      '@' outermost(mapNest(2))   ';;'
4      fissionReduceMap '@' outermost(appliedReduce)';;'
5      split(4)         '@' innermost(appliedReduce)';;'
6      reorder(List(1,2,5,6,3,4)))
7  (blocking ';' lowerToC)(mm)
8
9  val loopPerm = (
10   tile(32,32)      '@' outermost(mapNest(2))    ';;'
11   fissionReduceMap '@' outermost(appliedReduce) ';;'
12   split(4)         '@' innermost(appliedReduce) ';;'
13   reorder(Seq(1,2,5,3,6,4))                     ';;'
14   vectorize(32)    '@' innermost(isApp(isApp(isMap))))
15 (loopPerm ';' lowerToC)(mm)
```

Listing 15: Expressing *blocking optimization* and *loop permutation* in Lift's optimization language Elevate.

Listing 15 shows how Lift's optimization language *Elevate* [68] is used for expressing *blocking optimization* and *loop permutation* (the example is taken from [68]). The user is in charge of explicitly expressing tiling optimization and reordering computations (lines 3, 10 and lines 6, 13), splitting the computations across threads (lines 5 and 12), etc. For a common application developer, it is challenging to identity the particular optimizations (blocking, loop permutation, ...) required to achieve high performance on the particular target architecture, and how these optimizations are expressed via Elevate's basic building blocks (`tile`, `reorder`, `split`, ...) and instantiated with well-performing values (e.g., 32 in line 3, or value 4 in line 5) – expert knowledge is required from the developer about both low-level hardware features (caches, cores, etc) and how these features are efficiently targeted via code optimizations, as well as how optimizations are expressed in Elevate.

```
1  // Scheduling commands for targeting GPU.
2  // Tile i and j and map the resulting dimensions
3  // to GPU
4  Var i0, j0, i1, j1;
5  by.tile_gpu(i, j, 32, 32, i0, j0, i1, j1);
6  bx.compute_at(by, j0);
7  bx.cache_shared_at(by, j0);
8
9  // Use struct-of-array data layout
10 // for bx and by.
11 bx.store_in({c,i,j});
12 by.store_in({c,i,j});
13
14 // Create data copy operations
15 operation cp1 = in.host_to_device();
16 operation cp2 = by.device_to_host();
17
18 // Specify the order of execution of copies
19 cp1.before(bx, root);
20 cp2.after(by, root);
```

Listing 16: Expressing optimizations *tiling* and *parallelization* in Tiramisu's optimization language.

Listing 16 shows how Tiramisu's optimization language is used for expressing optimizations *tiling* and *parallelization* (taken from [83]). Tiramisu's language works similarly to Elevate, but it is based on an imperative language design, rather than a functional design as Elevate. For example, in line 4, variables are declared which are used in lines 5-7 to express tiling, and data copy operations are performed in lines 15- 16.

We will see in this thesis that our introduced approach enables fully automatically generating optimized code, without requiring from the user to explicitly express optimizations (as in Listings 15 and 16), thereby significantly contributing to user's productivity.[2]x

[2]Our work-in-progress results [10] allow to optionally expose optimization decisions to the user such that expert knowledge can be incorporated into our optimization process, e.g., to reduce the time for conducting automatic optimization.

Part II

A NOVEL, HOLISTIC APPROACH TO CODE GENERATION & OPTIMIZATION & EXECUTION

3

OVERVIEW

This part introduces our novel, holistic approach to code *generation* (in Chapter 4), *optimization* (in Chapter 5), and *execution* (in Chapter 6), which is summarized in Figure 3.

1. The first section of this part (*generation*) introduces *Multi-Dimensional Homomorphism (MDH)* and deals with the question of:

   How to generate code that can be automatically optimized (auto-tuned) for a particular target architecture and characteristics of the input and output data (e.g., size memory and layout)?

2. The second section (*optimization*) afterward introduces *Auto-Tuning Framework (ATF)* and is about the question:

   How can automatically optimizable (auto-tunable) code eventually be optimized (auto-tuned)?

3. The third section (*execution*) finally introduces *Host Code Abstraction (HCA)* and deals with the question:

   How can (auto-tuned) program code be executed on the devices of distributed multi-device systems?

We conclude this part with Chapter 7 by introducing two different interface kinds for using our MDH+ATF+HCA approach holistically.

4

CODE GENERATION VIA ALGEBRAIC CONCEPTS

Data-parallel computations, such as linear algebra routines (BLAS) and stencil computations, constitute one of the most relevant classes in parallel computing, e.g., due to their importance for deep learning. Efficiently de-composing such computations for the memory and core hierarchies of modern architectures and re-composing the computed intermediate results back to the final result – we say *(de/re)-composition* for short – is key to achieve high performance for these computations on, e.g., GPU and CPU. Current high-level approaches to generating data-parallel code are often restricted to a particular subclass of data-parallel computations and architectures (e.g., only linear algebra routines on only GPU, or only stencil computations), and/or the approaches rely on a user-guided optimization process for a well-performing (de/re)-composition of computations, which is complex and error prone for the user.

We formally introduce a systematic (de/re)-composition approach, based on our novel algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)*[1]. Our approach is designed as general enough to be applicable to a wide range of data-parallel computations and for various kinds of target parallel architectures. To efficiently target the deep and complex memory and core hierarchies of contemporary architectures, we exploit our introduced (de/re)-composition approach for a correct-by-construction, parametrized cache blocking and parallelization strategy. We show that our approach is powerful enough to express, in the same formalism, the (de/re)-composition strategies of different classes of state-of-the-art approaches (scheduling-based, polyhedral, etc), and we demonstrate that the parameters of our strategies enable systematically generating code that can be fully automatically optimized (auto-tuned) for the particular target architecture and characteristics of the input and output data (e.g., their sizes and memory layouts). Particularly, our experiments confirm that via auto-tuning, we achieve higher performance than state-of-the-art approaches, including hand-optimized solutions provided by vendors (such as NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN), on real-world data sets and for a variety of data-parallel computations, including: linear algebra routines, stencil and quantum chemistry computations, data mining algorithms, and computations that recently gained high attention due to their relevance for deep learning.

[1] https://mdh-lang.org

4.1 INTRODUCTION

Data-parallel computations constitute one of the most relevant classes in parallel computing. Important examples of such computations include linear algebra routines (BLAS) [297], various kinds of stencil computations (e.g., Jacobi method and convolutions) [124], quantum chemistry computations [91], and data mining algorithms [116]. The success of many application areas critically depends on achieving high performance for their data-parallel building blocks, on a variety of parallel architectures. For example, highly-optimized BLAS implementations combined with the computational power of modern GPUs currently enable deep learning to significantly outperform other existing machine learning approaches (e.g., for speech recognition and image classification).

Data-parallel computations are characterized by applying the same function (a.k.a *scalar function*) to each point in a multi-dimensional grid of data (a.k.a. *array*), and combining the obtained intermediate results in the grid's different dimensions using so-called *combine operators*.

$$\begin{pmatrix} M_{1,1} & \dots & M_{1,K} \\ \vdots & \ddots & \vdots \\ M_{I,1} & \dots & M_{I,K} \end{pmatrix}, \begin{pmatrix} v_1 \\ \vdots \\ v_K \end{pmatrix} \xrightarrow{\texttt{MatVec}} \begin{array}{c} \overrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad} {\scriptstyle \circledast_2} \\ \begin{vmatrix} f(M_{1,1}, v_1) & \dots & f(M_{1,K}, v_K) \\ \vdots & \ddots & \vdots \\ f(M_{I,1}, v_1) & \dots & f(M_{I,K}, v_K) \end{vmatrix}_{\circledast_1} \downarrow \end{array} = \begin{pmatrix} M_{1,1} * v_1 + \dots + M_{1,K} * v_K \\ \vdots \\ M_{I,1} * v_1 + \dots + M_{I,K} * v_K \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_I \end{pmatrix}$$

Figure 7: Data parallelism illustrated using the example *Matrix-Vector Multiplication* (`MatVec`).

Figures 7 and 8 illustrate data parallelism using as examples two popular computations: i) linear algebra routine *Matrix-Vector multiplication (`MatVec`)*, and ii) stencil computation *Jacobi (`Jacobi1D`)*. In the case of `MatVec`, the grid is 2-dimensional and consists of pairs, each pointing to one element of the input matrix $M_{i,k}$ and the vector v_k. To each pair, scalar function $f(M_{i,k}, v_k) := M_{i,k} * v_k$ (multiplication) is applied, and results in the i-dimension are combined using combine operator $\circledast_1((x_1,\dots,x_n),(y_1,\dots,y_m)) := (x_1,\dots,x_n,y_1,\dots,y_m)$ (concatenation) and in k-dimension using operator $\circledast_2((x_1,\dots,x_n),(y_1,\dots,y_n)) := (x_1+y_1,\dots,x_n+y_n)$ (pointwise addition). Similarly, the scalar function of `Jacobi1D` is $f(v_{i+0}, v_{i+1}, v_{i+2}) := c * (v_{i+0} + v_{i+1} + v_{i+2})$ which computes the Jacobi-specific function for an arbitrary but fixed constant c; `Jacobi1D`'s combine operator $\circledast_1$ is concatenation. We formally define scalar functions and combine operators later in this chapter.

Achieving high performance for data-parallel computations is considered important in both academia and industry, but has proven to

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \xrightarrow{\texttt{Jacobi1D}} \begin{vmatrix} f(v_1, v_2, v_3) \\ f(v_2, v_3, v_4) \\ \vdots \end{vmatrix}_{\circledast_1} \downarrow = \begin{pmatrix} c * (v_1 + v_2 + v_3) \\ c * (v_2 + v_3 + v_4) \\ \vdots \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_{N-2} \end{pmatrix}$$

Figure 8: Data parallelism illustrated using the example *Jacobi 1D* (`Jacobi1D`).

be challenging. In particular, achieving *high performance* that is *portable* (i.e., the same program code achieves a consistently high level of performance across different architectures and characteristics of the input/output data, e.g., their size and memory layout) and in a *user-productive* way is identified as an ongoing, major research challenge. This is because for high performance, an efficient *(de/re)-composition* of computations (illustrated in Figure 9 and discussed thoroughly in this chapter) is required to efficiently break down a computation for the deep and complex memory and core hierarchies of state-of-the-art architectures, via efficient cache blocking and parallelization strategies. Moreover, to achieve performance that is portable across architectures, the programmer has to consider that architectures often differ significantly in their characteristics [107] – depth of memory and core hierarchies, automatically managed caches (as in CPUs) vs manually managed caches (as in GPUs), etc – which poses further challenges on identifying an efficient (de/re)-composition of computations. Productivity is often also hampered: state-of-the-art programming models (such as OpenMP [44] for CPU, CUDA [38] for GPU, and OpenCL [30] for multiple kinds of architectures) operate on a low abstraction level; thereby, the models require from the programmer explicitly implementing a well-performing (de/re)-composition, which involves complex and error-prone index computations, explicitly managing memory and threads on multiple layers, etc.

Current high-level approaches to generating data-parallel code usually struggle with addressing in one combined approach all three challenges: *performance*, *portability*, and *productivity*. For example, approaches such as Halide [217], Apache TVM [120], Fireiron [69], and LoopStack [52] achieve high performance, but incorporate the user into the optimization process – by requiring from the user explicitly expressing optimizations in a so-called *scheduling language* – which is error prone and needs expert knowledge about low-level code optimizations, thus hindering user's productivity. In contrast, *polyhedral approaches*, such as Pluto [262], PPCG [218], and Facebook's TC [110], are often fully automatic and thus productive, but usually specifically designed toward a particular architecture (e.g., only GPU as TC and PPCG, or only CPU as Pluto) and thus not portable. *Functional approaches*, e.g., Lift [192], are productive for functional programmers (e.g., with experience in *Haskell* [25] programming, which relies on small, functional building blocks for expressing computations), but the approaches often have difficulties in automatically achieving the full performance potential of architectures [115]. Furthermore, many of the existing approaches are specifically designed toward a particular subclass of data-parallel computations only, e.g., only tensor operations (as LoopStack and TC) or only matrix multiplication (as Fireiron), or they require significant extensions for new subclasses (as Lift for matrix multiplication [169] and stencil computations [124]), which further hinders the productivity of the user.

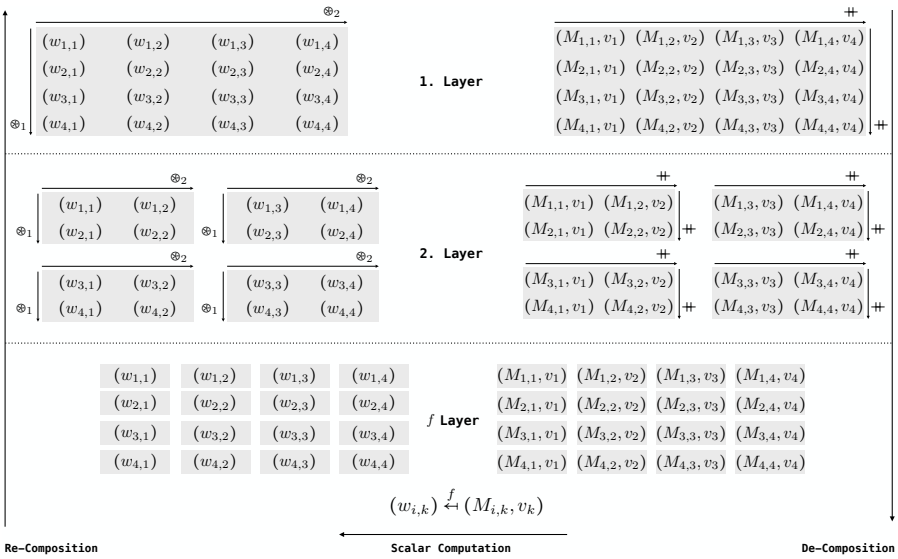


Figure 9: Example (de/re)-composition of `MatVec` (Figure 7) on a 4×4 input matrix M and a 4-sized vector v: i) the *de-composition phase* (right part of the figure) partitions the concatenated input data into parts (a.k.a. *tiles* in programming), where $+\!\!+$ denotes the concatenation operator; ii) to each part, scalar function f is applied in the *scalar phase* (bottom part of figure), which is defined for `MatVec` as: multiplying matrix element $M_{i,k}$ with vector element v_k, resulting in element $w_{i,k}$; iii) the *re-composition phase* (figure's left part) combines the computed parts to the final result, using combine operator $\circledast_1$ for the first dimension (defined as *concatenation* in the case of `MatVec`) and operator $\circledast_2$ (point-wise *addition*) for the second dimension. All basic building blocks (*scalar function*, *combine operator*, ...) and concepts (e.g. *partitioning*) are defined in this chapter, based on algebraic concepts. For simplicity, this example presents a (de/re)-composition on 2 layers only, and we partition the input for this example into parts that have straightforward, equal sizes. Optimized values of semantics-preserving parameters (a.k.a. *tuning parameters*), such as the number of parts and the application order of combine operators, are crucial for achieving high performance, as we discuss in this chapter. Phases are arranged from right to left, inspired by the application order of function composition, as we also discuss later.

In this chapter, we formally introduce a systematic (de/re)-composition approach for data-parallel computations targeting state-of-the-art parallel architectures. We express computations via *high-level functional expressions* (specifying *what* to compute), in the form of easy-to-use higher-order functions, based on our novel algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)*. Our higher-order functions are capable of expressing various kinds of data-parallel computations (linear algebra, stencils, etc), in the same formalism and on a high level of abstraction, independently of hardware and optimization details, thereby contributing to user's productivity[2]. As target for our high-level expressions, we introduce *functional low-level expressions* (specifying *how* to compute) to formally reason about (de/re)-compositions of data-parallel computations; our low-level expressions are designed such that they can be straightforwardly transformed to executable program code (e.g., in OpenMP, CUDA, and OpenCL). To systematically lower our high-level expressions to low-level expressions, we introduce a formally sound, parameterized *lowering process*. The parameters of our lowering process enable automatically computing low-level expressions that are optimized (auto-tuned [117]) for the particular target architecture and characteristics of the input/output data, thereby achieving fully automatically high, portable performance. For example, we formally introduce parameters for flexibly choosing the target memory regions for de-composed and re-composed computations, and also parameters for flexibly setting an optimized data access pattern.

We show that our high-level representation is capable of expressing various kinds of data-parallel computations, including computations that recently gained high attention due to their relevance for deep learning [84]. For our low-level representation, we show that it can express the cache blocking and parallelization strategies of state-of-the-art parallel implementations – as generated by scheduling approach TVM and polyhedral compilers PPCG and Pluto – in one uniform formalism. Moreover, we present experimental results to confirm that based on our parameterized lowering process in combination with auto-tuning, we are able to achieve higher performance than the state of the art, including hand-optimized implementations provided by vendors (e.g., NVIDIA cuBLAS and Intel oneMKL for linear algebra routines, and NVIDIA cuDNN and Intel oneDNN for deep learning computations).

[2]We consider as main users of our approach compiler engineers and library designers. Rasch, Schulze, and Gorlatch [81] show that our approach can also take straightforward, sequential code as input, which makes our approach attractive also to end users.

Summarized, we make the following three major contributions (illustrated in Figure 10):

1. We introduce a *high-level functional representation*, based on our novel algebraic formalism of Multi-Dimensional Homomorphisms (MDHs), that enables uniformly expressing data-parallel computations on a high level of abstraction.

2. We introduce a *low-level functional representation* that enables formally expressing and reasoning about (de/re)-compositions of data-parallel computations; our low-level representation is designed such that it can be straightforwardly transformed to executable program code in state-of-practice parallel programming models, including OpenMP, CUDA, and OpenCL.

3. We introduce a *systematic lowering process* to fully automatically lower an expression in our high-level representation to a device- and data-optimized expression in our low-level representation, in a formally sound manner, based on auto-tuning.

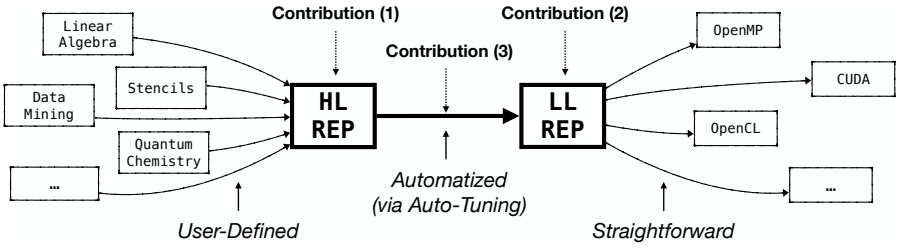


Figure 10: Overall structure of our approach (contributions highlighted in bold).

Our three contributions aim to answer the following questions:

1. *How can data parallelism be formally defined, and how can data-parallel computations be uniformly expressed via higher-order functions that are agonistic from of hardware and optimization details while still capturing all information relevant for generating high-performing, executable program code?* (Contribution 1);

2. *How can optimizations for the memory and core hierarchies of state-of-the-art parallel architectures be formally expressed and generalized such that they apply to arbitrary data-parallel computations?* (Contribution 2);

3. *How can optimizations for data-parallel computations be expressed and structured so that they can be automatically identified (auto-tuned) for a particular target architecture and characteristics of the input and output data?* (Contribution 3).

The rest of this chapter is structured as follows. We introduce our high-level functional representation (Contribution 1) in Section 4.2, and we show how this representation is used for expressing various kinds of popular data-parallel computations. In Section 4.3, we discuss our low-level functional representation (Contribution 2) which is powerful enough to express the optimization decisions of state-of-practice approaches (e.g., scheduling approach TVM and polyhedral compilers PPCG and Pluto) and beyond. Section 4.4 shows how we systematically lower a computation expressed in our high-level representation to an expression in our low-level representation, in a formally sound and auto-tunable manner (Contribution 3). We present experimental results in Section 4.5, discuss related work in Section 4.6, and we conclude in Section 4.7.

Our Appendix provides details for the interested reader that should not be required for understanding the basic concepts introduced in this chapter. In particular, our appendix contains formal details – for all the following definition, examples, and theorems in Sections 4.2-4.4 – whereas the formalism in this chapter is simplified for better illustration and easier understanding of our basic ideas and concepts.

4.2 HIGH-LEVEL REPRESENTATION FOR DATA-PARALLEL COMPUTATIONS[3]

We introduce functional building blocks, in the form of higher-order functions, that express data-parallel computations on a high abstraction level. The goal of our high-level abstraction is to express computations agnostic of hardware and optimization details, and thus in a user-productive manner, while still capturing all information relevant for generating high-performance program code. We introduce the building blocks of our abstraction based on our novel algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)* which formalizes data parallelism.

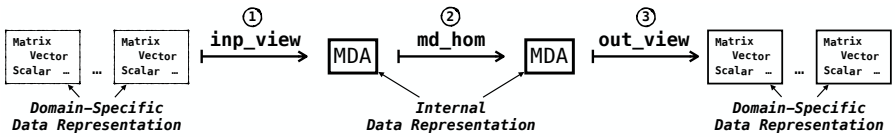


Figure 11: High-level representation (overview).

[3]The full version of this section, which contains all formal details, is provided in the Appendix, Section .2, for the interested reader.

Figure 11 shows a basic overview of our high-level representation. We express data-parallel computations using exactly three higher-order functions only (a.k.a. *patterns* or *skeletons* [243] in programming terminology): 1) `inp_view` transforms the domain-specific input data (e.g., a matrix and a vector in the case of matrix-vector multiplication) to a *Multi-Dimensional Array (MDA)* which is our internal data representation and defined later in this section; 2) `md_hom` expresses the data-parallel computation; 3) `out_view` transforms the computed MDA back to the domain-specific data representation.

In the following, after informally discussing an introductory example in Section 4.2.1, we formally define and discuss each higher-order function in detail in Section 4.2.2 (function `md_hom`) and Section 4.2.3 (functions `inp_view` and `out_view`). Note that Section 4.2.2 and Section 4.2.3 introduce and present the internals and formal details of our approach, which are not relevant for the end user of our system – the user only needs to operate on the abstraction level discussed in Section 4.2.1.

4.2.1 *Introductory Example*

Figure 12 shows how our MDH-based high-level representation is used for expressing the example of matrix-vector multiplication `MatVec`[5](Figure 7). Computation `MatVec` takes as input a matrix $M \in T^{I \times K}$ and vector $v \in T^{K}$ of arbitrary scalar type[6] T and sizes $I \times K$ (matrix) and K (vector), for arbitrary but fixed positive natural numbers $I, K \in \mathbb{N}$[7]. In the figure, based on index function $(i,k) \to (i,k)$ and $(i,k) \to (k)$, high-level function `inp_view` computes a function that takes M and v as input and maps them to a 2-dimensional array of size $I \times K$ (referred to as *input MDA* in the following and defined formally in the next subsection). The MDA contains at each point (i,k) the pair $(M_{i,k}, v_k) \in T \times T$ comprising element $M_{i,k}$ within

```
MatVec<T∈TYPE | I,K∈ℕ> :=
        out_view<T>( w:(i,k)↦(i) ) ∘
            md_hom<I,K>( ∗, (+,+) ) ∘
                inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

Figure 12: High-level expression for Matrix-Vector Multiplication (`MatVec`).[4]

[4]Our technical implementation takes as input a representation that is equivalent to Figure 12, expressed via straightforward program code (see Appendix, Section .1.4).

[5]The expression in Figure 12 can also be extracted from straightforward, annotated sequential code [81, 82].

[6]We consider as *scalar types* integers $\mathbb{Z}$ (a.k.a. `int` in programming), floating point numbers $\mathbb{Q}$ (a.k.a. `float` or `double`), any fixed collection of types (a.k.a. *record* or *struct*), etc. We denote the set of scalar types as `TYPE` in the following.

[7]We denote by $\mathbb{N}$ the set of positive natural number $\{1, 2, \dots\}$, and we use $\mathbb{N}_0$ for the set of natural numbers including 0.

matrix M (first component) and element v_k within vector v (second component). The input MDA is then mapped via function `md_hom` to an output MDA of size $I \times 1$, by applying multiplication $*$ to each pair $(M_{i,k}, v_k)$ within the input MDA, and combining the obtained intermediate results within the MDA's first dimension via $+\!\!+$ (concatenation – also defined formally in the next subsection) and in second dimension via $+$ (point-wise addition). Finally, function `out_view` computes a function that straightforwardly maps the output MDA, of size $I \times 1$, to `MatVec`'s result vector $w \in T^I$, which has scalar type T and is of size I. For the example of `MatVec`, the output view is trivial, but it can be used in other computations (such as matrix multiplication) to conveniently express more advanced variants of computations (e.g., computing the result matrix of matrix multiplication as transposed, as we demonstrate later).

4.2.2 *Function `md_hom`*

We introduce higher-order function `md_hom` to express *Multi-Dimensional Homomorphisms (MDHs)* – our formal representation of data-parallel computations – in a convenient and structured way. In the following, we illustrate the definition of MDHs and function `md_hom`.

To define MDH functions, we first need to introduce two central building blocks used in the definition of MDHs: i) *Multi-Dimensional Arrays (MDAs)* – the data type on which MDHs operate and which uniformly represent domain-specific input and output data (scalar, vectors, matrices, ...), and ii) *Combine Operators* which we use to combine elements within a particular dimension of an MDA.

Multi-Dimensional Arrays

Definition 1 (Multi-Dimensional Array)**.** A *Multi-Dimensional Array (MDA)* a that has *dimensionality* $D \in \mathbb{N}$, *size* $N \in \mathbb{N}^D$, *index sets* $I_1, \ldots, I_D \subset \mathbb{N}_0$, and *scalar type* $T \in \mathtt{TYPE}$ is a function with the following signature:

$$a : I_1 \times \ldots \times I_D \to T$$

We refer to $I_1 \times \ldots \times I_D \to T$ as the *type* of MDA a.

Notation 1. For better readability, we denote MDAs' types and accesses to them using a notation close to programming. We often write:

- $a \in T[\, I_1, \ldots, I_D \,]$ instead of $a : I_1 \times \ldots \times I_D \to T$ to denote the type of MDA a;
- $a \in T[\, N_1, \ldots, N_D \,]$ instead of $a : [0, N_1)_{\mathbb{N}_0} \times \cdots \times [0, N_D)_{\mathbb{N}_0} \to T$;[8]
- $a[\, i_1, \ldots, i_D \,]$ instead of $a(i_1, \ldots, i_D)$ to access MDA a at position $(i_1, \ldots, i_D)$.

[8] We denote by $[L, U)_{\mathbb{N}_0} := \{\, n \in \mathbb{N}_0 \mid L \leq n < U \,\}$ the half-open interval of natural numbers (including 0) between L (incl.) and U (excl.).

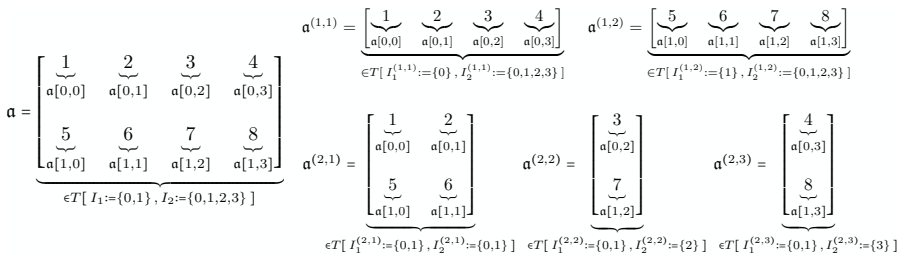


Figure 13: MDA examples.

Figure 13 shows six MDAs for illustration. For example, the left part of the figure shows MDA $\mathfrak{a}$ which is of type $\mathfrak{a} : I_1 \times I_2 \to T$, for $I_1 = \{0, 1\}$, $I_2 = \{0, 1, 2, 3\}$, and $T = \mathbb{Z}$ (integer numbers).

Note that MDAs named $\mathfrak{a}^{(1,1)}, \mathfrak{a}^{(1,2)}, \mathfrak{a}^{(2,1)}, \mathfrak{a}^{(2,2)}, \mathfrak{a}^{(2,3)}$ in Figure 13 can be considered as *parts* (a.k.a. *tiles* in programming) of MDA $\mathfrak{a}$: the MDA named $\mathfrak{a}^{(1,1)}$ represents the first row of $\mathfrak{a}$, MDA $\mathfrak{a}^{(2,2)}$ the third column of $\mathfrak{a}$, etc. We formally define and use *partitionings* of MDAs in Section 4.3.

Combine Operators

A central building block in our definition of MDHs is a *combine operator*. Intuitively, we use a combine operator to combine all elements within a particular dimension of an MDA. For example, in Figure 7 (matrix-vector multiplication), we combine elements of the 2-dimensional MDA via combine operator *concatenation* in MDA's first dimension and via operator *point-wise addition* in the second dimension. Technically, combine operators are functions that take as input two MDAs and yield a single MDA as their output.

We now define *combine operators* formally, and we illustrate this formal definition afterward using the example operators *concatenation* and *point-wise combination*.

Definition 2 (Combine Operator)**.** We refer to any binary function $\circledast$ of type

$$\circledast : T[\, I_1, \ldots, \underset{\uparrow d}{P}, \ldots, I_D \,] \times T[\, I_1, \ldots, \underset{\uparrow d}{Q}, \ldots, I_D \,] \to T[\, I_1, \ldots, \underset{\uparrow d}{R}, \ldots, I_D \,]$$

as *combine operator* that has *scalar type* $T \in \mathtt{TYPE}$, *dimensionality* $D \in \mathbb{N}$, and *operating dimension* $d \in [1, D]_{\mathbb{N}}$. We denote combine operator's type concisely as $\mathtt{CO}$.

Example 1 (Concatenation)**.** We define *concatenation* (in dimension d) as function $+\!\!+_d$ of type

$$+\!\!+_d : T[\, I_1, \ldots, \underset{\uparrow d}{P}, \ldots, I_D \,] \times T[\, I_1, \ldots, \underset{\uparrow d}{Q}, \ldots, I_D \,] \to T[\, I_1, \ldots, \underset{\uparrow d}{P \uplus Q}, \ldots, I_D \,]$$

and that is computed as:

$$+\!\!+_d(a_1, a_2)[i_1, \ldots, i_d, \ldots, i_D] := \begin{cases} a_1[\ldots, i_d, \ldots] & , i_d \in P \\ a_2[\ldots, i_d, \ldots] & , i_d \in Q \end{cases}$$

The function is well defined when P and Q are disjoint. We usually use an infix notation for $+\!\!+_d$, i.e., we write $a_1 +\!\!+_d a_2$ instead of $+\!\!+_d(a_1, a_2)$, and we refrain from $+\!\!+_d$'s subscript d when it is clear from the context.

Example 2 (Point-Wise Combination)**.** We define *point-wise combination* (in dimension d), according to a binary function $\oplus : T \times T \to T$[9] (e.g. addition), as function $\vec{\bullet}_d$ of type

$$\vec{\bullet}_d : \underbrace{T \times T \to T}_{\oplus} \to$$
$$\underbrace{T[I_1, \ldots, \{0\}, \ldots, I_D] \times T[I_1, \ldots, \{0\}, \ldots, I_D] \to T[I_1, \ldots, \{0\}, \ldots, I_D]}_{\texttt{point-wise combination (according to } \oplus\texttt{)}}$$
($\{0\}$ at dimension d)

that is computed as:

$$\vec{\bullet}_d(\oplus)(a_1, a_2)[i_1, \ldots, 0, \ldots, i_D] \quad (0 \text{ at dimension d})$$
$$:= a_1[i_1, \ldots, 0, \ldots, i_D] \oplus a_2[i_1, \ldots, 0, \ldots, i_D] \quad (0 \text{ at dimension d})$$

The input MDAs are assumed to have index set $\{0\}$ in the operating dimension d; otherwise, $\vec{\bullet}(\oplus)$ is undefined. We refrain from $\vec{\bullet}_d(\oplus)$'s subscript d when it is clear from the context. For brevity, we often write $\oplus$ only, instead of $\vec{\bullet}_d(\oplus)$, and we usually use an infix notation for $\oplus$.

Multi-Dimensional Homomorphisms

Now that we have defined MDAs (Definition 1) and combine operators (Definition 2), we can define *Multi-Dimensional Homomorphisms (MDHs)*. Intuitively, a function h operating on MDAs is an MDH iff we can apply the function independently to parts of its input MDA and combine the obtained intermediate results to the final result using combine operators; this can be imagined as a typical divide-and-conquer pattern. Compared to classic approaches, e.g., *list homomorphisms* [289, 306, 313], a major characteristic of MDH functions is that they allow (de/re)-composing computations in multiple dimensions (e.g., in Figure 7, in both the concatenation dimension as well as in the point-wise addition dimensions), rather than being limited to a particular dimension only (e.g., only the concatenation dimension or only point-wise addition dimension, respectively). We will see later that a

[9] In practice, the binary function is usually associative and commutative, which we exploit in our optimizations (as discussed later in this chapter).

multi-dimensional (de/re)-composition approach is essential to efficiently exploit the hardware of modern architectures which require fine-grained cache blocking and parallelization strategies to achieve their full performance potential.

Figure 14 illustrates the MDH property informally on a simple, two-dimensional input MDA. In the left part of the figure, we split the input MDA in dimension 1 (i.e., horizontally) into two parts a_1 and a_2, apply the MDH function h independently to each part, and combine the obtained intermediate results to the final result using the MDH function h's combine operator $\circledast_1$. Similarly, in the right part of Figure 14, we split the input MDA in dimension 2 (i.e., vertically) into parts and combine the results via MDH function h's second combine operator $\circledast_2$.

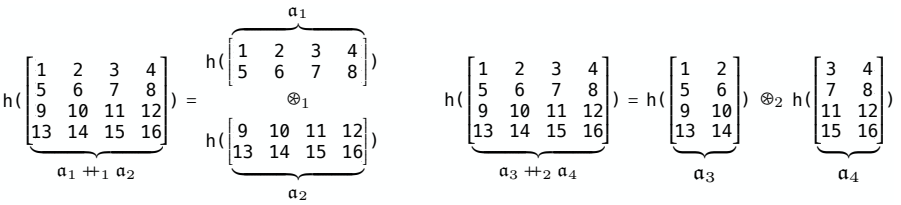


Figure 14: MDH property illustrated on a two-dimensional example computation

Figure 15 shows an artificial example in which we apply the MDH property (illustrated in Figure 14) recursively. We refer in Figure 15 to the part above the horizontal dashed lines as *de-composition phase* and to the part below dashed lines as *re-composition* phase.

Definition 3 (Multi-Dimensional Homomorphism)**.** A function

$$h : T^{INP}[\, I_1, \ldots, I_D \,] \to T^{OUT}[\, J_1, \ldots, J_D \,]$$

is a *Multi-Dimensional Homomorphism (MDH)* that has *input scalar type* $T^{INP} \in \mathtt{TYPE}$, *output scalar type* $T^{OUT} \in \mathtt{TYPE}$, and *dimensionality* $D \in \mathbb{N}$, iff for each $d \in [1, D]_{\mathbb{N}}$, there exists a combine operator $\circledast_d$ (Definition 2), such that for any concatenated input MDA $a_1 +\!\!+_d a_2$ in dimension d, the *homomorphic property* is satisfied:

$$h(\, a_1 +\!\!+_d a_2 \,) \;=\; h(a_1) \circledast_d h(a_2)$$

We denote the type of MDHs concisely as `MDH`.

MDHs are defined such that applying them to a concatenated MDA in dimension d can be computed by applying the MDH h independently to the MDA's parts a_1 and a_2 and combining the intermediate results afterward by using its combine operator $\circledast_d$, as also informally discussed above.

Example 3 (Function Mapping)**.** A simple example MDH is *function mapping* [251], computed by higher-order function $\mathtt{map}(f)(a)$, which applies a user-defined scalar function $f : T^{INP} \to T^{OUT}$ to each element within a D-dimensional MDA a. Function $\mathtt{map}(f)$ is an MDH whose combine operators are concatenation $+\!\!+$ in all of its D dimensions (Example 1).

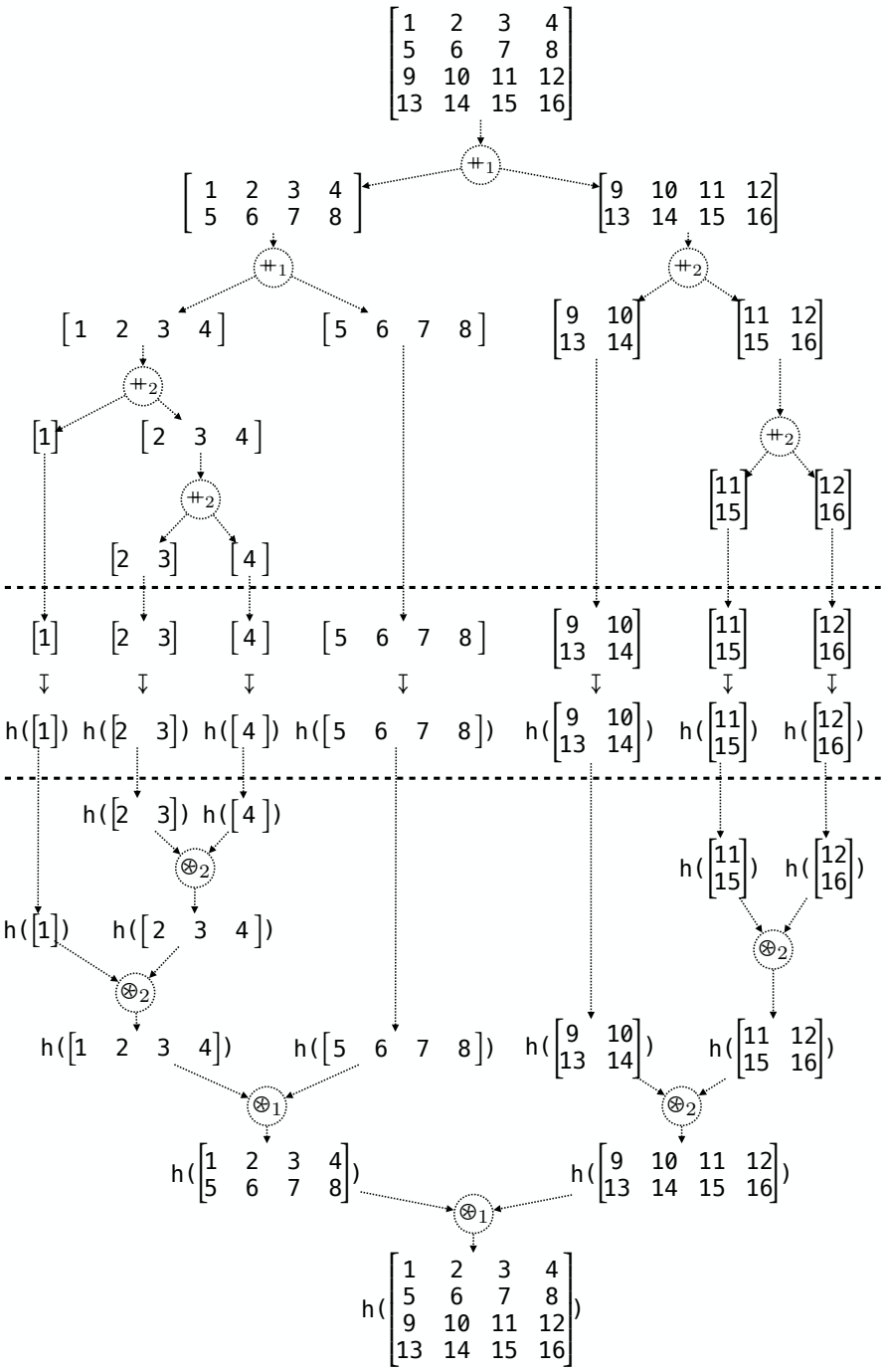


Figure 15: MDH property recursively applied to a two-dimensional example computation.

Example 4 (Reduction)**.** A further MDH function is *reduction* [251], implemented as higher-order function $\texttt{red}(\oplus)(\mathtt{a})$, which combines all elements within a D-dimensional MDA $\mathtt{a}$ using a user-defined binary function $\oplus : T \times T \to T$. Reduction's combine operators are point-wise combination $\vec{\bullet}(\oplus)$ in all dimensions (Example 2).

We show how Examples 3 and 4 (and particularly also more advanced examples) are expressed in our high-level representation in Section 4.2.4, based on higher-order functions `md_hom`, `inp_view`, and `out_view` (Figure 11) which we introduce in the following.

Higher-Order Function `md_hom`

We define higher-order function `md_hom` which conveniently expresses MDH functions in a uniform and structured manner. For this, we exploit that any MDH function is uniquely determined by its combine operators and its behavior on singleton MDAs, as informally illustrated in the following figure:

$$\mathtt{h}\left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}\right) = \begin{array}{c} \xrightarrow{\otimes_2} \\ \begin{matrix} \mathtt{h}([1]) & \mathtt{h}([2]) & \mathtt{h}([3]) & \mathtt{h}([4]) \\ \mathtt{h}([5]) & \mathtt{h}([6]) & \mathtt{h}([7]) & \mathtt{h}([8]) \\ \mathtt{h}([9]) & \mathtt{h}([10]) & \mathtt{h}([11]) & \mathtt{h}([12]) \\ \mathtt{h}([13]) & \mathtt{h}([14]) & \mathtt{h}([15]) & \mathtt{h}([16]) \end{matrix} \\ \downarrow_{\otimes_1} \end{array} = \begin{array}{c} \xrightarrow{\otimes_2} \\ \begin{matrix} \mathtt{f}(1) & \mathtt{f}(2) & \mathtt{f}(3) & \mathtt{f}(4) \\ \mathtt{f}(5) & \mathtt{f}(6) & \mathtt{f}(7) & \mathtt{f}(8) \\ \mathtt{f}(9) & \mathtt{f}(10) & \mathtt{f}(11) & \mathtt{f}(12) \\ \mathtt{f}(13) & \mathtt{f}(14) & \mathtt{f}(15) & \mathtt{f}(16) \end{matrix} \\ \downarrow_{\otimes_1} \end{array}$$

Here, $\mathtt{f}$ is the function on scalar values that behaves the same as $\mathtt{h}$ when restricted to singleton MDAs: $\mathtt{f}(\mathtt{a}[i_1,\ldots,i_D]) := \mathtt{h}(\mathtt{a})$, for any MDA $\mathtt{a} \in T[\{i_1\},\ldots,\{i_D\}]$ consisting of only one element that is accessed by (arbitrary) indices $i_1,\ldots,i_D \in \mathbb{N}_0$. For singleton MDAs, we usually use $\mathtt{f}$ instead of $\mathtt{h}$, because $\mathtt{f}$ can be defined more conveniently by the user as $\mathtt{h}$ (which needs to handle MDAs of arbitrary sizes, and not only singleton MDAs as $\mathtt{f}$). Also, since $\mathtt{f}$ takes as input a scalar value (rather than a singleton MDA, as $\mathtt{h}$), the type of $\mathtt{f}$ also becomes simpler, which further contributes to simplicity.

We now formally introduce function `md_hom` which uniformly expresses any MDH function, by using only the MDH's behavior $\mathtt{f}$ on scalar values and the MDH's combine operators.

Definition 4 (Higher-Order Function `md_hom`)**.** The higher-order function `md_hom` is of type

$$\texttt{md\_hom} : \underbrace{\texttt{SF}}_{\mathtt{f}} \times \underbrace{(\,\texttt{CO} \times \ldots \times \texttt{CO}\,)}_{\otimes_1,\ldots,\otimes_D} \to_p \underbrace{\texttt{MDH}}_{\texttt{md\_hom}(\,\mathtt{f}\,,\,(\otimes_1,\ldots,\otimes_D)\,)}$$

where SF denotes the set of scalar functions of type $T^{\texttt{INP}} \to T^{\texttt{OUT}}$. Function `md_hom` is partial (indicated by $\to_p$ instead of $\to$), which we motivate after this definition. The function takes as input a scalar function $\mathtt{f}$ and a tuple of D-many combine operators $(\otimes_1,\ldots,\otimes_D)$, and it yields a function $\texttt{md\_hom}(\,\mathtt{f}\,,\,(\otimes_1,\ldots,\otimes_D)\,)$ which is defined as

$$\texttt{md\_hom}(\,\mathtt{f}\,,\,(\otimes_1,\ldots,\otimes_D)\,)(\,\mathtt{a}\,) := \underset{i_1 \in I_1}{\otimes_1} \ldots \underset{i_D \in I_D}{\otimes_D} \mathtt{f}(\,\mathtt{a}[i_1,\ldots,i_D]\,)$$

The combine operators' underset notation denotes straightforward iteration[10]. For md_hom, we require by definition the homomorphic property (Definition 3), i.e., for each $d \in [1, D]_{\mathbb{N}}$, it must hold:

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\, a_1 +\!\!+_d\, a_2\,) \;=$$

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\, a_1\,)\;\circledast_d\;\texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\, a_2\,)$$

Using Definition 4, we express any MDH function uniformly via higher-order function md_hom using only the MDH's behavior f on scalar values and its combine operators $\circledast_1,\ldots,\circledast_D$. The other direction also holds: each function expressed via md_hom is an MDH function, because we require the homomorphic property for md_hom.

Note that function md_hom is defined as partial function, because the homomorphic property is not met for all potential combinations of combine operators, e.g., $\circledast_1 = +$ (point-wise addition) and $\circledast_2 = *$ (point-wise multiplication). However, in many real-world examples, an MDH's combine operators are a mix of concatenations and point-wise combinations according to the same binary function. The following lemma proves that any instance of the md_hom higher-order function for such a mix of combine operators is a well-defined MDH function.

Lemma 1. Let $\oplus : T \to T$ be an arbitrary but fixed associative and commutative binary function on scalar type $T \in$ TYPE. Let further $\circledast_1,\ldots,\circledast_D$ be combine operators of which any is either concatenation (Example 1) or point-wise combination according to binary function $\oplus$ (Example 2).

It holds that $\texttt{md\_hom}(\, f, (\circledast_1,\ldots,\circledast_D)\,)$ is well defined.

Proof. Proved in the Appendix, Section .3.5. □

MDH functions are defined (Definition 3) such that they uniformly operate on MDAs (Figure 11). We introduce higher-order function inp_view to transform domain-specific inputs (e.g., a matrix and a vector in the case of matrix-vector multiplication) to an MDA, and we use function out_view to transform the output MDA back to the domain-specific data requirements (like storing it as a transposed matrix in the case of matrix multiplication, or splitting it into multiple outputs as we will see later with examples). We introduce both higher-order functions in the following.

[10] We implicitly interpret the output scalar of function f as a singleton MDA, as combine operators operate on MDAs and not on scalars (formal details provided in the Appendix, Definition 28).

4.2.3 *View Functions*

In the following, after introducing *Buffers (BUF)* which represent domain-specific input and output data in our approach (scalars, vectors, matrices, etc), we define in Sections 4.2.3.1 and 4.2.3.2 the concepts of *input views* and *output views* – both are central building blocks in our approach. We define *input views* as arbitrary functions that map a collection of user-defined BUFs to our internal MDA data representation (Figure 11); higher-order function `inp_view` is then introduced to conveniently compute an important class of input view functions that are relevant for expressing real-world computations. Correspondingly, Section 4.2.3.2 defines *output views* as functions that transform an MDA to a collection of BUFs, and higher-order function `out_view` is introduced to conveniently compute important output views.

We discuss in Section 4.2.3.3 the relationship between higher-order function `inp_view` and `out_view`: we prove that both functions are inversely related to each other, allowing arbitrarily switching between our internal MDA representation and our domain-specific BUF representation (as required for our code generation process discussed later).

Definition 5 (Buffer)**.** A *Buffer (BUF)* $\mathfrak{b}$ that has *dimensionality* $D \in \mathbb{N}_0$[11], *size* $N := \{N_1, \ldots, N_D\} \in \mathbb{N}^D$, and *scalar type* $T \in \texttt{TYPE}$ is a function with the following signature:

$$\mathfrak{b} : [0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\bot\}$$

Here, we use $\bot$ to denote the *undefined value*. We refer to $[0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\bot\}$ as the *type* of BUF $\mathfrak{b}$, which we also denote as $T^{N_1 \times \ldots \times N_D}$. Analogously to Notation 1, we write $\mathfrak{b}[i_1, \ldots, i_D]$ instead of $\mathfrak{b}(i_1, \ldots, i_D)$ to avoid a too heavy usage of parentheses.

In contrast to MDAs, a BUF always operates on a contiguous range of natural numbers starting from 0, and a BUF may contain undefined values. These two differences allow straightforwardly transforming BUFs to data structures provided by low-level programming languages (e.g., *C arrays*, as used in OpenMP, CUDA, and OpenCL).

Note that in our generated program code (discussed later in Section 4.3), we implement MDAs on top of BUFs, as straightforward aliases that access BUFs, so that we do not need to transform MDAs to low-level data structures and/or store them otherwise physically in memory.

[11] We use the case $D = 0$ to represent scalar values (details provided in the Appendix, Section .3.7).

4.2.3.1 *Input Views*

We define *input views* as any function that compute an MDA from a collection of (user-defined) BUFs. For example, in the case of `MatVec`, its input view takes as input two BUFs – a matrix and a vector – and it yields a two-dimensional MDA containing pairs of matrix and vector elements (illustrated in Figure 7). In contrast, the input view of `Jacobi1D` takes as input a single BUF (representing a vector) only, and it computes an MDA containing triples of BUF elements (Figure 8).

Definition 6 (Input View)**.** An *input view* from B-many BUFs, $B \in \mathbb{N}$, of arbitrary but fixed types $T_b^{N_1^b \times \ldots \times N_{D_b}^b}$, $b \in [1,B]_{\mathbb{N}}$, to an MDA of arbitrary but fixed type $T[I_1,\ldots,I_D]$ is any function iv of type:

$$iv : \underbrace{\bigtimes_{b=1}^{B} T_b^{N_1^b \times \ldots \times N_{D_b}^b}}_{\text{BUFs}} \to_p \underbrace{T[\, I_1, \ldots, I_D \,]}_{\text{MDA}}$$

We denote the type of iv as `IV`.

Example 5 (Input View – `MatVec`)**.** The input view of `MatVec` on a 1024×512 matrix and 512-sized vector (sizes chosen arbitrarily) is defined as:

$$\underbrace{[\, M(i,k) \,]_{i \in [0,1024)_{\mathbb{N}_0}, k \in [0,512)_{\mathbb{N}_0}}}_{\text{BUF (Matrix)}}, \underbrace{[\, v(k) \,]_{k \in [0,512)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}}$$

$$\mapsto \underbrace{[\, M(i,k), v(k) \,]_{i \in [0,1024)_{\mathbb{N}_0}, k \in [0,512)_{\mathbb{N}_0}}}_{\text{MDA}}$$

Example 6 (Input View – `Jacobi1D`)**.** The input view of `Jacobi1D` on a 512-sized vector is defined as:

$$\underbrace{[\, v(i) \,]_{i \in [0,512)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}} \mapsto \underbrace{[\, v(i+0), v(i+1), v(i+2) \,]_{i \in [0,512-2)_{\mathbb{N}_0}}}_{\text{MDA}}$$

In the following, we introduce higher-order function `inp_view` which conveniently computes important input views from user-defined index functions $id_{f_{b,a}} : \{0,1,\ldots\} \to \{0,1,\ldots\}$, $b \in [1,B]_{\mathbb{N}}$, $a \in [1,A_b]_{\mathbb{N}}$, in a uniform, structured manner. Here, $B \in \mathbb{N}$ represents the number of BUFs that the computed input view will take as input, and A_b represents the number of accesses to the b-th BUF required for computing an individual MDA element.

In the case of `MatVec` (Figure 7), we use $B := 2$ because `MatVec` has two input BUFs: a matrix M (the first input of `MatVec` and thus identified by $b = 1$) and a vector v (identified by $b = 2$). For the number of accesses, we use for the matrix $A_1 := 1$, as one element is accessed within matrix M to compute an individual MDA element – matrix element $M[i,k]$ for computing MDA element at position (i,k). For the vector, we use $A_2 := 1$, as the single element $v[k]$ is accessed within the vector. The index functions of `MatVec` are: $id\mathfrak{x}_{1,1}(i,k) := (i,k)$ (used to access the matrix) and $id\mathfrak{x}_{2,1}(i,k) := (k)$ (used for the vector).

In contrast, for `Jacobi1D` (Figure 8), we use $B := 1$ because `Jacobi1D` has vector v as its only input, and we use $A_1 := 3$ because the vector is accessed three times to compute an individual MDA element at arbitrary position i: first access $v[i+0]$, second access $v[i+1]$, and third access $v[i+2]$. The index functions of `Jacobi1D` are: $id\mathfrak{x}_{1,1}(i) := (i+0)$, $id\mathfrak{x}_{1,2}(i) := (i+1)$, and $id\mathfrak{x}_{1,3}(i) := (i+2)$.

Figures 16 and 17 use the examples `MatVec` and `Jacobi1D` to informally illustrate how function `inp_view` uses index functions to compute input views. In the two figures, we use domain-specific identifiers for better clarity: in the case of `MatVec`, we use for its two input BUFs the identifiers M and v instead of b_1 and b_2, as well as identifiers i and j instead of i_1 and i_2 for index variables; for `Jacobi1D`, we use identifier v instead of b_1 and i instead of i_1.

Definition 7 (Higher-Order Function `inp_view`)**.** Function `inp_view` is of type

$$\texttt{inp\_view} : \Big(\underbrace{\underbrace{\bigtimes_{b=1}^{B}}_{\text{Buffer Access}} \underbrace{\bigtimes_{a=1}^{A_b}}_{\text{Buffer Access}} \underbrace{\texttt{IDX-FCT}}_{\text{Index Function: } id\mathfrak{x}_{b,a}}}_{\text{Index Functions: } id\mathfrak{x}_{1,1},\ldots,id\mathfrak{x}_{B,A_B}} \Big) \rightarrow \underbrace{\texttt{IV}}_{\text{Input View: } iv}$$

and it is defined as:

$$\underbrace{(id\mathfrak{x}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Functions}} \mapsto \underbrace{\underbrace{(b_1,\ldots,b_B)}_{\text{BUFs}} \overset{iv}{\mapsto} \underbrace{a}_{\text{MDA}}}_{\text{Input View}}$$

for $a[i_1,\ldots,i_D] := (a_{b,a}[i_1,\ldots,i_D])_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}$ and $a_{b,a}[i_1,\ldots,i_D] := b_b[\, id\mathfrak{x}_{b,a}(i_1,\ldots,i_D)\,]$

Higher-order function `inp_view` takes as input a collection of index functions of types `IDX-FCT`, and it computes an input view of type `IV` (Definition 6) based on the index functions, as illustrated in Figures 16 and 17.

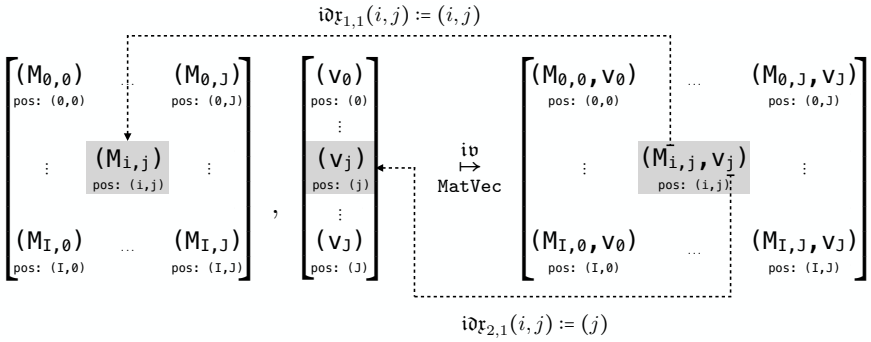


Figure 16: Input view illustrated using the example `MatVec`.

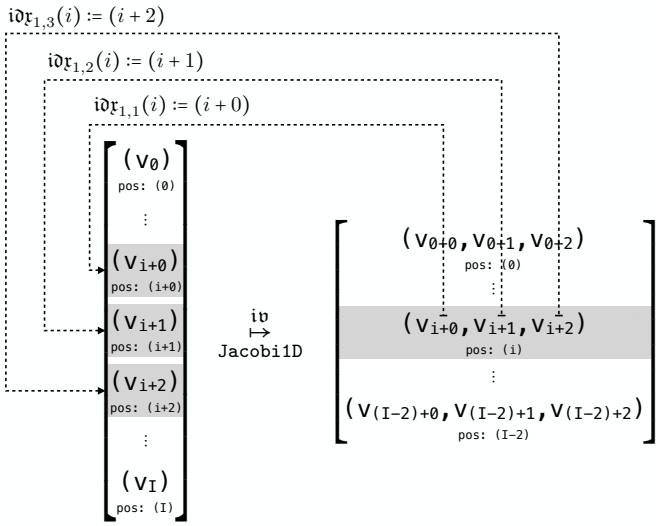


Figure 17: Input view illustrated using the example `Jacobi1D`.

Note that function `inp_view` is not capable of computing every kind of input view function (Definition 6). For example, `inp_view` cannot be used for computing MDAs that are required for expressing computations on sparse data formats [70], because such MDAs need dynamically accessing BUFs. This limitation of `inp_view` can be relaxed by generalizing our index functions toward taking additional, dynamic input arguments, which we consider as future work (as outlined in Chapter 8).

Notation 2 (Input Views)**.** For better readability, we use the following notation for the 2-dimensional structure of index functions taken as input by function `inp_view`, inspired by Lattner et al. [56]:

$$\texttt{inp\_view}(\ \texttt{ID}_1 : \mathfrak{id}\mathfrak{x}_{1,1}, \ldots, \mathfrak{id}\mathfrak{x}_{1,A_1} \quad , \ldots , \quad \texttt{ID}_B : \mathfrak{id}\mathfrak{x}_{B,1}, \ldots, \mathfrak{id}\mathfrak{x}_{B,A_B}\ )$$

Here, $\texttt{ID}_1, \ldots, \texttt{ID}_B$ denote arbitrary, user-defined identifiers (e.g., $\texttt{ID}_1$ = `"M"` and $\texttt{ID}_2$ = `"v"` for `MatVec`).

Example 7. Function `inp_view` is used for `MatVec` and `Jacobi1D` (in Notation 2) as follows:

$$\underline{\texttt{MatVec:}} \quad \texttt{inp\_view}(\ \underbrace{\underbrace{\texttt{M:}\ \underbrace{(\texttt{i},\texttt{k}) \mapsto (\texttt{i},\texttt{k})}_{\texttt{a=1}}}_{\texttt{b=1}}, \underbrace{\texttt{v:}\ \underbrace{(\texttt{i},\texttt{k}) \mapsto (\texttt{k})}_{\texttt{a=1}}}_{\texttt{b=2}}}\ )$$

$$\underline{\texttt{Jacobi1D:}} \quad \texttt{inp\_view}(\ \underbrace{\texttt{v:}\ \underbrace{(\texttt{i}) \mapsto (\texttt{i}+0)}_{\texttt{a=1}}, \underbrace{(\texttt{i}) \mapsto (\texttt{i}+1)}_{\texttt{a=2}}, \underbrace{(\texttt{i}) \mapsto (\texttt{i}+2)}_{\texttt{a=3}}}_{\texttt{b=1}}\ )$$

4.2.3.2 *Output Views*

An *output view* is the counterpart of an input view: in contrast to an input view which maps BUFs to an MDA, an output view maps an MDA to a collection of BUFs. In the following, we define output views, and we introduce higher-order function `out_view` which computes output views in a structured manner (analogously to function `inp_view` for input views).

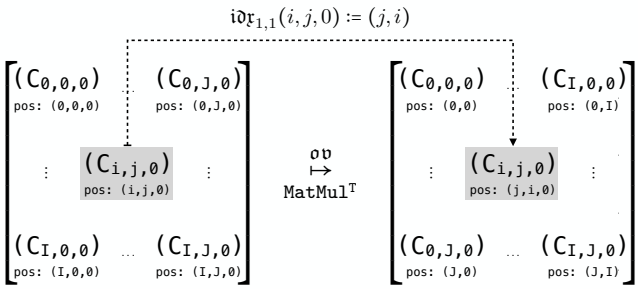


Figure 18: Output view illustrated using the example *transposed Matrix Multiplication*.

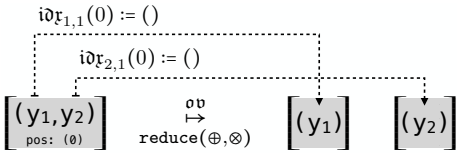


Figure 19: Output view illustrated using the example *Double Reduction*.

Figures 18 and 19 illustrate output views informally using the examples *transposed Matrix Multiplication* and *Double Reduction*.

In the case of transposed matrix multiplication (Figure 18), the computed output MDA (the computation of matrix multiplication is presented later and not relevant for our following considerations) is stored via an output view as a matrix in a transposed fashion, using index function $(i, j, 0) \mapsto (j, i)$. Here, the MDA's third dimension (accessed via index 0) represents the so-called reduction dimension of matrix multiplication, and it contains only one element after the computation, as all elements in this dimension are combined via addition.

For double reduction (Figures 19), we combine the elements within the vector twice – once using operator $\oplus$ (e.g., $\oplus = +$ addition) and once using operator $\otimes$ (e.g, $\otimes = *$ multiplication). The final outcome of double reduction is a singleton MDA containing a pair of two elements that represent the combined vector elements (e.g., the elements' sum and product). We store this MDA via an output view as two individual scalar values, using index functions $(0) \mapsto ()$[12] for both pair elements.

Definition 8 (Output View)**.** An *output view* from an MDA of arbitrary but fixed type $T[I_1, \ldots, I_D]$ to B-many BUFs, $B \in \mathbb{N}$, of arbitrary but fixed types $T_b^{N_1^b \times \ldots \times N_{D_b}^b}$, $b \in [1, B]_{\mathbb{N}}$, is any function $\mathfrak{ov}$ of type:

$$\mathfrak{ov} : \underbrace{T[\, I_1, \ldots, I_D \,]}_{\text{MDA}} \to_p \underbrace{\bigtimes_{b=1}^{B} T_b^{N_1^b \times \ldots \times N_{D_b}^b}}_{\text{BUFs}}$$

We denote the type of $\mathfrak{ov}$ as $\mathtt{OV}$.

Example 8 (Output View – `MatVec`)**.** The output view of `MatVec` computing a 1024-sized vector (size is chosen arbitrarily), of integers $\mathbb{Z}$, is defined as:

$$\underbrace{[\, w(i) \,]_{i \in [0,1024)_{\mathbb{N}_0}, k \in \{0\}}}_{\text{MDA}} \mapsto \underbrace{[\, w(i) \,]_{i \in [0,1024)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}}$$

Example 9 (Output View – `Jacobi1D`)**.** The output view of `Jacobi1D` computing a $(512 - 2)$-sized vector is defined as:

$$\underbrace{[\, w(i) \,]_{i \in [0,512-2)_{\mathbb{N}_0}, k \in \{0\}}}_{\text{MDA}} \mapsto \underbrace{[\, w(i) \,]_{i \in [0,512-2)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}}$$

[12] The empty braces denote accessing a scalar value (details provided in the Appendix, Section .3.7).

We define higher-order function `out_view` formally as follows.

Definition 9 (Higher-Order Function `out_view`)**.** Function `out_view` is of type

$$\texttt{out\_view}: \underbrace{\underbrace{\overset{B}{\underset{b=1}{\times}}\ \overset{A_b}{\underset{a=1}{\times}}}_{\text{Buffer Access}}\ \underbrace{\texttt{IDX-FCT}}_{\text{Index Function: } i\mathfrak{d}\mathfrak{x}_{b,a}}}_{\text{Index Functions: } i\mathfrak{d}\mathfrak{x}_{1,1},\ldots,i\mathfrak{d}\mathfrak{x}_{B,A_B}} \rightarrow \underbrace{\texttt{OV}}_{\text{Output View: } \mathfrak{ov}}$$

which differs from `inp_view`'s type only in mapping index functions to `OV` (Definition 8), rather than `IV` (Definition 6). Function `out_view` is defined as:

$$\underbrace{(i\mathfrak{d}\mathfrak{x}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Functions}} \mapsto \underbrace{\underbrace{\mathfrak{a}}_{\text{MDA}} \overset{\mathfrak{ov}}{\mapsto} (\underbrace{\mathfrak{b}_1,\ldots,\mathfrak{b}_B}_{\text{BUFs}})}_{\text{Output View}}$$

for $\mathfrak{b}_b[\ i\mathfrak{d}\mathfrak{x}_{b,a}(i_1,\ldots,i_D)\ ] := \mathfrak{a}_{b,a}[i_1,\ldots,i_D]$ and $(\mathfrak{a}_{b,a}[i_1,\ldots,i_D])_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}} := \mathfrak{a}[i_1,\ldots,i_D]$ i.e., $\mathfrak{a}_{b,a}[i_1,\ldots,i_D]$ is the element at point $i_1,\ldots,i_D$ within MDA $\mathfrak{a}$ that belongs to the a-th access of the b-th BUF. We set $\mathfrak{b}_b[j_1,\ldots,j_{D_b}] := \perp$ (symbol $\perp$ denotes the undefined value) for all BUF indices which are not in the function range of the index functions.

Note that the computed output view $\mathfrak{ov}$ is partial (indicated by $\rightarrow_p$ in Definition 8), because for non-injective index functions, it must hold $i\mathfrak{d}\mathfrak{x}_{b,a}(i_1,\ldots,i_D) = i\mathfrak{d}\mathfrak{x}_{b,a'}(i'_1,\ldots,i'_D) \Rightarrow \mathfrak{a}_{b,a}[i_1,\ldots,i_D] = \mathfrak{a}_{b,a'}[i'_1,\ldots,i'_D]$ which may not be satisfied for each potential input MDA of the computed view.

Notation 3 (Output Views)**.** Analogously to Notation 2, we denote `out_view` for a particular choice of index functions as:

$$\texttt{out\_view}(\ \texttt{ID}_1 : i\mathfrak{d}\mathfrak{x}_{1,1},\ldots,i\mathfrak{d}\mathfrak{x}_{1,A_1}\ ,\ldots,\ \texttt{ID}_B : i\mathfrak{d}\mathfrak{x}_{B,1},\ldots,i\mathfrak{d}\mathfrak{x}_{B,A_B}\ )$$

Example 10. Function `out_view` is used for `MatVec` and `Jacobi1D` (in Notation 3) as follows:

$$\underline{\texttt{MatVec:}}\quad \texttt{out\_view}(\ \texttt{w:}\underbrace{\underbrace{(i,k)\mapsto(i)}_{a=1}}_{b=1}\ )$$

$$\underline{\texttt{Jacobi1D:}}\quad \texttt{out\_view}(\ \texttt{w:}\underbrace{\underbrace{(i)\mapsto(i)}_{a=1}}_{b=1}\ )$$

4.2.3.3 *Relation between View Functions*

We use view functions to transform data from their domain-specific representation (represented in our formalism as BUFs, Definition 5) to our internal, MDA-based representation (via input views) and back (via output views), as also illustrated in Figure 11. In our implementation presented later, we aim to access data uniformly in the form of MDAs, thereby being independent of domain-specific data representations. However, we aim to store the data physically in the domain-specific format, as such format is usually the more efficient representation. For example, we aim to store the input data of `MatVec` in the domain-specific matrix and vector format, rather than as an MDA, because the input MDA of `MatVec` contains many redundancies – each vector element once per row of the input matrix (as illustrated in Figure 16).

The following lemma proves that functions `inp_view` and `out_view` are invertible and that they are each others inverses. Consequently, the lemma shows how we can arbitrarily switch between the domain-specific and our MDA-based representation, and consequently also that we can implicitly identify MDAs with the domain-specific data representation. For example, for computing `MatVec`, we will specify the computations via pattern `md_hom` which operates on MDAs (see Figure 11), but we use the view functions in our implementation to implicitly forward the MDA accesses to the physically stored BUF representation.

Lemma 2. Let

$$\texttt{inp\_view}(\ \texttt{ID}_1 : \mathfrak{id}_{1,1}, \ldots, \mathfrak{id}_{1,A_1}\ , \ldots ,\ \texttt{ID}_B : \mathfrak{id}_{B,1}, \ldots, \mathfrak{id}_{B,A_B}\ ),$$
$$\texttt{out\_view}(\ \texttt{ID}_1 : \mathfrak{id}_{1,1}, \ldots, \mathfrak{id}_{1,A_1}\ , \ldots ,\ \texttt{ID}_B : \mathfrak{id}_{B,1}, \ldots, \mathfrak{id}_{B,A_B}\ )$$

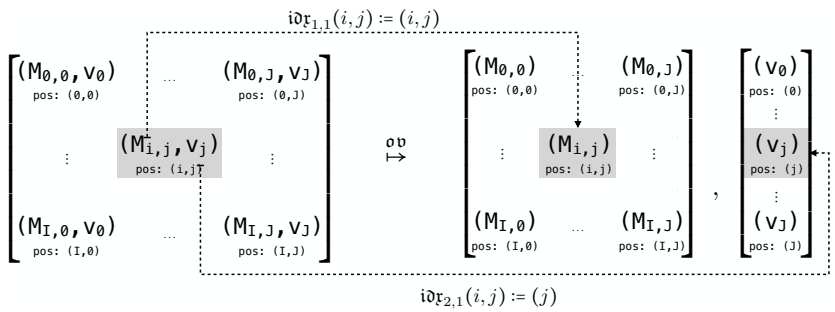
be two arbitrary instances of functions `inp_view` and `out_view` (in Notations 2 and 3), both using the same index functions $\mathfrak{id}_{1,1}, \ldots, \mathfrak{id}_{B,A_B}$.

It holds (index functions omitted via ellipsis for brevity):

$$\texttt{inp\_view}(\ \ldots\ ) \circ \texttt{out\_view}(\ \ldots\ ) \ = \ \texttt{id}$$
$$\texttt{out\_view}(\ \ldots\ ) \circ \texttt{inp\_view}(\ \ldots\ ) \ = \ \texttt{id}$$

Proof. Follows immediately from Definitions 7 and 9. □

The following figure illustrates the lemma using as example the inverse of `MatVec`'s input view (shown in Figure 16):



4.2.4 *Examples*

Figure 20 shows how our high-level representation is used for expressing different kinds of popular data-parallel computations. For brevity, we state only the index functions, scalar function, and combine operators of the higher-order functions; an expression as in Figure 12 is then obtained by straightforwardly inserting these building blocks into the higher-order functions.

SUBFIGURE 1 We show how our high-level representation is used for expressing linear algebra routines: 1) `Dot` (*Dot Product*); 2) `MatVec` (*Matrix-Vector Multiplication*); 3) `MatMul` (*Matrix Multiplication*); 4) $\texttt{MatMul}^\intercal$ (*Transposed Matrix Multiplication*) which computes matrix multiplication on transposed input and output matrices; 5) `bMatMul` (*batched Matrix Multiplication*) where multiple matrix multiplications are computed using matrices of the same sizes.

We can observe from the subfigure that our high-level expressions for the routines naturally evolve from each other. For example, the `md_hom` instance for `MatVec` differs from the `md_hom` instance for `Dot` by only containing a further concatenation dimension $+\!\!+$ for its `i` dimension. We consider this close relation between the high-level expressions of `MatVec` and `Dot` in our approach as natural and favorable, as `MatVec` can be considered as computing multiple times `Dot` – one computation of `Dot` for each value of `MatVec's` `i` dimension. Similarly, the `md_hom` instance for `MatMul` is very similar to the expression of `MatVec`, by containing the further concatenation dimension `j` for `MatMul`'s `j` dimension. The same applies to `bMatMul`: its `md_hom` instance is the expression of `MatMul` augmented with one further concatenation dimension.

Regarding $\texttt{MatMul}^\intercal$, the basic computation part of $\texttt{MatMul}^\intercal$ and `MatMul` are the same, which is exactly reflected in our formalisms: both $\texttt{MatMul}^\intercal$ and `MatMul` are expressed using exactly the same `md_hom` instances. The differences between $\texttt{MatMul}^\intercal$ and `MatMul` lies only in the data accesses – transposed accesses in the case of $\texttt{MatMul}^\intercal$ and non-transposed accesses in the case of `MatMul`. Data accesses are expressed in our formalism, in a structured way, via view functions (as discussed in Section 4.2.3): for example, for $\texttt{MatMul}^\intercal$, we use for its first input matrix A the index function $(\mathtt{i},\mathtt{j},\mathtt{k}) \mapsto (\mathtt{k},\mathtt{i})$ for transposed access, instead of using index function $(\mathtt{i},\mathtt{j},\mathtt{k}) \mapsto (\mathtt{i},\mathtt{k})$ as for `MatMul`'s non-transposed accesses.

Note that all `md_hom` instances in the subfigure are well defined according to Lemma 1.

SUBFIGURE 2 We show how convolution-style stencil computations are expressed in our high-level representation: 1) `Conv2D` expresses a standard convolution that uses a 2D sliding window [270]; 2) `MCC` expresses a so-called *Multi-Channel Convolution* [121] – a generalization of `Conv2D` that is heavily used in the area of deep learning; 3) `MCC_Capsule` is a recent generalization of `MCC` [126] which attracted high attention due to its relevance for advanced deep learning neural networks [84].

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$
Dot	*	+			
MatVec	*	++	+		
MatMul	*	++	++	+	
MatMul^T	*	++	++	+	
bMatMul	*	++	++	++	+

Views	inp_view		out_view
	A	B	C
Dot	(k) ↦ (k)	(k) ↦ (k)	(k) ↦ ()
MatVec	(i,k) ↦ (i,k)	(i,k) ↦ (k)	(i,k) ↦ (i)
MatMul	(i,j,k) ↦ (i,k)	(i,j,k) ↦ (k,j)	(i,j,k) ↦ (i,j)
MatMul^T	(i,j,k) ↦ (k,i)	(i,j,k) ↦ (j,k)	(i,j,k) ↦ (j,i)
bMatMul	(b,i,j,k) ↦ (b,i,k)	(b,i,j,k) ↦ (b,k,j)	(b,i,j,k) ↦ (b,i,j)

1) Linear Algebra Routines

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$	$\circledast_5$	$\circledast_6$	$\circledast_7$	$\circledast_8$	$\circledast_9$	$\circledast_{10}$
Conv2D	*	++	++	+	+						
MCC	*	++	++	++	++	+	+	+			
MCC_Capsule	*	++	++	++	++	+	+	+	++	++	+

Views	inp_view		out_view
	I	F	O
Conv2D	(p,q,r,s) ↦ (p+r,q+s)	(p,q,r,s) ↦ (r,s)	(p,q,r,s) ↦ (p,q)
MCC	(n,p,...) ↦ (n,p+r,q+s,c)	(n,p,...) ↦ (k,r,s,c)	(n,p,...) ↦ (n,p,q,k)
MCC_Capsule	(n,p,...) ↦ (n,p+r,q+s,c,mi,mk)	(n,p,...) ↦ (k,r,s,c,mk,mj)	(n,p,...) ↦ (n,p,q,k,mi,mj)

2) Convolution Stencils

md_hom	f	$\circledast_1$	...	$\circledast_6$	$\circledast_7$
CCSD(T)	*	++	...	++	+

Views	inp_view		out_view
	A	B	C
I1	(a,...,g) ↦ (g,d,a,b)	(a,...,g) ↦ (e,f,g,c)	(a,...,g) ↦ (a,...,f)
I2	(a,...,g) ↦ (g,d,a,c)	(a,...,g) ↦ (e,f,g,b)	(a,...,g) ↦ (a,...,f)

3) Quantum Chemistry

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$
Jacobi1D	J_{1D}	++		
Jacobi2D	J_{2D}	++	++	
Jacobi3D	J_{3D}	++	++	++

Views	inp_view	out_view
	I	O
Jacobi1D	(i1) ↦ (i1+0) , (i1) ↦ (i1+1) , ...	(i1) ↦ (i1)
Jacobi2D	(i1,i2) ↦ (i1+0,i2+1) , ...	(i1,i2) ↦ (i1,i2)
Jacobi3D	(i1,i2,i3) ↦ (i1+0,i2+1,i3+1) , ...	(i1,i2,i3) ↦ (i1,i2,i3)

4) Jacobi Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
PRL	wght	++	$\max_{PRL}$

Views	inp_view		out_view
	N	E	M
PRL	(i,j) ↦ (i)	(i,j) ↦ (j)	(i,j) ↦ (i)

5) Probabilistic Record Linkage

md_hom	f	$\circledast_1$	$\circledast_2$
Histo	f_{Histo}	+	++
GenHisto	f	$\oplus$	++

Views	inp_view		out_view
	Elems	Bins	Out
Histo	(e,b) ↦ (e)	(e,b) ↦ (b)	(e,b) ↦ (b)
GenHisto	(e,b) ↦ (e)	(e,b) ↦ (b)	(e,b) ↦ (b)

6) Histogram

md_hom	f	$\circledast_1$
map(f)	f	++
reduce($\oplus$)	id	$\oplus$
reduce($\oplus$,$\otimes$)	(x) ↦ (x,x)	($\oplus$,$\otimes$)

Views	inp_view	out_view	
	I	O_1	O_2
map(f)	(i) ↦ (i)	(i) ↦ (i)	
reduce($\oplus$)	(i) ↦ (i)	(i) ↦ ()	
reduce($\oplus$,$\otimes$)	(i) ↦ (i)	(i) ↦ ()	(i) ↦ ()

7) Map/Reduce Patterns

md_hom	f	$\circledast_1$	$\circledast_2$
scan($\oplus$)	id	$\leftrightarrow_{prefix\text{-}sum}(\oplus)$	
MBBS	id	$\leftrightarrow_{prefix\text{-}sum}(+)$	+

Views	inp_view	out_view
	A	Out
scan($\oplus$)	(i) ↦ (i)	(i) ↦ (i)
MBBS	(i,j) ↦ (i,j)	(i,j) ↦ (i)

8) Prefix Sum Computations

Figure 20: Data-parallel computations expressed in our high-level representation.

While our `md_hom` instances for convolutions are quite similar to those of linear algebra routines (they all use multiplication $*$ as scalar function, and a mix of concatenations $+\!\!+$ and point-wise additions $+$ as combine operators), the index functions used for the view functions of convolutions are notably different from those used for linear algebra routines: the index functions of convolutions contain arithmetic expressions (e.g., `p+r` and `q+s`), thereby access neighboring elements in their input – a typical access pattern in stencil computations that requires special optimizations [124]. Moreover, convolution-style computations are often high-dimensional (e.g., 10 dimensions in the case of `MCC_Capsule`), whereas linear algebra routines usually rely on less dimensions. Our experiments in Section 4.5 confirm that respecting the data access patterns and the high dimensionality of convolutions in the optimization process (as in our approach, which we discuss later) often achieves significantly higher performance than using optimizations chosen toward linear algebra routines, as in vendor libraries provided by NVIDIA and Intel for convolutions [165].

SUBFIGURE 3 We show how quantum chemistry computation *Coupled Cluster* (`CCSD(T)`) [91] is expressed in our high-level representation. The computation of `CCSD(T)` notably differs from those of linear algebra routines and convolution-style stencils, by accessing its high-dimensional input data in sophisticated transposed fashions: for example, the view function of `CCSD(T)`'s *instance one* (denoted as `I1` in the subfigure) uses indices `a` and `b` to access the last two dimensions of its A input tensor (rather than the first two dimensions of the tensor, as would be the case for non-transposed accesses).

For brevity, the subfigure presents only two `CCSD(T)` instances – in our experiments in Section 4.5, we present experimental results for nine different real-world `CCSD(T)` instances.

SUBFIGURES 4-6 The subfigures present computations whose scalar functions and combine operators are different from those used in Subfigures 1-3 (which are in Subfigures 1-3 straightforward multiplications $*$, concatenation $+\!\!+$, and point-wise additions $+$ only). For example, Jacobi stencils (Subfigure 4) use as scalar function the Jacobi-specific computation $\mathtt{J_{nD}}$ [220], and *Probabilistic Record Linkage (PRL)* [221], which is heavily used in data mining to identify duplicate entries in a data base, uses a PRL-specific both scalar function `wght` and combine operator $\mathtt{max_{PRL}}$ (point-wise combination via the PRL-specific binary operator $\mathtt{max_{PRL}}$) [116]. Histograms, in their generalized version [71] (denoted as `GenHisto` in Subfigure 6), use an arbitrary, user-defined scalar function `f` and a user-defined associative and commutative combine operator $\oplus$; the standard histogram variant `Histo` is then a particular instance of `GenHist`, for $\oplus = +$ (point-wise addition) and $\mathtt{f} = \mathtt{f_{Histo}}$, where $\mathtt{f_{Histo}}(e, \mathtt{b}) = 1$ iff $e = \mathtt{b}$ and $\mathtt{f_{Histo}}(e, \mathtt{b}) = 0$ otherwise.

SUBFIGURE 7 We show how typical `map` and `reduce` patterns [251] are implemented in our high-level representation. Examples `map(f)` and `reduce(⊕)` (discussed in Examples 3 and 4) are simple and thus straightforwardly expressed in our representation. In contrast, example `reduce(⊕,⊗)` is more complex and shows how `reduce(⊕)` is extended to combine the input vector simultaneously twice – once combining vector elements via operator ⊕ and once using operator ⊗. The outcome of `reduce(⊕,⊗)` are two scalars – one representing the result of combination via ⊕ and the other of combination via ⊗ – which we map via the output view to output elements $\mathtt{O}_1$ (result of ⊕) and $\mathtt{O}_2$ (result of ⊗), correspondingly; this is also illustrated in Figure 19.

SUBFIGURE 8 We present *prefix-sum computations* [312] which differ from the computations in Subfigures 1-7 in terms of their combine operators: the operator used for expressing computations in Subfigure 8 is different from concatenation (Example 1) and point-wise combinations (Example 2). Computation `scan(⊕)` uses as combine operator $+\!\!+_{\texttt{prefix-sum}}(\oplus)$ which computes prefix-sum [299] (formally defined in the Appendix, Section .3.9) according to binary operator ⊕, and `MBBS` (Maximum Bottom Box Sum) [86] uses a particular instance of prefix-sum for ⊕ = + (addition).

4.3 LOW-LEVEL REPRESENTATION FOR DATA-PARALLEL COMPUTATIONS[13]

We introduce our low-level representation for expressing data-parallel computations. In contrast to our high-level representation, our low-level representation explicitly expresses the de-composition and re-composition of computations (informally illustrated in Figure 9). Moreover, our low-level representation is designed such that it can be straightforwardly transformed to executable program code, because it explicitly captures and expresses the optimizations for the memory and core hierarchies of the target architecture.

In the following, after briefly discussing an introductory example in Section 4.3.1, we introduce in Section 4.3.2 our formal representation of computer systems, to which we refer to as *Abstract System Model (ASM)*. Based on this model, we define *low-level MDAs*, *low-level BUFs*, and *low-level combine operators* in Section 4.3.3, which are basic building blocks of our low-level representation.

Note that all details and concepts discussed in this section are not exposed to the end users of our system and therefore transparent for them: expressions in our low-level representation are generated fully automatically for the user, from expressions in our high-level representation (Figure 10), according to the methodologies presented later in Section 4.4 and our auto-tuning techniques introduced and discussed in Chapter 5 of this thesis.

[13] The full version of this section, which contains all formal details, is provided in the Appendix, Section .4, for the interested reader.

4.3.1 *Introductory Example*

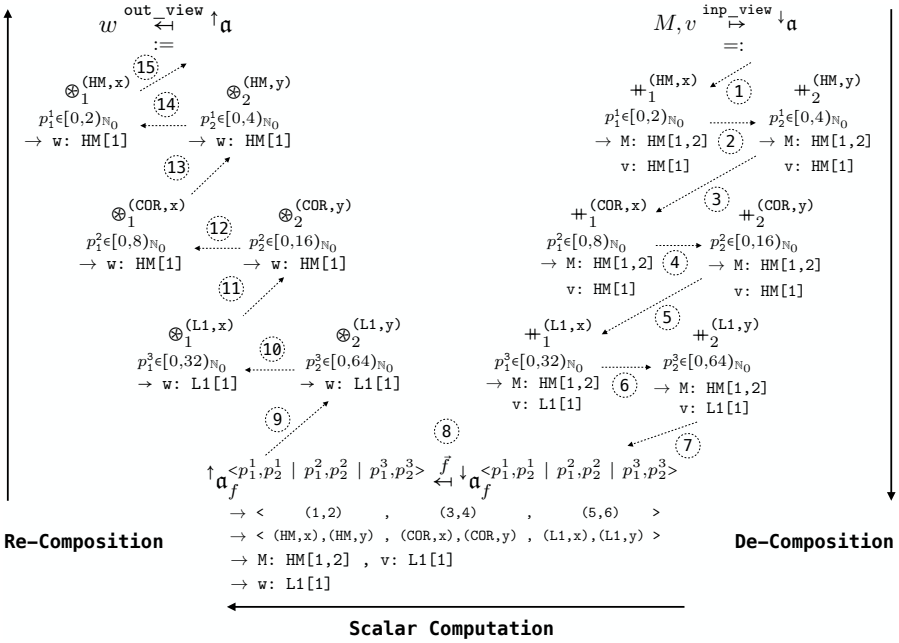


Figure 21: Low-level expression for straightforwardly computing Matrix-Vector Multiplication (`MatVec`) on a simple, artificial architecture with two memory layers (`HM` and `L1`) and one core layer (`COR`). Dotted lines indicate data flow.

Figure 21 illustrates our low-level representation by showing how `MatVec` (Matrix-Vector Multiplication) is expressed in our representation. In our example, we use an input matrix $M \in T^{512\times4096}$ of size 512×4096 (size chosen arbitrarily) that has an arbitrary but fixed scalar type $T \in$ `TYPE`; the input vector $v \in T^{4096}$ is of size 4096, correspondingly.

For better illustration, we consider for this introductory example a straightforward, artificial target architecture that has only two memory layers – *Host Memory (`HM`)* and *Cache Memory (`L1`)* – and one *Core Layer* (`COR`) only; our examples presented and discussed later in this section target real-world architectures (e.g., CUDA-capable NVIDIA GPUs). The particular values of tuning parameters (discussed in detail later in this section), such as the number of threads and the order of combine operators, are chosen by hand for this example and as straightforward for simplicity.

Our low-level representations works in three phases: 1) *decomposition* (steps 1-7, in the right part of Figure 21), 2) *scalar* (step 8, bottom part of the figure), 3) *re-composition* (steps 9-15, left part). Steps are arranged from right to left, inspired by the application order of function composition.

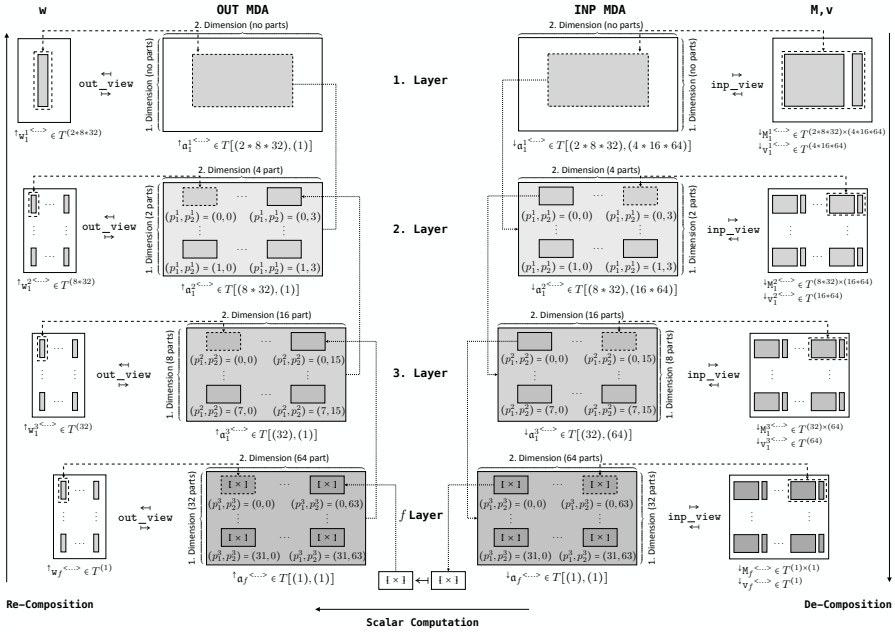


Figure 22: Illustration of multi-layered, multi-dimensional MDA partitioning using the example MDA from Figure 21. In this example, we use three layers and two dimensions, according to Figure 21.

1. DE-COMPOSITION PHASE: The de-composition phase (steps 1-7 in Figure 21) partitions input MDA ${}^{\downarrow}\mathfrak{a}$ (in the top right of Figure 21) to the structure ${}^{\downarrow}\mathfrak{a}_f^{<\cdots>}$ (bottom right) to which we refer to as *low-level MDA* and define formally in the next subsection. The low-level MDA represents a partitioning of MDA ${}^{\downarrow}\mathfrak{a}$ (a.k.a *hierarchical, multi-dimensional tiling* in programming), where each particular choice of indices $p_1^1 \in [0,2)_{\mathbb{N}_0}$, $p_2^1 \in [0,4)_{\mathbb{N}_0}$, $p_1^2 \in [0,8)_{\mathbb{N}_0}$, $p_2^2 \in [0,16)_{\mathbb{N}_0}$, $p_1^3 \in [0,32)_{\mathbb{N}_0}$, $p_2^3 \in [0,64)_{\mathbb{N}_0}$ refers to an MDA that represents an individual part of MDA ${}^{\downarrow}\mathfrak{a}$ (a.k.a. *tile* in programming – informally illustrated in Figure 13). The partitions are arranged on multiple layers (indicated by the p's superscripts) and in multiple dimensions (indicated by subscripts) – as illustrated in Figure 22 – according to the memory/core layers of the target architecture and dimensions of the MDH computation: we partition for each of the target architecture's three layers (`HM`, `L1`, `COR`) and in each of the two dimensions of the MDH (dimensions 1 and 2, as we use example `MatVec` in Figure 21, which represents a two-dimensional MDH computation). Consequently, our partitioning approach allows efficiently exploiting each particular layer of the target architecture (both memory and core layers), and also optimizing for both dimensions of the target computation (in the case of `MatVec`, the i-dimension and also the k-dimension – see Figure 7), allowing fine-grained optimizations.

We compute the partitionings of MDAs by applying the concatenation operator (Example 1) inversely (indicated by using =: instead of := in the top right part of Figure 21). For example, we partition in Figure 21 MDA ${}^{\downarrow}\mathfrak{a}$ first via the inverse of $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ in dimension 1 (indicated by the subscript 1 of $+\!\!+_1^{(\texttt{HM},\texttt{x})}$; the superscript `(HM,x)` is explained later) into 2 parts, as p_1^1 iterates over interval $[0,2)_{\mathbb{N}_0} = \{0,1\}$ which consists of two elements (0 and 1) – the interval is chosen ar-

bitrarily for this example. Afterward, each of the obtained parts is further partitioned, in the second dimension, via $+\!\!+_2^{(\texttt{HM},\texttt{y})}$ into 4 parts (p_2^1 iterates over $[0,4)_{\mathbb{N}_0} = \{0,1,2,3\}$ which consists of four elements). The $(2 * 4)$-many `HM` parts are then each further partitioned in both dimensions for the `COR` layer into $(8 * 16)$ parts, and each individual `COR` part is again partitioned for the `L1` layer into $(32 * 64)$ parts, resulting in $(2 * 8 * 32) * (4 * 16 * 64) = 512 * 4096$ parts in total.

We always use a *full partitioning* in our low-level expressions[14], i.e., each particular choice of indices p_1^1, p_2^1, p_1^2, p_2^2, p_1^3, p_2^3 points to an MDA that contains a single element only (in Figure 22, the individual elements are denoted via symbol $\times$, in the bottom part of the figure). By relying on a full partitioning, we can apply scalar function `f` to the fully partitioned MDAs later in the scalar phase (described in the next paragraph). This is because function `f` is defined on scalar values (Definition 4) to make defining scalar functions more convenient for the user (as discussed in Section 4.2.2).

The superscript of combine operators, e.g., `(COR,x)` of operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$, is a so-called *operator tag* (formal definition given in the next subsection). A tag indicates to our code generator whether its combine operator is assigned to a memory layer (and thus computed sequentially in our generated code) or to a core layer (and thus computed in parallel). For example, tag `(COR,x)` indicates that parts processed by operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$ should be computed by cores `COR`, and thus in parallel; the dimension tag `x` indicates that `COR` layer's `x` dimension should be used for computing the operator (we use dimension `x` for our example architecture as an analogous concept to CUDA's thread/block dimensions `x,y,z` for GPU architectures [38]), as we also discuss in the next subsection. In contrast, tag `(HM,x)` refers to a memory layer (host memory `HM`) and thus, operator $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ is computed sequentially. Since the current state-of-practice programming approaches (OpenMP, CUDA, OpenCL, ...) have no explicit notion of memory tiles (e.g., by offering the potential variables `tileIdx.x/tileIdx.y/tileIdx.z`, as analogous concepts to CUDA variables `threadIdx.x/threadIdx.y/threadIdx.z`), the dimensions tag `x` in `(HM,x)` is currently ignored by our code generator, because `HM` refers to a memory layer.

Note that the number of parts (2 parts on layer 1 in dimension 1; 4 parts on layer 1 in dimension 2; ...), the combine operators' tags, and our partition order (e.g. first partitioning in MDA's dimension 1 and afterward in dimension 2) are chosen arbitrarily for this example. These choices are critical for performance and should be optimized[15] for a particular target architecture and characteristics of the input and output data (size, memory layouts, etc.), as we discuss in detail later in this section.

[14] Our future work (outlined in Chapter 8) aims to additionally allow coarser-grained partitioning schemas, e.g., to target domain-specific hardware extensions (such as *NVIDIA Tensor Cores* [148] which compute 4×4 matrices immediately in hardware, rather than 1×1 matrices as obtained in the case of a full partitioning).

[15] We currently rely on auto-tuning [117] for choosing optimized values of performance-critical parameters, as we discuss in Section 4.5.

2. SCALAR PHASE: In the scalar phase (step 8 in Figure 21), we apply MDH's scalar function f to the individual MDA elements

$$\downarrow_{a_f}^{<p_1^1,p_2^1 \mid p_1^2,p_2^2 \mid p_1^3,p_2^3>}$$

for each particular choice of indices p_1^1, p_2^1, p_1^2, p_2^2, p_1^3, p_2^3, which results in

$$\uparrow_{a_f}^{<p_1^1,p_2^1 \mid p_1^2,p_2^3 \mid p_1^3,p_2^3>}$$

In the figure, $\vec{f}$ is the slight adaption of function f that operates on a singleton MDA, rather than a scalar (see Footnote 10).

Annotation → `< (1,2) , ... >` indicates the application order of applying scalar function (in this example, first iterating over p_1^1, then over p_2^1, etc), and we use annotation → `< (HM,x) , ... >` to indicate how the scalar computation is assigned to the target architecture (this is described in detail later in this section). Annotations → `M: HM , v: L1` and → `w: L1` (in the bottom part of Figure 21) indicate the memory regions to be used for reading and writing the input scalar of function f (also described later in detail).

3. RE-COMPOSITION PHASE: Finally, the re-composition phase (steps 9-15 in Figure 21) combines the computed parts $\uparrow_{a_f}^{<p_1^1,p_2^1 \mid p_1^2,p_2^2 \mid p_1^3,p_2^3>}$ (bottom left in the figure) to the final result $\uparrow a$ (top left) via MDH's combine operators, which are in the case of matrix-vector multiplication $\circledast_1 := +\!\!+$ (concatenation) and $\circledast_2 := +$ (point-wise addition). In this example, we first combine the `L1` parts in dimension 2 and then in dimension 1; afterward, we combine the `COR` parts in both dimensions, and finally the `HM` parts. Analogously to before, this order of combine operators and their tags are chosen arbitrarily for this example and should be auto-tuned for high performance.

In the de- and re-composition phases, the arrow notation below combine operators allow efficiently exploiting architecture's memory hierarchy, by indicating the memory region to read from (de-composition phase) or to write to (re-composition phase); the annotations also indicate the memory layouts to use. We exploit these memory and layout information in both: i) our code generation process to assign combine operators' input and output data to memory regions and to chose memory layouts for the data (row major, column major, etc); ii) our formalism to specify constraints of programming models, e.g., that in CUDA, results of GPU cores can only be combined in designated memory regions [37]. For example, annotation → `M: HM[1,2] , v: L1[1]` below an operator in the de-composition phase indicates to our code generator that the parts (a.k.a tiles) of matrix M used for this computation step should be read from host memory `HM` and that parts of vector v should be copied to and accessed from fast `L1` memory. The annotation also indicates that `M` should be stored using a row-major memory layout (as we use `[1,2]` and not `[2,1]`). The memory regions and layouts are chosen arbitrarily for this example

and should be chosen as optimized (auto-tuned) for the particular target architecture and characteristics of the input and output data. Formally, the arrow notation of combination operators is a concise notation to hide MDAs and BUFs for intermediate results (discussed in the Appendix, Section .5.3, for the interested reader).

Excursion: Code Generation[16]

Our low-level expressions can be straightforwardly transformed to executable program code in imperative-style programming languages (such as OpenMP, CUDA, and OpenCL). As code generation is not the focus of this work, we outline our code generation approach briefly using the example of Figure 21. Details about our code generation process are provided in the Appendix, Section .8, and will be presented and illustrated in detail in our future work (as outlined in Chapter 8).

We implement combine operators as sequential or parallel loops. For example, the operator $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ is assigned to memory layer `HM` and thus implemented as a sequential loop (loop range indicated by $[0,2)_{\mathbb{N}_0}$), and operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$ is assigned to core layer `COR` and thus implemented as a parallel loop (e.g., a loop annotated with `#pragma omp parallel for` in OpenMP [44], or variable `threadIdx.x` in CUDA [38]). Correspondingly, our three phases (de-composition, scalar, and re-composition) each correspond to an individual loop nest; we generate the nests as fused when the tags of combine operators have the same order in phases, as in Figure 21. Note that our currently targeted programming models (OpenMP, CUDA, and OpenCL) have no explicit notion of *tiles*, e.g., by offering the potential variable `tileIdx.x` for managing tiles automatically in the programming model (similarly as variable `threadIdx.x` automatically manages threads in CUDA). Consequently, when the operator tag refers to a memory layer, the dimension information within tags are currently ignored by our code generator (such as dimension `x` in tag `(HM,x)` which refers to memory layer `HM`).

Operators' memory regions correspond to straightforward allocations (e.g., in CUDA's `device`, `shared`, or `register` memory [38], according to the arrow annotations in our low-level expression). Memory layouts are implemented as aliases, e.g., *preprocessor directives* such as `#define M(i,k) M[k][i]` for storing `MatVec`'s input matrix M as transposed.

We implement MDAs also as aliases (according to Definition 7), e.g., `#define inp_mda(i,k) M[i][k],v[k]` for `MatVec`'s input MDA.

Code optimizations that are applied on a lower abstraction level than proposed by our representation in Example 21 are beyond the scope of this work and outlined in the Appendix, Section .9, e.g., loop fusion and loop unrolling which are applied on the loop-based abstraction level.

[16] Our implementation of MDH is open source: https://mdh-lang.org

We provide an open source *MDH compiler* for code generation [3]. Our compiler takes as input a high-level MDH expression (as in Figure 12), in the form of a Python program, and it fully automatically generates auto-tuned program code from it.

In the following, we introduce in Section 4.3.2 our formal representation of a computer system (which can be a single device, but also a multi-device or a multi-node system, as we discuss soon), and we illustrate our formal system representation using the example architectures targeted by programming models OpenMP, CUDA, and OpenCL. Afterward, in Section 4.3.3, we formally define the basic building blocks of our low-level representation – *low-level MDAs*, *low-level BUFs*, and *low-level combine operators* – based on our formal system representation.

4.3.2 *Abstract System Model (ASM)*

Definition 10 (Abstract System Model)**.** An L-*Layered Abstract System Model (ASM)*, $L \in \mathbb{N}$, is any pair of two positive natural numbers

$$(\,\texttt{NUM\_MEM\_LYRs}\,,\,\texttt{NUM\_COR\_LYRs}\,) \in \mathbb{N} \times \mathbb{N}$$

for which $\texttt{NUM\_MEM\_LYRs} + \texttt{NUM\_COR\_LYRs} \;=\; L$.

Our ASM representation is capable of modeling architectures with arbitrarily deep memory and core hierarchies[17]: `NUM_MEM_LYRs` denotes the target architecture's number of memory layers and `NUM_COR_LYRs` the architecture's number of core layers, correspondingly. For example, the artificial architecture we use in Figure 21 is represented as an ASM instance as follows (bar symbols denote set cardinality):

$$\texttt{ASM}_{\texttt{artif.}} := (\;\; |\{\texttt{HM},\texttt{L1}\}| \;\;,\;\; |\{\texttt{COR}\}| \;\;) = (2,1)$$

The instance is a pair consisting of the numbers 2 and 1 which represent the artificial architecture's two memory layers (`HM` and `L1`) and its single core layers (`COR`).

[17] We deliberately do not model into our ASM representation an architecture's particular number of cores and/or sizes of memory regions, because our optimization process is designed to be generic in these numbers and sizes, for high flexibility.

Example 11. We show particular ASM instances that represent the device models of the state-of-practice approaches OpenMP, CUDA, and OpenCL[18]::

$$\mathtt{ASM}_{\mathtt{OpenMP}} := (\ |\{\mathtt{MM},\mathtt{L2},\mathtt{L1}\}|\ ,\ |\{\mathtt{COR}\}|\ ) = (3,1)$$

$$\mathtt{ASM}_{\mathtt{OpenMP+L3}} := (\ |\{\mathtt{MM},\mathtt{L3},\mathtt{L2},\mathtt{L1}\}|\ ,\ |\{\mathtt{COR}\}|\ ) = (4,1)$$

$$\mathtt{ASM}_{\mathtt{OpenMP+L3+SIMD}} := (\ |\{\mathtt{MM},\mathtt{L3},\mathtt{L2},\mathtt{L1}\}|\ ,\ |\{\mathtt{COR},\mathtt{SIMD}\}|\ ) = (4,2)$$

$$\mathtt{ASM}_{\mathtt{CUDA}} := (\ |\{\mathtt{DM},\mathtt{SM},\mathtt{RM}\}|\ ,\ |\{\mathtt{SMX},\mathtt{CC}\}|\ ) = (3,2)$$

$$\mathtt{ASM}_{\mathtt{CUDA+WRP}} := (\ |\{\mathtt{DM},\mathtt{SM},\mathtt{RM}\}|\ ,\ |\{\mathtt{SMX},\mathtt{WRP},\mathtt{CC}\}|\ ) = (3,3)$$

$$\mathtt{ASM}_{\mathtt{OpenCL}} := (\ |\{\mathtt{GM},\mathtt{LM},\mathtt{PM}\}|\ ,\ |\{\mathtt{CU},\mathtt{PE}\}|\ ) = (3,2)$$

OpenMP is often used to target $(3+1)$-layered architectures which rely on 3 memory regions (main memory `MM`, and caches `L2` and `L1`) and 1 core layer (`COR`). OpenMP-compatible architectures sometimes also contain the `L3` memory region, and they may allow exploiting SIMD parallelization (a.k.a. *vectorization* [227]), which are expressed in our ASM representation as a further memory or core layer, respectively.

CUDA's target architectures are $(3+2)$-layered: they consist of *Device Memory (`DM`)*, *Shared Memory (`SM`)*, and *Register Memory (`RM`)*, and they offer as cores so-called *Streaming Multiprocessors (`SMX`)* which themselves consist of *Cuda Cores (`CC`)*. CUDA also has an implicit notion of so-called *Warps (`WRP`)* which are not explicitly represented in the CUDA programming model [38], but often exploited by programmers – via special intrinsics (e.g., *shuffle* and *tensor core intrinsics* [129, 148]) – to achieve highest performance.

OpenCL-compatible architectures are designed analogously to those targeted by the CUDA programming model; consequently, both OpenCL- and CUDA-compatible architectures are represented by the same ASM instance in our formalism. Apart from straightforward syntactical differences between OpenCL and CUDA [172], we see as the main differences between the two programming models (from our ASM-based abstraction level) that OpenCL has no notion of warps, and it uses a different terminology – *Global/Local/Private Memory (`GM`/`LM`/`PM`)* instead of device/shared/register memory, and *Compute Unit (`CU`)* and *Processing Element (`PE`)*, rather than `SMX` and `CC`.

[18] Differences between Example 11 and Figure 4 are outlined in the Appendix, Section .5.4.

In the following, we consider memory regions and cores of ASM-represented architectures as arrangeable in an arbitrary number of dimensions. Programming models for such architectures often have native support for such arrangements. For example, in the CUDA model, memory is accessed via arrays which can be arbitrary-dimensional (a.k.a *multi-dimensional C arrays*), and cores are programmed in CUDA via threads which are arranged in CUDA's so-called dimensions `x`, `y`, `z`; further thread dimensions can be explicitly programmed in CUDA, e.g., by embedding them in the last dimension `z`.

We express constraints of programming models – for example, that in CUDA, `SMX` can combine their results in `DM` only [37] – via so-called *tuning parameter constraints*, which we discuss later in this section.

Note that we call our abstraction *Abstract System Model* (rather than *Abstract Architecture Model*, or the like), because it can also represent systems consisting of multiple devices and/or nodes, etc. For example, our ASM representation of a multi-GPU system is:

$$\texttt{ASM}_{\texttt{Multi-GPU}} := (\,|\{\texttt{HM},\texttt{DM},\texttt{SM},\texttt{RM}\}|\,,\,|\{\texttt{GPU},\texttt{SMX},\texttt{CC}\}|\,) = (4,3)$$

It extends our ASM-based representation of CUDA devices (Example 11) by *Host Memory (`HM`)* which represents the memory region of the system containing the GPUs (and in which the intermediate of different GPUs are combined), and it introduces the further core layer `GPU` representing the system's GPUs. Analogously, our ASM representation of a multi-node, multi-GPU system is:

$$\texttt{ASM}_{\texttt{Multi-Node/GPU}} := (\,|\{\texttt{NM},\texttt{HM},\texttt{DM},\texttt{SM},\texttt{RM}\}|\,,\,|\{\texttt{NOD},\texttt{GPU},\texttt{SMX},\texttt{CC}\}|\,) = (5,4)$$

It adds to $\texttt{ASM}_{\texttt{Multi-GPU}}$ the memory layer *Node Memory (`NM`)* which represents the memory region of the host node, and it adds core layer *Node* (`NOD`) which represents the compute nodes. Our approach is currently designed for *homogeneous systems*, i.e., all devices/nodes/... are assumed to be identical. We aim to extend our approach to *heterogeneous systems* (which may consist of different devices/nodes/...) as future work, inspired by dynamic load balancing approaches [248].

4.3.3 *Basic Building Blocks*

We introduce the three main basic building blocks of our low-level representation: 1) *low-level MDAs* which we use to partition MDAs and that represent multi-layered, multi-dimensionally arranged collection of ordinary MDAs (Definition 1) – one ordinary MDA per memory/core layer of their target ASM and for each dimension of the MDH computation (as illustrated in Figure 22); 2) *low-level BUFs* which are a collection of ordinary BUFs (Definition 5) and that are augmented with a *memory region* and a *memory layout*; 3) *low-level combine operators* which represent combine operators (Definition 2) to which the layer and dimension of their target ASM is assigned to be used to compute the operator in our generated code (e.g., a core layer to compute the operator in parallel).

Definition 11 (Low-Level MDA)**.** An L-layered, D-dimensional, P-partitioned *low-level MDA* that has scalar type T and index sets I is any function a_{ll} of type:

$$a_{ll}^{<\overbrace{(p_1^1,\ldots,p_D^1)\in P_1^1\times\ldots\times P_D^1}^{\text{Partitioning: Layer 1}}\mid\ldots\mid\overbrace{(p_1^L,\ldots,p_D^L)\in P_1^L\times\ldots\times P_D^L}^{\text{Partitioning: Layer L}}>}:$$

$$I_1^{<p_1^1,\ldots,p_D^1\mid\ldots\mid p_1^L,\ldots,p_D^L>}\times\ldots\times I_D^{<p_1^1,\ldots,p_D^1\mid\ldots\mid p_1^L,\ldots,p_D^L>}\to T$$

We use low-level MDAs in the following to represent partitionings of MDAs (as illustrated soon).

Next, we introduce *low-level BUFs* which work similarly as BUFs (Definition 5), but are tagged with a memory region and a memory layout. While these tags have no effect on the operators' semantics, they indicate later to our code generator in which memory region the BUF should be stored and accessed, and which memory layout to chose for storing the BUF. Moreover, we use these tags to formally define constraints of programming models, e.g., that according to the CUDA specification [37], `SMX` cores can combine their results in memory region `DM` only.

Definition 12 (Low-Level BUF)**.** An L-layered, D-dimensional, P-partitioned *low-level BUF* that has scalar type T and size N is any function b_{ll} of type ($\twoheadrightarrow$ denotes bijection):

$$b_{ll}^{<\overbrace{\texttt{MEM}\in[1,\texttt{NUM\_MEM\_LYRs}]_{\mathbb{N}}}^{\text{Memory Region}}\mid\overbrace{\sigma:[1,D]_{\mathbb{N}}\twoheadrightarrow[1,D]_{\mathbb{N}}}^{\text{Memory Layout}}><\ldots\mid\overbrace{(p_1^l,\ldots,p_D^l)\in P_1^l\times\ldots\times P_D^l}^{\text{Partitioning: Layer l}}\mid\ldots>}:$$

$$[0,N_1^{<\ldots\mid p_1^l,\ldots,p_D^l\mid\ldots>})_{\mathbb{N}_0}\times\ldots\times[0,N_D^{<\ldots\mid p_1^l,\ldots,p_D^l\mid\ldots>})_{\mathbb{N}_0}\to T$$

We refer to `MEM` as low-level BUF's *memory region* and to σ as its *memory layout*, and we refer to the function

$$b_{ll}^{\texttt{trans}<\overbrace{\ldots}^{\text{Memory Region}}\mid\overbrace{\ldots}^{\text{Memory Layout}}><\ldots\mid\overbrace{\ldots}^{\text{Partitioning: Layer l}}\mid\ldots>}:$$

$$[0,N_{\sigma(1)}^{<\ldots>})_{\mathbb{N}_0}\times\ldots\times[0,N_{\sigma(D)}^{<\ldots>})_{\mathbb{N}_0}\to T$$

that is defined as

$$\mathfrak{b}_{ll}^{\texttt{trans}<\dots><\dots>}(i_{\sigma(1)},\dots,i_{\sigma(D)}) \;:=\; \mathfrak{b}_{ll}^{<\dots><\dots>}(i_1,\dots,i_D)$$

as $\mathfrak{b}_{ll}$'s *transposed function representation* (which we use to store the buffer in our generated code).

Finally, we introduce *low-level combine operators*. We define such operators to behave the same as ordinary combine operators (Definition 2), but we additionally tag them with a layer of their target ASM. Similarly as for low-level BUFs, the tag has no effect on semantics, but it is used in our code generation process to assign the computation to the hardware (e.g., indicating that the operator is computed by either an `SMX`, `WRP`, or `CC` when targeting CUDA – see Example 11). Also, we use the tags to define model-specific constraints in our formalism (as also discussed for low-level BUFs). We also tag the combine operator with a dimension of the ASM layer, enabling later in our optimization process to express advanced data access patterns (a.k.a. *swizzles* [101]). For example, when targeting CUDA, flexibly mapping ASM dimensions on `CC` layer (in CUDA terminology, the dimensions are called `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`) to array dimensions enables the well-performing *coalesced global memory accesses* [37] for both transposed and non-transposed data layouts, by only using different dimension tags.

Definition 13 (ASM Level)**.** We refer to pairs $(l_{\texttt{ASM}}, d_{\texttt{ASM}})$ – consisting of an ASM layer $l_{\texttt{ASM}} \in [1, L]_{\mathbb{N}}$ and an ASM dimension $d_{\texttt{ASM}} \in [1, D]_{\mathbb{N}}$ – as *ASM Levels* (`ASM-LVL`)[19]:

$$\texttt{ASM-LVL} := \{\ (l_{\texttt{ASM}}, d_{\texttt{ASM}}) \mid l_{\texttt{ASM}} \in [1, L]_{\mathbb{N}}, d_{\texttt{ASM}} \in [1, D]_{\mathbb{N}}\}$$

Definition 14 (Low-Level Combine Operator)**.** Let be $L \in \mathbb{N}$ (representing an ASM's number of layers) and $D \in \mathbb{N}$ (representing an MDH's number of dimensions).

A *low-level combine operator*

$$\circledast^{<(l_{\texttt{ASM}},d_{\texttt{ASM}})\in\texttt{ASM-LVL}=\{\ (l,d)\ \mid\ l\in[1,L]_{\mathbb{N}}\,,\ d\in[1,D]_{\mathbb{N}}\ \}>}$$

is a function for which $\circledast^{<(l_{\texttt{ASM}},d_{\texttt{ASM}})>}$ is an ordinary combine operator (Definition 2), for each $(l_{\texttt{ASM}}, d_{\texttt{ASM}}) \in \texttt{ASM-LVL}$.

Note that in Figure 21, for better readability, we use domain-specific identifiers for ASM layers: `HM:=1` as an alias for the ASM layer that has id 1, `L1:=2` for the layer with id 2, and `COR:=3` for the layer with id 3. For dimensions, we use aliases $x := 1$ for ASM dimension 1 and $y := 2$ for ASM dimension 2, correspondingly.

[19] For simplicity, we refrain from annotating identifier `ASM-LVL` with values L and D (e.g., $\texttt{ASM-LVL}^{<L,D>}$), because both values will usually be clear from the context.

4.4 LOWERING: FROM HIGH LEVEL TO LOW LEVEL[20]

We have designed our formalism such that an expression in our high-level representation (as in Figure 12) can be *systematically lowered* to an expression in our low-level representation (as in Figure 21). For this, we parameterize our high-level representation, step-by-step, in tuning parameters; thereby, we obtain for concrete tuning parameter values a particular expression in our low-level representation – this is formally discussed and demonstrated in the Appendix, Section .6, for the interested reader. We chose optimized values of tuning parameters fully automatically, using the auto-tuning techniques presented in Chapter 5 of this thesis; Chapter 8 outlines alternative approaches for parameter selection.

Table 1 lists the tuning parameters of our lowering process – different values of tuning parameters lead to semantically equal expressions in our low-level representation (which is proven formally in the Appendix, Section .6), but the expressions will be translated to differently optimized code variants.[21]

In the following, we explain the 15 tuning parameters in Table 1. We give our explanations in a general, formal setting that is independent of a particular computation and programming model. Dotted lines in Table 1 separate parameters for different phases: parameters `D1-D4` customize the de-composition phase, parameters `S1-S6` the scalar phase, and parameters `R1-R4` the re-composition phase, correspondingly; the parameter `0` impacts all three phases (separated by a straight line in the table).

Our tuning parameters in Table 1 have constraints: 1) *algorithmic constraints* which have to be satisfied by all target programming models, and 2) *model constraints* which are specific for particular programming models only (CUDA-specific constraints, OpenCL-specific constraints, etc), e.g., that the results of CUDA's thread blocks can be combined in designated memory regions only [37]. We discuss algorithmic constraints in the following, together with our tuning parameters; model constraints are discussed in the Appendix, Section .5.1, for the interested reader.

Note that our parameters do not aim to introduce novel optimization techniques, but to unify, generalize, and combine together well-proven optimizations, based on a formal foundation, toward an efficient, overall optimization process that applies to various combinations of data-parallel computations, architectures, and characteristics of input and output data (e.g., their size and memory layout).

In Table 1, we point to combine operators in Figure 21 using pairs (l, d) to which we refer as *MDH Levels*. We use the pairs as enumeration for operators in the de-composition and re-composition phases.

[20] The full version of this section, which contains all formal details, is provided in the Appendix, Section .6, for the interested reader.

[21] Our Appendix (Section .4.5) shows that by choosing particular tuning parameter values, we can express in our formalism the (de/re)-compositions of different, existing state-of-the-art approaches, including scheduling-based approach TVM [120] and polyhedral compilers PPCG [218] and Pluto [262].

No.	Name	Range	Description
0	#PRT	MDH-LVL → $\mathbb{N}$	number of parts
D1	$\sigma_{\downarrow\text{-ord}}$	MDH-LVL ↛ MDH-LVL	de-composition order
D2	$\leftrightarrow_{\downarrow\text{-ass}}$	MDH-LVL ↛ ASM-LVL	ASM assignment (de-composition)
D3	$\downarrow\text{-mem}^{\langle ib\rangle}$	MDH-LVL → MR	memory regions of input BUFs (ib)
D4	$\sigma^{\langle ib\rangle}_{\downarrow\text{-mem}}$	MDH-LVL → $[1,\ldots,D^{IB}_{ib}]_{\mathcal{S}}$	memory layouts of input BUFs (ib)
S1	$\sigma_{f\text{-ord}}$	MDH-LVL ↛ MDH-LVL	scalar function order
S2	$\leftrightarrow_{f\text{-ass}}$	MDH-LVL ↛ ASM-LVL	ASM assignment (scalar function)
S3	$f^{\downarrow}\text{-mem}^{\langle ib\rangle}$	MR	memory region of input BUF (ib)
S4	$\sigma^{\langle ib\rangle}_{f^{\downarrow}\text{-mem}}$	$[1,\ldots,D^{IB}_{ib}]_{\mathcal{S}}$	memory layout of input BUF (ib)
S5	$f^{\uparrow}\text{-mem}^{\langle ob\rangle}$	MR	memory region of output BUF (ob)
S6	$\sigma^{\langle ob\rangle}_{f^{\uparrow}\text{-mem}}$	$[1,\ldots,D^{OB}_{ob}]_{\mathcal{S}}$	memory layout of output BUF (ob)
R1	$\sigma_{\uparrow\text{-ord}}$	MDH-LVL ↛ MDH-LVL	re-composition order
R2	$\leftrightarrow_{\uparrow\text{-ass}}$	MDH-LVL ↛ ASM-LVL	ASM assignment (re-composition)
R3	$\uparrow\text{-mem}^{\langle ob\rangle}$	MDH-LVL → MR	memory regions of output BUFs (ob)
R4	$\sigma^{\langle ob\rangle}_{\uparrow\text{-mem}}$	MDH-LVL → $[1,\ldots,D^{OB}_{ob}]_{\mathcal{S}}$	memory layouts of output BUFs (ob)

Table 1: Tuning parameters of our low-level expressions.

Definition 15 (MDH Level)**.** We refer to pairs (l_{MDH}, d_{MDH}) – consisting of a layer $l_{MDH} \in [1, L]_{\mathbb{N}}$ and dimension $d_{MDH} \in [1, D]_{\mathbb{N}}$ – as *MDH Levels* (MDH-LVL):

$$\text{MDH-LVL} := \{\ (l_{MDH}, d_{MDH}) \mid l_{MDH} \in [1, L]_{\mathbb{N}}, d_{MDH} \in [1, D]_{\mathbb{N}}\}^{22}$$

We use the pairs to say, for example, that the MDH computation is partitioned on level $(1,1)$ (i.e., layer $l = 1$, dimension $d = 1$) into two parts, as in Figure 21.

PARAMETER 0: Parameter #PRT is a function that maps pairs in MDH-LVL to natural numbers; the parameter determines *how much* data are grouped together into parts in our low-level expression (and consequently also in our generated code later), by setting the particular number of parts (a.k.a. *tiles*) used in our expression. For example, in Figure 21, we use $\#\text{PRT}(1,1) := 2$ which causes combine operators $+\!\!+_1^{(HM,x)}$ and $\circledast_1^{(HM,x)}$ to iterate over interval $[0,2)_{\mathbb{N}_0}$ (and thus partitioning the MDH computation on level $(1,1)$ into two parts), and we use $\#\text{PRT}(1,2) := 4$ to let operators $+\!\!+_2^{(HM,y)}$ and $\circledast_2^{(HM,x)}$ iterate over interval $[0,4)_{\mathbb{N}_0}$ (partitioning into four parts on level $(1,2)$), etc.

To ensure a full partitioning (so that we obtain singleton MDAs to which scalar function f can be applied in the scalar phase, as discussed above), we require the following algorithmic constraint for the parameter (N_d denotes the input size in dimension d):

$$\prod_{l \in [1,L]_{\mathbb{N}}} \texttt{\#PRT}(l,d) = N_d, \text{ for all } d \in [1,D]_{\mathbb{N}}$$

In our generated code, the number of parts directly translates to the number of *tiles* which are computed either sequentially (a.k.a. *cache blocking* [310]) or in parallel, depending on the combine operators's tags (which are chosen via Parameters `D2,S2,R2`, as discussed soon). In our example from Figure 21, we process parts belonging to combine operators tagged with `HM` and `L1` sequentially, via `for`-loops, because `HM` and `L1` correspond to ASM's memory layers (note that Parameter `0` only chooses the number of tiles; the parameter has no effect on explicitly copying data into fast memory resources, which is the purpose of Parameters `D3,R3,S1,S2`). The `COR` parts are computed in parallel in our generated code, because `COR` corresponds to ASM's core layer, and thus, the number of `COR` parts determines the number of threads used in our code.

An optimized number of tiles is essential for achieving high performance [307], e.g., due to its impact for locality-aware data accesses (number of sequentially computed tiles) and efficiently exploiting parallelism (number of tiles computed in parallel, which corresponds to the number of threads in our generated code).

PARAMETERS `D1,S1,R1`: These three parameters are permutations on `MDH-LVL` (indicated by symbol ↠ in Table 1), determining *when* data are accessed and combined. The parameters specify the order of combine operators in the de-composition and re-composition phases (parameters `D1` and `R1`), and the order of applying scalar function f to parts (parameter `S1`). Thereby, the parameters specify when parts are processed during the computation.

In our generated code, combine operators are implemented as sequential/parallel loops such that the parameters enable optimizing loop orders (a.k.a. *loop permutation* [303]). For combine operators assigned to ASM's core layer (via parameter `R2` discussed in the next paragraph) and thus computed in parallel, parameter `R1` particularly determines when the computed results of threads are combined: if we used in the re-composition phase of Figure 21 combine operators tagged with `(COR,x)` and `(COR,y)` immediately after applying scalar function f (i.e., in steps ⑩ and ⑪, rather than steps ⑫ and ⑬), we would combine the computed intermediate results of threads multiple times, repeatedly after each individual computation step of threads, and using the two operators at the end of the re-composition phase (in steps ⑭ and ⑮) would combine the result of threads only once, at the end of the re-composition phase. Combining the results of threads early in the computation usually has the advantages of

reduced memory footprint, because memory needs to be allocated for one thread only, but at the cost of more computations, because the results of threads need to be combined multiple times. In contrast, combing the results of threads late in the computation reduces the amount of computations, but at the cost of higher memory footprint. Our parameters make this trade-off decision generic in our approach such that the decision can be left to an auto-tuning system, for example.

Note that each phase corresponds to an individual loop nest which we fuse together when parameters `D1,S1,R1` (as well as parameters `D2,S2,R2`) coincide (as also outlined in the Appendix, Section .9).

PARAMETERS `D2,S2,R2`: These parameters (symbol $\twoheadrightarrow$ in the table denotes bijection) assign MDH levels to ASM levels, by setting the tags of low-level combine operators (Definition 14). Thereby, the parameters determine *by whom* data are processed (e.g., threads or `for`-loops), similar to the concept of `bind` in scheduling languages [16]. Consequently, the parameters determine which parts should be computed sequentially in our generated code and which parts in parallel. For example, in Figure 21, we use $\leftrightarrow_{\downarrow\texttt{-ass}}\ (2,1) := (\texttt{COR},\texttt{x})$ and $\leftrightarrow_{\downarrow\texttt{-ass}}\ (2,2) := (\texttt{COR},\texttt{y})$, thereby assigning the computation of MDA parts on layer 2 in both dimensions to ASM's `COR` layer in the decomposition phase, which causes processing the parts in parallel in our generated code. For multi-layered core architectures, the parameters particularly determine the thread layer to be used for the parallel computation (e.g., `block` or `thread` in CUDA).

Using these parameters, we are able to flexibly set data access patterns in our generated code. In Figure 21, we assign parts on layer 2 to `COR` layers, which results in a so-called *block access* pattern of cores: we start $8 * 16$ threads, according to the $8 * 16$ core parts, and each thread processes a part of the input MDA representing a block of 32×64 MDA elements within the input data. If we had assigned in the figure the first computation layer to ASM's `COR` layer (in the figure, this layer is assigned to ASM's `HM` layer), we would start $2 * 4$ threads and each thread would process MDA parts of size $(8 * 32) \times (16 * 64)$; assigning the last MDH layer to `COR`s would result in $(2 * 8 * 32) \times (4 * 16 * 64)$ threads each processing a singleton MDA (a.k.a. *strided access*).

The parameters also enable expressing so-called *swizzle* access patterns [101]. For example, in CUDA, processing consecutive data elements in data dimension 1 by threads that are consecutive in thread dimension 2 (a.k.a `threadIdx.y` dimension in CUDA) can achieve higher performance due to the hardware design of fast memory resources in NVIDIA GPUs. Such swizzle patterns can be easily expressed and auto-tuned in our approach; for example, by interchanging in Figure 21 tags `(COR,x)` and `(COR,y)`. For memory layers (such as `HM` and `L1`), the dimension tags `x` and `y` currently have no effect on our generated code, as the programming models we target at the moment (OpenMP, CUDA, and OpenCL) have no explicit notion of tiles. However, this might change in the future when targeting new kinds of programming models, e.g., for upcoming architectures.

PARAMETERS `D3,R3` AND `S3,S5`: Parameters `D3` and `R3` set for each BUF the memory region to be used, thereby determining *where* data are read from or written to, respectively. In the table, we use `ib` $\in \mathbb{N}$ to refer to a particular input BUF (e.g., `ib=1` to refer to the input matrix of matrix-vector multiplication, and `ib=2` to refer to the input vector), and `ob` $\in \mathbb{N}$ refers to an output BUF, correspondingly. Parameter `D3` specifies the memory region to read from, and parameter `R3` the regions to write to. The set `MR` $:= [1, \texttt{NUM\_MEM\_LYRs}]_{\mathbb{N}}$ denotes the ASM's memory regions.

Similarly to parameters `D3` and `R3`, parameters `S3` and `S5` set the memory regions for the input and output of scalar function `f`.

Exploiting fast memory resources of architectures is a fundamental optimization [9, 64, 203, 300], particularly due to the performance gap between processors' cores and their memory systems [57, 285].

PARAMETERS `D4,R4` AND `S4,S6`: These parameters set the memory layouts of BUFs, thereby determining *how* data are accessed in memory; for brevity in Table 1, we denote the set of all BUF permutations $[1, D]_{\mathbb{N}} \twoheadrightarrow [1, D]_{\mathbb{N}}$ (Definition 12) as $[1, \ldots, D]_{\mathcal{S}}$ (symbol $\mathcal{S}$ is taken from the notation of *symmetric groups* [284]). In the case of our matrix-vector multiplication example in Figure 21, we use a standard memory layout for all matrices, which we express via the parameters by setting them to the identity function, e.g., $\sigma^{\texttt{<M>}}_{\downarrow\texttt{-mem}}(1,1) := \texttt{id}$ (Parameter `D4`) for the matrix read by operator $+\!\!+^{(\texttt{HM},\texttt{x})}_{1}$.

An optimized memory layout is important to access data in a locality-aware and thus efficient manner.

4.5 EXPERIMENTAL EVALUATION

All experiments described in this section can be reproduced using our artifact implementation [33].

We experimentally evaluate our approach by comparing it to popular representatives of four important classes:

1. *Scheduling Approach*: TVM [120] which generates GPU and CPU code from programs expressed in TVM's own high-level program representation;

2. *Polyhedral Compilers*: PPCG [218] for GPUs[23] and Pluto [262] for CPUs, which automatically generate executable program code in CUDA (PPCG) or OpenMP (Pluto) from straightforward, unoptimized C programs;

3. *Functional Approach*: Lift [192] which generates OpenCL code from a Lift-specific, functional program representation;

[23] We cannot compare to polyhedral compiler TC [110] which is optimized toward deep learning computations on GPUs, because TC is not under active development anymore and thus is not working for newer CUDA architectures [22]. Rasch, Schulze, and Gorlatch [115] show that our approach achieves higher performance than TC for popular computations on real-world data sets.

4. *Domain-Specific Libraries*: NVIDIA cuBLAS [43] and NVIDIA cuDNN [36], as well as Intel oneMKL [28] and Intel oneDNN [27], which offer the user easy-to-use, domain-specific building blocks for programming. The libraries internally rely on pre-implemented assembly code that is optimized by experts for their target application domains: linear algebra (cuBLAS and oneMKL) or convolutions (cuDNN and oneDNN), respectively. To make comparison against the libraries challenging for us, we compare to all routines provided by the libraries. For example, the cuBLAS library offers three, semantically equal but differently optimized routines for computing `MatMul`: `cublasSgemm` (the default `MatMul` implementation in cuBLAS), `cublasGemmEx` which is part of the `cuBLASEx` extension of cuBLAS [41], and the most recent `cublasLtMatmul` which is part of the `cuBLASLt` extension [42]; each of these three routines may perform differently on different problem sizes (NVIDIA usually recommends to naively test which routine performs best for the particular target problem). To make comparison further challenging for us, we exhaustively test for each routine all of its so-called `cublasGemmAlgo_t` variants, and report the routine's runtime for the best performing variant. In the case of oneMKL, we compare also to its *JIT engine* [90] which is specifically designed and optimized for small problem sizes. We also compare to library *EKR* [265] which computes data mining example `PRL` (Figure 20) on CPUs – the library is implemented in the Java programming language and parallelized via *Java Threads*, and the library is used in practice by the *Epidemiological Cancer Registry* in North Rhine-Westphalia (Germany) which is the currently largest cancer registry in Europe.

We compare to the approaches experimentally in terms of:

i) *Performance*: via a runtime comparison of our generated code against code that is generated according to the related approaches;

ii) *Portability*: based on the *Pennycook Metric* [99] which mathematically defines portability[24] as:

$$\Phi(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported, } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

In words: "for a given set of platforms H, the *performance portability* Φ of an application a solving problem p is defined as $\Phi(a,p,H)$, where $e_i(a,p)$ is the performance efficiency (i.e. a ratio of observed performance relative to some proven, achievable level of performance) of application a solving problem p

[24] Pennycook's metric is actually called *Performance Portability (PP)*. Since performance portability particularly includes functional portability, we refer to Pennycook's PP also more generally as *Portability* only.

on platform i; value $\Phi(a, p, H)$ is 0, if any platform in H is unsupported by a running p." [99]. Consequently, Pennycook defines portability as a real value in the interval $[0, 1]_{\mathbb{R}}$ such that a value close to 1 indicates *high* portability and a value close to 0 indicates *low* portability. Here, platforms H represents a set of devices (CPUs, GPUs, ...), an application a is in our context a framework (such as TVM, a polyhedral compiler, or our approach), problems p are our case studies, and $e_i(a, p)$ is computed as the runtime $a^{best}_{p,i}$ of the application that achieves the best observed runtime for problem p on platform i, divided by the runtime of application a for problem p running on platform i.

iii) *Productivity*: by intuitively arguing that our approach achieves the same/lower/higher productivity as the related approaches, using the representative example computation *Matrix-Vector Multiplication (`MatVec`)* (Figure 12). classic code metrics, such as *Lines of Code (LOC)*, COCOMO [304], McCabe's cyclomatic complexity [324], and Halstead development effort [322] are not meaningful for comparing the short and concise programs in high-level languages as proposed by the related work as well as our approach.

In the following, after discussing our application case studies, experimental setup, auto-tuning system, and code generator, we compare our approach to each of the four above mentioned classes of approaches (1)-(4) in Sections 4.5.1-4.5.4.

Application Case Studies

We use for experiments in this section popular example computations from Figure 20 that belong to different classes of computations:

- Linear Algebra Subroutines (BLAS): *Matrix Multiplication* (`MatMul`) and *Matrix-Vector Multiplication* (`MatVec`);
- Stencil Computations: *Jacobi Computation* (`Jacobi3D`) and *Gaussian Convolution* (`Conv2D`) which differ from linear algebra routines by accessing neighboring elements in their input data;
- Quantum Chemistry: *Coupled Cluster (`CCSD(T)`)* computations which differ from linear algebra routines and stencil computations by accessing their high-dimensional input data in complex, transposed fashions;
- Data Mining: *Probabilistic Record Linkage* (`PRL`) which differs from the previous computations by relying on a PRL-specific combine operator and scalar function (instead of straightforward additions or multiplications as the previous computations);

- Deep Learning: the most time-intensive computations within the popular neural networks `ResNet-50` [182], `VGG-16` [205], and `MobileNet` [144], according to their TensorFlow implementations [47–49]. Deep learning computations rely on advanced variants of linear algebra routines and stencil computations, e.g., `MCC` and `MCC_Capsule` for computing convolution-like stencils, instead of the classic `Conv2D` variant of convolution (Figure 20) – the deep learning variants are considered as significantly more challenging to optimize than their classic variants [84].

We use for experiments this subset of computations from Figure 20 to make experimenting challenging for us: the computations differ in major characteristics (as discussed in Section 4.2.4), e.g., accessing neighboring elements in their input data (as stencil computations) or not (as linear algebra routines), thus usually requiring fundamentally different kinds of optimizations. Consequently, we consider it challenging for our approach to achieve high performance for our studies, because our approach relies on a generalized optimization process (discussed in Section 4.4) that uniformly applies to any kind of data-parallel computation and also parallel architecture. In contrast, the optimization processes of the related approaches are often specially designed and tied to a particular application class and often also architecture. For example, NVIDIA cuBLAS and Intel oneMKL are highly optimized specifically for linear algebra routines on either GPU or CPU, respectively, and TVM is specifically designed and optimized for deep learning computations.

To make experimenting further challenging for us, we consider data sizes and characteristics either taken from real-world computations (e.g., from the *TCCG* benchmark suite [170] for quantum chemistry computations) or sizes that are preferable for our competitors, e.g., powers of two for which many competitors are highly optimized, e.g., vendor libraries. For the deep learning case studies, we use data characteristics (sizes, strides, padding strategy, image/filter formats, etc.) taken from the particular implementations of the neural networks when computing the popular *ImageNet* [229] data set (the particular characteristics are listed in the Appendix, Section .7.1, for the interested reader). For all experiments, we use single precision floating point numbers (a.k.a. `float` or `fp32`), as such precision is the default in TensorFlow and many other frameworks.

Experimental Setup

We run our experiments on a cluster containing two different kinds of GPUs and CPUs:

- `NVIDIA Ampere GPU A100-PCIE-40GB`
- `NVIDIA Volta GPU V100-SXM2-16GB`
- `Intel Xeon Broadwell CPU E5-2683 v4 @ 2.10GHz`
- `Intel Xeon Skylake CPU Gold-6140 @ 2.30GHz`

We represent the two CUDA GPUs in our formalism using model $\texttt{ASM}_{\texttt{CUDA+WRP}}$ (Example 11). We rely on model $\texttt{ASM}_{\texttt{CUDA+WRP}}$, rather than the CUDA's standard model $\texttt{ASM}_{\texttt{CUDA}}$ (also in Example 11), to exploit CUDA's (implicit) warp level for a fair comparison to the related approaches: warp-level optimizations are exploited by the related approaches (such as TVM), e.g., for *shuffle operations* [129] which combine the results of threads within a warp with high performance. To fairly compare to TVM and PPCG, we avoid exploiting warps' *tensor core intrinsics* [148], in all experiments, which compute the multiplication of small matrices with high performance [5], because these intrinsics are not used in the TVM- and PPCG-generated CUDA code. For the two CPUs, we rely on model $\texttt{ASM}_{\texttt{OpenCL}}$ (Example 11) for generating OpenCL code. The same as our approach, TVM also generates OpenCL code for CPUs; Pluto relies on the OpenMP approach to target CPUs.

For all experiments, we use the currently newest versions of frameworks, libraries, and compilers, as follows. We compile our generated GPU code using library `CUDA NVRTC` [39] from `CUDA Toolkit 11.4`, and we use `Intel's OpenCL runtime` version `18.1.0.0920` for compiling CPU code. For both compilers, we do not set any flags so that they run in their default modes. For the related approaches, we use the following versions of frameworks, libraries, and compilers:

- `TVM` [18] version `0.8.0` which also uses our system's `CUDA Toolkit` version `11.4` for GPU computations and `Intel's runtime` version `18.1.0.0920` for computations on CPU;
- `PPCG` [34] version `0.08.04` using flag `-target=cuda` for generating CUDA code, rather than OpenCL, as CUDA is usually better performing than OpenCL on NVIDIA GPUs, and we use flag `-sizes` followed by auto-tuned tile sizes – we rely on our *Auto-Tuning Framework (ATF)* (discussed in Chapter 5) for choosing optimized tile size values (as we discuss in the next subsection);
- `Pluto` [50] commit `12e075a` using flag `-parallel` for generating OpenMP-parallelized C code (rather than sequential C), as well as flag `-tile` to use ATF-tuned tile sizes for Pluto; the Pluto-generated OpenMP code is compiled via `Intel's icx` compiler version `2022.0.0` using the Pluto-recommended optimization flags `-O3 -qopenmp`;

- NVIDIA cuBLAS [43] from CUDA Toolkit 11.4, using the NVIDIA-recommended compiler flags -fast -O3 -DNDEBUG;
- NVIDIA cuDNN [36] from CUDA Toolkit 11.4, using the NVIDIA-recommended compiler flags -fast -O3 -DNDEBUG;
- Intel oneMKL [28] compiled with Intel's icpx compiler version 2022.0.0, using flags -DMKL_ILP64 -qmkl=parallel -L${MKLROOT}/lib/intel64 -liomp5 -lpthread -lm -ldl, as recommended for oneMKL by Intel's *Link Line Advisor* tool [26], as well as standard flags -O3 -NDEBUG;
- Intel oneDNN [27] also compiled with Intel's icpx compiler version 2022.0.0, using flags -I${DNNLROOT}/include -L${DNNLROOT}/lib -ldnnl, according to oneDNN's documentation, as well as standard flags -O3 -NDEBUG;
- EKR [265] executed via Java SE 1.8.0 Update 281.

We profile runtimes of CUDA and OpenCL programs using the corresponding, event-based profiling APIs provided by CUDA and OpenCL. For Pluto which generates OpenMP-annotated C code, we measure runtimes via system call clock_gettime [24]. In the case of C++ libraries Intel oneMKL and Intel oneDNN, we use the C++ chrono library [20] for profiling. Libraries NVIDIA cuBLAS and NVIDIA cuDNN are also based on the CUDA programming model; thus, we profile them also via CUDA events. To measure the runtimes of the EKR Java library, we use Java function System.currentTimeMillis().

All measurements of CUDA and OpenCL programs contain the pure program runtime only (a.k.a. *kernel runtime*). The runtime of *host code*[25] is not included in the reported runtimes, as performance of host code is not relevant for this work and the same for all approaches.

In all experiments, we collect measurements until the 99% confidence interval was within 5% of our reported means, according to the guidelines for *scientific benchmarking of parallel computing systems* by Hoefler and Belli [183].

Auto-Tuning

The auto-tuning process of our approach relies on our generic *Auto Tuning Framework (ATF)* which is introduced and discussed in Chapter 5 of this thesis. The ATF framework has proven to be efficient for exploring large search spaces of constrained tuning parameters (as our space introduced in Section 4.4). We use ATF, out of the box, exactly as described in Chapter 5: 1) we straightforwardly represent in ATF our search space (Table 1) via *tuning parameters* which express the parameters in the table and their constraints; 2) we use ATF's pre-implemented cost functions for CUDA and OpenCL to measure the

[25] *Host code* is required in approaches CUDA and OpenCL for program execution: it compiles the CUDA and OpenCL programs, performs data transfers between host and device, etc. We rely on our *HCA* approach (discussed in Chapter 6 of this thesis) for host code programming in this work.

cost of our generated OpenCL and CUDA codes (in this thesis, we consider as cost program's runtime, rather than its energy consumption or similar); 3) we start the tuning process using ATF's default search technique (*AUC bandit* [194]). ATF then fully automatically determines a well-performing tuning parameter configuration for the particular combination of a case study, architecture, and input/output characteristics (size, memory layout, etc).

For scheduling approach `TVM`, we use its *Ansor* [78] optimization engine which is specifically designed and optimized toward generating optimized `TVM` schedules. Polyhedral compilers `PPCG` and `Pluto` do not provide own auto-tuning systems; thus, we use for them also ATF for auto-tuning, the same as for our approach. For both compilers, we additionally also report their runtimes when relying on their internal heuristics, rather than on auto-tuning, to fairly compare to them.

To achieve the best possible performance results for `TVM`, `PPCG`, and `Pluto`, we auto-tune each of these frameworks individually, for each particular combination of case study, architecture, and input/output characteristics, the same as for our approach. For example, we start for `TVM` one tuning run when auto-tuning case study `MatMul` for the `NVIDIA Ampere GPU` on one input size, and another, new tuning run for a new input size, etc.

Hand-optimized libraries `NVIDIA cuBLAS/cuDNN` and `Intel oneMKL/oneDNN` rely on heuristics provided by experts, rather than auto-tuning. By relying on heuristics, the libraries avoid the time-intensive process of auto-tuning. However, auto-tuning is well amortized in many application areas (e.g., deep learning), because the auto-tuned implementations are re-used in many program runs. Moreover, auto-tuning avoids the complex and costly process of hand optimization by experts, and it often achieves higher performance than hand-optimized code (as we confirm later in our experiments), because well-performing optimizations are often not intuitive.

For a fair comparison, we use for each tuning run uniformly the same tuning time of 12h. Even though for many computations well-performing tuning results could often be found in less than 12h, for our approach as well as for other frameworks, we use such generous tuning time for all frameworks to avoid auto-tuning issues in our reported results – analyzing, improving, and accelerating the auto-tuning process is beyond the scope of this work and intended for our future work (as also outlined in Chapter 8). In particular, TVM's Ansor optimizer was often able to find well performing optimizations in 6h of tuning time or less. This is because Ansor explores a small search space that is designed and optimized for deep learning computations – Ansor's space is a proper subset of our space, as our space aims to capture general optimizations that apply to arbitrary data-parallel computations. However, the focus on deep learning causes Ansor to have difficulties with optimizing computations not taken from the deep learning area, as we confirm in our experiments.

To improve the auto-tuning efficiency for our implementations, we rely on a straightforward cost model that shrinks our search space in Table 1 before starting our ATF-based auto-tuning process: i) we always use the same values for Parameters `D1`, `S1`, `R1` as well as for Parameters `D2`, `S2`, `R2`, thereby generating the same loop structure for all three phases (de-composition, scalar, and re-composition) such that the structures can be generated as a fused loop nest; ii) we restrict Parameters `D2`, `S2`, `R2` to two values – one value that let threads process outer parts (a.k.a. *blocked access* or *outer parallelism*, respectively) and one to let threads process inner parts (*strided access* or *inner parallelism*); all other permutations are currently ignored for simplicity or because they have no effect on the generated code (e.g., permutations of Parameters `D2`, `S2`, `R2` that only differ in dimension tags belonging to memory layers, as discussed in the previous sections); iii) we restrict Parameters `D3`, `S3`, `S5`, `R3` such that each parameter is invariant under different values of `d` of its input pairs $(\texttt{l}, \texttt{d}) \in \texttt{MDH-LVL}$, i.e., we always copy full tiles in memory regions (and not a full tile of one input buffer and a half tile of another input buffer, which sometimes might achieve higher performance when memory is a limited resource).

Our cost model is straightforward and might filter out configurations from our search space that achieve potentially higher performance than we report for our approach in Sections 4.5.1-4.5.4. We aim to substantially improve our naive cost model in future work, based on *operational semantics* for our low-level representation, to improve the auto-tuning quality and to reduce (or even avoid) tuning time.

Code Generation

We provide an open source *MDH compiler* [3] for generating executable program code from expressions in our high-level representation (as illustrated in Figure 10). Our compiler takes as input the high-level representation of the target computation (Figure 20), in the form of a Python program, and it fully automatically generates auto-tuned program code, based on the concepts and methodologies introduced and discussed in this chapter and our ATF auto-tuning framework (Chapter 5).

In our future work, we aim to integrate our code generation approach into the *MLIR* compiler framework [56], building on work-in-progress results [66], thereby making our work better accessible to the community. We consider approaches such as *AnyDSL* [128] and *BuildIt* [55] as further, interesting frameworks in which our compiler could be implemented.

4.5.1 *Scheduling Approaches*

Performance

Figures 23-28 report the performance of the TVM-generated code, which is in CUDA for GPUs and in OpenCL for CPUs. We observe that we usually achieve with our approach the high performance of TVM and often perform even better. For example, in Figure 27, we achieve a speedup > 2× over TVM on NVIDIA Ampere GPU for matrix multiplications as used in the inference phase of the ResNet-50 neural network – an actually favorable example for TVM which is designed and optimized toward deep learning computations executed on modern NVIDIA GPUs. Our performance advantage over TVM is because we parallelize and optimize more efficiently reduction-like computations – in the case of MatMul (Figure 20), its 3rd-dimension (a.k.a. k-dimension). The difficulties of TVM with reduction computations become particularly obvious when computing dot products (Dot) on GPUs (Figure 23): the Dot's main computation part is a reduction computation (via point-wise addition, see Figure 20), thus requiring reduction-focussed optimization, in particular when targeting the highly-parallel architecture of GPUs: in the case of Dot (Figure 23), our generated CUDA code exploits parallelization over CUDA blocks, whereas the Ansor-generated TVM code exploits parallelization over threads within in a single block only, because TVM currently cannot use blocks for parallelizing reduction computations [11]. Furthermore, while TVM's Ansor rigidly parallelizes outer dimensions [78], our ATF-based tuning process has auto-tuned our tuning parameters D2, S2, R2 in Table 1 to exploit parallelism for inner dimensions, which achieves higher performance for this particular MatMul example used in ResNet-50. Also, for MatMul-like computations, Ansor always caches parts of the input in GPU's shared memory, and it computes these cached parts always in register memory. In contrast, our caching strategy is auto-tunable (via parameters D3, S3 S5, R3 in Table 1), and ATF has determined to not cache the input matrices into fast memory resources for the MatMul example in ResNet-50. Surprisingly, Ansor does not exploit fast memory resources for Jacobi stencils (Figure 24), as required to achieve high performance for them: our generated and auto-tuned CUDA kernel for Jacobi uses register memory for both inputs (image buffer and filter) when targeting NVIDIA Ampere GPU (small input size), thereby achieving a speedup over TVM+Ansor of 1.93× for Jacobi. Most likely, Ansor fails to foresee the potential of exploiting fast memory resources for Jacobi stencils, because Jacobi's index functions used for memory accesses (Figure 20) are injective. For the MatMul example of ResNet-50's training phase (Figure 27), we achieve a speedup over TVM on NVIDIA Ampere GPU of 1.26×, because auto-tuning determined to store parts of input matrix A as transposed into fast memory (via parameter D4 in Table 1). Storing parts of the input/output data as transposed is not considered by Ansor as optimization, perhaps because such optimization must be expressed in TVM's high-level

language, rather than in its scheduling language [13]. For MatVec on NVIDIA Ampere GPU (Figure 23), we achieve a speedup over TVM of 1.22× for the small input size, by exploiting a so-called *swizzle pattern* [101]: our ATF tuner has determined to assign threads that are consecutive in CUDA's x-dimension to the second MDA dimension (via parameters D2, S2, R2 in Table 1), thereby accessing the input matrix in a GPU-efficient manner (a.k.a *coalesced global memory accesses* [37]). In contrast, for MatVec computations, Ansor assigns threads with consecutive x-ids always to the first data dimension, in a non-tunable manner, causing lower performance.

Our positive speedups over TVM on CPU are for the same reasons as discussed above for GPU. For example, we achieve a speedup of $> 3\times$ over TVM on Intel Skylake CPU for MCC (Figure 28) as used in the training phase of the MobileNet neural network, because we exploit fast memory resources more efficiently than TVM: our auto-tuning process has determined to use register memory for the MCC's second input (the filter buffer F, see Table 20) and using no fast memory for the first input (image buffer I), whereas Ansor uses shared memory rigidly for both inputs of MCC. Moreover, our auto-tuning process has determined to parallelize the inner dimensions of MCC, while Ansor always parallelizes outer dimensions. We achieve the best speedup over TVM for MCC on an input size taken from TVM's own tutorials [17] (Figure 24), rather than from neural networks (as in Figures 27 and 28). This is because TVM's MCC size includes large reduction computations, which are not efficiently optimized by TVM (as discussed above).

The TVM compiler achieves higher performance than our approach for some examples in Figures 23-28. However, in most cases, this is for a technical reason only: TVM uses the NVCC compiler for compiling CUDA code, whereas our proof-of-concept code generator currently relies on NVIDIA's NVRTC library which surprisingly generates less efficient CUDA assembly than NVCC. In three cases, the higher performance of TVM over our approach is because our ATF auto-tuning framework was not able to find a better performing tuning configuration than TVM's Ansor optimization engine during our 12h tuning time; the three cases are: 1) MCC from VGG-16's inference phase on NVIDIA Ampere GPU (Figure 27), 2) MCC (capsule variant) from VGG-16's training phase on NVIDIA Ampere GPU (Figure 27), and 3) MCC (capsule variant) from ResNet-50's training phase on Intel Skylake CPU (Figure 28). However, when we manually set the Ansor-found tuning configurations also for our approach, instead of using the ATF-found configurations, we achieve for these three cases exactly the same high performance as TVM+Ansor, i.e., the well-performing configurations are contained in our search space (Table 1). Most likely, Ansor was able to find this well-performing configuration within the 12h tuning time, because it explores a significantly smaller search space that is particularly designed for deep learning computations. To avoid such tuning issues in our approach, we aim to substantially improve our auto-tuning process in future work: we plan to introduce an analytical cost model that assists (or even replaces) our auto-tuner, as we also outline in Chapter 8.

Note that the TVM compiler crashes for our data mining example PRL, because TVM has difficulties with computations relying on user-defined combine operators [14].

Portability

Figure 29 reports the portability of the TVM compiler. Our portability measurements are based on the Pennycook metric where a value close to 1 indicates high portability and a value close to 0 indicates low portability, correspondingly. We observe that except for the example of transposed matrix multiplication $\texttt{GEMM}^{\texttt{T}}$, we always achieve higher portability than TVM. The higher portability of TVM for $\texttt{GEMM}^{\texttt{T}}$ is because TVM achieves for this example higher performance than our approach on NVIDIA Volta GPU. However, the higher performance of TVM is only due to the fact that TVM uses NVIDIA's NVCC compiler for compiling CUDA code, while we currently rely on NVIDIA's NVRTC library which surprisingly generates less efficient CUDA assembly, as discussed above.

Productivity

Listing 17 shows how matrix-vector multiplication (MatVec) is implemented in TVM's high-level program representation which is embedded into the Python programming language. In line 1, the input size $(I, K) \in \mathbb{N} \times \mathbb{N}$ of matrix $M \in T^{I \times K}$ (line 2) and vector $v \in T^{K}$ (line 3) are declared, in the form of function parameters; the matrix and vector are named M and v and both are assumed to contain elements of scalar type T = float32 (floating point numbers). Line 5 defines a so-called *reduction axis* in TVM, in which all values are combined in line 8 via te.sum (addition). The basic computation part of MatVec – multiplying matrix element M[i,k] with vector element v[k] – is also specified in line 8.

```
1  def MatVec(I, K):
2      M = te.placeholder((I, K), name='M', dtype='float32')
3      v = te.placeholder((K,), name='v', dtype='float32')
4
5      k = te.reduce_axis((0, K), name='k')
6      w = te.compute(
7          (I,),
8          lambda i: te.sum(M[i, k] * v[k], axis=k)
9      )
10     return [M, v, w]
```

Listing 17: TVM program expressing Matrix-Vector Multiplication (MatVec).

While we consider the MatVec implementations of TVM (Listing 17) and our approach (Figure 12) basically on the same level of abstraction, we consider our approach as more expressive in general. This is because our approach supports multiple reduction dimensions that may rely on different combine operators, e.g., as required for expressing the MBBS example in Figure 20. In contrast, TVM is struggling with different combine operators – adding support for multiple, different reduction dimensions is considered in the TVM community as a non-trivial extension of TVM [12, 63]. Also, we consider our approach as slightly less error-prone: we automatically compute the expected sizes of matrix M (as $I \times K$) and vector v (as K), based on the user-defined input size (I, K) in line 1 and index functions $(i, k) \mapsto (i, k)$ for the matrix and $(i, k) \mapsto (k)$ for the vector in line 8 (the formula for computing the sizes is given in the Appendix, Definition 32, for the interested reader). In contrast, TVM redundantly requests these matrix and vector sizes from the user: once in lines 2 and 3 of Listing 17, and again in lines 5 and 7. TVM uses these sizes for generating the function specification of its generated MatVec code, which lets TVM generate incorrect low-level code – without issuing an error message – when the user sets non-matching sizes in lines 2/3 and lines 5/7

4.5.2 *Polyhedral Compilers*

Performance

Figures 23-28 report the performance achieved by the PPCG-generated CUDA code for GPUs and of the OpenMP-annotated C code generated by polyhedral compiler Pluto for CPUs. For a fair comparison, we report for both polyhedral compilers their performance achieved for ATF-tuned tile sizes (denoted as PPCG+ATF/Pluto+ATF in the figures), as well as the performance of the two compilers when relying on their internal heuristics instead of auto-tuning (denoted as PPCG and Pluto). In some cases, PPCG's heuristic crashed with error "too many resources requested for launch", because the heuristic seems to not take into account device-specific constraints, e.g., limited availability of GPUs' fast memory resources.

We observe from Figures 23-28 that in all cases, our approach achieves better performance than PPCG and Pluto – sometimes by multiple orders of magnitude, in particular for deep learning computations (Figures 27 and 28). This is caused by the rigid optimization goals of PPCG and Pluto, e.g., always parallelizing outer dimensions, which causes severe performance losses. For example, we achieve a speedup over PPCG of $> 13\times$ on NVIDIA Ampere GPU and of $> 60\times$ over Pluto on Intel Skylake CPU for MCC as used in the inference phase of the real-world ResNet-50 neural network. Compared to PPCG, our better performance for this MCC example is because PPCG has difficulties with efficiently parallelizing computations relying on more than 3 dimension. Most likely, this is because CUDA offers per default 3 dimensions for parallelization (called x, y, z dimension in CUDA). However, MCC relies on 7 parallelizable dimensions (as shown in Fig-

ure 20), and exploiting the parallelization opportunities of the 4 further dimensions (as done in our generated CUDA code) is essential to achieve high performance for this MCC example from ResNet-50. Our performance advantage over Pluto for the MCC example is because Pluto parallelizes the outer dimensions of MCC only (whereas our approach has the potential to parallelize all dimensions); however, the dimension has a size of only 1 for this real-world example, resulting in starting only 1 thread in the Pluto-generated OpenMP code.

For dot products Dot (Figure 23), we can observe that PPCG fails to generate parallel CUDA code, because PPCG cannot parallelize and optimize computations which rely solely on combine operators different from concatenation, as we also discuss in Section 4.6.2. In Section 4.6.2, we particularly discuss that we do not consider the performance issues of PPCG and Pluto as weaknesses of the polyhedral approach in general, but of the particular polyhedral transformations chosen for PPCG and Pluto.

Note that Pluto crashes for our data mining example (Figure 26), with "Error extracting polyhedra from source file", because the scalar function of this example is too complex for Pluto (it contains if-statements). Moreover, Intel's icx compiler struggles with compiling the Pluto-generated OpenMP code for quantum chemistry computations (Figure 25): we aborted icx's compilation process after 24h compilation time. The icx's issue with the Pluto-generated code is most likely because of too aggressive loop unrolling of Pluto – the Pluto-generated OpenMP code has often a size > 50MB for our real-world quantum chemistry examples.

Portability

Since PPCG and Pluto are each designed for particular architectures only, they achieve the lowest portability of 0 for all our studies in Figure 29, according to the Pennycook metric. To simplify for PPCG and Pluto the portability comparison with our approach, we compute the Pennycook metric additionally also for two restricted sets of devices: only GPUs to make comparison against our approach easier for PPCG, and only CPUs to make comparison easier for Pluto.

Figures 30-34 report the portability of PPCG when considering only GPUs, as well as the portability of Pluto for only CPUs. We observe that we achieve higher portability for all our studies, as we constantly achieve higher performance than the two polyhedral compilers for the studies.

Note that even when restricting our set of devices to only GPUs for PPCG or only CPUs for Pluto, the two polyhedral compilers still achieve a portability of 0 for some examples, because they fail to generate code for them (as discussed above).

Productivity

Listing 18 shows the input program of polyhedral compilers `PPCG` and `Pluto` for `MatVec`. Both take as input easy-to-implement, straightforward, sequential C code. We consider these two polyhedral compilers as more productive than our approach (as well as scheduling and functional approaches, and also polyhedral compilers that take DSL programs as input, such as TC [110]), because both compilers fully automatically generate optimized parallel code from unoptimized, sequential program code.

Rasch, Schulze, and Gorlatch [81, 82] show that our approach can achieve the same high user productivity as polyhedral compilers, by using a polyhedral frontend for our approach: we can alternatively take as input the same sequential program code as `PPCG` and `Pluto`, instead of programs implemented in our high-level program representation (as in Figure 12). The sequential input program is then transformed via polyhedral tool *pet* [236] to its polyhedral representation which is then automatically transformed to our high-level program representation, according to the methodology presented by Rasch, Schulze, and Gorlatch [81, 82].

```
1 for( int i = 0 ; i < I ; ++i )
2   for( int k = 0 ; k < K ; ++k )
3       w[i] += M[i][k] * v[k];
```

Listing 18: `PPCG/Pluto` program expressing Matrix-Vector Multiplication (`MatVec`).

4.5.3 *Functional Approaches*

Rasch, Schulze, and Gorlatch [115] show that while functional approaches provide a solid formal foundation for computations, they typically suffer from performance and portability issues. For this, Rasch, Schulze, and Gorlatch [115] compare our approach to the state-of-the-art `Lift` [192] framework which, to the best of our knowledge, has so far not been improved toward higher performance and/or better portability. Consequently, we refrain from a further performance and portability evaluation of `Lift` and focus in the following on analyzing and discussing the productivity potentials of functional approaches, using again the state-of-the-art `Lift` approach as running example. We discuss the performance and portability issues of functional approaches, from a general perspective, in Section 4.6.3.

PERFORMANCE/PORTABILITY Already experimentally evaluated by Rasch, Schulze, and Gorlatch [115] and discussed in general terms in Section 4.6.3.

PRODUCTIVITY Listing 19 shows how MatVec is implemented in Lift. In line 1, type parameters n and m are declared, via the Lift building block nFun. Line 2 declares a function fun that takes as input a matrix of size m × n and a vector of size n, both consisting of floating point numbers (float). The computation of MatVec is specified in lines 3 and 4. In line 3, Lift's map pattern iterates over all rows of the matrix, and the zip pattern in line 4 combines each row pair-wise with the input vector. Afterward, multiplication ∗ is applied to each pair, using Lift's map pattern again, and the obtained products are finally combined via addition + using Lift's reduce pattern.

```
1 nFun(n => nFun(m =>
2   fun(matrix: [[float]n]m => fun(xs: [float]n =>
3     matrix :>> map(fun(row =>
4       zip(xs, row) :>> map(*) :>> reduce(+, 0)
5     )) )) ))
```

Listing 19: Lift program expressing Matrix-Vector Multiplication (MatVec).

Already for expressing MatVec, we can observe that Lift relies on a vast set of small, functional building blocks (five building blocks for MatVec: nFun, fun, map, zip, and reduce), and the blocks have to be composed and nested in complex ways for expressing computations. Consequently, we consider programming in Lift and Lift-like approaches as complex and their productivity for the user as limited. Moreover, the approaches often need fundamental extension for targeting new kinds of computations, e.g., so-called *macro-rules* which had to be added to Lift to efficiently target matrix multiplications [169] and primitives slide and pad together with optimization *overlapped tiling* for expressing stencil computations [124]. This need for extensions limits the expressivity of the Lift language and thus further hinders productivity.

In contrast to Lift, our approach relies on exactly three higher-order functions (Figure 11) to express various kinds of data-parallel computations (Figure 20): 1) inp_view (Definition 7) which prepares the input data; our inp_view function is designed as general enough to subsume, in a structured way, the subset of all Lift patterns intended to change the view on input data, including patterns zip, pad, and slide; 2) md_hom (Definition 3) expresses the actual computation part, and it subsumes the Lift patterns performing actual computations (fun, map, reduce, ...); 3) out_view (Definition 9) expresses the view on output data and is designed to work similarly as function inp_view (Lemma 2). Our three functions are always composed straightforwardly, in the same, fixed order (Figure 11), and they do not rely on complex function nesting for expressing computations.

Note that even though our language is designed as minimalistic, it should cover the expressivity of the `Lift` language[26] and beyond: for example, we are currently not aware of any `Lift` program being able to express the prefix-sum examples in Figure 20. For the above reasons, we consider programming in our high-level language as more productive for the user than programming in `Lift`-like, functional-style languages. Furthermore, as discussed in Section 4.5.2, our approach can take as input also straightforward, sequential program code, which further contributes to the productivity of our approach.

[26] This work is focussed on dense computations. `Lift` supports sparse computations [74] which we consider as future work for our approach (as also outlined in Chapter 8). We consider `Lift`'s approach, based on their so-called *position dependent arrays*, as a great inspiration for our future goal.

Linear Algebra	NVIDIA Ampere GPU							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
TVM+Ansor	172.48	128.22	1.74	1.23	1.00	1.00	1.00	1.17
PPCG	–	–	5.44	2.95	2.20	2.73	3.40	162.92
PPCG+ATF	–	–	4.22	2.77	1.20	1.87	1.32	3.06
cuBLAS	1.10	1.11	1.14	1.01	1.40	0.92	1.60	1.50
cuBLASEx	–	–	–	–	1.20	0.91	1.60	1.33
cuBLASLt	–	–	–	–	1.20	0.88	1.60	–

Linear Algebra	NVIDIA Volta GPU							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
TVM+Ansor	82.28	67.97	1.06	1.04	1.00	1.08	0.80	1.00
PPCG	–	–	2.67	1.71	1.40	3.07	2.60	111.98
PPCG+ATF	–	–	2.44	2.24	1.00	2.16	1.20	2.83
cuBLAS	1.06	1.09	1.10	1.07	2.60	1.11	1.80	1.83
cuBLASEx	–	–	–	–	1.80	0.30	1.40	1.17
cuBLASLt	–	–	–	–	1.20	0.96	1.40	–

Linear Algebra	Intel Skylake CPU							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
TVM+Ansor	5.07	6.14	1.03	3.39	1.06	1.15	1.02	1.10
Pluto	5.40	6.48	2.49	6.24	3.21	12.25	5.45	14.30
Pluto+ATF	5.39	6.01	1.43	3.38	2.98	4.78	4.79	2.14
oneMKL	0.64	0.57	0.42	3.83	6.27	0.69	3.42	0.98
oneMKL(JIT)	–	–	–	–	0.65	–	1.13	–

Linear Algebra	Intel Broadwell CPU							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
TVM+Ansor	5.60	8.46	1.21	1.63	1.20	1.11	1.11	1.00
Pluto	4.78	6.73	3.01	1.28	4.89	5.26	6.74	11.97
Pluto+ATF	4.75	6.72	2.91	1.21	1.94	2.85	3.46	1.23
oneMKL	1.03	0.41	0.57	0.59	2.00	0.66	1.98	0.84
oneMKL(JIT)	–	–	–	–	1.03	–	1.30	–

Figure 23: Speedup (higher is better) of our approach for linear algebra routines on GPUs and CPUs, for different (real-world) input sizes, over: i) scheduling approach `TVM`, ii) polyhedral compilers `PPCG` (GPU) and `Pluto` (CPU), as well as iii) hand-optimized libraries provided by vendors. Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Stencils	NVIDIA Ampere GPU				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096,5,5	1,512,7,7,512,3,3
TVM+Ansor	1.93	2.04	1.00	2.32	1.63
PPCG	4.19	5.27	1.58	2.36	–
PPCG+ATF	1.08	1.02	1.22	1.38	9.37
cuDNN	–	–	2.20	5.29	2.44

Stencils	NVIDIA Volta GPU				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096,5,5	1,512,7,7,512,3,3
TVM+Ansor	2.05	1.86	1.00	2.00	1.50
PPCG	7.01	13.87	1.45	1.75	–
PPCG+ATF	1.03	1.00	1.23	1.34	8.28
cuDNN	–	–	2.60	3.58	4.42

Stencils	Intel Skylake CPU				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096,5,5	1,512,7,7,512,3,3
TVM+Ansor	2.30	1.64	1.59	2.46	2.76
Pluto	3.65	2.66	2.39	1.38	143.80
Pluto+ATF	1.81	1.38	2.09	1.06	61.47
oneDNN	3.92	2.60	6.47	2.83	3.91

Stencils	Intel Broadwell CPU				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096,5,5	1,512,7,7,512,3,3
TVM+Ansor	2.21	1.78	3.14	3.98	3.99
Pluto	2.10	1.67	2.29	2.17	74.48
Pluto+ATF	1.29	1.05	1.74	1.25	74.47
oneDNN	16.09	15.02	7.29	16.42	7.69

Figure 24: Speedup (higher is better) of our approach for stencil computations on GPUs and CPUs, for different (real-world) input sizes, over: i) scheduling approach TVM, ii) polyhedral compilers PPCG (GPU) and Pluto (CPU), as well as iii) hand-optimized libraries provided by vendors. Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Quantum Chemistry	NVIDIA Ampere GPU							
	abcdef-gdab-efgc	abcdef-gdac-efgb	abcdef-gdbc-efga	abcdef-geab-dfgc	abcdef-geac-dfgb	abcdef-gebc-dfga	abcdef-gfab-degc	abcdef-gfbc-dega
TVM+Ansor	1.15	1.07	1.25	1.00	1.36	1.05	1.00	1.15
PPCG	10585.85	10579.40	9819.81	11211.57	10181.14	10482.81	11693.21	10585.85
PPCG+ATF	11.19	15.60	14.06	11.45	11.81	12.06	11.72	11.19

Quantum Chemistry	NVIDIA Volta GPU							
	abcdef-gdab-efgc	abcdef-gdac-efgb	abcdef-gdbc-efga	abcdef-geab-dfgc	abcdef-geac-dfgb	abcdef-gebc-dfga	abcdef-gfab-degc	abcdef-gfbc-dega
TVM+Ansor	1.09	0.93	1.04	1.03	1.01	1.11	1.01	1.09
PPCG	6466.22	6019.64	6300.31	6468.40	6608.80	5256.49	6602.22	6466.22
PPCG+ATF	8.28	9.61	9.38	7.21	6.60	5.14	7.77	8.28

Quantum Chemistry	Intel Skylake CPU							
	abcdef-gdab-efgc	abcdef-gdac-efgb	abcdef-gdbc-efga	abcdef-geab-dfgc	abcdef-geac-dfgb	abcdef-gebc-dfga	abcdef-gfab-degc	abcdef-gfbc-dega
TVM+Ansor	1.60	1.50	2.06	1.70	1.20	2.12	1.56	1.60
Pluto	147.45	151.55	206.60	162.58	157.43	145.17	321.66	147.45
Pluto+ATF	1.89	2.01	1.89	1.80	1.82	1.92	1.84	1.89

Quantum Chemistry	Intel Broadwell CPU							
	abcdef-gdab-efgc	abcdef-gdac-efgb	abcdef-gdbc-efga	abcdef-geab-dfgc	abcdef-geac-dfgb	abcdef-gebc-dfga	abcdef-gfab-degc	abcdef-gfbc-dega
TVM+Ansor	1.06	1.28	1.16	1.15	1.29	1.13	2.07	1.06
Pluto	–	–	–	–	–	–	–	–
Pluto+ATF	–	–	–	–	–	–	–	–

Figure 25: Speedup (higher is better) of our approach for quantum chemistry computations Coupled Cluster (`CCSD(T)`) on GPUs and CPUs, for different (real-world) input sizes, over: i) scheduling approach `TVM`, and ii) polyhedral compilers `PPCG` (GPU) and `Pluto` (CPU). Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Data Mining	NVIDIA Ampere GPU					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
TVM+Ansor	–	–	–	–	–	–
PPCG	1.49	1.05	1.12	1.22	1.37	1.56
PPCG+ATF	1.40	1.22	1.50	1.63	1.83	2.12

Data Mining	NVIDIA Volta GPU					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
TVM+Ansor	–	–	–	–	–	–
PPCG	1.11	1.15	1.10	1.30	1.51	1.82
PPCG+ATF	1.26	1.37	1.47	1.77	2.07	2.48

Data Mining	Intel Skylake CPU					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
TVM+Ansor	–	–	–	–	–	–
Pluto	–	–	–	–	–	–
Pluto+ATF	–	–	–	–	–	–
EKR	6.18	5.39	9.62	19.87	26.42	24.78

Data Mining	Intel Broadwell CPU					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
TVM+Ansor	–	–	–	–	–	–
Pluto	–	–	–	–	–	–
Pluto+ATF	–	–	–	–	–	–
EKR	8.01	9.17	23.58	66.90	119.33	167.19

Figure 26: Speedup (higher is better) of our approach for data mining algorithm Probabilistic Record Linkage (PRL) on GPUs and CPUs, for different (real-world) input sizes, over: i) scheduling approach TVM, and ii) polyhedral compilers PPCG (GPU) and Pluto (CPU), as well as the iii) hand-implemented Java CPU implementation used by *EKR* – the largest cancer registry in Europa. Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Deep Learning	NVIDIA Ampere GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.00	1.26	1.05	2.22	0.93	1.42	0.88	1.14	0.94	1.00
PPCG	3456.16	8.26	–	7.89	1661.14	7.06	5.77	5.08	2254.67	7.55
PPCG+ATF	3.28	2.58	13.76	5.44	4.26	3.92	9.46	3.73	3.31	10.71
cuDNN	0.92	–	1.85	–	1.22	–	1.94	–	1.81	2.14
cuBLAS	–	1.58	–	2.67	–	0.93	–	1.04	–	–
cuBLASEx	–	1.47	–	2.56	–	0.92	–	1.02	–	–
cuBLASLt	–	1.26	–	1.22	–	0.91	–	1.01	–	–

Deep Learning	NVIDIA Volta GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	0.75	1.21	0.72	1.79	1.00	1.11	1.06	1.00	1.00	1.00
PPCG	1976.38	5.88	–	5.64	994.16	3.41	8.21	2.51	1411.92	7.26
PPCG+ATF	3.43	3.54	3.42	4.93	3.85	3.15	8.13	2.05	3.49	3.56
cuDNN	1.21	–	1.29	–	2.80	–	3.50	–	2.32	3.14
cuBLAS	–	1.33	–	1.14	–	1.09	–	1.04	–	–
cuBLASEx	–	1.21	–	1.07	–	1.04	–	1.03	–	–
cuBLASLt	–	1.00	–	1.07	–	1.04	–	1.02	–	–

Deep Learning (Capsule)	NVIDIA Ampere GPU					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
TVM+Ansor	0.96	1.00	0.79	1.02	0.88	0.99
PPCG	4642.24	–	1013.55	–	4017.74	–
PPCG+ATF	25.98	85.33	4.41	13.64	8.89	22.12
cuDNN	–	–	–	–	–	–

Deep Learning (Capsule)	NVIDIA Volta GPU					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
TVM+Ansor	0.95	1.01	1.05	0.97	1.04	0.87
PPCG	2935.40	–	945.16	–	2885.90	–
PPCG+ATF	19.24	19.68	8.28	12.29	8.84	6.41
cuDNN	–	–	–	–	–	–

Figure 27: Speedup (higher is better) of our approach for the most time-intensive computations used in deep learning neural networks `ResNet-50`, `VGG-16`, and `MobileNet` on GPUs, for different (real-world) input sizes, over: i) scheduling approach `TVM`, ii) polyhedral compilers `PPCG` (GPU), as well as iii) hand-optimized libraries provided by vendors. Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Deep Learning	Intel Skylake CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.05	1.14	1.20	1.97	1.14	2.38	1.27	3.01	1.40
Pluto	355.81	49.57	364.43	13.93	130.80	93.21	186.25	36.30	152.14	75.37
Pluto+ATF	13.08	19.70	170.69	6.57	3.11	6.29	53.61	8.29	3.50	25.41
oneDNN	0.39	–	5.07	–	1.22	–	9.01	–	1.05	4.20
oneMKL	–	0.44	–	1.09	–	0.88	–	0.53	–	–
oneMKL(JIT)	–	6.43	–	8.33	–	27.09	–	9.78	–	–

Deep Learning	Intel Broadwell CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.60	1.29	1.53	1.32	1.00	1.27	1.02	2.42	1.92
Pluto	4349.20	40.41	137.21	15.96	1865.07	53.57	113.40	24.10	2255.00	53.85
Pluto+ATF	6.43	8.93	61.60	6.91	5.07	4.38	42.63	4.45	6.43	29.18
oneDNN	1.30	–	1.81	–	2.94	–	2.85	–	1.83	4.47
oneMKL	–	1.45	–	1.36	–	1.35	–	0.50	–	–
oneMKL(JIT)	–	19.78	–	9.77	–	50.58	–	10.70	–	–

Deep Learning (Capsule)	Intel Skylake CPU					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
TVM+Ansor	0.94	1.14	3.50	1.18	2.94	1.59
Pluto	209.36	265.77	–	166.45	160.49	159.34
Pluto+ATF	14.33	265.77	3.33	60.66	4.40	57.21
oneDNN	–	–	–	–	–	–

Deep Learning (Capsule)	Intel Broadwell CPU					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
TVM+Ansor	2.61	1.30	3.55	1.00	1.32	2.24
Pluto	–	–	–	–	–	–
Pluto+ATF	4418.82	56.17	75.77	2173.72	202.34	158.52
oneDNN	–	–	–	–	–	–

Figure 28: Speedup (higher is better) of our approach for the most time-intensive computations used in deep learning neural networks `ResNet-50`, `VGG-16`, and `MobileNet` on CPUs, for different (real-world) input sizes, over: i) scheduling approach `TVM`, ii) polyhedral compilers `Pluto` (CPU), as well as iii) hand-optimized libraries provided by vendors. Dash symbol "-" means that this framework does not support this particular combination of architecture, computation, and data characteristic.

Linear Algebra	Pennycook Metric							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
MDH+ATF	0.88	0.64	0.65	0.85	0.88	0.54	0.94	0.95
TVM+Ansor	0.01	0.02	0.54	0.47	0.83	0.50	0.97	0.89

Stencils	Pennycook Metric				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096,5,5	1,512,7,7,512,3,3
MDH+ATF	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.47	0.55	0.59	0.37	0.41

Quantum Chemistry	Pennycook Metric							
	abcdefg-gdab-efgc	abcdefg-gdac-efgb	abcdefg-gdbc-efga	abcdefg-geab-dfgc	abcdefg-geac-dfgb	abcdefg-gebc-dfga	abcdefg-gfab-degc	abcdefg-gfbc-dega
MDH+ATF	1.00	0.98	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.82	0.82	0.73	0.82	0.82	0.74	0.71	0.84

Data Mining	Pennycook Metric					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH+ATF	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.00	0.00	0.00	0.00	0.00	0.00

Deep Learning	Pennycook Metric									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
MDH+ATF	0.67	0.76	0.91	1.00	0.98	0.95	0.97	0.68	0.98	1.00
TVM+Ansor	0.53	0.62	0.89	0.59	0.76	0.81	0.70	0.61	0.54	0.75

Deep Learning (Capsule)	Pennycook Metric					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
MDH+ATF	0.96	1.00	0.94	0.99	0.97	0.96
TVM+Ansor	0.71	0.90	0.44	0.95	0.63	0.69

Figure 29: Portability (higher is better), according to Pennycook metric, of our MDH-based approach and TVM over GPUs and CPUs for case studies, using different (real-world) input sizes. Polyhedral compilers PPCG/Pluto and vendor libraries by NVIDIA and Intel are not listed: due to their limitation to certain architectures, all of them achieve the lowest portability of 0 only.

Linear Algebra	Pennycook Metric (GPUs only)							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
MDH+ATF	1.00	1.00	1.00	1.00	1.00	0.45	0.89	1.00
TVM+Ansor	0.01	0.01	0.71	0.88	1.00	0.42	1.00	0.92
PPCG	0.00	0.00	0.25	0.43	0.56	0.15	0.30	0.01
PPCG+ATF	0.00	0.00	0.30	0.40	0.91	0.21	0.71	0.34
cuBLAS	0.93	0.91	0.89	0.96	0.50	0.42	0.52	0.60
cuBLASEx	0.00	0.00	0.00	0.00	0.67	0.98	0.60	0.00
cuBLASLt	0.00	0.00	0.00	0.00	0.83	0.48	0.60	0.00

Linear Algebra	Pennycook Metric (CPUs only)							
	Dot		MatVec		MatMul		$MatMul^T$	bMatMul
	2^{24}	10^7	4096,4096	8192,8192	10,500,64	1024,1024,1024	10,500,64	16,10,500,64
MDH+ATF	0.78	0.48	0.48	0.74	0.79	0.67	1.00	0.90
TVM+Ansor	0.15	0.06	0.44	0.32	0.71	0.60	0.94	0.86
Pluto	0.15	0.07	0.18	0.24	0.20	0.08	0.16	0.07
Pluto+ATF	0.15	0.07	0.23	0.37	0.31	0.18	0.24	0.55
oneMKL	0.99	1.00	1.00	0.41	0.17	1.00	0.37	1.00
oneMKL(JIT)	0.00	0.00	0.00	0.00	0.98	0.00	0.83	0.00

Figure 30: Portability (higher is better), according to Pennycook metric, for linear algebra routines computed on only GPUs or CPUs, respectively, using different (real-world) input sizes. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

Stencils	Pennycook Metric (GPUs only)				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096	1,512,7,7,512,3,3
MDH+ATF	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.50	0.51	1.00	0.46	0.64
PPCG	0.18	0.10	0.66	0.49	0.00
PPCG+ATF	0.95	0.99	0.82	0.74	0.11
cuDNN	0.00	0.00	0.42	0.23	0.29

Stencils	Pennycook Metric (CPUs only)				
	Jacobi3D		Conv2D		MCC
	256,256,256	512,512,512	224,224,5,5	4096,4096	1,512,7,7,512,3,3
MDH+ATF	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.44	0.58	0.42	0.31	0.30
Pluto	0.35	0.46	0.43	0.56	0.01
Pluto+ATF	0.65	0.83	0.52	0.86	0.01
oneDNN	0.10	0.11	0.15	0.10	0.17

Figure 31: Portability (higher is better), according to Pennycook metric, for stencil computations computed on only GPUs or CPUs, respectively, using different (real-world) input sizes. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

Quantum Chemistry	Pennycook Metric (GPUs only)							
	abcdefg–gdab–efgc	abcdefg–gdac–efgb	abcdefg–gdbc–efga	abcdefg–geab–dfgc	abcdefg–geac–dfgb	abcdefg–gebc–dfga	abcdefg–gfab–degc	abcdefg–gfbc–dega
MDH+ATF	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.90	0.96	0.87	0.99	0.84	0.93	0.99	1.00
PPCG	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PPCG+ATF	0.10	0.08	0.09	0.11	0.11	0.12	0.10	0.15

Quantum Chemistry	Pennycook Metric (CPUs only)							
	abcdefg–gdab–efgc	abcdefg–gdac–efgb	abcdefg–gdbc–efga	abcdefg–geab–dfgc	abcdefg–geac–dfgb	abcdefg–gebc–dfga	abcdefg–gfab–degc	abcdefg–gfbc–dega
MDH+ATF	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.75	0.72	0.62	0.70	0.80	0.62	0.55	0.72
Pluto	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Pluto+ATF	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 32: Portability (higher is better), according to Pennycook metric, for quantum chemistry computation Coupled Cluster (CCSD(T)) computed on only GPUs or CPUs, respectively, using different (real-world) input sizes. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

Data Mining	Pennycook Metric (GPUs only)					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH+ATF	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.00	0.00	0.00	0.00	0.00	0.00
PPCG	0.77	0.91	0.90	0.80	0.69	0.59
PPCG+ATF	0.75	0.77	0.67	0.59	0.51	0.43

Data Mining	Pennycook Metric (CPUs only)					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH+ATF	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.00	0.00	0.00	0.00	0.00	0.00
Pluto	0.00	0.00	0.00	0.00	0.00	0.00
Pluto+ATF	0.00	0.00	0.00	0.00	0.00	0.00
EKR	0.14	0.14	0.06	0.02	0.01	0.01

Figure 33: Portability (higher is better), according to Pennycook metric, for data mining algorithm Probabilistic Record Linkage (PRL) computed on only GPUs or CPUs, respectively, using different (real-world) input sizes. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

Deep Learning	Pennycook Metric (GPUs only)									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
MDH+ATF	0.82	1.00	0.84	1.00	0.96	0.95	0.94	1.00	0.97	1.00
TVM+Ansor	0.96	0.81	0.98	0.50	1.00	0.75	0.97	0.93	1.00	1.00
PPCG	0.00	0.14	0.00	0.15	0.00	0.18	0.14	0.26	0.00	0.13
PPCG+ATF	0.24	0.33	0.11	0.19	0.24	0.27	0.11	0.35	0.28	0.14
cuBLAS	0.76	0.00	0.55	0.00	0.48	0.00	0.35	0.00	0.47	0.38
cuBLASEx	0.00	0.69	0.00	0.53	0.00	0.95	0.00	0.96	0.00	0.00
cuBLASLt	0.00	0.75	0.00	0.55	0.00	0.97	0.00	0.97	0.00	0.00
cuDNN	0.00	0.88	0.00	0.87	0.00	0.98	0.00	0.98	0.00	0.00

Deep Learning	Pennycook Metric (CPUs only)									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
MDH+ATF	0.56	0.61	1.00	1.00	1.00	0.94	1.00	0.51	1.00	1.00
TVM+Ansor	0.37	0.50	0.82	0.73	0.61	0.87	0.55	0.45	0.37	0.60
Pluto	0.00	0.01	0.00	0.07	0.00	0.01	0.01	0.02	0.00	0.02
Pluto+ATF	0.05	0.04	0.01	0.15	0.24	0.17	0.02	0.08	0.20	0.04
oneMKL	0.87	0.00	0.29	0.00	0.48	0.00	0.17	0.00	0.69	0.23
oneMKL(JIT)	0.00	0.82	0.00	0.82	0.00	0.85	0.00	1.00	0.00	0.00
oneDNN	0.00	0.06	0.00	0.11	0.00	0.02	0.00	0.05	0.00	0.00

Deep Learning (Capsule)	Pennycook Metric (GPUs only)					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
MDH+ATF	0.95	1.00	0.88	0.98	0.94	0.93
TVM+Ansor	1.00	0.99	0.98	0.99	0.98	1.00
PPCG	0.00	0.00	0.00	0.00	0.00	0.00
PPCG+ATF	0.04	0.02	0.14	0.08	0.11	0.07
cuDNN	0.00	0.00	0.00	0.00	0.00	0.00

Deep Learning (Capsule)	Pennycook Metric (CPUs only)					
	ResNet-50		VGG-16		MobileNet	
	Training	Inference	Training	Inference	Training	Inference
	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule	MCC_Capsule
MDH+ATF	0.97	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.55	0.82	0.28	0.92	0.47	0.52
Pluto	0.00	0.00	0.00	0.00	0.00	0.00
Pluto+ATF	0.00	0.01	0.03	0.00	0.01	0.01
oneDNN	0.00	0.00	0.00	0.00	0.00	0.00

Figure 34: Portability (higher is better), according to Pennycook metric, for deep learning computations computed on only GPUs or CPUs, respectively, using different (real-world) input sizes. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

4.5.4 *Domain-Specific Approaches*

Performance

Figures 23-28 report for completeness also performance results achieved by domain-specific approaches. Since domain-specific approaches are specifically designed and optimized for particular applications domains and often also architectures (e.g., only linear algebra routines on only GPU), we consider comparing to them as most challenging for us: our approach is designed and optimized for data-parallel computations in general, from arbitrary application domains (the same as also polyhedral compilers and many functional approaches), and our approach is also designed as generic in the target parallel architecture.

We observe in Figures 23-28 that the domain-specific libraries `NVIDIA cuBLAS/cuDNN` (for linear algebra routines and convolutions on GPUs) and `Intel oneMKL/oneDNN` (for linear algebra routines and convolutions on CPUs) sometimes perform better and sometimes worse than our approach.

The better performance of libraries over our approach is most likely[27] because the libraries internally rely on assembly-level optimizations, while we currently focus on the higher CUDA/OpenCL level of abstraction which offers less optimization opportunities [214, 264]. The `cuBLASEx` extension of `cuBLAS` achieves in one case – `MatMul` on `NVIDIA Volta GPU` for square 1024×1024 input matrices – significantly higher performance than our approach. The high performance is achieved by `cuBLASEx` when using its `CUBLAS_GEMM_ALGO1_TENSOR_OP` algorithm variant, which casts the `float`-typed inputs implicitly to the half precision type (a.k.a. `half` or `fp16`), allowing `cuBLASEx` to exploit the GPU's tensor core extension [148]. Thereby, `cuBLASEx` achieves significantly higher performance than our approach, because tensor cores compute small matrix multiplication immediately in hardware; however, at the cost of a significant precision loss: the `half` scalar type achieves only half the accuracy achieved by scalar type `float`. When using `cuBLASEx`'s default algorithm `CUBLAS_GEMM_DEFAULT` (rather than algorithm `CUBLAS_GEMM_ALGO1_TENSOR_OP`), which retains the `float` type and thus meets the accuracy expected from the computation, we achieve a speedup of 1.11× over `cuBLASEx`.[28]

[27] Since the Intel and NVIDIA libraries are not open source, we cannot explain their performance behavior with certainty.

[28] For the interested reader, our Appendix, Section .7.2, reports the runtime of `cuBLASEx` for all its algorithm variants, including reports for the accuracy achieved by the different variants.

The reason for the better performance of our approach over NVIDIA and Intel libraries is most likely because our approach allows generating code that is also optimized (auto-tuned) for data characteristics, which is important for high performance [155]. In contrast, the vendor libraries usually rely on pre-implemented code that is optimized toward only average high performance for a range of data characteristics (size, memory layout, etc). By relying on these fixed, pre-implemented code, the libraries avoid the auto-tuning overhead. However, auto-tuning is often amortized, particularly for deep learning computations – the main target of libraries `NVIDIA cuDNN` und `Intel oneDNN` – because the auto-tuned implementations are re-used in many program runs. Moreover, we achieve better performance for convolutions (Figure 24), because the libraries re-use optimizations for these computations originally intended for linear algebra routines [165], whereas our optimization space (Table 1) is designed for data-parallel computations in general and not as specifically oriented toward linear algebra.

Compared to the `EKR` library (Figure 26), we achieve higher performance, because the `EKR`'s Java implementation inefficiently handles memory: the library is implemented using Java's `ArrayList` data structure which is convenient to use for the Java programmer, but inefficient in terms of performance, because the structure internally performs costly memory re-allocations.

Portability

Similar to polyhedral compilers `PPCG` and `Pluto`, the domain-specific approaches work for certain architectures only and thus achieve the lowest portability of 0 only in Figure 29 for our studies. The domain-specific approaches are also restricted to a narrow set of studies, e.g., only linear algebra routines as `NVIDIA cuBLAS` and `Intel oneMKL` or only data mining example `PRL` as `EKR`. Consequently, the approaches achieve for these unsupported studies also a portability of only 0 in Figures 30-34 in which our portability evaluation is limited to only GPUs or CPUs, respectively, to make comparison against our approach easier for the vendor libraries.

For their target studies, domain-specific approaches achieve high portability. This is because the approaches are specifically designed and optimized toward these studies, e.g., via assembly-level optimizations which are currently beyond the scope of our work and considered as future work for our approach (see Chapter 8).

Productivity

Listing 20 shows the implementation of `MatVec` in domain-specific approach `NVIDIA cuBLAS`; the implementation of `MatVec` in other domain-specific approaches, e.g., `Intel oneMKL`, is analogous to the implementation in Listing 20.

We consider domain-specific approaches as most productive for their target domain: in the case of `MatVec`, the user simply calls the high-level function `cublasSgemv` and passes to it the input matrices (omitted via ellipsis in the listing) together with some meta information (memory layout of matrices, etc); `cuBLAS` then automatically starts the GPU computation for `MatVec`.

Besides the fact that domain-specific approaches typically target only particular target architectures, a further fundamental productivity issue of domain-specific approaches is that they can only be used for a narrow class of computations, e.g., only linear algebra routines as `NVIDIA cuBLAS` and `Intel oneMKL`. Moreover, in the case of domain-specific libraries from NVIDIA and Intel, it is often up to the user to manually choose among different, semantically equal but differently performing implementations for high performance. For example, the `cuBLAS` library offers three different routines for computing matrix multiplications: 1) `cublasSgemm` (part of standard `cuBLAS`), 2) `cublasGemmEx` (part of the `cuBLASEx` extension of `cuBLAS`), and 3) routine `cublasLtMatmul` (part of the `cuBLASLt` extension). These routines often also offer different, so-called *algorithms* (e.g., 42 algorithm variants in the case `cuBLASEx`) which impact the internal optimization process. When striving for the highest performance potentials of libraries, the user is in charge of naively testing each possible combination of routine and algorithm variant (as we have done in Figures 23-28 to make experimenting challenging for us). In addition, the user must be aware that different combinations of routines and algorithms can produce results of reduced accuracy (as discussed above), which can be critical for accuracy-sensitive use cases.

```
1 cublasSgemv( /* ... */ );
```

Listing 20: `cuBLAS` program expressing Matrix-Vector Multiplication (`MatVec`)

4.6 COMPARISON TO RELATED WORK

Three major classes of approaches currently focus on code generation and optimization for data-parallel computations: 1) scheduling, 2) polyhedral, and 3) functional. In the following, we compare in Sections 4.6.1-4.6.3 our approach to each of these three classes – in terms of *performance*, *portability*, and *productivity*. In contrast to Section 4.5, which has compared our approach against these classes experimentally, this section is focussed on discussions in a more general, non-experimental context. Afterward, we outline domain-specific approaches in Section 4.6.4, which are specifically designed and optimized toward their target application domains. Finally, in Section 4.6.5, we outline approaches focussing on optimizations that operate at the algorithmic level of abstraction (and thus at a higher abstraction level than our approach); we consider these higher level approaches as greatly combinable with our work.

4.6.1 *Scheduling Approaches*

Popular examples of scheduling approaches include *UTF* [294], *URUK* [272], *CHill* [212, 263], *Halide* [217], *Clay* [159], *TVM* [120], *TeML* [76], *Tiramisu* [83], *DaCe* [85], *Fireiron* [69], *Elevate* [68], *DISTAL* [53], and *LoopStack* [52]. While scheduling approaches usually achieve high performance, they often have difficulties with achieving portability and productivity, as we discuss in the following.[29]

PERFORMANCE Scheduling approaches usually achieve high performance. For this, the approaches incorporate human expert knowledge into their optimization process which is based on two major steps: 1) a human expert implements an optimization program (a.k.a *schedule*) in a so-called *scheduling language* – the program specifies the basic optimizations to perform, such as tiling and parallelization; 2) an auto-tuning system (or a human hardware expert) chooses values of performance-critical parameter of the optimizations implemented in the schedule, e.g., particular values of tile sizes and concrete numbers of threads.

Our experiments in Section 4.5 show that compared to the state-of-the-art scheduling approach TVM (using its recent Ansor optimizer [78] for schedule generation), our approach achieves competitive and sometimes even better performance, e.g., speedups up to 2.22× on GPU and 3.55× on CPU over TVM+Ansor for computations taken from TVM's favorable application domain (deep learning). Section 4.5 discusses that our better performance is due to the design and structure of our general optimization space (Table 1) which can

[29] Rasch et al. [10] introduce (optionally) a scheduling language for MDH to incorporate expert knowledge into MDH's optimization process, e.g., to achieve 1) better optimization, as an auto-tuning system might not always make the same high-quality optimization decisions as a human expert, and/or 2) faster auto-tuning, as some (or even all) optimization decisions might be made by the expert user and thus are not left to the costly auto-tuner.

be efficiently explored, fully automatically, using state-of-the-art auto tuning techniques (Chapter 5). We focus on TVM in our experiments (rather than, e.g. Halide) to make experimenting challenging for us: TVM+Ansor has proved to achieve higher performance on GPUs and CPUs than popular state-of-practice approaches [78], including Halide, pyTorch [98], and the recent FlexTensor optimizer [79].

The recent approach TensorIR [5] is a compiler for deep learning computations that achieves higher performance than TVM on NVIDIA GPUs. However, this performance gain over TVM is mainly achieved by exploiting the domain-specific *tensor core* [148] extensions of NVIDIA GPUs, which compute in hardware the multiplications of small, low-precision 4×4 matrices. For this, TensorIR introduces the concept of *blocks* which represent sub-computations, e.g., multiplying 4×4 matrices. These blocks are than mapped by TensorIR to domain-specific hardware extensions, which often leads to high performance.

While domain-specific hardware extensions are not targeted in this thesis, we can naturally exploit them in our approach, similar to TensorIR, as we plan for our future work: the sub-computations targeted by the current hardware extensions, such as matrix multiplication on 4×4 matrices, can be straightforwardly expressed in our approach (Figure 20). Thus, we can match these sub-expressions in our low-level representation and map them to hardware extensions in our generated code. For this, instead of relying on a full partitioning in our low-level representation (as in Figure 21) such that we can apply scalar function f to the fully de-composed data (consisting of a single scalar value only in the case of a full partitioning), we plan to rely on a coarser-grained partitioning schema, e.g., down to only 4×4 matrices (rather than 1×1 matrices, as in the case of a full partitioning). This allows us replacing scalar function f (which in the case of matrix multiplication is a simple scalar multiplication $*$) with the operation supported by the hardware extension, such as matrix multiplication on 4×4 matrices. We expect for our future work to achieve the same advantages over TensorIR as over TVM, because apart from supporting domain-specific hardware extensions, TensorIR is very similar to TVM.

PORTABILITY While scheduling approaches achieve high performance, they tend to struggle with achieving portability. This is because even though the approaches often offer different, pre-implemented backends (e.g., a CUDA backend to target NVIDIA GPUs and an OpenCL backend for CPUs), they do not propose any structured methodology about how new backends can be added, e.g., for potentially upcoming architectures, with potentially deeper memory and core hierarchies than GPUs and CPUs. This might be particularly critical (or requiring significant development effort) for the application area of deep learning which is the main target of many scheduling approaches, e.g., TVM and TensorIR, and for which new architectures are arising continuously [87].

In contrast, we introduce in this chapter a formally precise recipe for correct-by-construction code generation in different backends (including OpenMP, CUDA, and OpenCL), generically in the target architecture: we introduce an architecture-agnostic low-level representation (Section 4.3) as target for our high-level programs (Section 4.2), and we describe formally how our high-level programs are automatically lowered to our low-level representation (Section 4.4), based on the architecture-agnostic optimization space in Table 1. Our Appendix, Section .8, outlines how executable, imperative-style program code is straightforwardly generated from low-level expressions, which we plan to discuss and illustrate in detail in our future work.

PRODUCTIVITY Scheduling approaches rely on a two-step optimization process, as discussed above: implementing a schedule (first step) and choosing optimized values of performance-critical parameters within that schedule (second step). While the second step often can be easily automatized, e.g., via auto-tuning [119], the first step – implementing a schedule – usually has to be conducted manually by the user to achieve high performance, which requires expert knowledge and thus hinders productivity. The lack of formal foundation of many scheduling approaches further complicates implementing schedules for the user, as implementation becomes error prone and hardly predictable. For example, Fireiron's schedules can achieve high performance, close to GPUs' peak, but schedules in Fireiron can easily generate incorrect low-level code: Fireiron cannot guarantee that optimizations expressed in its scheduling language are semantics preserving, e.g., based on a formal foundation as done in this work, making programming Fireiron's schedules error prone and complex for the user. Similarly, TVM is sometimes unable to detect user errors in both its high-level language (as discussed in Section 4.5.1) as well as scheduling language [15]. Safety in parallel programming is an ongoing major demand, in particular from industry [29].

Auto schedulers, such as Halide's optimization engine [166] and TVM's recent Ansor [78], aim to automatically generate well-performing, correct schedules for the user. However, a major flaw of the current auto schedulers is that even though they work well for some computations (e.g., from deep learning, as TVM's Ansor), they may perform worse for others. For example, our approach achieves a speedup over TVM+Ansor of $> 100\times$ already for straightforward dot products (Figure 23). This is because Ansor does not exploit multiple thread blocks and uses only a small number of threads for reduction computations. While such optimization decisions are often beneficial for reductions as used in deep learning (e.g., within the computations of convolutions and matrix multiplications on deep learning workloads, because parallelization can be better exploited for outer loops of these computations), these rigid optimization decisions of Ansor may perform worse in other contexts (e.g., for computing dot product).

To avoid the productivity issues of scheduling approaches, we have designed our optimization process as fully auto-tunable, thereby freeing the user from the burden and complexity of making complex optimization decisions. Our optimization space (Table 1) is designed as generic in the target application area and hardware architecture, thereby achieving high performance for various combinations of data-parallel computations and architectures (Section 4.5). Correctness of optimizations is ensured in our approach by introducing a formal foundation that enables mathematical reasoning about correctness. Particularly, our optimization process is designed as *correct-by-construction*, meaning that any valid optimization decisions (i.e., a particular choice of tuning parameters in Table 1 that satisfy the constraints) leads to a correct expression in our low-level expression (as in Figure 21). In contrast, approaches such as introduced by Clément and Cohen [21] formally validate optimization decisions of scheduling approaches in already generated low-level code. Thereby, such approaches work potentially for arbitrary scheduling approaches (Halide, TVM, ...), but the approaches cannot save the user at the high abstraction level from implementing incorrect optimizations (e.g., via easy-to-understand, high-level error messages indicating that an invalid optimization decisions is made) or restricting the optimization space otherwise to valid decisions only, e.g., for an efficient auto-tuning process, because the approaches check already generated program code.

Scheduling approaches often also suffer from expressivity issues. For example, Fireiron is limited to computing only matrix multiplications on only NVIDIA GPUs, and TVM does not support computations that rely on multiple combine operators different from concatenation [12, 63], e.g., as required for expressing the *Maximum Bottom Box Sum* example in Figure 20. Also, TVM has difficulties with user-defined combine operators [14] and thus crashes for example *Probabilistic Record Linkage* in Figure 20. In contrast to TVM, we introduce a formal methodology about of how to manage different kinds of arbitrary, user-defined combine operators (Section 4.3), which is considered challenging [63].

4.6.2 *Polyhedral Approaches*

Polyhedral approaches, as introduced by Feautrier [309], as well as *Pluto* [262], *Polly* [224], *PPCG* [218], *Polyhedral Tensor Schedulers* [94], *TC* [110], and *AKG* [19] rely on a formal, geometrically inspired program representation, called *polyhedral model*. Polyhedral approaches often achieve high user productivity, e.g., by automatically parallelizing and optimizing straightforward sequential code. However, the approaches tend to have difficulties with achieving high performance and portability when used for generating low-level program code, as we outline in the following. In Section 4.6.5, we revisit the polyhedral approach as a potential frontend for our approach, as polyhedral transformations have proven to be efficient when used for high-level code optimizations (e.g., *loop skewing* [311]), rather than low-level code generation.

PERFORMANCE Polyhedral compilers tend to struggle with achieving their full performance potential. We argue that this performance issue of polyhedral compilers is mainly caused by the following two major reasons.

While we consider the set of polyhedral transformation (so-called *affine transformation*) as broad, expressive, and powerful, each polyhedral compiler implements a subset of expert-chosen transformations. This subset of transformations, as well as the application order of transformations, are usually fixed in a particular polyhedral compiler and chosen toward specific optimization goals only, e.g., coarse-grained parallelization and locality-aware data accesses (a.k.a. *Pluto algorithm* [261]), causing the search spaces of polyhedral compilers to be a proper subset of our space in Table 1. Consequently, computations that require for high performance other subsets of polyhedral transformations and/or application orders of transformations (e.g., transformations toward fine-grained parallelization) might not achieve their full performance potential when compiled with a particular polyhedral compiler [1].

In contrast to the currently existing polyhedral compilers, we have designed our optimization process as generic in goals: for example, our space is designed such that the degree of parallelization (coarse, fine, ...) is fully auto-tunable for the particular combination of target architecture and computation to optimize. We consider it as an interesting future work to investigate the strength and weaknesses of the polyhedral model for expressing our generic optimization space.

We see the second reason for potential performance issues in polyhedral compilers in their difficulties with reduction-like computations. This is mainly caused by the fact that the polyhedral model captures less semantic information than the high-level program representation introduced in Section 4.2: combine operators which are used to combine the intermediate results of computations (e.g., operator + from Example 2 for combining the intermediate results of the dot products within matrix multiplication) are not explicitly represented in the polyhedral model; the polyhedral model is rather focussed

on modeling memory accesses and their relative order only. Most likely, this semantic information is missing in the polyhedral model, because polyhedral approaches were originally intended to fully automatically optimize loop-based, sequential code (such as Pluto and PPCG) – extracting combine operators automatically from sequential code is challenging and often even impossible (Rice's theorem).

In contrast, our proposed high-level representation explicitly captures combine operators (Figure 20), by requesting these operators explicitly from the user. This is important, because the operators are often required for generating code that fully utilizes the highly parallel hardware of state-of-the-art architectures (GPUs, etc), as discussed in Section 4.5. Similarly to our approach, the polyhedral compiler TC also requests combine operators explicitly from the user. However, TC is restricted to operators + (addition), $*$ (multiplication), `min` (minimum), and `max` (maximum) only, thereby TC is not able to express important examples in Figure 20, e.g., PRL which is popular in data mining. Moreover, TC outsources the computation of its combine operators to the NVIDIA CUB library [35]; most likely as a workaround, because TC relies on the polyhedral model which is not designed to capture and exploit semantic information about combine operators for optimization. Thereby, TC is dependent on external approaches for computing combine operators, which might not always be available (e.g., for upcoming architectures).

Workarounds have been proposed by the polyhedral community to target reduction-like computations [168, 179]. However, these approaches are limited to a subset of computations, e.g., by not supporting user-defined scalar types [179] (as required for our PRL example in Figure 20), or by being limited to GPUs only [168]. Comparing the semantic information captured in the polyhedral model vs our MDH-based representation have been the focus of discussions between polyhedral experts and MDH developers [66].

PORTABILITY The polyhedral approach, in its general form, is a framework offering transformation rules (affine transformations), and each individual polyhedral compiler implements a set of such transformations which are then instantiated (e.g., with particular tile sizes) and applied when compiling a particular application. However, individual polyhedral compilers (e.g., PPCG and Pluto) apply a fixed set of affine transformations, thereby rigidly optimizing for a particular target architecture only, e.g., only GPU (as PPCG) or only CPU (as Pluto), and it remains open which affine transformations have to be used and how for other architectures, e.g., upcoming accelerators for deep learning computations [87] with potentially more complex memory and core hierarchies than GPUs and CPUs. Moreover, while we introduce an explicit low-level representation (Section 4.3), the polyhedral approach does not introduce representations on different abstraction levels: the model relies on one representation that is transformed via affine transformations. Apart from the ability of our low-level representation to handle combine operators (which we consider as complex and important), we see the advantages of our ex-

plicit low-level representation in, for example, explicitly representing memory regions, which allows formally defining important correctness constraints, e.g., that GPU architectures allow combining the results of threads in designated memory regions only. Furthermore, our low-level representation also allows straightforwardly generating executable code from it (shown in the Appendix, Section .8, and planned to be discussed thoroughly in future work). In contrast, code generation from the polyhedral model has proven challenging [19, 51, 181].

PRODUCTIVITY Most polyhedral compilers achieve high user productivity, by fully automatically parallelizing and optimizing straightforward sequential code (as Pluto and PPCG). Our approach currently relies on a DSL (Domain-Specific Language) for expressing computations, as discussed in Section 4.2; thus, our approach can be considered as less productive than many polyhedral compilers. However, Rasch, Schulze, and Gorlatch [81, 82] show that DSL programs in our approach can be automatically generated from sequential code (optionally annotated with simple, OpenMP-like directives for expressing combine operators, enabling advanced optimizations), by using polyhedral tool `pet` [236] as a frontend for our approach. Thereby, we are able to achieve the same, high user productivity as polyhedral compilers. We consider this direction – combing the polyhedral model with our approach – as promising, as it enables benefitting from the advantages of both directions: optimizing sequential programs and making them parallelizable using polyhedral techniques (like *loop skewing*, as also outlined in Section 4.6.5), and mapping the optimized and parallelizable code eventually to parallel architectures based on the concepts and methodologies introduced in this chapter.

4.6.3 *Functional Approaches*

Functional approaches map data-parallel computations that are expressed via small, formally defined building blocks (a.k.a. patterns [243], such as `map` and `reduce`) to the memory and core hierarchies of parallel architectures, based on a strong formal foundation. Notable functional approaches include Accelerate [238], Obsidian [246], so-called *skeleton libraries* [103, 122, 139, 141, 160, 245, 250], and the modern Lift approach [192] (recently also known as RISE [45]).

In the following, as functional approaches usually follow the same basic concepts and methodologies, we focus on comparing to Lift, because Lift is more recent than, e.g., Accelerate and Obsidian.

PERFORMANCE Functional approaches tend to struggle with achieving their full performance potential, often caused by the design of their optimization spaces. For example, analogously to our approach, functional approach Lift relies on an internal low-level representation [153] that is used as target for Lift's high-level programs. However, Lift's transformation process, from high level to low level, turned out to be challenging: Lift's lowering process relies on an infinitely large optimization space – identifying a well-performing configuration within that space is too complex to be done automatically in general, due to the space's large and complex structure. As a workaround, Lift currently uses approach Elevate [68] to incorporate user knowledge into the optimization process; however, at the cost of productivity, as manually expressing optimization is challenging, particularly for non-expert users.

In contrast, our optimization process is designed as auto-tunable (Table 1), thereby achieving fully automatically high performance, as confirmed in our experiments (Section 4.5), without involving the user for optimization decisions. In particular, Rasch, Schulze, and Gorlatch [115] show that our approach can significantly outperform Lift on GPU and CPU. Our performance advantage over Lift is mainly caused by the design of our optimization process: relying on formally defined tuning parameters (Table 1), rather than on formal transformation rules that span a too large and complex search space (as Lift), thereby contributing to a simpler, fully auto-tunable optimization process.

PORTABILITY The current functional approaches usually are designed and optimized toward code generation in a particular programming model only. For example, Lift inherently relies on the OpenCL programming model, because OpenCL works for multiple kinds of architectures: NVIDIA GPU, Intel CPU, etc. However, we see two major disadvantages in addressing the portability issue via OpenCL only: 1) GPU-specific optimizations (such as *shuffle operations* [129]) are available only in the CUDA programming model, but not in OpenCL; 2) the set of OpenCL-compatible devices is broad but still limited; in particular, in the *new golden age for computer ar-*

chitectures [87], upcoming architectures are arising continuously and may not support the OpenCL standard. We consider targeting new programming models as challenging for Lift, as its formal low-level representation is inherently designed for OpenCL [153]; targeting further programming models with Lift would require the design and implementation of new low-level representations, which we do not consider as straightforward.

To allow easily targeting new programming models with our approach, we have designed our formalism as generic in the target model: our low-level representation (Figure 21) and optimization space (Table 1) are designed and optimized toward an *Abstract System Model* (Definition 10) which is capable of representing the device models of important programming approaches, including OpenMP, CUDA, and OpenCL (Example 11). Furthermore, we have designed our high- and low-level representations as minimalistic (Figures 12 and 21), e.g., by relying on three higher-order functions only for expressing programs at the high abstraction level, which simplifies and reduces the development effort for implementing code generators for programming models.

In addition, we believe that compared to our approach, the following basic design decisions of Lift (and similar functional approaches) complicate the process of code generation for them and increase the development effort for implementing code generators: 1) relying on a vast set of small patterns for expressing computations, rather than aiming at a minimalistic design as we do (as also discussed in Section 4.5.3); 2) relying on complex function nestings and compositions for expressing computations, rather than avoiding nesting and relying on a fixed composition order of functions, as in our approach (Figure 11); 3) requiring new patterns for targeting new classes of data-parallel computations (such as patterns `slide` and `pad` for stencils [124]), which have to be non-trivially integrated into Lift's type and optimization system (often via extensions of the systems [124, 169]), instead of relying on a fixed set of expressive patterns (Figure 12) and generalized optimizations (Table 1) that work for various kinds of data-parallel computations (Figure 20); 4) expressing high-level and low-level concepts in the same language, instead of separating high-level and low-level concepts for a more structured and thus simpler code generation process (Figure 10). We consider these four design decisions as disadvantageous for code generation, because they require from a code generator handling various kinds of patterns (decision 1), and the patterns need to be translated to significantly different code variants, depending on their nesting level and composition order (decision 2). Moreover, each extension of patterns (decision 3) might affect code generation also for the already supported patterns which need to be be composable and nestable with the new ones (decision 2). We consider mixing up high-level and low-level concepts in the same language (decision 4) as further complicating the code generation process, because code generators cannot be implemented in clear, distinct stages: *high-level language → low-level language → executable program code*.

PRODUCTIVITY Functional approaches are expressive frameworks – to the best of our knowledge, the majority of these approaches should also be able to express (possibly after some extension) many of the high-level programs that can also be expressed via our high-level representation (e.g., those presented in Figure 20).

A main difference we see between the high-level representations of existing functional approaches and the representation introduced by our approach is that the existing approaches rely on a vast set of higher-order functions for expressing computations; these functions have to be functionally composed and nested in complex ways for expressing computations. For example, expressing matrix multiplication in Lift requires also involving Lift's pattern `transpose` (also when operating on non-transposed input matrices) [169], as per design in Lift, multi-dimensional data are considered as an array of arrays (rather than a multi-dimensional array, as in our approach as well as polyhedral approaches). In contrast, we aim to keep our high-level language minimalistic, by expressing data-parallel computations using exactly three higher-order functions and which are always used in the same, fixed order (shown in Figure 11). Rasch, Schulze, and Gorlatch [81, 82] confirm that due to the minimalistic and structured design of our high-level representation, programs in our representation can even be systematically generated from straightforward, sequential program code.

Functional approaches also tend to require extension when targeting new application areas, which hinders the expressivity of the frameworks and thus also their productivity. For example, functional approach Lift [192] required notable extension for targeting, e.g., matrix multiplications (so-called *macro-rules* had to be added to Lift [169]) and stencil computations (primitives `slide` and `pad` were added, and Lift's tiling optimization had to be extended toward *overlapped tiling* [124]). In contrast, we have formally defined our class of targeted computations (as MDH functions, Definition 3), and the generality of our approach allows expressing matrix multiplications and stencils out of the box, without relying on domain-specific building blocks.

4.6.4 *Domain-Specific Approaches*

Many approaches focus on code generation and optimization for particular domains. A popular domain-specific approach is *ATLAS* [297] for linear algebra routines on CPUs[30]. Similar to ATLAS, approach *FFTW* [293] targets *Fast Fourier Transform (FFT)*, and *SPIRAL* [277] works for *Digital Signal Processing (DSP)*.

[30] Section 5.7.5 shows that MDH achieves higher performance than ATLAS.

Nowadays, the best performing, state-of-practice domain-specific approaches are often provided by vendors and specifically designed and optimized toward their target application domain and also architecture. For example, the popular vendor library NVIDIA cuBLAS [43] is optimized by hand, on the assembly level, toward computing linear algebra routines on NVIDIA GPUs – cuBLAS is considered in the community as gold standard for computing linear algebra routines on GPUs. Similarly, Intel's oneMKL library [28] computes with high performance linear algebra routines on Intel CPUs, and libraries NVIDIA cuDNN [36] and Intel oneDNN [27] work well for convolution computations on either NVIDIA GPU (cuDNN) or Intel CPU (oneDNN), respectively.

In the following, we discuss domain-specific approaches in terms of *performance*, *portability*, and *productivity*.

PERFORMANCE Domain-specific approaches, such as cuBLAS and cuDNN, usually achieve high performance. This is because the approaches are hand-optimized by performance experts – on the assembly level – to exploit the full performance potential of their target architecture. In our experiments (Section 4.5), we show that our approach often achieves competitive and sometimes even better performance than domain-specific approaches provided by NVIDIA and Intel, which is mainly caused by their portability issues across different data characteristics, as we discuss in the next paragraph.

PORTABILITY Domain-specific approaches usually struggle with achieving portability across different architectures. This is because the approaches are often implemented in architecture-specific assembly code to achieve high performance, but thereby also being limited to their target architecture. The domain-specific approaches often also struggle with achieving performance portability across different data characteristics (e.g., their sizes): the approaches usually rely on a set of pre-implemented implementations that are each designed and optimized toward average high performance across a range of data characteristic. In contrast, our approach (as well as many scheduling and polyhedral approaches) allow automatically optimizing (auto-tuning) computations for particular data characteristics, which is important for achieving high performance [155]. Thereby, our approach often outperforms domain-specific approaches (as confirmed in Section 4.5), particularly for advanced data characteristics (small, uneven, irregularly shaped, ...), e.g., as used in deep learning. The costly time for auto-tuning is well amortized in many application areas, because the auto-tuned implementations are re-used in many program runs. Furthermore, auto-tuning avoids the time-intensive and costly process of hand-optimization by human experts.

PRODUCTIVITY Domain-specific approaches usually achieve highest productivity for their target domain (e.g., linear algebra), by providing easy to use high-level abstractions. However, the approaches suffer from significant expressivity issues, because – per design – they are inherently restricted to their target application domain only. Also, the approaches are often inherently bound to only particular architectures, e.g., only GPU (as NVIDIA cuBLAS and cuDNN) or only CPU (as Intel oneMKL and oneDNN). Domain-specific vendor libraries, such as NVIDIA cuBLAS and Intel oneMKL, also tend to offer the user differently performing variants of computations; the variants have to be naively tested by the user when striving for the full performance potentials of approaches (as discussed in Section 4.5.4), which is cumbersome for the user.

4.6.5 *Higher-Level Approaches*

There is a broad range of existing work that is focused on higher-level optimizations than proposed by this work. We consider such higher-level approaches as greatly combinable with our approach. For example, the polyhedral approach is capable of expressing algorithmic-level optimizations, such as *loop skewing* [311], to make programs parallelizable; such optimizations are beyond the scope of this work, but they can be combined with our approach as demonstrated by Rasch, Schulze, and Gorlatch [81, 82]. Similarly, we consider the approaches introduced by Yang, Atkinson, and Carbin [60], Farzan and Nicolet [86], Gunnels et al. [283], and Frigo et al. [288], which also focus on algorithmic-level optimizations, as greatly combinable with our approach: algorithmically optimizing user code according to the approaches' techniques, and using our methodologies to eventually map the optimized code to executable program code for parallel architectures.

Futhark [143], Dex [58], and ATL [32] are further approaches focussed on high-level program transformations, such as advanced *flattening* mechanisms [88], thereby optimizing programs at the algorithmic level of abstraction. We consider using our work as backend for these approaches as promising: the three approaches often struggle with mapping their algorithmically optimized program variants eventually to the multi-layered memory and core hierarchies of state-of-the-art parallel architectures, which is exactly the focus of this work.

4.7 SUMMARY

We introduce a formal (de/re)-composition approach for data-parallel computations targeting state-of-the-art parallel architectures, based on our novel algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)*. Our approach aims to combine three major advantages over related approaches – performance, portability, and productivity – by introducing formal program representations on both: 1) *high level*, for conveniently expressing – in one uniform formalism – various kinds of data-parallel computations (including linear algebra routines, stencil computations, data mining algorithms, and quantum chemistry computations), agnostic of hardware and optimization details, while still capturing all information relevant for generating high-performance program code; 2) *low level*, which allows uniformly reasoning – in the same formalism – about optimized (de/re)-compositions of data-parallel computations targeting different kinds of parallel architectures (GPUs, CPUs, etc). We *lower* our high-level representation to our low-level representation, in a formally sound manner, by introducing a generic search space that is based on performance-critical parameters. The parameters of our lowering process enable fully automatically optimizing (auto-tuning) our low-level representations for a particular target architecture and characteristics of the input and output data, and our low-level representation is designed such that it can be straightforwardly transformed to executable program code in imperative-style programming languages (including OpenMP, CUDA, and OpenCL). Our experiments confirm that due to the design and structure of our generic search space in combination with auto-tuning, our approach achieves higher performance on GPUs and CPUs than popular state-of-practice approaches, including hand-optimized libraries provided by vendors.

5 CODE OPTIMIZATION VIA AUTO-TUNING

Auto-tuning is a popular approach to program optimization (e.g., programs as generated by the MDH approach introduced in Chapter 4): it automatically finds good configurations of a program's so-called tuning parameters whose values are crucial for achieving high performance for a particular parallel architecture and characteristics of input/output data. In this chapter, we present the *Auto-Tuning Framework (ATF)*[1], which enables a key advantage in *general-purpose auto-tuning*: efficiently optimizing programs whose tuning parameters have *interdependencies* among them, e.g., the value of one parameter must be divisible by the value of another parameter.

We make the following contributions to the three main phases of general-purpose auto-tuning: 1) ATF *generates* the search space of interdependent tuning parameters with high performance, by efficiently exploiting our novel *parameter constraints*; 2) ATF *stores* such search spaces efficiently in memory, based on a novel *Chain-of-Trees (CoT)* search space structure; 3) ATF *explores* these search spaces faster, by introducing a multi-dimensional search strategy on its CoT search space representation.

Our experiments confirm that, compared to the existing, general-purpose auto-tuning frameworks, ATF substantially improves generating, storing, and exploring the search space of interdependent tuning parameters, thereby enabling an efficient overall auto-tuning process for important applications from popular domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

The concepts introduced in this chapter have already been adapted in further auto-tuning approaches, including [2, 341].

5.1 INTRODUCTION

High performance for parallel programs is difficult to achieve, because program code has to be optimized for different target architectures and for changing input/output characteristics (size, dimensionality, transposition layout, etc.) [156]. Typically, there are many parameters that influence a program's performance in different ways, such that finding the optimal parameter configuration is often a hardly manageable task even for experts.

[1] https://atf-tuner.org

Auto-tuning is a technique for automatically finding good values of a program's performance-critical parameters (a.k.a. *tuning parameters*), for example the number of threads and/or sizes of tiles, for a given architecture and input/output characteristics. Usually, the auto-tuning process consists of three major phases: the *generation*, *storing*, and *exploration* of the program-specific *search space* which consists of all possible parameter configurations.

There have been several successful *special-purpose auto-tuning* approaches, with an overview in [117]. They achieve impressive results for particular application classes on particular target architectures, by taking advantage of domain-specific knowledge to efficiently generate, store, and explore the program-specific search space. Notable examples of special-purpose auto-tuners are ATLAS [296] and PATUS [239], where auto-tuning is used for optimizing linear algebra routines on CPU architectures or for high-performance stencil computations on CPUs and GPUs, respectively.

Unfortunately, the implementation of a special-purpose auto-tuner is a cumbersome task. The developer has to manually manage the generation and storing of the parameter configurations, and tailor a search technique (like genetic algorithms or simulated annealing [290]) to the parameters' search space for its automatic exploration. Special-purpose auto-tuning becomes especially challenging when parameters have *interdependencies* among them, e.g., when the value of one parameter must be divisible by the value of another parameter. Such divisibility properties are often required for tuning parameters of recent parallel applications; for example, in order to correctly exploit the thread and memory hierarchies of modern architectures, as we discuss later in this chapter. Designing a special-purpose auto-tuner for such interdependent parameters requires expert knowledge and a significant implementation effort from the auto-tuning developer. We demonstrate that generating and storing the search spaces of interdependent tuning parameters is time-consuming and memory-intensive, and that the structure of such spaces significantly impacts the exploration efficiency of state-of-the-art search techniques. The demand for higher productivity in auto-tuning has been identified as a major research challenge in high-performance computing [117, 207].

Our work is inspired by the alternative approach, *general-purpose auto-tuning*, with the classic approaches Orio [257] and Active-Harmony [259], followed by the current state-of-the-art frameworks OpenTuner [195] and CLTune [190], and the most recent `libtuning` [100] and KernelTuner [112] approaches. General-purpose auto-tuning aims at simplifying the auto-tuning process: the software developer specifies program's tuning parameters by their names and ranges of possible values, and the general-purpose framework then automatically creates the corresponding special-purpose auto-tuner that generates, stores, and explores the program-specific search space.

The existing general-purpose auto-tuning approaches are efficient for many applications on a range of architectures; however, we demonstrate in this chapter that they still struggle with programs whose tuning parameters have interdependencies among them: the existing approaches either keep invalid configurations within their search spaces, which hinders search techniques from finding well-performing configurations (like Orio, OpenTuner, and `libtuning`), or the approaches have difficulties with efficiently generating, storing, and exploring the search spaces of only valid configurations for recent parallel applications (like ActiveHarmony, CLTune, and KernelTuner).

We introduce the *Auto-Tuning Framework (ATF)* to address the discussed weaknesses in state-of-the-art general-purpose auto-tuning for programs with interdependent tuning parameters. ATF makes new contributions in general-purpose auto-tuning to each particular phase of the auto-tuning process:

1. ATF *generates* the search space of interdependent parameters with higher performance than the current general-purpose auto-tuners by efficiently exploiting *parameter constraints*;

2. ATF *stores* the generated search space of such parameters more efficiently in memory by relying on a novel *Chain-of-Trees (CoT)* search space structure for representing these spaces;

3. ATF *explores* the generated and stored space of interdependent parameters faster by employing a multi-dimensional search strategy on its CoT search space representation.

Our experiments confirm that ATF substantially improves the state of the art in general-purpose auto-tuning, based on four popular application case studies: 1) stencil computation *Gaussian Convolution*, 2) linear algebra routine *General Matrix-Matrix Multiplication*, 3) quantum chemistry computation *Coupled Cluster*, and 4) data mining algorithm *Probabilistic Record Linkage*.

Furthermore, ATF introduces a novel programming interface for auto-tuning – based on a *Domain-Specific Language (DSL)* for conveniently auto-tuning at compile time and also interfaces implemented in *General-Purpose Languages (GPL)* (e.g., C++ and Python) for auto-tuning at runtime – thereby making auto-tuning also appealing to common application developers. We argue in this chapter that ATF's interfaces are arguably simpler to use than interfaces of the existing, state-of-the-art general-purpose auto-tuning systems.

The rest of this chapter is structured as follows. Section 5.2 recapitulates the state of the art in general-purpose auto-tuning. Sections 5.3, 5.4, and 5.5 introduce ATF's novel mechanisms for generating, storing, and exploring the search spaces of interdependent tuning parameters, and Section 5.6 introduces the ATF's user interface and compares it to interfaces of popular, state-of-the-art auto-tuning systems that are heavily used in practice. Our experimental evaluation is described in Section 5.7. We discuss related work in Section 5.8, and we conclude in Section 5.9.

5.2 GENERAL-PURPOSE AUTO-TUNING APPROACHES

We briefly recapitulate the state of the art in general-purpose auto-tuning: OpenTuner and `libtuning` in Section 5.2.1, and CLTune and KernelTuner in Section 5.2.2. classic approaches ActiveHarmony and Orio are discussed in Section 5.8.

5.2.1 *Auto-Tuners Designed Toward Independent Tuning Parameters*

OpenTuner and `libtuning` are auto-tuning frameworks designed and optimized toward applications, whose tuning parameters have no interdependencies among them. This restriction enables both approaches to rely on only straightforward mechanisms for search space generation, storing, and exploration: the user specifies tuning parameters by their name and range of possible values, and, per design, each possible combination of parameters' values is considered by the auto-tuner as a valid configuration within the search space. Therefore, explicitly generating and storing the entire search space is not required in these approaches, because configurations can be generated straightforwardly, on the fly, by arbitrarily combining parameters' values, and storing these spaces requires only storing parameters' ranges which have a small memory footprint. Furthermore, search techniques can be easily used for exploring these systems' search spaces: the spaces have rectangular shapes where each dimension of the space represents the range of a particular tuning parameter; this enables straightforwardly mapping the spaces to a *coordinate space* – a collection of equally-sized sequences of real numbers – for which numerical search techniques are specifically designed and optimized [290].

While auto-tuning systems for independent parameters work well for many applications, they struggle with programs whose tuning parameters have interdependencies among them. This is because arbitrarily combining the values of interdependent parameters leads to invalid parameter configurations which cannot be distinguished in OpenTuner and `libtuning` due to their inherent design – the user has to manually set a penalty value for invalid configurations, as a workaround [132]. Consequently, the tuners' spaces contain also invalid configurations which is often inefficient: we demonstrate later in this chapter for important parallel applications that often $> 99.999\%$ of configurations within their search spaces are invalid due to parameters' interdependencies, which hinders search techniques from finding well-performing configurations.

5.2.2 *Auto-Tuners Designed Toward Interdependent Tuning Parameters*

CLTune and KernelTuner are popular auto-tuners that take interdependencies among tuning parameters into account. For this, the frameworks generate, store, and explore *constrained search spaces* which contain valid configurations only such that search techniques do not have to struggle with invalid configurations. For example, we show in Section 5.7 that using the constrained search space, it is possible to find well-performing configurations for important applications (e.g., stencil computations and linear algebra routines) in reasonable 4h of tuning time, in which search techniques explore up to 20,000 valid configurations in the constrained space. In contrast, when relying on the *unconstrained search space* which contains also invalid configurations (as in OpenTuner and `libtuning`), it is not possible to even find a valid starting point within the space – independent of the chosen search technique – in 4h exploration time (in which, e.g., OpenTuner tested up to 190,000 configurations). This is because the unconstrained space contains a vast amount of invalid configurations.

CLTune and KernelTuner are efficient for many applications. However, both approaches rely on the same, straightforward processes to generating, storing, and exploring constrained search spaces, which are inefficient for the large spaces of recent parallel applications: the two approaches rely on a naive space representation which straightforwardly enumerates all valid configurations within a one-dimensional array. We demonstrate that such a representation causes large memory footprint and hinders the efficiency of state-of-the-art search techniques, because exploration can be performed in only one dimension. Furthermore, the two approaches generate these spaces based on a so-called *search space constraint* which has to be checked also for all possible configurations – valid and also invalid. This is inefficient when the number of invalid configurations is large, thereby severely hindering the applicability of both approaches.

5.3 GENERATING CONSTRAINED SEARCH SPACES

We address the first phase of general-purpose auto-tuning by introducing a novel generation algorithm for constrained search spaces. Our generation algorithm is based on *parameter constraints*, which we introduce in Section 5.3.1. Afterward, we show in Section 5.3.2 how we efficiently exploit our constraint design for fast search space generation.

5.3.1 *Parameter Constraints*

A key concept in auto-tuning is a *tuning parameter*. Typically, a tuning parameter p_i is represented by a pair containing parameter's name and a range which specifies the parameter's possible values:

$$p_i := (\ \langle \texttt{name} \rangle, \langle \texttt{range} \rangle\ )$$

We extend this traditional definition of a tuning parameter by adding to it a *parameter constraint*:

$$p_i := (\ \langle \texttt{name} \rangle, \langle \texttt{range} \rangle, \langle \texttt{constraint} \rangle\ )$$

A parameter constraint may be any arbitrary, unary, boolean function that takes as input an element of its parameter's range; values for which the function returns `false` are filtered out of the range. In our implementation of ATF (focus of Section 5.6), we rely on the syntax of C++ for expressing constraint functions.

Parameter constraints enable expressing arbitrary interdependencies among tuning parameters and consequently to avoid invalid configurations within the search space. For this, the constraint function of a parameter p_i may use in its definition all previously defined tuning parameters p_j, $j < i$, as common variables that have the same type as their corresponding range values, e.g., type `int` in case of a tuning parameter p_j whose range consists of integers.

For example, in OpenCL, the number of threads in a group (a.k.a. *local size* in OpenCL terminology) has to divide the overall number of threads (*global size*), and the global size usually has to be smaller than or equal to the input size N to avoid idling threads. To express this, we use the following boolean unnamed functions (in C++ syntax) as constraints for the global and local size tuning parameters, where `%` denotes the modulo operator:

```
// global size parameter constraint
( int global_size ){ return global_size <= N; }
```

```
// local size parameter constraint
( int local_size ){ return global_size % local_size == 0; }
```

The constraint function of the local size parameter uses the global size parameter `global_size` in its body, thereby expressing the interdependency among these two parameters. We discuss the definition and usage of parameter constraints in ATF's user interface in more detail in Section 5.6.

For comparison, a *search space constraint* in CLTune and KernelTuner that is equivalent to the ATF's two parameter constraints above is [112, 190]:

```
( auto c ){ return c.local_size * c.k <= N; }
```

A search space constraint has to be defined as a single function (with drawbacks discussed in the next subsection). In this example, the constraint takes as input a configuration `c` comprising tuning parameters `local_size` and `k`; the `global_size` is then computed as `local_size * k`.

Note that ATF's parameter constraints are as expressive as the traditional search space constraints: a search space constraint that is equivalent to a set of parameter constraints can always be generated by combining the parameter constraints via logical `and`. Moreover, we show in Sections 5.6.2 and 5.6.3 that ATF's user interface (based on parameter constraints) provides a better user experience as compared to the interfaces of CLTune (which relies on search space constraints) and OpenTuner (which supports no constraints at all).

In the following, we show how we exploit ATF's constraint design for fast search space generation.

5.3.2 *Algorithm for Generating Constrained Search Spaces*

We first briefly recapitulate the traditional generation algorithm for constrained search spaces, as used in CLTune and KernelTuner, which is based on a search space constraint. Afterward, we introduce our novel algorithm for generating constrained search spaces, which is based on ATF's parameter constraints.

TRADITIONAL APPROACH Listing 21 shows as pseudocode the original search space generation algorithm of CLTune and KernelTuner (taken from [190] and [112]), which is based on a search space constraint and the traditional definition of tuning parameters.

```
1  // iteration over tuning parameter configurations
2  for ( v1 : r1 )
3    ⋱
4      for ( vk : rk )
5
6        // checking search space constraint
7        if ( sc(v1,...,vk) )
8
9          // adding configuration to the search space
10         add_config( v1,...,vk );
```

Listing 21: Traditional algorithm (pseudocode) for generating constrained search spaces [112, 190].

Configurations are added to the search space (line 10 in Listing 21) if the search space constraint sc (line 7) is satisfied. In the listing, we use the C++ syntax for range-based `for`-loops, where r_i denotes the range of the i-th tuning parameter (lines 2-4). A major drawback of the traditional approach is that the search space constraint (line 7) has to be checked at the deepest level of the loop nest, causing high search space generation time.

NOVEL APPROACH Listing 22 shows as pseudocode our optimized algorithm for generating constrained search spaces, which relies on parameter constraints, rather than on search space constraints as in the traditional approach. Compared to Listing 21, our algorithm in Listing 22 exploits ATF's constraint design for two major optimizations: 1) generating independently and also in parallel the search space parts of tuning parameters that can be grouped together according to their interdependencies, and 2) checking constraints early in the loop nest. We discuss both optimizations in the following.

```
1  // processing in parallel groups of interdependent parameters
2  parallel for ( G : {G_1,...,G_n} )
3
4    // processing an individual group also in parallel
5    parallel for ( v_1^G : r_1^G )
6      if ( pc_1^G⟨⟩(v_1^G) )
7        ⋱
8          parallel for ( v_tG^G : r_tG^G )
9            if ( pc_tG^G⟨n_1^G,...,n_tG-1^G⟩(v_tG^G) )
10
11             // sequential computation of a group
12             for ( v_tG+1^G : r_tG+1^G )
13               if ( pc_tG+1^G⟨n_1^G,...,n_tG^G⟩(v_tG+1^G) )
14                 ⋱
15                   for ( v_kG^G : r_kG^G )
16                     if ( pc_kG^G⟨n_1^G,...,n_kG-1^G⟩(v_kG^G) )
17
18                       // adding configuration to the search space
19                       search_space.group(G).add_par( v_1^G,...,v_tG^G ).
                             add_seq( v_tG+1^G,...,v_kG^G );
```

Listing 22: Novel algorithm (pseudocode) for generating constrained search spaces.

Optimization 1: In general, not all tuning parameters depend on each other. For example, in recent GPU and CPU implementations, e.g., for stencil computations and linear algebra routines, up to 39 parameters are used (as we discuss in Section 5.7); we partition these parameters into differently-sized groups of interdependent tuning parameters; for example, 6 groups in the case of stencil computations, which contain up to 4 parameters per group. We exploit ATF's constraint design to automatically identify the interdependent parameter groups: two parameters are interdependent and thus in the same group, iff one of them occurs in the syntax tree of the other parameter's constraint function. In the following, let $G_1,\ldots,G_n$ be the disjoint grouping of interdependent parameters, where each group $G \in \{G_1,\ldots,G_n\}$ contains k^G-many tuning parameters, i.e.: $G = \{p_1^G,\ldots,p_{k^G}^G\}$, and the i-th parameter within group G is defined (according to our definition of tuning parameters in Section 5.3.1) as: $p_i^G = (n_i^G , r_i^G , pc_i^G\langle n_1^G,\ldots,n_{i-1}^G\rangle)$, $1 \le i \le k^G$, where n_i^G is the parameter's name, r_i^G its range, and pc_i^G its constraint. We denote by $\langle n_1^G,\ldots,n_{i-1}^G\rangle$ that constraint pc_i^G may depend on all previously defined tuning parameter (as discussed in Section 5.3.1).

In Listing 22, for each group G, we generate its corresponding part of the search space independently of the other group's parts (line 2). This breaks the deep loop nests in Listing 21, leading to significantly faster space generation, as we confirm later in our experiments. Moreover, as the groups' search space parts can be generated independently of each other, we can generate them in parallel (indicated by keyword `parallel` in line 2). We use a further potential of parallelism in Listing 22 by generating a group G's first t^G `for`-loops, $t^G \in \{1, \ldots, k^G\}$, also in parallel (line 5-8 in Listing 22); here, t^G denotes an arbitrary, user-defined constant between 1 and k^G.

Usually, we set t^G to a default value of $t^G = 1$, i.e., we parallelize only the first loop of the nest (line 5), because, in most cases, this is sufficient for high parallelization: first parameter's range usually comprises more values than cores are available in the target system's CPU. However, in special cases, if the first tuning parameter has a small range (e.g., a boolean parameter), we set t^G to a higher value, and consequently parallelize more loops in the nest, in order to fully utilize the available hardware. To avoid a parallelization overhead which might be high even for $t^G = 1$ if the first parameter's range is large, our parallel implementation uses a thread pool comprising as many threads as cores are available in the target CPU.

Synchronization is not required in our parallel algorithm, because the subspaces of different groups (accessed via function `group` in line 19 of Listing 22) and the subspaces of a group G's first t^G parameters (added via `add_par` in line 19 of Listing 22) are disjoint.

Note that in general, groups of interdependent parameters cannot be identified automatically when using a traditional search space constraint as in CLTune and KernelTuner: by design, parameters' interdependencies are defined in a single search space constraint only, thus requiring a complicated semantic analysis of the constraint that in general cannot be automated (Rice's theorem [274]).

Optimization 2: In the traditional algorithm, constraints must be checked at the deepest level of the loop nest, because the algorithm relies on a search space constraint which checks full configurations (line 7 in Listing 21). In contrast, ATF's constraint design enables checking constraints early in the loop nest (Listing 22, lines 6-16), thereby avoiding iterations over entire subspaces, which often substantially accelerates search space generation, as confirmed by our experiments described in Section 5.7.

5.4 STORING CONSTRAINED SEARCH SPACES

In this section, we address the second phase of auto-tuning: storing the space of configurations, which is generated according to the first phase described in Section 5.3. If tuning parameters have no interdependencies among them, as assumed by OpenTuner and `libtuning`, then their space can be represented straightforwardly using only the ranges of the tuning parameters (as discussed in Section 5.2.1). This is because each combination of values in parameters' ranges represents a valid configuration. In contrast, representing the constrained search space of parameters with interdependencies among them is significantly more complex, because not all configurations of parameter values are valid.

CLTune and KernelTuner generate and store in memory a priori the entire constrained search space. This is important, because generating and storing the entire space allows search techniques to freely navigate over the space, as required by the techniques for high search efficiency [290]. However, both approaches store the space in a plain array of configurations, which wastes significant amount of memory space, because many configurations share the same parameter values.

To efficiently store constrained search spaces in memory, we introduce the novel *Chain-of-Trees (CoT)* search space structure. This structure chains multiple trees where each node in the trees represents a particular parameter value, and each tree represents the search space part of an interdependent tuning parameter group (defined in the previous section).

We explain our CoT search space structure using a simple, illustrative example of five tuning parameters:

$$
\begin{aligned}
p_1 &:= (\ n_1, && \{22, 35\}, && \_ &&) \\
p_2 &:= (\ n_2, && \{2, 5, 7, 11\}, && \texttt{divides}(n_1) &&) \\
p_3 &:= (\ n_3, && \{26, 51\}, && \_ &&) \\
p_4 &:= (\ n_4, && \{1, 3, 13, 17\}, && \texttt{divides}(n_3) &&) \\
p_5 &:= (\ n_5, && \{27, 39, 52, 54, 68\}, && \texttt{equals}(n_3 + n_4) &&)
\end{aligned}
$$

For an easy distinction, the parameters' ranges comprise different values, and p_1 and p_3 have no constraints. We use `divides(N)` as an alias for parameter constraint `(int i){return N % i == 0;}`, and we use `equals(N)` for the constraint `(int i){return N == i;}`. There are two groups of interdependent parameters in this example: the first group contains parameters $\{p_1, p_2\}$, while parameters $\{p_3, p_4, p_5\}$ form the second group.

Figure 35 illustrates our CoT search space structure for the example parameters $p_1, ..., p_5$. For each of the two parameter groups, we use a tree (`Tree 1` and `Tree 2`) to represent its part of the search space, and we chain these two trees by connecting the leaves of the first tree with the root of the second tree. To save memory, we store the connecting (dashed) edges as a single reference in `Tree 1`. Each path in `Tree 1` from the root to a leaf represents a valid configuration of parameters p_1 and p_2, for which their constraints are satisfied, and each path in `Tree 2` represents a valid configuration of parameters p_3, p_4, p_5. Consequently, each combination of a path in `Tree 1` and a path in `Tree 2` represents a valid, full configuration of parameters.

In our CoT structure, parameter values are often stored only once, whereas in a plain array of configurations (as in CLTune and KernelTuner), these values would be repeated many times. For example, we store values 22 and 35 only once at the top level of `Tree 1` while these would be stored 20 times in the traditional space representation (once per configuration in the search space), resulting in a high memory footprint. Furthermore, the configurations of parameters p_3, p_4, p_5, which are represented by `Tree 2`, would have to be stored for every leaf of `Tree 1` (4 times in this example). We avoid this significant waste of memory by storing `Tree 2` only once and chaining the two trees together.

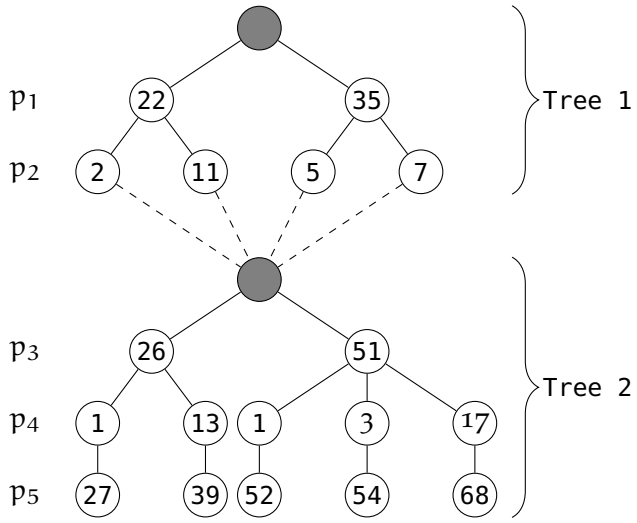


Figure 35: The example Chain-of-Trees (CoT) represents the search space of parameters $p_1, \ldots, p_5$.

Note that our CoT structure is efficient also for storing the spaces of parameters without interdependencies: if tuning parameters are independent, then each single parameter represents an own interdependent parameter group (comprising only one parameter), which corresponds to exactly the same range-based search space representation as used in OpenTuner and `libtuning`.

5.5 EXPLORING CONSTRAINED SEARCH SPACES

The third and final phase of auto-tuning is the exploration of the search space (generated and stored as described in Section 5.3 and 5.4) using some search technique. State-of-the-art general-purpose auto-tuning frameworks follow one of two basic approaches to the exploration phase, as described in the following two paragraphs.

CLTune and KernelTuner explore constrained search spaces, but they use a plain array of configurations. Consequently, they provide to the search techniques an only one-dimensional view on the search space, which often causes sub-optimal auto-tuning results, because locality information in the space's particular dimensions is lost [290]. For example, we demonstrate in Section 5.7 for the search by simulated annealing – CLTune's most efficient search technique [190] – that the selection time of the next candidate point is very high for large search spaces when relying on the one-dimensional space representation, leading to poor auto-tuning results.

OpenTuner and `libtuning` retain the multidimensionality of their search spaces, as required by search techniques for high search efficiency [290]. However, these two frameworks have to explore unconstrained search spaces which may contain also invalid configurations. This usually drastically worsens their efficiency for programs with interdependent tuning parameters, due to the often high amount of invalid configurations, as we confirm experimentally in Section 5.7.

We aim at combining the advantages of both state-of-the-art approaches: we explore constrained search spaces (as in CLTune and KernelTuner), and we search in multiple dimensions (as OpenTuner and `libtuning`). For this, we exploit the structure of our CoT search space representation (introduced in Section 5.4).

As search techniques usually explore coordinate spaces (i.e., spaces containing equally-sized sequences of real numbers that have no interdependencies among them), our basic idea for exploration is as follows: we map a coordinate space to our CoT representation; thereby, we reduce the challenge of exploring a CoT to the challenge of exploring a coordinate space – the most efficient structure for search techniques [290]. For a CoT with L-many levels (excluding the roots), we map to it a coordinate space of L dimensions. For each dimension in the coordinate space, we use real numbers in the interval $(0, 1]_{\mathbb{R}}$ – all points from 0 to 1, excluding 0. We denote L-dimensional coordinate spaces as $(0, 1]_{\mathbb{R}}^{L}$.

Figure 36 demonstrates an example of how we map a coordinate space to our CoT structure. For illustration, we use a CoT with 4 levels, i.e., $L = 4$ (roots excluded); correspondingly, we use a 4-dimensional coordinate space for mapping it to the 4-leveled CoT. The goal of our mapping is to assign to each arbitrary sequence in the coordinate space $(l_1, \ldots, l_4) \in (0, 1]_{\mathbb{R}}^{4}$ a path in the CoT (and thus a configuration – see Section 5.4), which we do intuitively as follows. In level 1, the CoT has 4 nodes, so in the first dimension of the coordinate space, we split interval $(0, 1]_{\mathbb{R}}$ evenly in 4 equally-sized blocks, where each block corresponds to one node in the CoT's first level.

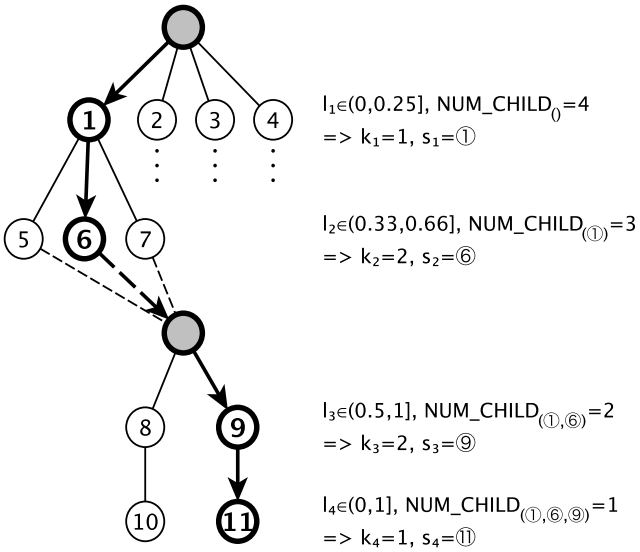


Figure 36: Example of exploring our Chain-of-Trees (CoT) structure in multiple dimensions, based on an L-dimensional coordinate space (in this example: L = 4). Subtrees in level 1 – for nodes 2, 3, 4 – are omitted for brevity.

In our example, we map each l_1 in block $(0, 0.25]$ to the root's first child ①, and if l_1 is in block $(0.25, 0.5]$, we map it to root's second child ②, etc. In level 2, after moving along the path (s_1) which comprises only node s_1 = ①, we have 3 nodes, so each $l_2 \in (0, 0.33]$ is mapped to s_1's first child node ⑤, and each $l_2 \in (0.33, 0.66]$ is mapped to the second child node ⑥, and so on.

In general, for an arbitrary L-leveled CoT, we map an L-dimensional coordinate space to this CoT as follows. Each sequence $(l_1, \ldots, l_L) \in (0, 1]^L_{\mathbb{R}}$ in the coordinate space is mapped to a path $(s_1, \ldots, s_L)$ in the CoT. To obtain node s_i, $1 \leq i \leq L$, we calculate $k_i := \lceil l_i * \texttt{NUM\_CHILD}_{(s_1,\ldots,s_{i-1})} \rceil$, where $\texttt{NUM\_CHILD}_{(s_1,\ldots,s_{i-1})}$ is the number of child nodes of s_{i-1} after moving along the path $(s_1, \ldots, s_{i-1})$. We round up k_i to the next higher integer value (indicated by $\lceil \ldots \rceil$) and set s_i as the k_i-th child of node s_{i-1}.

Pre-Implemented Search Techniques in ATF

ATF currently offers three pre-implemented search techniques which can be arbitrarily chosen by the ATF user for exploration: 1) *Exhaustive Search*, 2) *Simulated Annealing* [318], and 3) *AUC Bandit* [195]. All of them implement the same, generic interface; new techniques can be easily added to ATF by implementing this interface:

```
class search_technique
{
  void       initialize( CoT search_space );
  void       finalize();
  R_sequence get_next_config();
  void       report_cost( cost_t cost );
}
```

Here, function `initialize` takes as input the search space to explore in its CoT representation; the function is called by ATF before starting the exploration phase, and it is used to initialize the search technique for the concrete characteristics of the CoT to be explored, e.g., by preparing the technique's coordinate space with the required number of dimensions for exploring the levels of the CoT, as discussed above. Function `finalize` is the counterpart of `initialize`: it is called after the exploration phase, e.g., to free allocated memory, etc.

During exploration, ATF performs repeatedly the following two steps until the user-chosen *abort condition* (e.g. a certain time duration has been reached, as we discuss in detail in the next section) is satisfied: 1) ATF requests a sequence of floating point numbers from the search technique (as explained above) using function `get_next_config`, maps the returned sequence to a configuration of tuning parameter values (also explained above), and measures the configuration's cost according to a user-defined cost function (discussed in detail in the next section), e.g., a function to measure the runtime of the program to tune for that configuration and/or to measure the program's energy consumption for that configuration; 2) ATF reports the measured cost to the search technique using the `report_cost` function (`cost_t` represents an arbitrary C++ type for which operator < is defined – also discussed in the next section).

We now discuss the implementation of these four functions for each of ATF's three pre-implemented search techniques. Further search techniques can be easily added to ATF by implementing the interface `search_technique` shown above.

EXHAUSTIVE SEARCH This search straightforwardly iterates over the entire search space. Thereby, the technique finds provably the best tuning result, but at the cost of long tuning times if the search space is large. For exhaustive search, the implementation of `finalize` and `report_cost` is void; function `initialize` stores a reference to the passed CoT search space, and `get_next_config` successively returns sequences of real numbers within the coordinate space that correspond to the individual paths within the CoT.

SIMULATED ANNEALING This search has proven to be efficient for exploring OpenCL and CUDA search spaces when tuning time is limited [190]. The implementation of functions `initialize` and `finalize` is straightforward: memory for intermediate results is allocated/deallocated, etc. Function `get_next_config` returns in each call a random neighbor s' of the current sequence s, and the corresponding runtime t' of s' is reported using function `report_cost`. The configuration s' becomes the new current configuration with probability

$$P(t, t', T) = e^{-(t'-t)*T^{-1}} \in [0,1]_{\mathbb{R}} \text{ for } t' \geq t$$

and 1 if $t' < t$. Here, t and t' represent the runtimes of configurations s and s', and T is the so-called annealing temperature. ATF uses per default $T = 4$, because this value was reported as suitable for exploring OpenCL and CUDA search spaces [190].

AUC BANDIT This search is used as default in ATF – it combines multiple, well-proven search techniques, such as Nelder-Mead and Torczon hillclimbers, which are automatically selected by the AUC Bandit technique based on the specific characteristics of the particular search space to explore. Consequently, this technique achieves an average good tuning result for the search spaces of various various kinds of applications [195]. We use the AUC Bandit implementation of [195] which explores coordinate spaces and thus can be straightforwardly integrated into ATF, according to the methodology presented above: functions `initialize` and `finalize` straightforwardly initialize and finalize the AUC Bandit technique, function `get_next_config` returns the sequence of floating point numbers that is returned by AUC Bandit, and function `report_cost` straightforwardly reports the measured cost of that sequence back to the AUC Bandit technique.

5.6 USER INTERFACE OF AUTO-TUNING FRAMEWORK (ATF)

ATF introduces a *Domain-Specific Language (DSL)* for auto-tuning. The auto-tuning DSL of ATF relies on *tuning directives* which are used by the ATF programmer to straightforwardly annotate the program's source code to be tuned. The directives express, for example, the program's tuning parameters and the particular search technique that the user wants ATF to use for exploring the program's search space.

In the following, we introduce in Section 5.6.1 ATF's auto-tuning DSL, by discussing the required steps for implementing an ATF tuning program via its DSL language. Afterward, we compare ATF's user interface to the interfaces of CLTune (in Section 5.6.2, and OpenTuner (in Section 5.6.3) to argue that ATF achieves a better user experience as compared to the popular CLTune and OpenTuner frameworks. Finally, Section 5.6.4 shows how ATF is used for auto-tuning at the runtime of the program to be tuned (a.k.a. *online* auto-tuning). For this, ATF introduces auto-tuning interfaces implemented in *General Purpose Languages (GPL)*, such as C++ and Python, to work at the runtime of the program to be tuned, e.g., a C++ program or a Python program. Correspondingly, we refer to ATF's online auto-tuning interfaces as *GPL based*, which are used by the ATF user as an alternative to its DSL-based interface which works at program's compile time (*offline* auto-tuning).

5.6.1 *Illustration of ATF's User Interface*

We illustrate ATF's DSL-language by a simple example: auto-tuning the `saxpy` kernel of the *CLBlast* library [131], which is a popular case study in the literature on auto-tuning. The kernel is written in OpenCL: it takes as its input the size N of the input vectors, a floating point value a, and two N-sized vectors x and y of floating point values, and it computes:

$$y[i] = a * x[i] + y[i], \text{ for all } i \in [1, N]_{\mathbb{N}}$$

Listing 23 shows a slightly simplified OpenCL program code of the `saxpy` kernel than presented in [131]: for simplicity, we removed switching between single and double precision floating point inputs, as well as using OpenCL's vector data types (e.g., `float4` which represents in OpenCL a vector of four single-precision floating point numbers).

```
1  __kernel void saxpy( const          int    N,
2                       const          float  a,
3                       const __global float* x,
4                             __global float* y
5                     )
6  {
7    for( int w = 0; w < WPT; ++w )
8    {
9      const int index = w * get_global_size(0) + get_global_id(0);
10     y[ index ] += a * x[ index ];
11   }
12 }
```

Listing 23: Simplified `saxpy` kernel from CLBlast.

The kernel is executed on a device (e.g., a GPU) in parallel by several *Work-Items (WIs)* – the OpenCL term for thread. Each WI computes a chunk of `WPT`-many elements of the result vector – `WPT` stands for *Work-Per-Thread* in CLBlast, and it is a tuning parameter: the programmer has to replace it textually (e.g., using the OpenCL preprocessor) by a concrete value that is optimized for the particular target hardware architecture and characteristics of the input and output data (e.g., data's size and type) [156], or by using an auto-tuning system (like ATF) to automatically determine and replace `WPT` by an optimized value. The WIs iterate over their corresponding chunks of the input (line 7), and they compute in each iteration the index of the input elements of x and y (line 9) to be used in the computation of y (line 10). OpenCL requires work-items to be grouped in so-called *work-groups*. The number of work-items per work-group is called *local size* (`LS`) in OpenCL, and it represents a further tuning parameter of the `saxpy` kernel, i.e, the local size also has to be chosen as optimized for the target architecture and characteristics of the input and output data. The local size is set in the *host code* – C++ code that is required in OpenCL for invoking a kernel on a device (we thoroughly discuss host code in Chapter 6 of this thesis).

For the correctness of the `saxpy` kernel, `WPT` must divide the input size `N`, such that each WI processes an equal-sized chunk of the input. Moreover, the two tuning parameters `WPT` and `LS` are interdependent: the OpenCL specification [342] requires the local size `LS` to divide the *global size* – the total number of work-items – which in case of the `saxpy` kernel is `N/WPT`. Analogously to the local size, the global size is set in the host code when invoking the kernel.

Listing 24 demonstrates how ATF is used for auto-tuning the `saxpy` kernel in Listing 23. The user annotates the kernel's source code (in line 28 of Listing 24) with *ATF tuning directives* (lines 1-26): the directives specify the auto-tuning process, i.e., how the 1) *search space* is generated, 2) *cost function* is implemented, and 3) *search process* is started – the three steps are explained in detail in the following three subsections. The annotated source code in Listing 24 is passed to ATF which then automatically generates and executes a specific auto-tuning program for `saxpy`, according to the directives. The generated tuning program yields as its result the best found configuration for the two tuning parameters of `saxpy` in the form of a JSON file – a common file format for storing human-readable name-value pairs. Since the ranges of `WPT` and `LS` are dependent on the input size – both parameters are in the interval from `1` to `N`, where `N` is the input vector size – we auto-tune the `saxpy` kernel specifically for a fixed, user-defined input size, which is important for high performance of the kernel [155]. To be able to auto-tune the `saxpy` kernel for arbitrary input sizes with the ATF-generated auto-tuning program, the input size is passed by the user as the first command line argument to the generated tuning program, as indicated by `$1` in line 1 of Listing 24.

```
1  #atf::var::N $1
2
3  // Step 1: Describing the Search Space
4  #atf::tp name       "WPT"                    // name
5           range      interval<size_t>( 1,N ) // range
6           constraint divides( N )             // constraints
7
8  #atf::tp name       "LS"                     // name
9           range      interval<size_t>( 1,N ) // range
10          constraint divides( N/WPT )         // constraints
11
12 // Step 2: Implementing the Cost Function
13 #atf::ocl::platform "NVIDIA"    // OpenCL platform
14 #atf::ocl::device   "Tesla K20" // OpenCL device
15
16 #atf::ocl::input scalar<int>( N )   // N
17 #atf::ocl::input scalar<float>()    // a
18 #atf::ocl::input buffer<float>( N ) // x
19 #atf::ocl::input buffer<float>( N ) // y
20
21 #atf::ocl::global_size N/WPT // OpenCL global size
22 #atf::ocl::local_size  LS    // OpenCL local size
23
24 // Step 3: Starting the Search Process
25 #atf::search_technique auc_bandit
26 #atf::abort_condition  duration<minutes>( 10 )
27
28 // saxpy kernel's code of Listing 23
```

Listing 24: The `saxpy` kernel of Listing 23 annotated with ATF tuning directives.

To auto-tune a program with ATF, the programmer has to perform the following three steps: 1) describe the program-specific *search space*, via *tuning parameters*, which contains all possible, valid configurations of the tuning parameters, 2) implement the *cost function* for estimating program's cost for a particular parameter configuration in terms of the target *objective(s)*, e.g., high runtime performance and/or low energy consumption, and 3) start the *search process* based on a *search technique* to be used by ATF for exploring the parameters' search space and an *abort condition* which specifies when to stop the search process (e.g., after a specific time duration). The following three subsections describe how ATF's tuning directives are used for these three steps.

Step 1: Describing the Search Space

The ATF user describes the search space using *tuning parameters* (introduced in Section 5.3.1); ATF then automatically generates and stores the search space according to the methodologies presented in Sections 5.3 and 5.4 of this chapter.

In Listing 24, the tuning parameters are: 1) the Work-Per-Thread `WPT` (line 4-6) which is a `size_t` parameter whose range is in the interval $[1, N]_{\mathbb{N}}$ (line 5) that has to divide the input size N (line 6), and 2) the local size `LS` (line 8-10) – a `size_t` parameter in $[1, N]_{\mathbb{N}}$ (line 9) that divides the global size `N/WPT` (line 10). The generated search space contains the set of all valid parameter configurations, i.e., configurations for which the parameters' constraints (line 6 and 10) are satisfied.
The general form of an ATF tuning parameter is as follows:

```
#atf::tp name       /* name        */
         range      /* range       */
         constraint /* constraints */
```

It has a *name*, a *range*, and a *constraint*, which we explain in the following.

NAME A name is a tuning parameter's unique identifier. ATF textually replaces each occurrence of tuning parameter names by optimized values in the program's source code to be tuned.

RANGE A tuning parameter's range specifies the possible values of the parameter. It can be defined in ATF as either an `interval` or a `set`.

An ATF interval has the form `interval<T>(begin, end, step_size, generator)` and represents the values of an arbitrary primitive C++ type `T` (e.g., `bool`, `int`, or `float`) from `begin` to `end` using the optional `step_size` that has a default value of `1`. The parameter's `generator` function is also optional and allows conveniently defining special parameter ranges, e.g., powers of two. A generator can be any arbitrary lambda function (a.k.a. anonymous function) with the input type `T` and arbitrary primitive output type `T'`. We denote lambda functions using a simplified C++ syntax: we declare function's input parameters in parenthesis together with

their corresponding type, and we state the function's body in curly braces. For example, `(int i){return pow(2,i);}` is the lambda function that takes an integer `i` as input and returns the `i`-th power of `2`. When a generator function is used, the parameter's range type automatically changes to `T'`, and the interval contains the values `generator(i)` for each `i` from `begin` to `end` using step size `step_size`. For example, `interval<size_t>( 1,10 , atf::pow(2) )` represents the first ten powers of 2. Here, `atf::pow(2)` is a *generator function alias* which ATF provides for user's convenience: ATF automatically replaces the alias by the lambda function presented above for generating powers of `2`. Currently, ATF provides two pre-implemented generator function aliases: `atf::pow(N)` for generating the powers of an arbitrary integer `N`, and `atf::multiple_of(N)` for generating a sequence of numbers which are multiples of `N`. Further generator function aliases can be easily added to ATF by the user: ATF provides a configuration file which contains all generator function aliases in the form of the generator's alias name and the lambda function that implements the generator.

As an alternative to intervals, ATF provides range type `set` which explicitly lists all elements in the parameter's range, and thus is convenient to use for small parameter ranges. For example, `set<int>{3,7}` contains the integers `3` and `7`, and `set<string>{cat,dog}` contains the strings `cat` and `dog`.

CONSTRAINT Constraints are a major feature of ATF – they enable filtering a tuning parameter's range and thus expressing interdependencies among tuning parameters, as discussed in Section 5.3.1. We implement constraints in ATF as lambda functions. A constraint function in ATF takes as input a value of its parameter's range and returns a value of type `bool`: values for which the constraint returns `false` are automatically filtered out of the range. We use constraints to conveniently express parameter interdependencies. For example, in line 10 of Listing 24, we use the tuning parameter `WPT` in the constraint function of the tuning parameter `LS` to ensure that `LS` divides `N/WPT`. Here, `divides( N/WPT )` is a *constraint alias* that allows the ATF user conveniently defining divisibility constraints; constraint aliases are a similar concept to the generator functions aliases for interval ranges discussed above, and serve for convenience only. Currently, ATF provides six pre-implemented constraint aliases: `divides`, `multiple_of`, `less_than`, `greater_than`, `equal`, and `unequal`. Constraints (in the form of aliases as well as lambda functions) can be conveniently combined in ATF using the logical operators `&&` (*logical and*) and `||` (*logical or*). For example, to express that tuning parameter `p1` must divide tuning parameter `p2` and that `p1` has to be less than tuning parameter `p3`, the user uses as constraint of parameter `p1` the following expression: `divides(p1) && less_than(p3)`. The user can extend ATF easily by further constraint aliases, by adding the new aliases straightforwardly to an ATF configuration file.

Step 2: Implementing the Cost Function

ATF automatically generates the cost function for the user. For this, ATF provides two directives allowing to the user to let ATF generate cost functions for programs implemented in arbitrary programming languages: 1) `#atf::compile_script` followed by the path to a user-provided script that compiles the user's program, e.g., a bash script, and 2) `#atf::run_script` followed by the path to a user-provided script for running the compiled program. Optionally, the user can state the path to a *cost file* – a text file to which user's program will write its cost that ATF should minimize, e.g., its energy consumption – by using the directive `#atf::cost_file`. If no cost file is specified, ATF automatically measures and uses program's runtime as cost. For multi-objective tuning, e.g., auto-tuning for high runtime performance together with low energy consumption, the user program writes tuples to the cost file comprising the runtime of the code part to be tuned within the program (e.g., in ms) and the part's energy consumption (e.g., in millijoules [6]); ATF then minimizes these costs using the lexicographical order, i.e.: configuration `c` has a lower cost than configuration `c'` if either `c` has a lower runtime than `c'`, or when `c` and `c'` have the same runtime and `c` causes a lower energy consumption than `c'`. The user can easily use a self-defined order, by defining the order in a corresponding ATF configuration file. In Chapter 9 of this thesis, we discuss how ATF's multi-objective tuning can be extended in future work toward identifying the so-called *Pareto front*, inspired by [96] – the set of parameter configurations in which any configuration cannot be optimized further for one objective without worsening the configuration's quality for another objective (e.g., improving runtime performance, but at the cost of higher energy consumption).

For OpenCL, which requires host code for its execution (discussed in detail in Chapter 6 of this thesis), ATF provides special directives to further enhance user experience. The ATF's OpenCL directives allow ATF to automatically generate the host code for the user, e.g., by offering the user special directive to select the desired OpenCL device and to specify the input and output data of the OpenCL program to tune. In our example in Listing 24, we select the Tesla K20 device of system's OpenCL NVIDIA platform (lines 13 and 14) – this example device is chosen arbitrarily and could be any other OpenCL-compatible device of our system; the target OpenCL device can alternatively be chosen in ATF via its numerical platform and device id. As the kernel's input, we use the input size N (line 16), a random floating point number for `a` (line 17) – the random number is automatically generated by ATF since no value is explicitly stated for `a` – and two N-sized buffers for `x` and `y` that are also filled with random floating point numbers (lines 18 and 19) – random data is the default input when auto-tuning OpenCL kernels. The buffers can be generated with user-defined data of type `T` using `atf::ocl::buffer<T>( file_name("./path") )` where `./path` is the path to a text file that contains the buffer's values line by line as

strings; the strings are automatically parsed by ATF to values of type `T`. The kernel's global and local size are conveniently set in lines 21 and 22 as common arithmetic expressions that may contain tuning parameters (such as tuning parameter `WPT` in line 21 and parameter `LS` in line 22). The automatically generated cost function encapsulates kernel's source code, and it takes as input a configuration of tuning parameters comprising concrete values for the `WPT` and `LS` parameters of the `saxpy` kernel in Listing 23; the cost function returns the `saxpy` kernel's runtime for concrete `WPT` and `LS` values. For this, the cost function internally replaces in kernel's source code the tuning parameters' names by their corresponding values in the input configuration using the standard OpenCL preprocessor. Moreover, it uses a pre-implemented OpenCL host code for invoking the kernel with the passed global/local size, and for measuring and returning the kernel's runtime using the standard OpenCL profiling API. Optionally, ATF provides directives for error checking the computed results.

ATF also provides special directives for auto-tuning CUDA kernels – CUDA also require host code for execution, analogously to OpenCL. The CUDA directives are used by ATF to generate NVIDIA NVRTC host code [39] which executes CUDA kernels. The ATF's CUDA directives are analogous to those for OpenCL (in lines 1-26 of Listing 24) with the only difference that no platform name (line 13) is required, because CUDA targets NVIDIA devices only.

Step 3: Starting the Search Process

ATF automatically explores the search space that is described in Step 1, according to the cost function implemented in Step 2. For this, the user chooses one of ATF's pre-implemented *search techniques* and an *abort condition*, both described in the following.

In the example of Listing 24, we use the `AUC Bandit` search [195] (line 25) and abort condition `atf::duration` (line 26) which causes to stop tuning after a time interval – 10 minutes in the case of our `saxpy` example in Listing 24.

Currently, ATF allows the user to choose among three pre-implemented search techniques (as also discussed in Section 5.5):

1) *Exhaustive Search* which finds the provably best configuration, but probably at the cost of a long search time: every parameter configuration within the space is tested in exhaustive search, which becomes very time-intensive for large search spaces;

2) *Simulated Annealing* [318] which has proven to be efficient for auto-tuning OpenCL and CUDA programs if search spaces are too large to be explored exhaustively [190];

3) *AUC Bandit*, inspired by OpenTuner [195], which automatically combines various, well-proven search techniques, e.g., variants of Nelder-Mead search and Torczon hillclimbers, to achieve a good tuning result on average for arbitrary programs with large search spaces.

New search techniques can be easily added to ATF by the user, e.g., for domain-specific user requirements, by implementing a straightforward C++ interface (discussed in Section 5.5).

To stop the exploration phase, ATF offers six pre-implemented abort conditions:

1) `duration<D>(t)`: stops the tuning after a user-defined time interval `t`, where `D` is the time unit: `seconds`, `minutes`, etc.;
2) `evaluations(n)`: stops after `n` tested configurations;
3) `fraction(f)`: stops after `f*S` tested configurations, where `f` is a floating point value in $[0, 1]_{\mathbb{R}}$ and `S` the search space size;
4) `cost(c)`: stops when a configuration with a cost $\leq$`c` has been found;
5) `speedup<D>(s,t)`: stops when within the last time interval `t` the cost could not be lowered by a factor $\geq$`s`;
6) `speedup(s,n)`: stops when within the last `n` tested configurations the cost could not be lowered by a factor $\geq$`s`.

If no abort condition is specified, ATF uses `evaluations(S)`, where `S` is the search space size. Abort conditions can be combined by using the logical operators `&&` and `||` for complex user requirements. New abort conditions can be added easily to ATF by implementing a straightforward C++ interface.

5.6.2 *Comparison: ATF vs. CLTune*

While in the previous Sections 5.3-5.5, we compare ATF and CLTune in terms of their internal design, we now focus on comparing the two approaches regarding their user interfaces. For our comparison, we use again the `saxpy` OpenCL example in Listing 23, taken from the CLBlast library. We consider such a comparison as challenging for ATF, as CLTune is specifically designed toward auto-tuning OpenCL programs and the CLBlast library [131]. In contrast, ATF is designed and optimized toward generality and thus not specifically targeted to OpenCL and/or CLBlast.

Listing 25 shows the CLTune program for auto-tuning the `saxpy` kernel, taken from [161]. The CLTune's user interface is implemented in the C++ programming language[2], rather than based on a high-level DSL language as ATF (Listing 24). In Listing 25, the CLTune user chooses a target device, the OpenCL kernel, and prepares the kernel's input data (lines 6-24). Afterward, the `saxpy`-specific tuning parameters (lines 26-45) are defined, and the search process is started (lines 47-49).

[2]By relying on C++, CLTune can be used for auto-tuning C++ programs at runtime, but at the cost of a user interface that inherently relies on the low C++ abstraction level (rather high-level code annotations, as offered by ATF). ATF provides both a high-level user interface, based on code annotations, enabling conveniently auto-tuning program's at their compile time, and also interfaces implemented in general-purpose programming langues (such as C++ and Python) to enable auto-tuning at program's runtime, as we discuss in Section 5.6.4.

In the following, we compare ATF and CLTune for each of the three steps that the user has to implement for auto-tuning (discussed in Section 5.6.1): 1) describing the search space, 2) implementing the cost function, and 3) starting the search process. We will see that ATF's constraints conveniently express parameters' interdependencies and that ATF allows tuning parameters of arbitrary types, making ATF more flexible and less memory-intensive than CLTune. Moreover, we demonstrate that in contrast to ATF, the CLTune user has to use special functions for setting the OpenCL's global and local size, while ATF relies on common arithmetic expressions for expressing these sizes, which further contributes to ATF's usability and also expressiveness.

Step 1: Describing the Search Space

CLTune allows the user to express parameter interdependencies via user-defined C++ lambda functions. In the case of the `saxpy` example in Listing 25, these lambda functions are: `DividesN` (line 32) and `DividesNDivWPT` (line 39). CLTune's constraint functions take as input a configuration of all tuning parameters (technically represented as a C++ vector in CLTune – lines 32 and 36); the tuning parameters in this `saxpy` example are: `LS` (line 29) and `WPT` (line 30). Configurations for which one of the user-defined CLTune constraint returns `false` are filtered out of the space internally by CLTune.

ATF follows a different constraint design: the user constrains the parameters' ranges (Listing 24, line 6 and 10), rather than full configurations within the search space which may become very large. This design decision enables ATF to substantially accelerate the process of search space generation and, consequently, to auto-tune programs with significantly larger parameter ranges, as discussed in Section 5.3 and confirmed experimentally later in Section 5.7. Moreover, CLTune requires using C++ vectors for expressing constraints (Listing 25, lines 32 and 36). ATF frees the user from this technical burden, by allowing conveniently using tuning parameters' names in the definition of constraints (Listing 24, line 10), rather than relying on a vector representing the parameter names. Also, ATF allows tuning parameters of arbitrary fundamental types (e.g., also `bool` and `float`), while CLTune allows type `size_t` only, making ATF suitable for a broad range of applications with differently-typed tuning parameters.

Step 2: Implementing the Cost Function

ATF allows auto-tuning programs written in arbitrary programming languages using an arbitrary tuning objective (e.g., high runtime performance and/or low energy consumption). For this, ATF provides the user with tuning directives for expressing the program-specific cost function, independently of a particular programming language and tuning objective (as discussed in Section 5.6.1). In contrast, the CLTune user is limited to auto-tuning OpenCL programs only, and CLTune allows high runtime performance as the only tuning objective.

```
int main()
{
  const std::string saxpy = /* path to saxpy kernel    */;
  const size_t      N     = /* an arbitrary input size */;

  cltune::Tuner tuner(1,0);
  auto id = tuner.AddKernel(saxpy, "saxpy", {N}, {1} );

  float a;
  auto  x = std::vector<float>(N);
  auto  y = std::vector<float>(N);

  const auto random_seed = std::chrono::system_clock::now().
      time_since_epoch().count();
  std::default_random_engine generator( static_cast<unsigned int>(
      random_seed) );
  std::uniform_real_distribution<float> distribution(-2.0f,2.0f);

  a = distribution(generator);
  for (auto &item: x) { item = distribution(generator); }
  for (auto &item: y) { item = distribution(generator); }

  tuner.AddArgumentScalar( N );
  tuner.AddArgumentScalar( a );
  tuner.AddArgumentInput( x );
  tuner.AddArgumentOutput( y );

  auto range = std::vector<size_t>( N );
  for( size_t i = 0; i < N ; ++i )
    range[ i ] = i;
  tuner.AddParameter( id, "LS" , range );
  tuner.AddParameter( id, "WPT", range );

  auto DividesN = []( std::vector<size_t> v )
                  {
                    return  N % v[0] == 0;
                  };
  auto DividesNDivWPT = []( std::vector<size_t> v )
                        {
                          return ( N / v[0] ) % v[1] == 0;
                        };

  tuner.AddConstraint( id, DividesN      , {"WPT"}       );
  tuner.AddConstraint( id, DividesNDivWPT, {"WPT", "LS"} );

  tuner.DivGlobalSize(id, {"WPT" } );
  tuner.MulLocalSize(id, {"LS"} );

  tuner.UseAnnealing( 1.0f/2048.0f , 4.0 );
  tuner.Tune();
  const auto parameters = tuner.GetBestResult();
}
```

Listing 25: CLTune program for auto-tuning `saxpy`.

ATF provides OpenCL-specific tuning directives for auto-tuning OpenCL programs (Listing 24, line 13-22). We argue that the ATF's OpenCL-directives improve the user experience of CLTune, for the following reasons.

In CLTune, the values of the global and local size have to be set via function `AddKernel` (Listing 25, line 7). Since these two values are usually dependent on tuning parameters, CLTune provides the two special functions `DivGlobalSize` (line 38) and `MulLocalSize` (line 39) to override the initial value by dividing/multiplying the global and local size with the value of a tuning parameter. In contrast, the ATF user sets the global and local size more generally, by defining them as common arithmetic expressions; the expressions may contain tuning parameters' names to conveniently make the global and local size dependent on the values of tuning parameters (Listing 24, lines 21 and 22). Consequently, the ATF user is not required to set the global and local size to initial values and modify them later using special functions to make the two sizes dependent on tuning parameters, as in CLTune. The generality of ATF makes it particularly more expressive. For example, in addition to `saxpy`, the CLBlast library also contains kernel `GEMM` for computing matrix multiplication. The `GEMM` kernel's global size is computed in CLBlast as an arithmetic expression based tuning parameters and constants. This global size cannot be expressed in CLTune, i.e., CLBlast (which currently relies on CLTune for auto-tuning) has to use a simplified global size for auto-tuning its `GEMM` kernel, thereby missing the kernel's full performance potential [114].

A further convenience feature of ATF is that it automatically generates random input data – the default input in auto-tuning – for arbitrary fundamental types. For example, `scalar<T>()` (Listing 24, line 17) generates a random value of type `T`, and a buffer containing `N` random elements of type `T` is generated via `atf::ocl::buffer<T>(N)` (lines 18 and 19). In comparison, the CLTune user is responsible for explicitly generating its input data (Listing 25, lines 13-19). If the ATF user aims at auto-tuning for concrete input data, then `scalar<T>(a)` represents the concrete scalar value `a` of an arbitrary fundamental type `T` (Listing 24, line 16), and `atf::ocl::buffer<T>( file_name("./path") )` represents a buffer with particular input values, as discussed above.

Step 3: Starting the Search Process

In contrast to CLTune, apart from using a multi-dimensional exploration approach for higher search efficiency (discussed in Section 5.5), ATF implements as an additional search technique the popular AUC Bandit technique [195] which combines a broad range of other search methods for effectively exploring large search spaces [195]. ATF also allows auto-tuning for an arbitrary objective and also multi-objective auto-tuning (as discussed in Section 5.6.1), while CLTune is restricted to auto-tuning for high runtime performance only. Furthermore, ATF offers a broad range of abort conditions (also discussed in Sec-

tion 5.6.1), e.g., to stop the search process depending on the tuning result (`cost` and `speedup`), and ATF also allows combining abort conditions by logical operators to meet complex user requirements. In contrast, the CLTune user can choose as abort condition only testing a fraction of the total search space (line 47 in Listing 25) which corresponds to our `fraction` condition in Section 5.6.1.

5.6.3 *Comparison: ATF vs. OpenTuner*

We compare ATF's user interface to the interface of OpenTuner which also supports auto-tuning programs written in arbitrary languages and targeting flexible tuning objectives. To make comparison challenging for ATF, we use for comparison the running example of the OpenTuner developers: auto-tuning *GNU Compiler Collection's (GCC)* optimization options for application `raytracer` [195]. The GCC's optimization options have no interdependencies among them, making the example preferable for OpenTuner which is designed and optimized toward programs without interdependencies among tuning parameters.

Listing 26 shows the OpenTuner program (which has to be implemented in the Python programming language[3]) for GCC's optimization options, taken from [146]. The OpenTuner user defines one tuning parameter per option (lines 4-14, 18-29) – 322 parameters in total – by overriding OpenTuner's `manipulator` function (line 18) of OpenTuner's class `MeasurementInterface` (line 16), and by using a so-called OpenTuner *configuration manipulator* (line 19). Due to the high number of parameters, the OpenTuner developers provide a straightforward Python script for automatically extracting the GCC's options used in lines 4-14. To define the cost function in OpenTuner, the user overrides OpenTuner's `run` function (lines 31-48). In the `run` function of the GCC example, the OpenTuner user has to explicitly construct a GCC command line call in Python (lines 32-42); the call has to contain the GCC's optimization options according to the particular values of tuning parameters. Moreover, the `run` function has to compile `raytracer` using the constructed GCC call, as well as to run, measure and return the compiled `raytracer`'s runtime (lines 44-48).

Listing 27 shows the ATF directives for auto-tuning the OpenTuner's GCC example; the directives are the ATF's equivalent to the OpenTuner program in Listing 26. In Listing 27, we specify the ATF directives in form of a JSON file. ATF offers the JSON file format as an alternative to the code annotations used in Figure 24 for ATF: for programs with a large number of tuning parameter (e.g., 322 parameters as in the GCC example), the ATF's JSON interface is often more convenient to use, because the JSON file format is well suited to be generated automatically and widely supported in all common pro-

[3] Similarly to CLTune, the OpenTuner's user interface is implemented in a general-purpose programming language, namely Python, rather than relying on a high-level DSL for auto-tuning as ATF (Listing 24). This enables OpenTuner to be used for auto-tuning Python programs at runtime, but at the cost of an increased user effort when targeting programs implemented in languages different from Python (we present ATF's user interface for runtime auto-tuning in Section 5.6.4).

```
import opentuner
# from opentuner import: ConfigurationManipulator, ...

GCC_FLAGS = [
  'align-functions', 'align-jumps', 'align-labels',
  'branch-count-reg', 'branch-probabilities',
  # ... (176 total)
]

GCC_PARAMS = [
  ('early-inlining-insns', 0, 1000),
  ('gcse-cost-distance-ratio', 0, 100),
  # ... (145 total)
]

class GccFlagsTuner(MeasurementInterface):

  def manipulator(self):
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
      IntegerParameter('opt_level', 0, 3))
    for flag in GCC_FLAGS:
      manipulator.add_parameter(
        EnumParameter(flag,
                      ['on', 'off', 'default']))
    for param, min, max in GCC_PARAMS:
      manipulator.add_parameter(
        IntegerParameter(param, min, max))
    return manipulator

  def run(self, desired_result, input, limit):
    cfg = desired_result.configuration.data
    gcc_cmd = 'g++ apps/raytracer.cpp -o ./tmp.bin'
    gcc_cmd += ' -O{0}'.format(cfg['opt_level'])
    for flag in GCC_FLAGS:
      if cfg[flag] == 'on':
        gcc_cmd += ' -f{0}'.format(flag)
      elif cfg[flag] == 'off':
        gcc_cmd += ' -fno-{0}'.format(flag)
    for param, min, max in GCC_PARAMS:
      gcc_cmd += ' --param {0}={1}'.format(
        param, cfg[param])

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0
    run_result = self.call_program('./tmp.bin')
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])

if __name__ == '__main__':
  argparser = opentuner.default_argparser()
  GccFlagsTuner.main(argparser.parse_args())
```

Listing 26: OpenTuner program for auto-tuning GCC's flags for raytracer.

gramming languages. For generating the JSON file in Listing 27, we use a straightforward Python script. Our script is a slightly modified version of the Python script that OpenTuner provides and uses for extracting the GCC's options in Listing 26, lines 4-14 (discussed above). Our only modification is that the script outputs its result in the JSON file format, rather than in the OpenTuner's Python syntax, and that we have added to the script's output the lines 30-35 in Listing 27.

In the following, we compare ATF and OpenTuner in terms of the three steps required for auto-tuning, analogously to Section 5.6.2.

Step 1: Describing the Search Space

A major advantage of ATF over OpenTuner is that ATF offers a more expressive user interface: ATF allows the user to express interdependencies among tuning parameters – via ATF's concept of parameter constraints (Sections 5.3.1 and 5.6.1) – thereby substantially improving the auto-tuning process for important applications, as discussed in Sections 5.3- 5.5 and confirmed experimentally in Section 5.7. Moreover, in comparison to the OpenTuner program in Listing 26, the ATF directives in Listing 27 express the tuning parameters (line 2-28) without the boilerplate code for setting up a configuration manipulator, as required in OpenTuner's approach (Listing 26, lines 18-29). Also, ATF offers the range type `interval` which allows expressing intervals of arbitrary C++ types with user-defined step sizes and generator functions for conveniently expressing domain-specific requirements.

Step 2: Implementing the Cost Function

ATF frees the user from the burden of explicitly implementing the cost function. For example, in Listing 26, the OpenTuner user has to explicitly construct the GCC command, pass the options to the GCC command according to the configuration of tuning parameters passed to the function as input, execute the command, and measure the runtime of the GCC-compiled `raytracer` application (lines 31-48). In contrast, ATF automatically generates in Listing 27 the cost function for the user, based on the following user-defined information: 1) path to the source code of the target program to tune (line 30); 2) command for compiling the program (line 31); 3) command for running the compiled program (line 32). In Listing 27, the target program is a bash script (shown in Listing 28) containing the GCC command for compiling `raytracer`. Consequently, we use in Listing 27 the bash script as both program's source (line 30) as well as compile script (line 31). The command for running `raytracer` is straightforward (line 32).

For generating the bash script in Listing 28, we use again a modified version of OpenTuner's Python script for extracting GCC's options: it straightforwardly generates the GCC command with options that have parametrized values. For example, we use as value for option `early-inlining-insns` (Listing 28, line 5) the name of the corresponding tuning parameter `early-inlining-insns_val` (Listing 27, line 20) instead of a concrete integer value; the parameter name is then replaced by ATF with concrete values for the GCC option during auto-tuning.

```
{
  "tuning_parameters" :
  {
    {
      "name" : "opt_level",
      "range" : {"-O0", "-O1", "-O2", "-O3" }
    },

    {
      "name" : "align_functions",
      "range" : { "-falign_functions", "-fno-align_functions"}
    },
    {
      "name" : "align-jumps",
      "range" : { "-falign-jumps", "-fno-align-jumps" }
    },
    // ... (176 total)

    {
      "name" : "early-inlining-insns_val",
      "range" : "interval<int>( 0, 1000 )"
    },
    {
      "name" : "gcse-cost-distance-ratio_val",
      "range" : "interval<int>( 0, 100 )"
    },
    // ... (145 total)
  },

  "program_source" : "./bash_script.sh",
  "compile_script" : "./bash_script.sh",
  "run_script"     : "./raytracer.bin"

  "search_technique" : "open_tuner",
  "abort_condition"  : "duration<minutes>(30)",
}
```

Listing 27: ATF directives (as JSON file) for auto-tuning GCC's optimization options.

```
#!/bin/bash
g++ apps/raytrayer.cpp -o ./raytracer.bin opt_level
  align-functions align-jumps
  # ... 176 total
  --param early-inlining-insns=early-inlining-insns_val
  --param gcse-cost-distance-ratio=gcse-cost-distance-ratio_val
  # ... 145 total
```

Listing 28: Auto-tunable bash script for compiling `raytracer` via GCC.

To further increase the user experience of the ATF user for auto-tuning modern parallel programming approaches, ATF offers special directives for OpenCL and CUDA which both require host code for their execution (discussed in Section 5.6.1). In contrast, the OpenTuner user has to implement host code explicitly, making auto-tuning OpenCL and CUDA programs tedious and cumbersome with OpenTuner (we thoroughly discuss host code and its implementation in Chapter 6 of this thesis).

Step 3: Starting the Search Process

OpenTuner and ATF both support the AUC Bandit search technique, which has proven to be effective for exploring the spaces of a broad range of applications [195]. However, regarding abort conditions, ATF offers a notably richer set of conditions than OpenTuner: in OpenTuner, the auto-tuning process can only be stopped based on either the tuning time (corresponding to ATF's abort condition `duration` in Section 5.6.1) or based on the number of tested configurations (abort condition `evaluations` in Section 5.6.1). In contrast, ATF allows stopping the tuning also dependent on the tuning result (e.g., via abort condition `cost`) or based on the both tuning time and also tuning result (condition `speedup`). In particular, ATF allows arbitrarily combining abort condition via logical operators, thereby being able to express complex user requirements for stopping the auto-tuning process.

5.6.4 *Online Auto-Tuning via ATF*

While ATF's DSL-based interface (presented in Section 5.6.1) works at compile time (a.k.a. *offline auto-tuning*), ATF also supports auto-tuning at runtime (*online auto-tuning*). For this, ATF offers its auto-tuning interface also as implemented in *General-Purpose Languages (GPLs)*, as an alternative to its DSL-based interface, e.g., implemented in C++ and Python (similarly to the interfaces of CLTune and OpenTuner discussed in Sections 5.6.2 and 5.6.3).

In this section, we present and discuss ATF's GPL-based interface using the example of online auto-tuning C++ programs; other target GPLs, besides C++ (such as ATF's interface for online-tuning Python programs), are currently a work in progress and outlined in Chapter 9.

Listing 29 shows how ATF's GPL-based C++ interface is used for online auto-tuning the `saxpy` kernel in Listing 23; the program is functionally equivalent to ATF's DSL program in Listing 24. The same as ATF's DSL-based interface, the ATF's C++ interface works in three steps that are analogous to the steps described in Section 5.6.1 for ATF's DSL interface. In contrast to ATF's DSL program in Listing 24, the ATF program in Listing 29 can be used as part of a larger C++ program (e.g., a C++ math library containing the `saxpy` kernel in Listing 23) and thus be conveniently executed during the runtime of the large C++ program.

```
int main()
{
  const std::string saxpy = /* saxpy OpenCL kernel code */;
  const int         N     = /* an arbitrary input size  */;

  // Step 1: Describing the Search Space
  auto WPT = atf::tuning_parameter(
               "WPT"                         , // name
               atf::interval<size_t>( 1,N ) , // range
               atf::divides( N )              // constraints
             );

  auto LS  = atf::tuning_parameter(
               "LS"                          , // name
               atf::interval<size_t>( 1,N ) , // range
               atf::divides( N/WPT )          // constraints
             );

  // Step 2: Implementing the Cost Function
  auto saxpy_kernel = atf::ocl::kernel<
                        // kernel's type specification
                        atf::ocl::scalar<int>   , // N
                        atf::ocl::scalar<float> , // a
                        atf::ocl::buffer<float> , // x
                        atf::ocl::buffer<float>   // y
                      >(  saxpy  ,  // kernel code
                         "saxpy" ); // kernel name

  auto cf_saxpy = atf::ocl::cost_function( saxpy_kernel )
                  .platform( "NVIDIA" )   // OpenCL platform
                  .device( "Tesla K20" )  // OpenCL device
                  .inputs( atf::ocl::scalar<int>( N )    , // N
                           atf::ocl::scalar<float>()     , // a
                           atf::ocl::buffer<float>( N ) , // x
                           atf::ocl::buffer<float>( N ) ) // y
                  .global_size( N/WPT ) // OpenCL global size
                  .local_size( LS )     // OpenCL local size

  // Step 3: Starting the Search Process
  auto tuning_result = atf::tuner( cf_saxpy, {WPT,LS} )
                       .search_technique( auc_bandit )
                       .abort_condition( duration<minutes>(10) )
                       .tune();
}
```

Listing 29: ATF program for auto-tuning the saxpy kernel.

In the following, we describe the three steps for auto-tuning in ATF's C++ interface. Since the steps are very similar to the steps described in Section 5.6.1 for ATF's DSL interface, ATF's C++ interface achieves the same advantages over related approaches discussed in Sections 5.6.2 and 5.6.3 for ATF's DSL interface.

Step 1: Describing the Search Space

The same as for ATF's DSL programs in Listing 24, the search space is described in Listing 29 using tuning parameters (lines 7-17). The parameters are specified in Listing 29 via their names, ranges, and constraints, analogously to lines 4-10 in Listing 24. ATF's C++ interface supports the same range types as the DSL interface – `interval` and `set` (see Section 5.6.1) – and ATF's C++ interface also allows the same kinds of constraint functions as DSL programs (also discussed in Section 5.6.1): arbitrary, user-defined lambda functions and ATF's pre-implemented constraint functions, such as `divides`.

Step 2: Implementing the Cost Function

The user implements the cost functions as a C++ *callable* (e.g., an ordinary C++ function or a C++ lambda expression) – the callable takes as input a configuration of tuning parameter values, and it returns a value of an arbitrary C++ type for which operator < is defined (for example, type `size_t`). ATF interprets the function's return value (e.g., program's runtime) as the configuration's cost that has to be minimized. By allowing cost functions to have an arbitrary, user-defined return type, we enable *multi-objective tuning*, e.g., minimizing first for high runtime performance and then for low energy consumption. For example, to auto-tune for both, runtime performance and low energy consumption, the user chooses tuples as return type containing runtime (e.g., in ms) and energy consumption (e.g., in millijoules [6]), and < is defined on tuples as lexicographical order.

For user's convenience, ATF provides the pre-implemented cost function `atf::ocl::cost_function` (Listing 29, lines 29-37) for auto-tuning OpenCL kernels in terms of runtime performance, because OpenCL requires host code for its execution, which is tedious to program (host code is discussed thoroughly in Chapter 6 of this thesis). The ATF's OpenCL cost function works analogously to the OpenCL directives offered by ATF's DSL-based interface (shown in Listing 24, lines 13-22): the user chooses the target OpenCL device via its platform and device name (or, alternatively, via its numerical platform and device id – see Section 5.6.1), sets the OpenCL kernel's input and output data (random data in this example), and sets the kernel's global and local size as arithmetic expressions that may contain tuning parameters for high expressivity (also discussed in Section 5.6.1).

ATF provides a pre-implemented cost function also for auto-tuning CUDA kernels. The ATF's CUDA cost function is based on the NVIDIA NVRTC library [39] and used analogously to the ATF's OpenCL cost function in Listing 29: the only difference is that platform's name (line 30 in Listing 29) is not required in the CUDA cost function, because CUDA targets NVIDIA devices only.

Moreover, ATF's C++ interface offers a generic cost function to simplify auto-tuning programs written in an arbitrary programming language and for an arbitrary objective in the ATF's C++ interface. The generic cost function is initialized with: 1) the path to program's source file, 2) paths to two user-provided scripts for compiling and running the program to tune, and optionally 3) the path to a text file to which the user program writes its cost that ATF should minimize (if no text file is stated, ATF automatically measures and uses program's runtime as cost). For multi-objective tuning, the auto-tuned program writes comma-separated costs to the text file; ATF then minimizes these costs using the lexicographical order, or, alternatively, a user-defined order.

Step 3: Starting the Search Process

The search process is managed in ATF's C++ interface via a `tuner` object (line 40 in Listing 29). The object is initialized using the cost function to minimize and the tuning parameters to tune for (line 40); the `tuner` object is then configured with the desired search technique (line 41) and abort condition (line 42). The search process is eventually started via function `tune` (line 43). ATF's C++ interface supports the same kinds of search techniques and abort condition as its DSL-based interface discussed in Section 5.6.1. ATF's tuner object also supports error checking the computed results, the same as ATF's DSL-based interface.

Program-Guided Online Auto-Tuning

Since online auto-tuning happens at the runtime of the program that is being tuned, many kinds of programs can benefit from accessing the results of different tuning iterations: instead of running auto-tuning on random input data and discarding the computed results of the different tuning iterations, each tuning iteration is run on meaningful data, such that the computed results of the different iterations can be used for the actual computation. This is in particular useful for *iterative applications* where the same computation is repeatedly called on different data: each tuning iteration then can compute a call of the computation – the average performance of each call improves over time, because the auto-tuning process tries to use a better performing configuration of tuning parameters for each new call of the computation.

Listing 30 shows a straightforward C++ implementation of *Gaussian Convolution* [209] which is popular in signal and image processing. The implementation is auto-tunable: it uses tuning parameter `CB_SIZE` for flexibly choosing a cache block size, allowing better exploiting locality [310].

```
1  class gaussian_at
2  {
3    public:
4      gaussian_at( atf::configuration &config )
5        : _CB_SIZE( config["CB_SIZE"].value() )
6      {}
7
8      // Gaussian Convolution
9      void operator()( size_t N, std::vector<float>& data )
10     {
11       for( size_t cbo = 0 ; cbo < N ; cbo += _CB_SIZE )
12         for( size_t i = cbo; i < cbo + _CB_SIZE; ++i )
13           for( size_t j = 0; j < N; ++j )
14             data[ /* ... */ ] = /* ... */ ;
15     };
16
17   private:
18       size_t _CB_SIZE;
19 };
```

Listing 30: ATF-Tunable Gaussian Convolution in C++.

Listing 31 contains in lines 5-22 a straightforward ATF program for online auto-tuning the Gaussian kernel in Listing 30; the ATF program in lines 5-22 works analogously to the ATF program in Listing 29 and thus does not use the auto-tuning process for the actual computation part (which is shown in lines 24-31 of Listing 30): first the kernel is auto-tuned in lines 5-22, and only afterward the auto-tuned kernel is called on the actual input data in lines 24-31.

Listing 32 shows how ATF is used for auto-tuning the Gaussian example in Listing 30 when using the actual input data and thus computing meaningful results during the auto-tuning process. In lines 6-11 of Listing 32, we describe Gaussian's search space – this is done the same as in Listing 31. Afterward in Listing 32, in line 14, we implement the cost function. In contrast to Listing 31, we refrain from using fixed input data for the cost function, because we aim to call the function in Listing 32 later in line 21 on the actual input of Gaussian. In lines 17-18, we initialize the tuner object with the desired search technique, but we do not set an abort condition for the tuner (as in Listing 31), because the tuning process is intended to be guided by the user program. The computation of Gaussian is performed in lines 20-21 of Listing 32 as part of the tuning process: the cost function is iteratively called on the actual input data, and the function's cost (in this case, its runtime which is measured via the pre-implemented ATF cost function `atf::cpp::cost_function` for C++) is reported back to the `tuner` object via its function `make_step`. The `make_step` function makes one tuning step: it passes to the Gaussian's cost function a configuration of tuning parameters, runs the function, measures its cost, and reports the cost to the search technique which then determines internally a new configuration of tuning parameters for the next call of the `make_step` function.

```
1  int main()
2  {
3    auto N = /* input size */;
4
5    // ATF Program for Gaussian
6    // 1. Step: Describing the Search Space
7    auto CB_SIZE = atf::tuning_parameter(
8                     "CB_SIZE"                    ,
9                     atf::interval<size_t>( 1,N ) ,
10                    atf::divides( N )
11                  );
12
13   // 2. Step: Implementing the Cost Function
14   auto random_data = /* random input data */;
15   auto cf_gaussian = atf::cpp::cost_function( gaussian_at )
16                        .inputs( N, random_data );
17
18   // 3. Step: Starting the Search Process
19   auto tuning_result = atf::tuner( cf_gaussian, {CB_SIZE} )
20                          .search_technique( auc_bandit )
21                          .abort_condition( evaluations(1000) )
22                          .tune();
23
24   // Using Auto-Tuned Gaussian Iteratively
25   auto gaussian =
26     gauassian_at( tuning_result.get_configuration() );
27
28   auto data = /* input data */;
29
30   for (int iter = 0 ; iter < 1000 ; ++iter)
31     gaussian( N, data );
32 }
```

Listing 31: Straightforward Online Auto-Tuning of Gaussian Convolution (Listing 30) via ATF.

Compared to Listing 31, the program in Listing 32 has a significantly lower total runtime: the Gaussian function is called in Listing 32 1000-times only (lines 20-21 of Listing 32), whereas in Listing 31, the function is called 1000-times as part of the auto-tuning process (in lines 5-22) and a further 1000-times for computing Gaussian on the actual input data (in lines 24-31).

We call auto-tuning as in Listing 32 *Program-Guided Online Auto-Tuning*, because the auto-tuning process is guided by the user program: instead of letting the auto-tuner internally run the cost function (as in lines 40-43 of Listing 31), in program-guided auto-tuning, the user program is in charge of running the cost function, thereby being able to use the function's computed results which are otherwise discarded when the function is called internally within the tuner (as in Listing 31).

```
1  int main()
2  {
3    auto N    = /* input size */;
4    auto data = /* input data */;
5
6    // 1. Step: Describing the Search Space
7    auto CB_SIZE = atf::tuning_parameter(
8                     "CB_SIZE"                    ,
9                     atf::interval<size_t>( 1,N ) ,
10                    atf::divides( N )
11                  );
12
13   // 2. Step: Implementing the Cost Function
14   auto cf_gaussian = atf::cpp::cost_function( gaussian_at );
15
16   // 3. Step: Iterative Gaussian as part of Tuning Process
17   auto tuner =
18     atf::tuner( CB_SIZE ).search_technique( auc_bandit );
19
20   for (int iter = 0 ; iter < 1000 ; ++iter)
21     tuner.make_step( cf_gaussian.inputs( N, data ) );
22 }
```

Listing 32: *Program-Guided Online Auto-Tuning* of Gaussian Convolution (Listing 30) via ATF.

5.7 EXPERIMENTAL EVALUATION

All experiments described in this section can be reproduced using our artifact implementation [62].

In the following, after describing our experimental setup in Section 5.7.1, we introduce four application case studies in Section 5.7.2 which are popular in auto-tuning and used for experiments in this section. Afterward, in Section 5.7.3, we compare ATF which implements our novel mechanisms for search space generation (Section 5.3), storing (Section 5.4), and exploration (Section 5.5) against state-of-the-art competitors. In Section 5.7.4, we experimentally analyze ATF regarding each particular phase of the auto-tuning process. Finally, we discuss in Section 5.7.5 ATF's auto-tuning efficiency for application classes different from our four studies in Section 5.7.2, and we present in Section 5.7.6 ATF's efficiency for auto-tuning a real-world deep learning neural network.

5.7.1 *Experimental Setup*

We use for experiments a system equipped with an Intel Xeon Gold 6140 18-core CPU, tacted at 2.3GHz with 192 GB main memory and hyper-threading enabled, and an NVIDIA Tesla V100-SXM2-16GB GPU. Time measurements are made using the C++ chrono library and the OpenCL profiling API, respectively.

5.7.2 *Application Case Studies for Experiments*

Our experiments rely on four case studies:

1. *Gaussian Convolution (`CONV`)* – a popular stencil computation;
2. *General Matrix-Matrix Multiplication (`GEMM`)* from the area of linear algebra;
3. *Coupled Cluster (`CCSD(T)`)* which is important in quantum chemistry [286];
4. *Probabilistic Record Linkage (`PRL`)* which is widely used in data mining [116], e.g., to identify duplicate entries in databases.

We use the recent GPU and CPU implementations of these four applications presented in [115]. In particular, we show experimentally that when using ATF for auto-tuning these implementations, the implementations can be auto-tuned to higher performance than current state-of-practice solutions, including Facebook's TensorComprehensions framework [111] which relies on state-of-the-art polyhedral techniques combined with a special-purpose auto-tuner, as well as hand-optimized vendor libraries such as Intel MKL/MKL-DNN [337, 338] and NVIDIA cuBLAS/cuDNN [36, 43] for linear algebra routines and stencil computations, respectively.

The implementations in [115] are written in OpenCL in order to target different kinds of architectures, and they rely on multiple tuning parameters, including sizes of tiles and numbers of threads on different memory and core layers. The parameters have various interdependencies among them, e.g., the value of a tile size tuning parameter on an upper memory layer has to be a multiple of a tile size on a lower memory layer – an interdependency among the tile size parameters – because a lower-layer tile is a chunk of an upper-layer tile. We refer the reader to [115] for more details about the tuning parameters and their particular interdependencies, as this chapter is focussed on auto-tuning, rather than generating auto-tunable implementations (which is the focus of Chapter 4 of this thesis).

	App.	#TPs	TP Groups	Min. RS	Max. RS	Avg. RS	|SP|	FT
1	`CONV`	14	{1,1,2,2,4,4}	2	4092	2339.29	$1.69 * 10^{8}$	$1.49 * 10^{-23}$
2	`GEMM`	19	{1,1,2,3,4,4,4}	2	500	121.84	$2.51 * 10^{8}$	$2.47 * 10^{-17}$
3	`CCSD(T)`	39	{1,1,2,4,4,4,4,4,4,4,7}	2	24	15.46	$8.81 * 10^{18}$	$1.47 * 10^{-25}$
4	`PRL`	14	{1,1,2,2,4,4}	2	1024	586.14	$2.31 * 10^{7}$	$1.33 * 10^{-19}$

Table 2: Auto-tuning characteristics of our four application studies.

Relevant for the evaluation in this section are the auto-tuning characteristics of our four studies, which are summarized in Table 2. For high performance, the implementations in [115] span different search spaces [155] for different sizes of the input; we use for experiments in this section the same real-world input sizes also used in [115] for experiments:

1. `CONV`: (4096×4096) input images
2. `GEMM`: (10×64) and (64×500) input matrices
3. `CCSD(T)`: $(24 \times 16 \times 16 \times 24 \times 16 \times 16 \times 24)$ tensors
4. `PRL`: (1024×1024)-sized database entries

The columns of the table have the following meanings:

- `#TPs`: number of tuning parameters
- `TP Groups`: number of groups of interdependent tuning parameters (defined in Section 5.3), and the groups' individual sizes
- `Min. RS`: minimum range size of tuning parameters
- `Max. RS`: maximum range size of tuning parameters
- `Avg. RS`: average range size of tuning parameters
- `|SP|`: size of the constrained search space
- `FT`: fraction of the unconstrained search space that represents valid configurations

For example, application `CONV` has 14 tuning parameters which are automatically split by ATF into 6 groups of interdependent parameters: two groups comprising 1 parameter, two groups comprising 2 parameters, and two groups comprising 4 parameters. Characteristics `Min. RS`, `Max. RS`, `Med. RS`, `Avg. RS`, `|SP|`, and `FT` depend on the particular input size of the applications, because the input size is used to calculate the upper bound of some of the tuning parameters' ranges [115]. For example, the range of the tile size tuning parameter is defined as all values between 1 and the input size. In the table, we present values of the input-dependent characteristics for the same input sizes as also used in [115] (listed above), e.g., real-world sizes taken from deep learning. For example, for input size 4096×4096, `CONV`'s parameters have a minimum range size of 2 (a boolean parameter) and a maximum range size of 4092 (tile size); `CONV`'s average range size is 2339.29. The constrained search space of `CONV` (which comprises only valid configurations, as in ATF, CLTune, and KernelTuner) has a size of $1.69 * 10^8$ for input size 4096×4096. The `FT` value ($1.49 * 10^{-23}$ in this example) denotes which fraction of the unconstrained search space represents valid configurations. For example, the unconstrained search space of `CONV` has a size of $1.13 * 10^{31}$ and only a fraction of $1.49 * 10^{-23}$ of configurations within the space ($1.69 * 10^8$ many) are valid.

We observe from Table 2 that our case studies have very different auto-tuning characteristics, thus enabling a thorough evaluation of the ATF framework.

5.7.3 *Comparison of Auto-Tuning Efficiency*

We compare the overall auto-tuning efficiency of ATF (i.e., the performance achieved by ATF-tuned case studies) which implements our novel mechanisms presented in Sections 5.3-5.5 to the auto-tuning efficiency of two representative examples: i) OpenTuner, as a representative for auto-tuners which are designed for programs whose tuning parameters have no interdependencies among them, and ii) CLTune, as a representative for tuners supporting interdependencies among tuning parameters. For brevity, we do not present our experimental results for the general-purpose auto-tuning frameworks `libtuning` and `KernelTuner`, because our results for them are analogous to those that we obtain for OpenTuner and CLTune: the same as OpenTuner, `libtuning` is optimized toward programs whose tuning parameters have no interdependencies, and KernelTuner relies on exactly the same mechanisms for search space generation and storing as CLTune. Thus, even though `libtuning` and `KernelTuner` use other kinds of search techniques than OpenTuner and CLTune, both have difficulties with auto-tuning our case studies for the same reasons as OpenTuner and CLTune discussed in Sections 5.3-5.5 and confirmed experimentally in the remainder of this section.

Additionally, we compare the performance of our application case studies auto-tuned using our ATF approach against the newest versions of state-of-practice, high-performance libraries that use their own optimized implementations of these applications (rather than implementations in [115]):

- NVIDIA `cuDNN` 7.6.5 [36] and `cuBLAS` 10.2 [43] for GPUs, as well as Intel `MKL-DNN` 0.21.5 [337] and `MKL` 2020 [338] for CPUs, which are architecture-specific approaches for high-performance convolution computations or linear algebra routines, respectively; the libraries rely on hand-optimized assembly code, rather than auto-tuned OpenCL programs as we use for ATF experiments in this section based on OpenCL implementations in [115];
- `Conv2D` and `CLBlast` [131, 190] which are auto-tunable OpenCL implementations for GPUs and CPUs of convolution or matrix multiplication, respectively; both implementations rely on the CLTune auto-tuning framework, which is specifically designed and optimized toward auto-tuning these two implementations [190];
- `TensorComprehensions (TC)` [111] and `COGENT` [92] which are recent approaches optimized toward efficiently computing `CCSD(T)` on NVIDIA GPUs; `TC` generates its own `CCSD(T)` implementation based on state-of-the-art polyhedral techniques, and it is tightly coupled to a self-provided, special-purpose auto-tuner; `COGENT` generates CUDA code for `CCSD(T)` based on hand-crafted heuristics for optimization, rather than auto-tuning;
- the hand-optimized, parallel Java implementation of `PRL` for multi-core CPU that is currently used by `EKR` [265] – the largest cancer registry in Europe.

Tables 3 and 4 show the measured runtimes of our four case studies on GPU (Table 3) and CPU (Table 4) when auto-tuned using ATF, compared to auto-tuning the application studies using the existing general-purpose auto-tuners OpenTuner and CLTune, as well as the state-of-practice, high-performance libraries listed above (highlighted using italic in the tables and separated via double horizontal line). We auto-tune each of our four studies for 4h with each auto-tuning framework; the frameworks are denoted in the table as: `ATF`, `OpenTuner`, `CLTune`, and `CLTune (pruned)` where `pruned` means that the search space is hand-pruned by a CLTune expert. For a fair comparison, we conduct each tuning run for ten times, and we present in the tables for each framework the results of the best run (the result of the other 9 runs are presented and discussed later in Section 5.7.4).

In addition to the runtimes of the application studies, we present in Tables 3 and 4 for each framework and library also: i) the time required for generating a study's search space, ii) the search space size for each particular study, iii) the number of valid configurations explored in the 4h of tuning time for each study, and iv) the number of invalid configurations explored. Note that `ATF`, `CLTune`, and `CLTune (pruned)` explore only valid configurations, thereby requiring search space generation time (see Section 5.3). In contrast, `OpenTuner` relies on the unconstrained search spaces and thus, it requires no search space generation time; however, at the cost of invalid configurations within its search space. The high-performance libraries Intel `MKL-DNN/MKL` and NVIDIA `cuDNN/cuBLAS` (for `CONV` and `GEMM`), as well as libraries `COGENT` (for study `CCSD(T)`) and `EKR` (for `PRL`) do not exploit auto-tuning; consequently, they do not generate or explore search spaces. The `TC` library uses internally a domain-specific, special-purpose auto-tuner which explores valid configurations only. To make comparison more challenging for `ATF`, we use for `TC` a tuning time of 12h, rather than 4h as for `ATF`. Note that we cannot report the search space size of `TC`, because the size is not listed in `TC`'s log files.

We observe in Tables 3 and 4 that frameworks `OpenTuner` and `CLTune` have difficulties with auto-tuning most of our application studies. In the case of `CLTune`, this is because of a too high search space generation time, which we discuss and analyze in detail in the next subsection. `OpenTuner` cannot find a single valid configuration in the tuning time of 4h – in all 10 tuning runs per particular application study – because it relies on the unconstrained search space which contains too many invalid configurations.

For CLTune in Tables 3 and 4, we use also a hand-pruned search space (denoted as `CLTune (pruned)` in the tables), because pruning is usually required in CLTune for generating its search spaces in adequate time [190]. To generate the pruned CLTune search spaces, we use exactly the same, restricted ranges of tuning parameters that are recommended by the CLTune experts in [190]: range $\{1, 8, 16, 32\}$ for the number of threads tuning parameters on different thread layers, rather than the parameters' complete range $\{1, \ldots, N\}$ which we use for `ATF`, where N denotes the input size; for sizes of tiles, we use for `CLTune (pruned)` range $\{1, 16, 32, 64, 128\}$ instead of $\{1, \ldots, N\}$. For the

	Framework	NVIDIA V100 GPU				
		SP Gen.Time	SP Size	Valid Configs.	Invalid Configs.	Runtime on GPU
CONV	**ATF**	68 ms	1.69E+08	5,178	0	**0.20 ms**
	OpenTuner	--	1.13E+31	0	161,355	**FAILED**
	CLTune	>4.0 h	1.69E+08	0	0	**FAILED**
	CLTune (pruned)	1481 ms	48	48	0	**5400.07 ms**
	NVIDIA cuDNN	auto-tuning not supported				**3.14 ms**
	Conv2D	3986 ms	3,684	3,684	0	**0.80 ms**
GEMM	**ATF**	4 ms	2.51E+08	7,426	0	**5.12 us**
	OpenTuner	--	1.02E+25	0	158,066	**FAILED**
	CLTune	>4.0 h	2.51E+08	0	0	**FAILED**
	CLTune (pruned)	1.6 h	3,024	2,300	0	**613.99 us**
	NVIDIA cuBLAS	auto-tuning not supported				**12.29 us**
	CLBlast	134 ms	8,420	8,420	0	**23.00 us**
CCSD(T)	**ATF**	11 ms	8.81E+18	2,270	0	**0.23 ms**
	OpenTuner	--	5.99E+43	0	168,090	**FAILED**
	CLTune	>4.0 h	8.81E+18	0	0	**FAILED**
	CLTune (pruned)	>4.0 h	2.32E+09	0	0	**FAILED**
	TensorComprehensions	--	?	3,190	0	**0.54 ms**
	COGENT	auto-tuning not supported				**0.48 ms**
PRL	**ATF**	18 ms	2.31E+07	105	0	**0.74 ms**
	OpenTuner	--	1.74E+26	0	163,849	**FAILED**
	CLTune	>4.0 h	2.31E+07	0	0	**FAILED**
	CLTune (pruned)	1818 ms	1.75E+05	162	0	**0.79 ms**
	EKR	gpu not supported				

Table 3: Auto-tuning efficiency of ATF vs state-of-the-art auto-tuners and high-performance libraries on GPU for application studies: `CONV` (Gaussian Convolution), `GEMM` (GEneral Matrix-Matrix multiplication), `CCSD(T)` (Coupled Cluster), and `PRL` (Probabilistic Record Linkage). Here, `SP` abbreviates *Search Space*.

	Framework	Intel Xeon CPU				
		SP Gen.Time	SP Size	Valid Configs.	Invalid Configs.	Runtime on CPU
CONV	**ATF**	63 ms	1.69E+08	15,804	0	**4.35 ms**
	OpenTuner	--	1.13E+31	0	182,918	**FAILED**
	CLTune	>4.0 h	1.69E+08	0	0	**FAILED**
	CLTune (pruned)	1227 ms	48	48	0	**194.14 ms**
	Intel MKL-DNN	auto-tuning not supported				**13.49 ms**
	Conv2D	3358 ms	3,536	3,536	0	**16.20 ms**
GEMM	**ATF**	5 ms	2.51E+08	22,463	0	**26.74 us**
	OpenTuner	--	1.02E+25	0	178,008	**FAILED**
	CLTune	>4.0 h	2.51E+08	0	0	**FAILED**
	CLTune (pruned)	1.3 h	3,024	2,142	0	**58.92 us**
	Intel MKL	auto-tuning not supported				**65.03 us**
	CLBlast	107 ms	3839	3839	0	**160.00 us**
CCSD(T)	**ATF**	12 ms	8.81E+18	7,702	0	**3.67 ms**
	OpenTuner	--	5.99E+43	0	190,651	**FAILED**
	CLTune	>4.0 h	8.81E+18	0	0	**FAILED**
	CLTune (pruned)	>4.0 h	2.32E+09	0	0	**FAILED**
	TensorComprehensions	cpu not supported				
	COGENT	cpu not supported				
PRL	**ATF**	17 ms	2.31E+07	291	0	**0.41 ms**
	OpenTuner	--	1.74E+26	0	188,030	**FAILED**
	CLTune	>4.0 h	2.31E+07	0	0	**FAILED**
	CLTune (pruned)	1450 ms	1.75E+05	442	0	**0.68 ms**
	EKR	auto-tuning not supported				**183.00 ms**

Table 4: Auto-tuning efficiency of ATF vs state-of-the-art auto-tuners and high-performance libraries on CPU for application studies: `CONV` (Gaussian Convolution), `GEMM` (GEneral Matrix-Matrix multiplication), `CCSD(T)` (Coupled Cluster), and `PRL` (Probabilistic Record Linkage). Here, `SP` abbreviates *Search Space*.

further tuning parameters of our four studies, the CLTune experts provide no pruning recommendations; thus, to avoid missing well-performing values in these parameters' ranges, we use for them the original, unrestricted ranges which we also use for ATF.

We observe in Tables 3 and 4 that search space pruning enables CLTune to generate its search space in acceptable time. For example, for application CONV, CLTune (pruned) requires 1.227s to generate CONV's search space, while without pruning, CLTune needs > 4h. However, pruning severely affects applications' performance: slowdowns ranging from 1.06× (for PRL) to up to $> 10^4\times$ (for CONV) when comparing CLTune (pruned) to ATF on GPU in Table 3. This is because pruning by hand is a complex task and thus, it often excludes well-performing configurations out of the search space, even when using the pruned parameter ranges recommended by the CLTune experts. For example, for application CONV on GPU, ATF determines as optimal number of threads the (counter-intuitive [73]) value of 372, which is not represented in CLTune's recommended, hand-pruned ranges. ATF is able to find such counter-intuitive parameter values, because classification of configurations in well-performing and not well-performing is left entirely to ATF (without relying on hand-pruning by humans).

As compared to high-performance libraries, we observe in Tables 3 and 4 that ATF auto-tunes our applications to better performance. In case of MKL-DNN, MKL, cuDNN, cuBLAS, COGENT, and EKR, this is because we rely on auto-tuning for the particular input size, while the libraries use hand-crafted heuristics optimized toward average high performance over different sizes; thereby, the libraries avoid the time-intensive process of auto-tuning for the particular size. However, as we demonstrate in Section 5.7.6, auto-tuning for the input size is well amortized in many application areas [156], e.g., deep learning, where the same sizes are re-used in each program run.

High-performance libraries Conv2D and CLBlast in Tables 3 and 4 use their own, auto-tunable OpenCL implementations that rely on CLTune for auto-tuning. In contrast to these two libraries, we achieve better performance by auto-tuning with ATF the implementations provided in [115]; these implementations have larger search spaces than Conv2D and CLBlast and thus, they enable more fine-grained optimization for the target architecture and input/output characteristics – this is discussed in detail in [115]. The larger spaces in [115] cannot be generated using CLTune (as confirmed in Tables 3 and 4), because CLTune relies on a straightforward search space generation algorithm (discussed in Section 5.3).

In the following, we compare ATF to the state-of-the-art auto-tuning frameworks in terms of each particular phase of the auto-tuning process.

5.7.4 *Generating, Storing, and Exploring Constrained Search Spaces*

This section experimentally evaluates ATF's contributions presented in Sections 5.3-5.5 by measuring and assessing the search space generation time (Section 5.3), the memory footprint (Section 5.5), and the exploration efficiency (Section 5.5) of ATF for constrained search spaces as compared to CLTune. In particular, we show that even when improving search space generation in the state-of-the-art auto-tuning frameworks, which is one of their main limitations (as discussed in the previous subsection), the approaches would still suffer from severe weaknesses regarding the storing and exploration of constrained search spaces.

Note that a comparison with OpenTuner and `libtuning` for generating/storing/exploring constrained search spaces is not possible, because both approaches inherently rely on the unconstrained search space (see Section 5.2.1), with the major drawbacks discussed in Section 5.7.2. We also refrain from presenting our experimental results for KernelTuner, because it relies on exactly the same mechanisms for generating, storing, and exploring constrained search spaces as CLTune; thus, both approaches achieve analogous results, even though KernelTuner uses other kinds of search techniques.

In the following, we use for our four studies again the state-of-the-art GPU and CPU implementions presented in [115], which achieve high performance (as confirmed in Tables 3 and 4) by relying on complex search spaces.

GENERATING CONSTRAINED SEARCH SPACES Figure 37 reports the measured search space generation time for our four application case studies when using ATF which implements our novel mechanism for generating constrained search spaces (introduced in Section 5.3), compared to the search space generation mechanisms of CLTune. In addition, we compare also to a state-of-the-art constraint solver [130] which is designed for combinatorial problems and thus can be exploited also for search space generation in auto-tuning. We show for each application the generation time for different square, power-of-two input sizes. The generation times are growing with the input sizes, because the sizes are used to calculate the upper bounds for some of the tuning parameter ranges, e.g., tile size parameters, as discussed above. For each combination of application and input size, we measure the search space generation times up to 12h on our system. For larger input sizes, the generation times of our competitors often exceed 12h (e.g., in case of study `CONV`, for sizes $2^4, 2^5, \ldots$). In these cases, we report a theoretically computed generation time (highlighted as dashed lines in Figure 37) which we compute based on the average times measured for smaller input sizes.

We observe in Figure 37 that ATF generates constrained search spaces faster than its competitors, by several orders of magnitude, already on small input sizes (note the logarithmic scale in the figure). For example, CLTune requires > 1h for generating the search space of the `CONV` implementation in [115], even for small 8×8 input images,

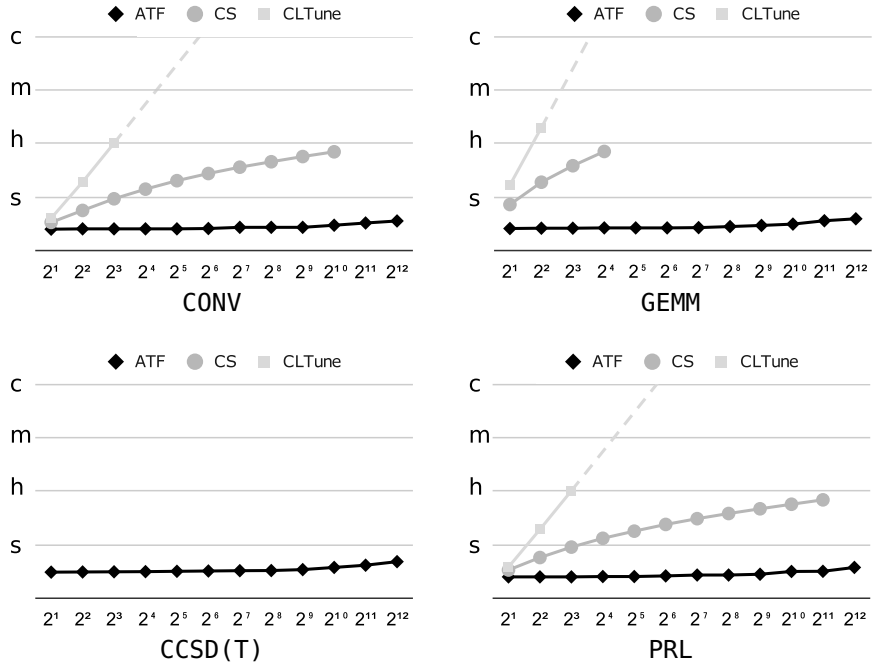


Figure 37: Search space generation time (lower is better) of ATF vs. Constraint Solver (CS) [130], and CLTune [190] for our four case studies using different square, power-of-two input sizes. We use a logarithmic scale on the y-axis: seconds (`s`), hours (`h`), months (`m`), centuries (`c`). When search space generation time exceeds 12h, we use a theoretically computed generation time (highlighted as dashed line). For study `CCSD(T)`, we cannot present the search space generation time of `CS` and `CLTune`, because `CS` crashes due to large memory footprint, and `CLTune` requires > 12h generation time for each input size of `CCSD(T)`. `CS` also crashes for other applications on large sizes because of its large memory footprint.

while ATF requires only 21ms to generate the same space – a speedup of $> 10^5$. This is because CLTune relies on a naive search space generation algorithm (discussed in Section 5.3) which iterates over the huge unconstrained search space containing more than 10^9 configurations for 8×8 input images. In contrast, ATF uses our novel search space generation algorithm which enables generating groups of interdependent parameters independently and in parallel (referred to as *Optimization 1* in Section 5.3), and ATF also checks constraints early in the search space generation phase (*Optimization 2*). The constraint solver `CS` shown in Figure 37 does not exploit parameter grouping and parallelization (ATF's *Optimization 1*), causing significantly higher search space generation time than our novel space generation algorithm in ATF.

Asymptotic behavior differs over approaches, because ATF and solver `CS` check constraints early, for each particular parameter, while CLTune checks constraints simultaneously for all parameters and thus late in the generation phase.

STORING CONSTRAINED SEARCH SPACES We compare the memory requirements of the constrained search space built by ATF which relies on its novel CoT structure (introduced in Section 5.4) against CLTune for our four case studies using again different square, power-of-two input sizes. CLTune relies on a one-dimensional search space representation which is memory intensive (explained in Section 5.4); consequently, CLTune suffers from a memory crash for already quite small input sizes.

In the following, to make comparison challenging for ATF, we compute theoretically the minimum memory requirement of the one-dimensional search space representation in CLTune for each particular combination of application and input size, as follows. The search space in CLTune is a flat array of configurations which contains particular values of the tuning parameters (e.g., 14 parameters in case of application `CONV` – see column `#TPs` in Table 2). Each tuning parameter value has at least a size of 1 `byte` (usually more, e.g., 4 `byte` in case of an `integer` parameter); this results in the following minimum memory requirement of the one-dimensional CLTune search space: |`SP`| $*$ `#TPs` $*$ `1 Byte`. Here, |`SP`| denotes the number of configurations within the space (listed in Table 2), which is equal to the search space size; `#TPs` denotes the number of parameter values per configuration, which is equal to the number of tuning parameters (also listed in Table 2). The |`SP`| values in Table 2 are all computed using ATF which (in contrast to CLTune) is capable of generating and storing large spaces.

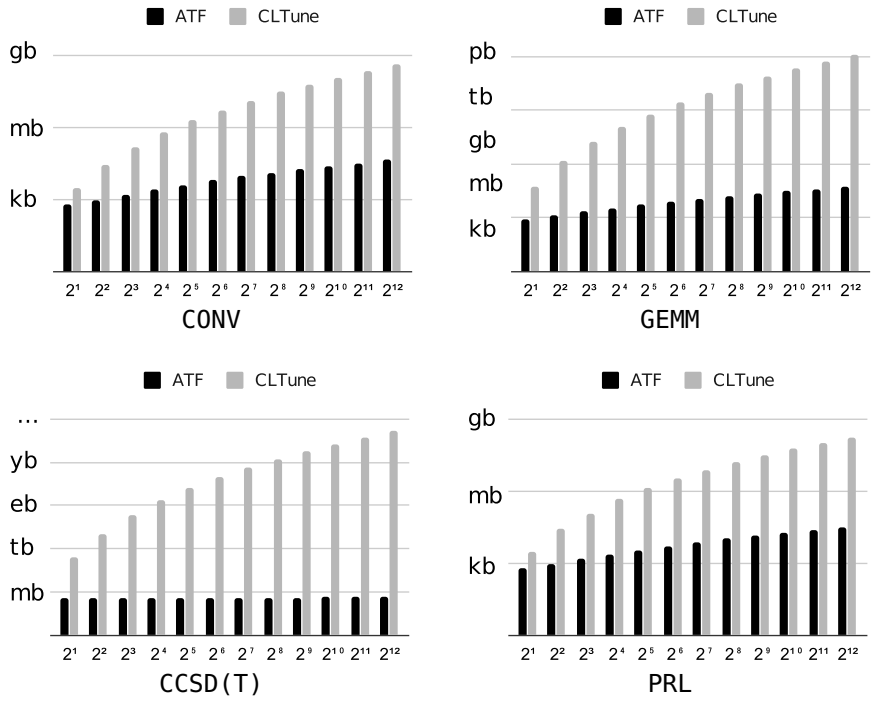


Figure 38: Memory footprint (lower is better) of ATF vs. CLTune [190] for our four application case studies using different square, power-of-two input sizes. We use a logarithmic scale on the y-axis.

In Figure 38, we observe for all four applications that our CoT search space representation in ATF requires a significantly smaller memory footprint than the one-dimensional search space representation used in CLTune (note the logarithmic scale in the figure). For example, application `CCSD(T)`'s one-dimensional search space representation used in CLTune requires for input size 2^{12} the prohibitively high amount of $>10^{19}$GB (calculated according to the formula above), while ATF's CoT search space structure requires only 256KB for storing the same space – a memory consumption reduced by a factor of $>10^{22}$.

EXPLORING CONSTRAINED SEARCH SPACES Figure 39 shows the advantages of ATF's multi-dimensional exploration strategy (introduced in Section 5.5) over a one-dimensional strategy (as used in CLTune), drawn as box plots. Each plot shows 10 runtimes of a particular study, obtained after 10 independent auto-tuning runs of 4h each. A box depicts the 25% – 75% quartiles, i.e., half of the configurations obtained after the 10 independent auto-tuning runs achieve a runtime that lays within the box. The vertical lines connect for each study the runtime of the worst auto-tuning run (i.e., the final configuration after 10 runs that leads to the highest runtime for the study) with the runtime of the best run. We auto-tune all applications for NVIDIA Tesla V100 GPU using the input sizes listed in Section 5.7.2. Note that different tuning runs usually find different final configurations, because search techniques are not deterministic; for example, techniques usually start exploration at a random configuration within the search space.

Figure 39 shows that when exploring a constrained search space in multiple dimensions (as described in Section 5.5), by exploiting the structure of ATF's CoT search space representation, we find better-performing parameter configurations for our studies in the same auto-tuning time (4h in the figure) as compared to the traditional, one-dimensional exploration strategy used in CLTune.

For the simulated annealing search – CLTune's most efficient search technique [190] – we observe considerably better tuning results for the multi-dimensional exploration strategy in ATF, compared to exploration in only one dimension used in CLTune: an average speedup of $10^2\times$, i.e., we can improve the runtime of our case studies on average by $10^2\times$ when using ATF for 4h of auto-tuning as compared to auto-tuning the applications for 4h using CLTune. This is because simulated annealing requires a long time for selecting the next configuration when the search space is one-dimensional and large [65]. When exploiting ATF's CoT structure, we perform exploration in multiple dimensions; in each dimension, we explore a corresponding level of the CoT structure, rather than exploring the large search space represented as one-dimensional structure of entire configurations only. This allows more configurations to be explored and results in better configurations being found in 4h of tuning time.

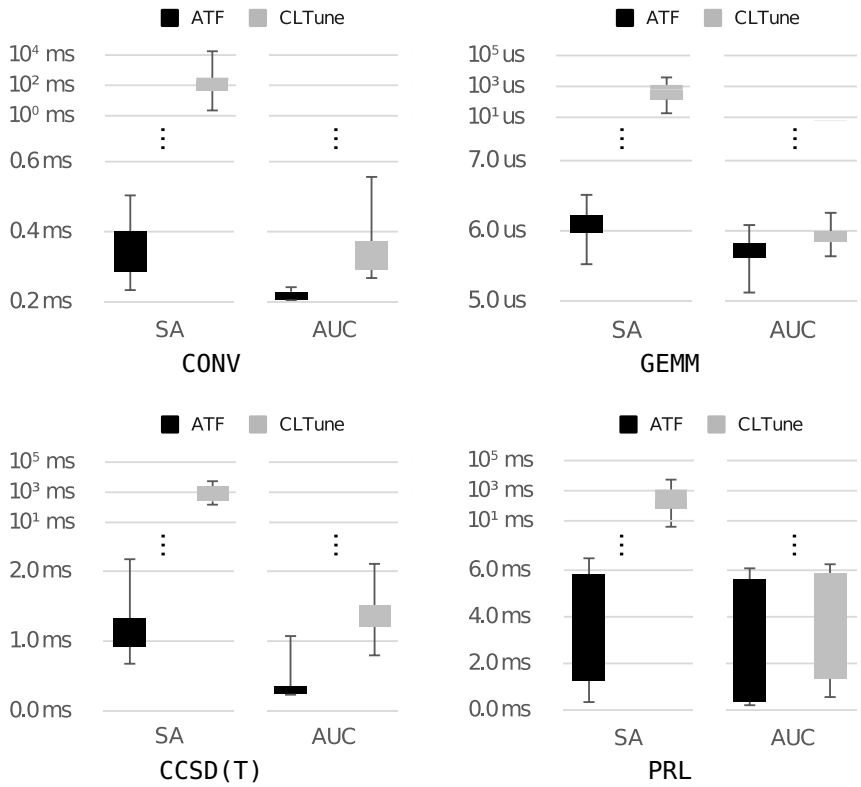


Figure 39: Search space exploration efficiency (lower is better) of ATF vs. CLTune [190] for our four application case studies using search techniques *Simulated Annealing (SA)* and *AUC Bandit (AUC)*. We use a logarithmic scale on the y-axis.

We also observe in Figure 39 better exploration efficiency for the AUC Bandit search – currently one of the most efficient search techniques [195] – when relying on ATF's multi-dimensional exploration approach. This is because exploration in multiple dimensions enables better exploiting locality information within the space's particular dimensions [290], which is especially beneficial for large spaces, e.g., the space of `CCSD(T)`.

Note that our CoT search space structure allows at least the same exploration efficiency as the traditional exploration strategy in one dimension, for any search technique: search techniques can straightforwardly access our CoT search space structure in a one-dimensional fashion (exactly as in CLTune) by iterating over the CoT's leaves.

	Domains	Applications
1	Compiler Optimization	GCC Flags [114], SIMD Vectorization [106]
2	Data Mining	Probabilistic Record Linkage [116]
3	Quantum Chemistry	CCSD(T) [115]
3	Deep Learning	BLAS [114, 115]
4	Sparse Computations	SpMM, SpMV, TTV [2]
5	DSL Compiler Optimizations	Lift [93, 125], MDH [115, 138], RISE [2]
6	Polyhedral Compilation	PPCG [125]
7	Signal Processing	FFT [93]
8	Stencil Computations	Conv2D [114, 115], Jacobi2D/3D, ... [105, 115, 125]
9	FPGA Programming	BFS, Audio, PreEuler [2]

Table 5: Applications auto-tuned using ATF.

5.7.5 *ATF for Further Application Classes*

ATF has been successfully used for auto-tuning applications from different important domains, summarized in Table 5. For example, ATF has proved to achieve tuning results of the same or even higher quality as OpenTuner and CLTune for application classes that are favorable for them [114]: i) GCC compiler's optimization options [195] (favorable for OpenTuner), as an example application whose tuning parameters have no interdependencies among them, and ii) GEMM implementation of the CLBlast library [131] (favorable for CLTune) whose tuning parameters have interdependencies.

Compared to OpenTuner, this is because GCC's tuning parameters have no interdependencies, causing both OpenTuner and ATF to generate, store, and explore the same (unconstrained) search space, and because ATF provides (among others) the highly-efficient AUC Bandit search technique which is also used by OpenTuner for search space exploration. Consequently, both OpenTuner and ATF achieve for GCC the same good auto-tuning results.

As compared to CLTune, ATF is able to auto-tune the CLBlast's GEMM routine to better performance than CLTune, by up to 17× (as shown in [114]), even though CLTune is specifically designed toward auto-tuning this routine [190]. This is because the CLTune user has to massively hand-prune the ranges of GEMM's tuning parameters (i.e., remove valid values out of the ranges) in order to generate CLTune's search space in acceptable time (similarly as in Section 5.7.4). However, such pruning massively shrinks GEMM's search space, by factors $> 1000\times$, thereby usually losing well-performing configurations within the space [114].

5.7.6 *ATF for a Real-World Application*

ATF can significantly speedup real-world applications that rely on compute-intensive kernels like those discussed in Section 5.7.3. We demonstrate this for the real-world example `siamese` which is used for handwriting recognition within the popular deep-learning framework Caffe [201].

No.	Input Size			num calls	Intel Xeon Gold 6140 CPU				
					ATLAS runtime	CLBlast		ATF	
	M	N	K			runtime	speedup over ATLAS	runtime	speedup over ATLAS
1.	50	64	500	8420128	0.2760	0.4720	***0.58***	0.0366	***7.55***
2.	20	576	25	8420128	0.0848	0.1165	***0.73***	0.0281	***3.01***
3.	20	576	1	8420128	0.0263	0.0539	***0.49***	0.0102	***2.57***
4.	50	64	1	8420128	0.0038	0.0160	***0.24***	0.0025	***1.54***
5.	500	64	50	6400000	0.2705	0.2147	***1.26***	0.0224	***12.06***
6.	50	500	64	6400000	0.2668	0.2539	***1.05***	0.0455	***5.86***
7.	20	25	576	6400000	0.0612	0.5655	***0.11***	0.0269	***2.28***
8.	64	500	800	100002	1.1326	2.3305	***0.49***	0.1581	***7.17***
9.	64	10	500	100002	0.0724	0.4678	***0.15***	0.0224	***3.23***
10.	64	500	1	100002	0.0419	0.0652	***0.64***	0.0157	***2.67***
11.	64	10	1	100002	0.0047	0.0109	***0.43***	0.0012	***4.03***
12.	64	2	10	100002	0.0031	0.0176	***0.18***	0.0008	***3.91***
13.	64	2	1	100002	0.0015	0.0095	***0.16***	0.0012	***1.25***
14.	500	800	64	100000	1.0922	1.2813	***0.85***	0.0863	***12.65***
15.	64	800	500	100000	1.1098	1.9104	***0.58***	0.0850	***13.06***
16.	64	500	10	100000	0.0749	0.0889	***0.84***	0.0227	***3.30***
17.	10	500	64	100000	0.0686	0.1517	***0.45***	0.0312	***2.20***
18.	64	10	2	100000	0.0053	0.0122	***0.43***	0.0017	***3.14***
19.	2	10	64	100000	0.0036	0.0218	***0.16***	0.0013	***2.66***
20.	100	500	800	20200	1.5435	2.3467	***0.66***	0.2584	***5.97***
21.	100	10	500	20200	0.0753	0.4875	***0.15***	0.0256	***2.95***
22.	100	500	1	20200	0.0629	0.0687	***0.92***	0.0173	***3.64***
23.	100	10	1	20200	0.0073	0.0141	***0.52***	0.0017	***4.21***
24.	100	2	10	20200	0.0049	0.0279	***0.18***	0.0017	***2.87***
25.	100	2	1	20200	0.0025	0.0125	***0.20***	0.0011	***2.18***

Table 6: Auto-tuning efficiency of ATF for the `siamese` neural network (runtimes in ms) on CPU.

No.	Input Size			num calls	NVIDIA V100 GPU				
					cuBLAS runtime	CLBlast		ATF	
	M	N	K			runtime	speedup over cuBLAS	runtime	speedup over cuBLAS
1.	50	64	500	8420128	0.0133	0.0331	***0.40***	0.0061	***2.17***
2.	20	576	25	8420128	0.0123	0.0213	***0.58***	0.0041	***3.00***
3.	20	576	1	8420128	0.0061	0.0198	***0.31***	0.0041	***1.50***
4.	50	64	1	8420128	0.0072	0.0199	***0.36***	0.0041	***1.75***
5.	500	64	50	6400000	0.0082	0.0225	***0.36***	0.0061	***1.33***
6.	50	500	64	6400000	0.0225	0.0224	***1.00***	0.0061	***3.67***
7.	20	25	576	6400000	0.0143	0.0345	***0.42***	0.0061	***2.33***
8.	64	500	800	100002	0.0195	0.0440	***0.44***	0.0195	***1.00***
9.	64	10	500	100002	0.0133	0.0329	***0.40***	0.0061	***2.17***
10.	64	500	1	100002	0.0061	0.0198	***0.31***	0.0041	***1.50***
11.	64	10	1	100002	0.0061	0.0195	***0.31***	0.0041	***1.50***
12.	64	2	10	100002	0.0123	0.0200	***0.62***	0.0041	***3.00***
13.	64	2	1	100002	0.0072	0.0193	***0.37***	0.0041	***1.75***
14.	500	800	64	100000	0.0123	0.0296	***0.41***	0.0164	***0.75***
15.	64	800	500	100000	0.0174	0.0375	***0.46***	0.0215	***0.81***
16.	64	500	10	100000	0.0133	0.0212	***0.63***	0.0041	***3.25***
17.	10	500	64	100000	0.0184	0.0215	***0.86***	0.0051	***3.60***
18.	64	10	2	100000	0.0072	0.0199	***0.36***	0.0041	***1.75***
19.	2	10	64	100000	0.0174	0.0215	***0.81***	0.0041	***4.25***
20.	100	500	800	20200	0.0256	0.0500	***0.51***	0.0256	***1.00***
21.	100	10	500	20200	0.0133	0.0325	***0.41***	0.0061	***2.17***
22.	100	500	1	20200	0.0061	0.0203	***0.30***	0.0041	***1.50***
23.	100	10	1	20200	0.0072	0.0197	***0.36***	0.0041	***1.75***
24.	100	2	10	20200	0.0113	0.0200	***0.56***	0.0041	***2.75***
25.	100	2	1	20200	0.0072	0.0201	***0.36***	0.0041	***1.75***

Table 7: Auto-tuning efficiency of ATF for the `siamese` neural network (runtimes in ms) on GPU.

Tables 6 and 7 (left parts of tables) show our analysis of `siamese` on GPU (Table 7) and CPU (Table 6). We observe that GEMM is called in `siamese` over 50 million times in total, on 25 input sizes (which remain fixed for different inputs of `siamese` [202]). For computing GEMM on CPU, the `siamese` implementation in Caffe relies on the ATLAS library [296] which uses a self-provided special-purpose auto-tuner for optimization; for GEMM on GPU, `siamese` uses the hand-optimized NVIDIA cuBLAS library which we discussed in Section 5.7.3. Alternatively to ATLAS and cuBLAS, the Caffe user can choose the CLBlast library for both GPUs and CPUs (also discussed in Section 5.7.3), which relies on CLTune for auto-tuning.

Tables 6 and 7 (right parts) show that the `siamese` application requires for computing GEMM on CPU via ATLAS in total 2.1h, which makes up 53% of `siamese`'s total runtime on CPU (3.9h); on GPU, `siamese` requires 10min for GEMM via cuBLAS, which is 83% of `siamese`'s total GPU runtime (13min). However, when replacing ATLAS and cuBLAS by the ATF-optimized GEMM implementation in [115] (which we discussed in Section 5.7.3), we can speedup `siamese`'s total runtime by 1.78× on CPU (to 2.2h) and by 1.85× on GPU (to 7 min). This is because ATF is capable of auto-tuning the GEMM implementation in [115] to higher performance than ATLAS and cuBLAS, by up to 13× on CPU and 4× on GPU, as confirmed in Tables 7 and 6.

ATF achieves better performance than ATLAS, because ATLAS relies on small search spaces and ignores correlations among tuning parameters by auto-tuning parameters independently of each other; most likely, this is done in ATLAS to simplify the auto-tuning process. Compared to cuBLAS, our better performance is because we rely on auto-tuning for the particular input size, while cuBLAS uses hand-crafted heuristics optimized toward average high-performance over various sizes, thereby avoiding the auto-tuning overhead. However, auto-tuning is an amortized one-time overhead in many application areas. For example, in `siamese`, the same 25 input sizes (listed in Tables 6 and 7) are re-used in each program run, such that auto-tuning becomes an acceptable one-time overhead per target architecture only.

5.8 COMPARISON TO RELATED WORK

We classify auto-tuning approaches into two major categories: 1) *special-purpose* approaches which are designed and optimized toward a particular application class (e.g., linear algebra routines or stencil computations), and 2) *general-purpose* approaches which target a broad range of arbitrary (possibly emerging/upcoming) application classes.

Special-purpose auto-tuning has proved to achieve impressive tuning results for important applications, including: FFT [276], DSP [278], chemistry computations [275], linear algebra [230, 296], self-adaptive architectures [184], geophysics [152], code mapping [215], multi-GPU systems [258], hardware synthesis [208], compiler optimization [241, 268, 280], code variant tuning [108, 109, 204], networks-on-chip [213], stencil computations [188, 239], resource virtualization [174], loop optimization [263], shared memory multiprocessors [301], load balancing [137], threading models [134, 163], machine learning [127], graph analytics [158], dynamic parallelism [154], and execution policies [140]. However, programmer's productivity is severely hindered in special-purpose auto-tuning, because special-purpose auto-tuners have to be designed and implemented specifically for any particular application class.

The alternative approach, general-purpose auto-tuning, aims at tackling this productivity issue, by providing the programmer with a general framework for conveniently generating special-purpose auto-tuners. The popular general-purpose auto-tuning frameworks currently include: OpenTuner [195], CLTune [190], KernelTuner [112], and `libtuning` [100]. These approaches generate efficient special-purpose auto-tuners for applications whose tuning parameters are either independent of each other (OpenTuner and `libtuning`) or have small ranges of tuning parameters (CLTune and KernelTuner). However, the approaches have weaknesses regarding auto-tuning recent parallel applications for state-of-the-art architectures, because such applications rely on interdependent tuning parameters with large ranges [115]. OpenTuner and `libtuning` often fail for such applications because, by design, they assume tuning parameters to be independent of each other. In contrast, CLTune and KernelTuner are designed toward interdependent tuning parameters, but they struggle with large ranges for such parameters, because they rely on straightforward mechanisms for generating, storing, and exploring the search spaces of these parameters. The weaknesses of the state-of-the-art general-purpose approaches are discussed in Sections 5.3-5.5 and confirmed experimentally in Section 5.7.

classic approaches to general-purpose auto-tuning of programs with interdependent tuning parameters are ActiveHarmony [259] and Orio [189, 257]. ActiveHarmony uses for generating its constrained search space the constraint solver that we discussed in Section 5.7.4. However, ActiveHarmnoy suffers from similar high search space generation time as CLTune whose time is higher than the time we present for the solver in Figure 37. This is because ActiveHarmony internally relies on search space constraints, similarly as CLTune, thereby hindering the solver from achieving its full performance potential (which is still lower than ATF's efficiency for search space generation, as confirmed in Figure 37). Furthermore, ActiveHarmony suffers from a high memory footprint and a time-intensive search space exploration phase. This is because ActiveHarmony uses search techniques to explore the unconstrained search space and whenever an invalid configuration is found, it maps the configuration to a valid configurations in

its (previously generated) constrained search space using the *Approximate Nearest Neighbour (ANN)* search [292]. However, ANN has two major drawbacks when used in auto-tuning: 1) high memory footprint: for `d` tuning parameters and a search space size of `n`, ANN requires `O(d·n)` memory space, similarly as CLTune, while ATF's memory footprint is in general substantially smaller (Figure 38); 2) time-intensive initialization: ANN requires additional, significant initialization time, of `O(d·n·log(n))`, for preparing its internal data structures. In contrast to ActiveHarmony, we introduce an exploration strategy for ATF toward directly exploring the constrained search space, based on our novel CoT search space structure introduced in Section 5.4, thereby avoiding the memory and time intensive process of ANN.

The other classic approach Orio supports interdependent parameters, by exploring the unconstrained search space (similarly as OpenTuner and `libtuning`) and setting a penalty value for invalid configurations; thereby, Orio avoids generating and storing the entire search space [189] as inherently required by our ATF approach (as well as CLTune and KernelTuner). However, by relying on the unconstrained search space, Orio suffers from the same severe weaknesses as discussed and experimentally shown in this chapter for OpenTuner and `libtuning`.

We present the Auto-Tuning Framework (ATF) which addresses the weaknesses in state-of-the-art general-purpose auto-tuning for programs with interdependent tuning parameters via novel mechanisms for *generating*, *storing*, and *exploring* the search space of interdependent tuning parameters. Our new mechanisms are especially important for online auto-tuning where auto-tuning is performed at program runtime: our novel search space generation mechanism contributes to significantly lower program initialization time than competitors (Figure 37), because in online auto-tuning, the search space is usually generated at program start based on runtime values (e.g., the input size). Moreover, our exploration mechanism finds well-performing configurations faster (Figure 39), thereby further contributing to lower program runtime. Related approaches, such as ActiveHarmony, when used for the special case of online auto-tuning rely on penalty values to avoid their time-intensive processes of constrained search space generation and exploration (discussed above). However, relying on penalty values is unfeasible when aiming at auto-tuning modern parallel applications, as we discussed in detail and showed experimentally in this chapter at the examples of OpenTuner and `libtuning`.

5.9 SUMMARY

The Auto-Tuning Framework (ATF) is a general-purpose auto-tuning approach for programs with interdependent tuning parameters. This chapter presents novel mechanisms for efficiently generating, storing, and exploring the search spaces of such parameters. Compared to the state-of-the-art general-purpose auto-tuning frameworks, ATF's new contributions improve each particular phase of the auto-tuning process: 1) ATF generates the search spaces of interdependent tuning parameters faster, 2) ATF requires less memory for storing these spaces, and 3) ATF achieves a higher exploration efficiency for such spaces. Our experiments confirm that ATF substantially enhances general-purpose auto-tuning as compared to the state of the art, and ATF enables efficiently auto-tuning applications from popular application domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

6

CODE EXECUTION VIA HIGH-LEVEL PROGRAMMING ABSTRACTIONS

Program code in state-of-the-art programming approaches, such as OpenCL and CUDA code (e.g., as generated by the MDH approach introduced in Chapter 4 and optimized via ATF which is introduced in Chapter 5) requires so-called *host code* for its execution. Efficiently implementing host code is often a cumbersome task, especially when targeting systems with multiple nodes, each comprising multiple, different devices, such as multi-core CPU and GPU: the programmer is responsible for explicitly managing and synchronizing data in the memory regions of different nodes and devices, synchronizing computations with data transfers between devices of potentially different nodes, and for optimizing data transfers, e.g., by using so-called *pinned main memory* [231] for accelerating data transfers and overlapping the transfers with computations. The programmer is also responsible for using and combining multiple programming models, e.g. MPI for inter-node programming, OpenCL for programming CPUs, and CUDA for GPU programming, which further complicates implementing host code for the programmer.

In this chapter, we introduce our *Host-Code Abstraction (HCA)*[1] approach, currently implemented as the C++ library *dOCAL (distributed OpenCL/CUDA Abstraction Layer)* for OpenCL and CUDA[2].

HCA combines important advantages over the state-of-the-art high-level approaches, such as providing a uniform high-level programming interface for node and device programming, automatically managing node-to-node communications, and performing host-code optimizations transparently for the user (such as overlapping data transfers with computations). We show that our HCA approach significantly simplifies the development of host code for systems consisting of multiple nodes and devices (as well as for single-device systems), and our experiments confirm that HCA imposes a negligible runtime overhead only.

[1] https://hca-project.org

[2] Our HCA approach is designed such that it can be easily extended to other target programming models, e.g., OpenMP, as we discuss in Chapter 10.

6.1 INTRODUCTION

We consider modern systems (a.k.a. *heterogeneous multi-device systems*) which may comprise one or several nodes, each equipped with potentially multiple, different devices, such as multi-core CPU and accelerator devices, such as GPUs. State-of-practice approaches to programming such systems are, e.g., MPI in combination with OpenCL and CUDA (a.k.a. *MPI+X* [177]). A common problem of these approaches is that they require the programmer to implement the so-called *host code* for managing MPI and executing OpenCL and CUDA device code (a.k.a. *kernel*).

Implementing host code is often a tedious task: boilerplate low-level commands are required, e.g., for allocating memory on the target device and for performing data transfers between the device's memory and node's main memory. Especially when targeting complex systems which consist of multiple nodes, each equipped with different devices, e.g., two or more GPUs and CPU, host code's implementation becomes cumbersome and error-prone even for experienced programmers: the memory regions of different devices of potentially different nodes have to be explicitly managed by the programmer, and data transfers have to be synchronized with kernel computations running on different devices.

Host code development becomes additionally complex for systems equipped with devices from different vendors: e.g., non-NVIDIA devices are often programmed using OpenCL, while NVIDIA devices mostly rely on CUDA for performance reasons [147] and because CUDA provides better profiling and debugging tools [349]. Therefore, to program a system with both NVIDIA and non-NVIDIA devices, the programmer has to mix CUDA and OpenCL host code and explicitly program the communication between CUDA and OpenCL data structures, e.g., to combine the results of different GPUs (computed using CUDA) on a multi-core CPU (using OpenCL). When the devices belong to different nodes, the OpenCL and CUDA host code has also to be mixed with a multi-node programming model, such as MPI, again increasing complexity.

To achieve high host code performance, the code must be optimized: using *pinned* and/or *unified memory* (a.k.a. *zero-copy buffer* in OpenCL terminology) can accelerate, hide or even avoid data transfers between devices' memory regions and nodes' main memories [200, 231]. However, using these specially-optimized memory regions requires from the programmer expert knowledge about low-level OpenCL/CUDA host code functions and flags (which we discuss later), thus making host code programming even more tedious and challenging.

We introduce the *Host-Code Abstraction (HCA)* to address the challenges described above – a high-level approach to host code programming, currently implemented as the C++ library *dOCAL (the distributed OpenCL/CUDA Abstraction Layer)* for OpenCL and CUDA devices that may belong to different nodes. HCA combines major advantages over the state of the art: 1) it simplifies implementing host code, by automatically managing low-level details such as data transfers and synchronization via a uniform high-level programming interface hiding the complexity of low-level APIs (such as MPI, OpenCL, and CUDA); 2) it works for arbitrary application classes by being designed toward executing arbitrary device code (a.k.a. *kernel code*); 3) it enables conveniently targeting the devices of multi-node systems by automatically managing the node-to-node communication; 4) it simplifies host code optimizations, e.g., by providing different, specially-allocated memory classes, such as pinned main memory for overlapping data transfers with computations, and automatically detecting and avoiding unnecessary data transfers; 5) it enables interoperability, e.g., between OpenCL and CUDA host code by automatically handling the communication between OpenCL and CUDA data structures and by automatically translating between the OpenCL and CUDA kernel programming languages. Moreover, HCA is i) compatible with existing programming libraries (such as NVIDIA cuBLAS and cuDNN), ii) it supports interconnecting with auto-tuning systems, and iii) it allows conveniently profiling the runtime behavior of programs.

The remainder of Chapter 6 is organized as follows. In Section 6.2, we illustrate the usage of our HCA approach, using a simple single-node example. Afterward, in Section 6.3, we demonstrate HCA's interoperability feature. In Section 6.4, we show how HCA is used for multi-node systems. After presenting advanced features of HCA in Section 6.5, we present our experimental results in Section 6.6. Section 6.7 compares our HCA approach to related work, and Section 6.8 concludes Chapter 6.

6.2 ILLUSTRATION OF HCA

We illustrate HCA using a simple, demonstrative example: summing all elements of a vector (a.k.a. *reduction*) on multiple devices, such as GPUs and/or CPUs. In the following, we first show in Section 6.2.1 how HCA is used for programming CUDA host code using all of the system's CUDA-capable GPUs. Afterward, Section 6.2.2 shows how HCA is used for programming OpenCL host code, which supports devices of different architectures and vendors, e.g., also Intel CPU, AMD CPU/GPU, etc.

```
1  __global__ static
2  void reduceKernel(float *d_Result, float *d_Input, int N)
3  {
4      const int     tid = blockIdx.x * blockDim.x + threadIdx.x;
5      const int threadN = gridDim.x * blockDim.x;
6      float         sum = 0;
7
8      for (int pos = tid; pos < N; pos += threadN)
9          sum += d_Input[pos];
10
11     d_Result[tid] = sum;
12 }
```

Listing 33: NVIDIA's original CUDA kernel for reduction taken from [349].

6.2.1 *HCA for Programming CUDA Host Code*

Listing 33 shows the original CUDA *reduction* kernel provided by NVIDIA in [349]; we aim to execute this kernel via HCA host code in the following. The kernel takes as input a vector `d_Input` of `N`-many floating point numbers (line 2), and it computes in parallel a partial sum of the vector's elements – one result per started thread (lines 4-9); the partial results are stored in `d_Result` (line 11) and are then summed up to the final result in the host code (shown in Listing 34 and discussed in the following) after kernel's execution.

Listing 34 shows for completeness in CUDA's standard host programming language an excerpt of the low-level host code for executing the reduction kernel in Listing 33 cooperatively on all of system's CUDA-capable devices; the CUDA host code in Listing 34 is provided by NVIDIA in [349] and required when not using using HCA for executing the CUDA kernel in Listing 33. The host code in Listing 34 contains boilerplate low-level functions, such as `cudaMalloc` and `cudaMallocHost` (lines 13-16) required for allocating device and main memory, `cudaMemcpyAsync` (lines 23 and 25) for performing data transfers between main memory and devices' memories, `cudaStreamCreate` (line 12) for creating the so-called *CUDA streams* which are required to coordinate data transfers and the execution of kernels on the CUDA devices, and `cudaStreamSynchronize` (line 31) for synchronization.

```
int main(int argc, char **argv)
{
    // initialization
    int i, j, gpuBase, GPU_N;
    cudaGetDeviceCount(&GPU_N);

    /* ... prepare input data ... */

    // Allocate device and host memory
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i);
        cudaStreamCreate(&plan[i].stream);
        cudaMalloc((void **)&plan[i].d_Data, plan[i].dataN * sizeof(
            float));
        cudaMalloc((void **)&plan[i].d_Sum, ACCUM_N * sizeof(float));
        cudaMallocHost((void **)&plan[i].h_Sum_from_device, ACCUM_N *
            sizeof(float));
        cudaMallocHost((void **)&plan[i].h_Data, plan[i].dataN *
            sizeof(float));
        for (j = 0; j < plan[i].dataN; j++)
            plan[i].h_Data[j] = (float)rand()/(float)RAND_MAX;
    }
    // Perform data transfers and start device computations
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i);
        cudaMemcpyAsync(plan[i].d_Data, plan[i].h_Data, plan[i].dataN
            * sizeof(float), cudaMemcpyHostToDevice, plan[i].stream);
        reduceKernel<<<BLOCK_N, THREAD_N, 0, plan[i].stream>>>(plan[i
            ].d_Sum, plan[i].d_Data, plan[i].dataN);
        cudaMemcpyAsync(plan[i].h_Sum_from_device, plan[i].d_Sum,
            ACCUM_N *sizeof(float), cudaMemcpyDeviceToHost, plan[i].
            stream);
    }
    // combine GPUs' results
    for (i = 0; i < GPU_N; i++) {
        float sum;
        cudaSetDevice(i);
        cudaStreamSynchronize(plan[i].stream);
        sum = 0;
        for (j = 0; j < ACCUM_N; j++)
            sum += plan[i].h_Sum_from_device[j];
        *(plan[i].h_Sum) = (float)sum;
        cudaFreeHost(plan[i].h_Sum_from_device);
        cudaFree(plan[i].d_Sum);
        cudaFree(plan[i].d_Data);
        cudaStreamDestroy(plan[i].stream);
    }

    /* ... Compare GPU and CPU results ... */
}
```

Listing 34: Excerpt of NVIDIA's original CUDA host code taken from [349] for executing the CUDA reduction kernel in Listing 33. Boilerplate low-level commands make the development of host code tedious and cumbersome for the CUDA programmer.

Listing 35 demonstrates the HCA host code that is equivalent to the NVIDIA's low-level host code in Listing 34, i.e., the HCA code in Listing 35 has exactly the same semantics as the CUDA host code in Listing 34. We have implemented HCA as a C++ header-only library, thereby freeing the user from the burden of compiling, packaging and installing; to use HCA, the user only includes the corresponding header file (line 1) and implements a C++ program which performs four major steps, 1.–4., discussed in the following.

```
1  #include "HCA.hpp"
2
3  int main()
4  {
5    int N = /* arbitrary chunk size */;
6
7    // 1. choose devices
8    auto devices = HCA::get_all_local_devices<HCA::CUDA>();
9
10   // 2. declare kernel
11   HCA::kernel reduceKernel = HCA::cuda::source(/* reduction kernel */)
         ;
12
13   const int GS = 32, BS = 256;
14
15   // 3. prepare kernel inputs
16   HCA::buffer<float> in ( N     * devices.size() );
17   HCA::buffer<float> out( GS*BS * devices.size() );
18
19   std::generate( in.begin(), in.end(), std::rand );
20
21   // 4. start device computations
22   for( auto& dev : devices )
23     dev.kernel( reduceKernel )
24        .execution_configuration( HCA::dim3(GS), HCA::dim3(BS) )
25        .data( HCA::write(out.begin() + dev.id()* GS*BS, GS*BS),
26               HCA::read(  in.begin() + dev.id()* N    , N    ),
27               N ).compute();
28
29   auto res = std::accumulate( out.begin(), out.end(), std::plus<float
         >() );
30
31   std::cout << res << std::endl;
32 }
```

Listing 35: The HCA host code for executing the CUDA reduction kernel in Listing 33. As compared to low-level CUDA host code (Listing 34), HCA frees the user from using boilerplate low-level commands, thus making the host code simpler.

1. CHOOSE DEVICES. In HCA, system's devices are represented as objects of the class `HCA::device`; they allow the user to conveniently perform device computations, as we demonstrate later in Step 4.

In our HCA host program in Listing 35, we execute the reduction kernel in Listing 33 on all of system's CUDA-capable devices, according to the CUDA host code in Listing 34. For this, we use the HCA function `get_all_local_devices<HCA::CUDA>` (line 8 in Listing 35) which constructs one `HCA::device<HCA::CUDA>` object per system's CUDA device, and it returns the constructed HCA device objects in the form of a C++ standard vector. For constructing the device objects, HCA automatically interacts with the low-level CUDA API to automatically determine and manage the target devices' CUDA ids (Listing 34, lines 4-5, 11, 22, 30) and to initialize and handle the low-level CUDA streams (lines 12, 23-25, 31, 39) – per default, HCA uses 32 streams per device, thus enabling simultaneously executing multiple kernels on a device and consequently a better hardware utilization (a.k.a. *Hyper-Q* in NVIDIA terminology [216]). The device id and CUDA streams are encapsulated in the HCA device objects to hide them from the user.

The user can also choose a specific CUDA device. For this, the user initializes `HCA::device<HCA::CUDA>` object by using either 1) device's name as string, e.g., `"Tesla K20"`, 2) its numerical CUDA device id, or 3) some of its device properties, e.g., the first found device with support for double precision and atomic operations.

2. DECLARE KERNELS. The HCA user declares an object of class `HCA::kernel` (Listing 35, line 11) for each CUDA kernel to be executed on one of system's CUDA devices. HCA kernels are currently initialized by the kernel's source code in its string representation[3], using either 1) the HCA-provided function `cuda::source` (line 11), or 2) function `HCA::cuda::path` to use the path to kernel's source file. If the source code contains only a single kernel, HCA automatically extracts kernel's name using the C++ regular expression library [362]; otherwise, the user passes the target kernel's name to the HCA kernel object. Optionally, the user can also pass CUDA compiler flags to the HCA kernel, e.g., `-maxrregcount` to specify the maximum number of registers to use, or `-D name=definition` to replace in kernel's code each textual occurrence of `name` by `definition`.

By passing kernels in their string representation to HCA, we enable *Just-In-Time (JIT)* compilation and thus benefiting from runtime values (a.k.a. *multi-stage programming* [302]) for a better performance. For example, the user can replace the input size `N` in kernel's code (Listing 33, line 8) by its actual value (Listing 35, line 5), thereby enabling more aggressive compiler optimizations, e.g., loop unrolling. For such replacement the user can conveniently use the CUDA compiler flag `-D` (discussed above). Internally, the HCA kernel class contains pre-implemented low-level code – based on NVIDIA's *Runtime Compilation Library (NVRTC)* [39] – which is automatically called by

[3] In future work, we aim to additionally allow passing kernels as common C++ code, rather than strings, inspired by [191, 197], as also discussed in Chapter 10.

HCA for compiling the CUDA kernel code. To minimize the cost for runtime compilation, HCA stores the compiled kernels in the HCA kernel object, and also on the system's hard drive, and reuses them for further computations; this happens transparently for the user.

3. PREPARE KERNEL INPUTS. CUDA kernels take as their input the values of fundamental types (e.g., `int` and `float`), vector types (e.g., `int2` and `float4`) and/or device buffers, i.e., pointers to a contiguous range of memory on a particular device (a.k.a. *device array* in CUDA). While values of fundamental and vector types are passed straightforwardly to a kernel, passing buffers requires preparation and thus programming effort from the CUDA user: the special low-level functions `cudaMalloc` and `cudaFree` (Listing 34, lines 13-14, 37-38) have to be used for allocating/de-allocating memory on the target device, and function `cudaMemcpyAsync` (lines 23 and 25) is used for transferring data between main memory and devices' memories. The effort for programming in CUDA increases for complex applications where a buffer's content is read and/or written on multiple devices, e.g., the partial results of one device are combined in parallel on another device. For such complex applications, the programmer is in charge of explicitly managing multiple buffers – one per device – and performing the device-to-device data transfers. Moreover, synchronization is a further challenge that has to be explicitly managed by the CUDA programmer, via a careful management of multiple CUDA streams. For example, a data transfer from main memory to a device's memory has to be completed before a kernel on that device reads the data, and the kernel has to be finished before its computed data are transferred from the device's memory to main memory (Listing 34, lines 12, 23-25, 31, 39). For the complex applications where devices' computations have interdependencies, e.g., the result of one device is used as input on another device, the user has to also use and manage so-called CUDA *events* which are created as synchronization points in the different devices' streams. Events have to be carefully managed by the user to avoid race conditions, which becomes especially challenging when multiple streams are used per device (as done in HCA for a better hardware utilization – see discussion before).

To free the user from the burdens of preparing and managing low-level CUDA buffers and explicitly handling synchronization, HCA provides the high-level buffer class `HCA::buffer` – it represents a portion of data that can be used for kernel computations on any CUDA device of the system. For this, an HCA buffer encapsulates one low-level CUDA buffer per used device as well as a region of main memory – the buffers and main memory mirror the same data. The HCA buffer class automatically manages the encapsulated CUDA buffers and main memory region by: 1) allocating memory on a device when the buffer is used for kernel computations on that particular device (see Step 4) and by de-allocating the memory when the buffer is destructed; 2) updating an encapsulated low-level CUDA device buffer or main memory before reading or writing it, by automatically iden-

tifying and performing the corresponding data transfers; 3) managing synchronization across multiple streams, i.e., HCA ensures transparently for the user that device and/or main memory can be simultaneously read but not be simultaneously written or read and written, and HCA ensures correct synchronization for complex applications with interdependent device computations, by automatically generating and maintaining a data-dependency graph. HCA buffers contribute to high host code efficiency, by allocating device memory on demand only and performing data transfers only when required, based on its automatically managed data-dependency graph, as we discuss later in more detail.

An HCA buffer (Listing 35, lines 16-17) is passed to an HCA device object (lines 25-26) to use the buffer's data as kernel input, and the buffer is accessed in the host code via a convenient user interface that we have designed as analogous to the interface of the well-known C++ standard vector type [362]. HCA is implemented to be compatible with the popular C++ *Standard Template Library (STL)*. For example, we use the STL function `std::generate` (Listing 35, line 19) to conveniently fill the HCA buffer `in` with random numbers, and we use STL function `std::accumulate` (line 29) to combine the GPUs' partial results in our HCA host code after kernels' execution. In our reduction example (Listing 35), the HCA buffer `in` (line 16) contains the input values – `N` random floating point numbers (line 19) per device, according to the original CUDA example in [349] – and the HCA buffer `out` (line 17) is used for storing the devices' partial results.

4. START DEVICE COMPUTATIONS. To start computations on a device, the user chooses an HCA device object (this is described in Step 1) and passes to it: i) the `HCA::kernel` to be executed (declared in Step 2), ii) the kernel's *execution configuration* – in the case of CUDA, the number of thread blocks and threads per block (a.k.a. *grid* and *block size*), and iii) kernel's input arguments, i.e., values of fundamental/vector types such as `float` and `float4`, and/or HCA buffer objects which represent low-level CUDA buffers (prepared in Step 3). HCA then uses the pre-implemented CUDA code of the high-level HCA classes to automatically allocate devices' memories and main memory, perform data transfers, compile and execute the kernel, etc.

In our reduction example (Listing 35), we process equally-sized chunks of the input, cooperatively on all of system's CUDA-capable devices (line 22), according to the NVIDIA's host code (Listing 34, lines 10, 21, 28). For this, we pass to each HCA device object (Listing 35, lines 22, 23): 1) the HCA reduction kernel (line 23), 2) the kernel's corresponding grid and block size `GS` and `BS` (line 24) which we have chosen (line 13) according to the NVIDIA's sample in Listing 34, and 3) kernel's three input arguments (lines 25-27). The input arguments are: the input buffer `in` (line 25) comprising the floating point numbers to be summed up, the output buffer `out` (line 26) in which the kernels' partial results are stored – one per thread – and the device's input size `N` (line 27). Since each device accesses only a chunk of buffers `in` and `out`, we pass, in a common STL-style, also

C++ *iterators* to chunk's first element – returned by function `begin()` – summed with the corresponding offset, and the chunk size, i.e., `GS*BS` elements in the case of buffer `out` (line 25) and `N` elements in the case of buffer `in` (line 26). Alternatively to the chunk size, the user can use an iterator pointing to chunk's end. By setting for each device an iterator to its corresponding chunk, HCA avoids the costly transferring of the entire buffers `in` and `out` between main memory and a device's memory and only transfers one chunk per device and buffer.

For high host code performance, we have implemented HCA functions (such as `.kernel` and `.compute` in lines 23 and 27 of Listing 35) as asynchronous, i.e., the control returns immediately to the main thread which only blocks when one of the kernel's output buffers is accessed in the HCA host code. To differentiate between kernels' input and output buffers, HCA provides the user with three different *buffer tags*: `read`, `write` and `read_write` (Listing 35, lines 25-26). The tags indicate to HCA how the executed kernel accesses a buffer, allowing HCA to internally generate and maintain a data-dependency graph. The graph enables HCA automatically: 1) coordinating device computations, e.g., a kernel computation does not start until other computations on its input/output buffers have been finished, and 2) minimizing unnecessary data transfers, e.g., HCA avoids data transfers from main memory to a device's memory or between different devices' memories if a buffer is only written by the kernel or if the data have been transferred previously to the device (a.k.a. *lazy-copy* [250]), and HCA avoids transferring the data back after kernel's execution if buffer was only read and thus not modified by the kernel. For example, in Listing 35, analogously to the NVIDIA's hand-optimized low-level host code in Listing 34, the content of buffer `out` is not copied to devices' memories by HCA as it is tagged with `write` (Listing 35, line 25) and as such not read by the devices, and the buffer `in` is not copied from devices' memories to main memory as it is only read by the kernel. HCA automatically blocks the main thread (in Listing 35, line 29) where kernel's output buffer `out` is accessed by function `begin()`; the computation of the main thread continues when the kernel finishes and its result is transferred by HCA from devices' memory to main memory, so that the result becomes accessible for function `begin()`.

6.2.2 *HCA for Programming OpenCL Host Code*

Analogously to its high-level host code interface for CUDA used in Listing 35, HCA provides an interface to simplify programming OpenCL host code. For example, for executing the OpenCL reduction kernel provided by NVIDIA in [353] (which is equivalent to the CUDA kernel in Listing 33), the user only has to slightly modify the HCA CUDA host code in Listing 35, as follows: 1) replace the HCA tag `HCA::CUDA` used for function `get_all_local_devices` (in line 8) by tag `HCA::OpenCL` to acquire all OpenCL-compatible devices from HCA, and 2) set the HCA kernel object (in line 11) to the OpenCL kernel's source code using the HCA-function `opencl::source`. HCA then automatically performs the low-level OpenCL commands for executing the OpenCL reduction kernel on all of system's OpenCL-capable devices which may be of different vendors (NVIDIA, Intel, AMD, ARM, etc. [342]). All HCA optimizations for CUDA host code, e.g., using multiple streams (a.k.a. *command queue* in OpenCL terminology) for better hardware utilization, avoiding unnecessary data transfers, and caching kernel binaries for reducing the overhead of JIT compilation, are also provided by HCA for OpenCL.

6.3 INTEROPERABILITY IN HCA

HCA supports developing host code for programs that require multiple programming models for their implementation, e.g., CUDA for GPU computations and OpenCL for computations on the CPU. For such programs, HCA allows to arbitrarily combine HCA host code for different models – we call this *interoperability*. For example, in HCA, a buffer with the results of a CUDA kernel can be passed to an OpenCL device object in order to be further processed in parallel on the system's multi-core CPU. Furthermore, HCA allows executing a CUDA kernel on an OpenCL device to achieve portability [223], e.g., performing a CUDA kernel on an Intel multi-core CPU. HCA also allows executing an OpenCL kernel on a CUDA device for higher performance: CUDA compilers often generate more efficient machine code than OpenCL compilers for NVIDIA devices [147]. For this, HCA automatically performs source-to-source translation between the CUDA and OpenCL kernel programming languages. Our source code translation engine is currently a proof-of-concept implementation that is based on the C++ regular expression library [362] and has some limitations: advanced C++ features such as automatic type deduction and template meta programming are currently not supported.

In the following, we discuss interoperability using the examples of CUDA and OpenCL for both directions – *CUDA to OpenCL* as well as *OpenCL to CUDA*.

```
1  HCA::device<HCA::OpenCL_CPU> cpu;
2
3  int NUM_CORES = /* CPU's number of cores    */;
4  int SIMD_VL   = /* CPU's SIMD vector length */;
5
6  HCA::buffer cpu_res( NUM_CORES*VL );
7
8  cpu.kernel( reduceKernel )
9     .execution_configuration( HCA::dim3(NUM_CORES), HCA::dim3(SIMD_VL)
          )
10    .data( write( cpu_res ), read( out ), out.size() ).compute();
11
12 auto res = std::accumulate( cpu_res.begin(), cpu_res.end(), std::plus<
       float>() );
```

Listing 36: HCA's interoperability feature allows using OpenCL for summing up GPU's partial results obtained with CUDA, in parallel on system's multi-core CPU. This code is a replacement for line 29 in Listing 35.

CUDA TO OPENCL: Listing 36 demonstrates how HCA is used to utilize system's multi-core CPU in our reduction example of Listing 35: we use OpenCL to further sum the GPUs' partial results (obtained with CUDA) in parallel on system's multi-core CPU, rather than summing them only sequentially as done in line 29 of Listing 35 (and also in the original CUDA host code in Listing 34, lines 28-39). For this, we replace line 29 of our HCA program in Listing 35 by the CPU-parallel code presented in Listing 36. In this CPU-parallel code, we use system's multi-core CPU (line 1), and we declare buffer `cpu_res` (line 6) for storing CPU's partial results. We then start parallel computations on the CPU by passing to the HCA OpenCL device object: 1) the reduction kernel (line 8) – it comprises the CUDA device code in Listing 33 which is automatically translated by HCA to equivalent OpenCL code, according to HCA's interoperability feature, so that the code becomes executable on the multi-core CPU via OpenCL; 2) the execution configuration (line 9) which we choose as one thread group per CPU's core (line 3), and we choose the thread group size as CPU's SIMD vector length (line 4); 3) the kernel's input arguments (line 10) which are: i) buffer `cpu_res` for storing CPU's partial results (declared in line 6), ii) HCA buffer `out` (Listing 35, line 17), and iii) input size, i.e, the number of floating numbers in buffer `out`. The HCA buffer `out` contains the GPUs' partial results which are obtained with CUDA (Listing 35, line 25) and thus reside internally in a low-level CUDA data structure – HCA automatically copies the results, according to its interoperability feature, transparently for the user, to an OpenCL data structure so that HCA buffer `out` can be used by the OpenCL reduction kernel in line 10 of Listing 36.

Note that in Listing 36, we set the execution configuration (line 9), analogously to before (Listing 35, line 24), according to CUDA's approach as grid and block size using HCA function `dim3`. In OpenCL, the execution configuration (a.k.a. *NDRange*) is usually set as *global* and *local size* – the total number of threads and thread group size – which can be done in HCA by using the HCA function `nd_range`, rather than `dim3`. HCA allows the user to arbitrarily choose between either setting the execution configuration as grid and block size (using HCA's function `dim3`) or as global and local size (using function `nd_range`) for both CUDA and OpenCL device objects.

OPENCL TO CUDA: While we have shown in the previous paragraph that HCA's feature – using CUDA kernels for OpenCL devices – enables portability, we demonstrate in the remainder of this section that the other direction – from OpenCL to CUDA – often contributes to better kernel performance due to the usually higher efficiency of CUDA over OpenCL for NVIDIA devices [147]. In the following, we first show that HCA's interoperability feature allows conveniently executing OpenCL kernels on CUDA devices; afterward, we report experimentally confirmed performance advantages of executing OpenCL on CUDA devices, using the example the popular OpenCL GEMM kernel (GEneral Matrix Multiplication) provided by the CLBlast [131] library.

Listing 37 demonstrates that using HCA, the CLBlast's OpenCL GEMM kernel can be easily executed in the CUDA programming framework. As shown in line 5, the user only passes the kernel's OpenCL code (line 1) to an HCA CUDA device object (declared in line 3); HCA then automatically translates the OpenCL code to CUDA, and uses the CUDA framework for executing the translated kernel.

```
1 HCA::kernel opencl_gemm = HCA::opencl::source(/* OpenCL GEMM */);
2
3 HCA::device<HCA::CUDA> cuda_gpu( "Tesla K20" );
4
5 cuda_gpu.kernel( opencl_gemm )
6         .execution_configuration( /* ... */ )
7         .data( /* ... */ ).compute();
```

Listing 37: Using HCA for conveniently executing the CLBlast's OpenCL GEMM kernel in the CUDA framework for higher performance.

Figure 40 shows the measured speedups of the CLBlast OpenCL GEMM kernel on an NVIDIA Tesla K20 GPU; the bars show the speedup (higher is better) of the OpenCL kernel when executed via CUDA (as shown in Listing 37), as compared to executing the kernel via OpenCL, on the NVIDIA K20 device. We report results for the 20 input sizes used in the deep learning framework Caffe [202]; as concrete neural network, we use Caffe's *siamese* sample for handwriting recognition [185].

We observe in Figure 40 that executing the OpenCL CLBlast kernel via CUDA leads to speedups of up to 2× as compared to executing the kernel via OpenCL, because the CUDA compiler often generates more efficient NVIDIA machine code as compared to the OpenCL compiler [147]. The overhead for HCA's internal OpenCL-to-CUDA translation of the kernel code (250ms on our system – not included in our measurements in Figure 40) is negligible because once the kernel is translated from OpenCL to CUDA, it is automatically stored by HCA within the HCA kernel object (and also on the system's hard drive) and reused for each new kernel call – in the siamese sample, the GEMM kernel is called over $> 10^6$ times on each input size in Figure 40.

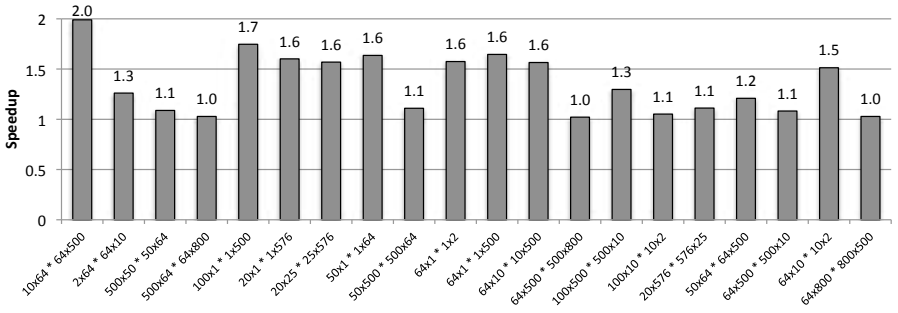


Figure 40: Speedup (higher is better) of CLBlast's OpenCL GEMM kernel [131] when executed via CUDA, rather than OpenCL, on an NVIDIA Tesla K20 GPU for the 20 input sizes used in the *siamese* sample of the deep learning framework Caffe [202].

6.4 HCA FOR MULTI-NODE SYSTEMS

For targeting multi-node systems (a.k.a. *distributed systems* or *clusters*), our HCA approach enables conveniently executing device computations on nodes that are connected via TCP/IP. For this, the user starts an HCA daemon process on the target nodes; HCA then automatically handles the required node-to-node data transfers and starts kernel computations on the nodes' devices, based on the `Boost.Asio` C++ networking library [332].

Our single-node example in Listing 35 can be easily extended to use the devices of all nodes available in the system: the user only replaces the function `HCA::get_all_local_devices<HCA::CUDA>()` in line 8 by function `HCA::get_all_devices<HCA::CUDA>()`; HCA then automatically acquires the devices of different nodes, transfers the devices' input and output data over the TCP/IP network, and synchronizes the different nodes' computations.

The user can also target specific remote devices, rather than using all devices available in the system. For this, an `HCA::device` object (see Section 6.2) is initialized additionally with the target node's name (or IP address), rather than with only the device name, device id, or device properties. For example, the user declares an HCA device as `HCA::device<HCA::CUDA>("gpu_node", 0)` to get the CUDA device with id `0` on the node with name `gpu_node`.

6.5 ADVANCED HCA USAGE

6.5.1 *Special Buffer Types*

In addition to its standard buffer type `HCA::buffer` (introduced and discussed in Section 6.2), HCA provides two further buffer types: 1) `HCA::pinned_buffer`, and 2) `HCA::unified_buffer`. Both are used the same as HCA's standard buffer type, but are internally optimized differently.

HCA's pinned buffer uses internally so-called *pinned main memory* [231] which enables fast data transfers between a node's main memory and its devices' memories, and pinned memory is also required for overlapping data transfers with device computations [232]. However, since pinned memory has a high allocation time, it should only be used if many data transfers are performed.

HCA's unified buffer type uses *unified memory* [149] which is beneficial when kernels access main memory sparsely and when the target device provides hardware support for unified memory. Especially when targeting CPUs, using unified memory (a.k.a. *zero-copy buffer* in OpenCL terminology) avoids data transfers between devices' memory and main memory, because for CPUs, devices' memories and main memory coincide [200].

Optimization guides [200, 231] often recommend the programmer to empirically test which allocation type – naive, pinned, or unified – suits best for their applications, dependent on the target hardware. However, testing these special allocation types – pinned and unified – requires a significant technical effort from the programmer. For example, for using pinned memory in low-level OpenCL host code, the user has to initialize an OpenCL-specific `cl_mem` object using the special flag `CL_MEM_ALLOC_HOST_PTR` and access the pinned memory region within the `cl_mem` object via special function `clEnqueueMapBuffer`. Moreover, the user is in charge of explicitly synchronizing the buffer (e.g., before it is read by a kernel), using the `clEnqueueUnmapMemObject` function, and use multiple *command queues* – the OpenCL equivalent to CUDA streams – to enable overlapping data transfers with computations [232]. HCA considerably simplifies the testing of different allocation types for the programmer, who only has to change the buffer type for testing.

The two optimized HCA buffer types automatically handle the inconvenient interactions with the low-level APIs of CUDA and OpenCL for allocating and using these special memory regions. Moreover, the user can easily switch between different allocation types, by only changing the HCA buffer type, e.g., from `HCA::buffer` to buffer type `HCA::pinned_buffer` for using pinned main memory, instead of the naively allocated memory region used by the HCA's standard buffer type.

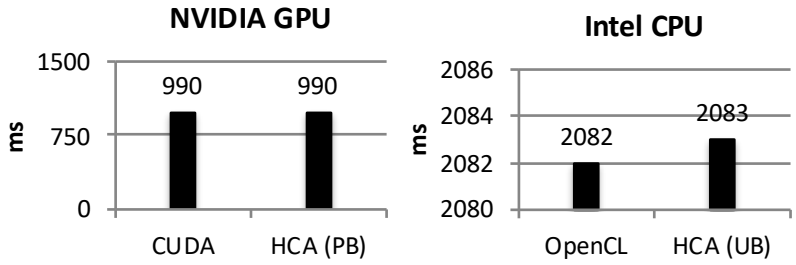


Figure 41: Runtime comparison (lower is better) of HCA with low-level CUDA and OpenCL host code for benchmarking pinned memory on NVIDIA GPU (left) and unified memory on Intel CPU (right). HCA achieves competitive performance with the low-level code.

Figure 41 (left) shows for an NVIDIA Tesla K20 GPU the runtime comparison of NVIDIA's benchmark `overlap-data-transfers` [232] with HCA using HCA's pinned buffer type; the benchmark computes trigonometric functions and is provided by NVIDIA to evaluate the performance of pinned main memory. Exactly as in the NVIDIA's implementation of the benchmark, we measure both the runtime for data transfers and the kernel's execution, but we ignore the runtimes for initializing OpenCL, compiling the kernel, etc. The experiments show that HCA achieves the same performance as NVIDIA's hand-implemented low-level CUDA host code: HCA's pinned buffers use internally pinned main memory, analogously to the NVIDIA's benchmark, thus enabling fast data transfers and overlapping the transfers with computations.

Figure 41 (right) shows for an Intel Xeon E5 CPU the runtime of Intel's OpenCL `ZeroCopy` benchmark [199] which is provided by Intel for evaluating unified memory – the benchmark computes *Ambient Occlusion* which is popular in the field of visual computing. We compare in the figure the benchmark's runtime to an equivalent HCA program which uses HCA's unified buffer type. Analogously to before, we measure only the runtime for data transfers and the kernel executions, for a fair comparison with our reference benchmark. We observe that HCA achieves competitive runtime with Intel's low-level OpenCL code. This is because HCA's unified buffers use, analogously to the Intel's benchmark, unified memory which enables avoiding data transfers when targeting CPU architectures (as discussed above).

6.5.2 *Compatibility with Existing Libraries*

There is a broad range of expert-implemented programming libraries, such as the OpenCL linear algebra library CLBlast [131] and the CUDA library cuDNN for Deep Learning computations [36]. To enable compatibility between HCA and such libraries, HCA's three buffer types (discussed in Sections 6.2 and 6.5.1) can be cast to their internally capsuled low-level buffer representation, e.g., a `cl_mem` object in the case of OpenCL and a `void*` pointer in the case of CUDA. This cast happens either automatically in HCA, and the low-level buffer is returned that belongs to the most recently used device. Alternatively, the user can use the HCA buffers' function `get_low_level_buffer`, e.g., `get_low_level_buffer<HCA::CUDA>(dev)` to get the CUDA buffer for the specific device `dev`, where `dev` is either an HCA device object, the device's name as string, or device's numerical CUDA device id. For OpenCL, function `get_low_level_buffer` is used with tag `HCA::OpenCL` instead of `HCA::CUDA`.

6.5.3 *Auto-Tuning Support*

HCA supports the user in finding well-performing values of parameters that are critical for kernel's performance, e.g., parameters for controlling the granularity of parallelism or the usage of fast memory resources, as thoroughly discussed in Chapter 4 of this thesis. For this, HCA allows to be conveniently interconnected with *auto-tuning systems* [117] which use advanced search methods to automatically explore a kernel's optimization space.

Auto-tuner can be conveniently generated using the *Auto-Tuning Framework (ATF)* which we discussed in detail in Chapter 5 of this thesis: the user annotates the kernel code with *tuning directives* which specify its performance-critical parameters by their: 1) types (e.g., `int` or `float`), 2) ranges of possible values, and 3) possible interdependencies (e.g., a parameter has to evenly divide another parameter). ATF then automatically generates the corresponding auto-tuner that optimizes the kernel for a target hardware and particular data characteristics.

For connecting HCA with an auto-tuner, the user provides to HCA the concrete auto-tuner for its kernel, e.g., generated using ATF, by storing the tuner to a corresponding path on the hard drive. HCA then manages transparently for the user the cumbersome tasks of: 1) calling the auto-tuner for each device on which the kernel is executed, 2) storing on the hard drive the auto-tuned kernel that is obtained by the auto-tuner, and 3) reusing the auto-tuned version of the kernel in each following kernel execution.

To incorporate runtime values into the auto-tuning process (e.g., the input size), as required for high-quality tuning results [156], the HCA user passes the concrete runtime values to the auto-tuner using the HCA function `auto_tuning`. For example, to auto-tune the reduction kernel in Listing 33 also for the input size `N` (Listing 35, line 5), the HCA user declares the kernel (in line 11) with the input size `N` as: `HCA::kernel reduceKernel = { HCA::cuda::source(/*...*/), auto_tuning(N) };` HCA then automatically forwards input size `N` to the auto-tuner.

6.5.4 *Kernel Profiling*

HCA enables conveniently profiling kernels, by automatically interacting with low-level profiling functions, such as `cudaEventRecord` and `cudaEventSynchronize` (for CUDA) or `clGetEventProfilingInfo` and `clWaitForEvents` (for OpenCL). To activate profiling in HCA, the user simply sets the C preprocessor macro `HCA_ENABLE_PROFILING`; HCA then automatically measures and outputs the runtimes for, e.g., initializating low-level objects, data transfers, and kernel compilations and executions. Additionally, HCA stores the measured runtimes in a JSON file – a popular file format for human-readable name-value pairs.

6.6 EXPERIMENTAL EVALUATION

All experiments described in this section can be reproduced using our artifact implementation [97].

We experimentally confirm that HCA simplifies implementing host code for both CUDA and OpenCL, with a low runtime overhead for abstraction. After describing our experimental setup in Section 6.6.1, we first report experimental results for a single-node system (Section 6.6.2) and afterward for a multi-node system (Section 6.6.3).

6.6.1 *Experimental Setup*

We use a system with two nodes, each equipped with two Intel Xeon E5-2640 v2 8-core CPUs, clocked at 2GHz with 128GB main memory and hyper-threading enabled, as well as two NVIDIA Tesla K20m GPUs; the two nodes are connected via an InfiniBand FDR network. We perform experiments using HCA for programming both CUDA for GPU programming and OpenCL for programming CPUs. A node's two CPUs are represented in OpenCL as a single device with 32 compute units, corresponding to the overall 2×16 logical cores in the node. For runtime measurements, we use the unix `time` command. As C++ compiler, we use `clang` version 3.8.1 with its `-O3` optimization flag enabled on the CentOS operating system version 7.4.

6.6.2 *Single-Node Experiments*

We perform our single-node experiments by comparing to all of the expert-implemented, multi-device code samples provided by NVIDIA and Intel for CUDA [349] and OpenCL [145], against equivalent HCA programs. The NVIDIA CUDA samples are: 1) `simpleMultiGPU` for reduction, 2) `MonteCarloMultiGPU` for a Monte Carlo experiment, and 3) `nbody` for N-body simulation. For OpenCL, we use the two Intel's samples: 1) `intel_ocl_multidevice_basic` for computing scaled dot product, and 2) `intel_ocl_tone_mapping_multidevice` for high dynamic range tone mapping. We compare each sample against the equivalent HCA program in terms of both code complexity and runtime.

We measure code complexity using four classic metrics: 1) Lines of Code (LOC), excluding blank lines and comments, 2) COCOMO development effort (DE) in person-months [305], 3) McCabe's cyclomatic complexity (CC) [324], and 4) the Halstead development effort (HDE) [323]. McCabe's cyclomatic complexity is the number of linearly independent paths through the source code, while the Halstead development effort metric is based on the number of operators and operands in the source code. Low cyclomatic complexity and Halstead development effort imply that code is simpler to develop and debug. We measure the metrics LOC and CC with the tool provided in [171], the DE with [279], and HDE with [175].

Sample	Code	LOC	DE	CC	HDE
Reduction	**CUDA**	110	0,26	14	19.980
	HCA	56	0,12	13	11.974
Monte-Carlo	**CUDA**	336	0,82	32	131.259
	HCA	190	0,45	24	76.337
N-body	**CUDA**	812	1,96	80	412.182
	HCA	434	1,03	37	226.962
Scaled-Dot-Prodcut	**OpenCL**	293	0,68	21	57.523
	HCA	54	0,12	8	10.729
HDR-Tone-Mapping	**OpenCL**	523	1,25	88	290.102
	HCA	246	0,57	32	114.451

Figure 42: Code complexity of the low-level CUDA and OpenCL host code samples from NVIDIA and Intel as compared to their HCA equivalents, using classic code metrics. The metrics indicate that HCA host code is significantly simpler than the low-level code.

Figure 42 compares the code complexity of the original CUDA and OpenCL samples from the vendors with their HCA counterparts. The kernel code is excluded in our measurements, because HCA and the OpenCL/CUDA samples use the same kernels. We observe that HCA programs are significantly simpler: on average they 1) require 2.72× fewer lines of code (LOC) in the case of OpenCL and 1.85× fewer lines in the case of CUDA, 2) require a 2.8× less development effort (DE) in the case of OpenCL and 1.9× less effort in the case of CUDA, 3) have a cyclomatic complexity (CC) that is reduced by a factor of 2.73× for OpenCL and 1.7× for CUDA, and 4) their Halstead development effort (HDE) is reduced by the factor 2.78× (OpenCL)

and 1.79× (CUDA). Even for simple applications, e.g., *scaled dot product* and *reduction* (which rely on one, simple kernel only), HCA programs are significantly simpler than their low-level OpenCL/CUDA equivalents, because of the boilerplate code required by the low-level approaches, e.g., for initializing OpenCL/CUDA and for performing data transfers. We observe that HCA programs achieve more reduction in complexity for OpenCL than for CUDA, because OpenCL requires more boilerplate commands to handle devices of different vendors while CUDA targets NVIDIA devices only.

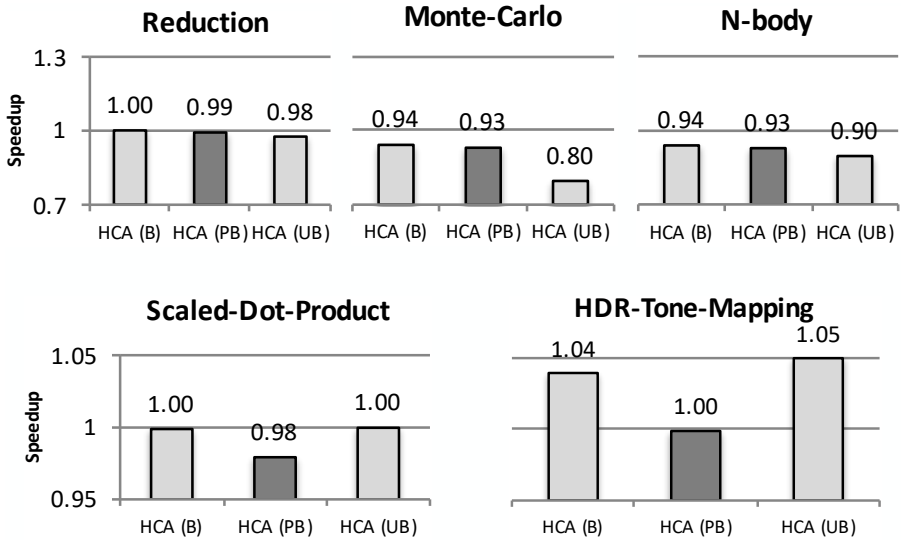


Figure 43: Speedup (higher is better) of HCA over NVIDIA's three CUDA samples [349] on two NVIDIA Tesla K20m GPUs (top part of the figure) and Intel's two OpenCL samples [145] on two Intel Xeon E5 CPUs (bottom part). We show experiments for each of HCA's three buffer types: buffer (B), pinned buffer (PB) and unified buffer (UB). The buffer type that corresponds to the memory used in the low-level samples is filled dark grey (which is PB for all samples). Speedups are computed using the median runtime of 30 runs. We observe that HCA's performance is competitive to low-level CUDA and OpenCL host code.

Figures 43 reports the speedups of our high-level HCA programs as compared to their corresponding low-level counterparts in CUDA (top part of the figure) and OpenCL (bottom part). We report results for each of HCA's three buffer types – buffer (B), pinned buffer (PB), and unified buffer (UB) – for which the CUDA and OpenCL documents [200, 231] recommend to naively test which type suits best for a particular combination of target application and hardware architecture. The low-level samples all use pinned main memory which corresponds to using HCA's pinned buffer type – we highlight this in the figure by filling the corresponding bars dark grey. The Intel's OpenCL samples run on our node's two Intel CPUs and the NVIDIA CUDA samples run on the node's two NVIDIA GPUs.

We observe that HCA's high-level approach causes a quite low runtime overhead of < 7% in comparison to CUDA and < 2% in comparison to OpenCL when using pinned memory for HCA (dark grey bars) as in the low-level samples. This is due to modern compilers efficiency – in our case, the `clang` compiler – which significantly optimize HCA's abstraction overhead, e.g., by performing optimizations like inline expansion [314]. For the two further HCA's buffer types – buffer and unified buffer – we observe the same or sometimes even slightly better performance of HCA as compared to the low-level samples. This is caused by the high allocation time for pinned memory which is used by the original NVIDIA and Intel samples. In contrast, HCA's standard buffer (B) and unified buffer (UB) types use straightforwardly allocated memory or unified memory, correspondingly, causing a lower allocation time (as discussed in Section 6.5.1). The better performance of HCA for OpenCL as compared to CUDA is because the OpenCL samples implement and use several helper functions, e.g., for selecting the OpenCL platform, which causes runtime overhead.

6.6.3 *Multi-Node Experiment*

We use the example of General Matrix Multiplication (GEMM) to demonstrate HCA's efficiency on multi-node systems. For this, we use the OpenCL GEMM kernel provided by NVIDIA in [353].

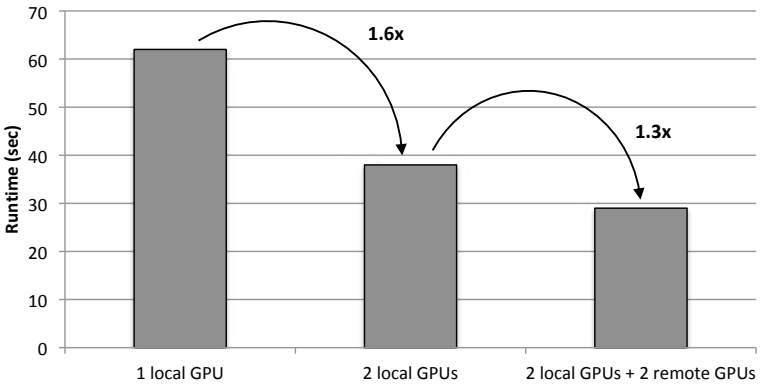


Figure 44: Runtime comparison (lower is better) of NVIDIA's General Matrix Multiplication (GEMM) in OpenCL when executed 1) on a single, local GPU, 2) on two local GPUs, and 3) on two local GPUs and two remote GPUs. Doubling the number of local GPUs speeds up performance by a factor of 1.6×; using in addition two remote GPUs increases performance further by a factor of 1.3×.

Figure 44 reports the GEMM's runtime on 16384×16384 input matrices of single precision floating point numbers (`float`) when executed 1) on a single, local GPU, 2) on both local GPUs, and 3) on all four GPUs of both nodes: two local GPUs (first node) and two remote GPUs (second node).

We observe in the figure that switching from a single local GPU to two local GPUs increases performance by a factor of 1.6×; when using the second node's two remote GPUs as well (i.e., four GPUs in total), performance is increased further by a factor of 1.3×. Performance increases more significantly when doubling the number of local GPUs, rather than doubling the number of remote GPUs, because using remote GPUs requires communication between different nodes. For example, in the case of GEMM, chunks of the input matrices have to be transferred over the network from the local node to the remote node, requiring nearly 5 seconds of runtime. If excluding this overhead, we would achieve again a speedup of nearly 1.6× (instead of a speedup of only 1.3×), i.e., the overhead for using the remote GPUs is mainly caused by the (inherent) node-to-node data transfers over the InfiniBand network of our system.

6.7 COMPARISON TO RELATED WORK

There are several successful high-level approaches to simplifying host code programming. However, these approaches focus on only particular host programming challenges, e.g., only data-transfer optimizations or only OpenCL or CUDA, respectively, thereby being restricted to particular application classes. For example, skeleton approaches [103, 141, 160, 176, 245, 250] simplify host code programming, but they are restricted to programs that can be expressed via so-called algorithmic skeletons [242] only. Directive-based approaches such as OpenACC [355], OpenMP [357], and OpenMPC [252] automatically generate the host code for the programmer, but they also automatically generate the kernel code, thereby preventing the programmer from hand-optimizing the kernels as often required for high performance [147]. The systems built on top of OpenCL – Maat [167], ViennaCL [253], Maestro [254], Boost.Compute [173], and HPL [193] – simplify executing user-defined OpenCL kernels, but they do not support CUDA. The approaches pyOpenCL and pyCUDA [228] allow implementing OpenCL/CUDA host code in the easy-to-use Python programming language, but they still require from the programmer to explicitly deal with low-level details, such as data transfers and synchronization. Multi-device Controllers [95], PACXX [197], SYCL [191], and OmpSs [240] allow conveniently programming OpenCL and/or CUDA-capable devices, while StarPU [237], PEPPHER [222] and ClusterSs [247] focus on simplifying task scheduling over multi- and many-core devices. However, these approaches do not support data-transfer optimizations, e.g., overlapping data transfers with computations. Moreover, the majority of the related work targets single-node systems only, thereby missing the full performance potential of modern computer systems with multiple nodes. Approaches SnuCL [226], rCUDA [249], dOpenCL [225] and LibWater [210] target multi-node systems, but they extend the low-level OpenCL or CUDA user API, rather than providing high-level abstraction to ease host programming.

6.8 SUMMARY

In this chapter, we present HCA – a high-level programming approach for conveniently implementing host code which is required in modern low-level programming approaches like CUDA and OpenCL for code execution. HCA conveniently targets multi-device and multi-node systems, by automatically managing different nodes' main memories and their devices' memories, performing node-to-node communication, handling synchronization, automatically minimizing data transfers, and supporting data transfer optimization between device and main memory. Furthermore, HCA allows interoperability between low-level programming models by automatically moving data between low-level data structures and by performing source-to-source translation between models' kernel languages. Our experiments on popular samples from NVIDIA and Intel confirm that HCA significantly simplifies host code programming, as compared to low-level approaches, while incurring only negligible runtime overhead for its higher level of abstraction.

7

USER INTERFACE: GENERATING/OPTIMIZING/EXECUTING CODE VIA THE MDH+ATF+HCA APPROACH

We offer two kinds of user interfaces for using our MDH+ATF+HCA approach introduced in Chapters 4-6 of this thesis. Our first interface kind relies on a *Domain-Specific Language (DSL)* for expressing computations; we have designed our DSL as close to our MDH formalism (presented in Chapter 4) such that formal reasoning in MDH also applies to DSL programs. Since users experience programming in the widely used C programming language usually as more amenable than DSL programming, we offer a second interface kind, based on auto-parallelization and code annotations of straightforward, sequential C programs. Our C-based user interface is implemented as a frontend for our DSL interface and incorporates internally techniques from *polyhedral compilation* [235].

We discuss both interface kinds in the following – our DSL-based interface in Chapter 7.1 and our interface based on auto-parallelization and the C programming language in Chapter 7.2.

7.1 A DOMAIN-SPECIFIC LANGUAGE FOR MDH+ATF+HCA

We offer a DSL that technically represents our MDH high-level program representation introduced in Chapter 4. The same as our formal MDH representation our DSL conveniently expresses data-parallel computations, such as linear algebra routines and stencil computations, agnostic of hardware and optimization details. In particular, by keeping the design of our DSL close to the MDH formalism, we ensure that all formal reasoning for MDH in Chapter 4 also applies to DSL programs.

In the following, we outline our DSL using the example of *Matrix-Vector Multiplication (MatVec)* which is presented in its formal MDH representation in Figure 12 of Chapter 4.

```
MatVec<T in TYPE | I,K in IN> := out_view<T>( w:(i,k)->(i) ) o
                                 md_hom<I,K>( *, (++,+) ) o
                                  inp_view<T,T>( M:(i,k)->(i,k),
                                                 v:(i,k)->(k)  )
```

Listing 38: DSL program for MatVec in MDH+ATF+HCA.

Listing 38 shows how MatVec is implemented in our DSL[1]. We observe from the listing that our DSL program for MatVec is very close to our formal MDH expression of MatVec in Figure 12 – the two representations differ only slightly in some syntactical aspects (e.g., using keyword `in` instead of math symbol $\in$). Consequently, we consider our DSL programmes to be an equivalent, technical representation of our formal MDH expressions presented in chapter 4.

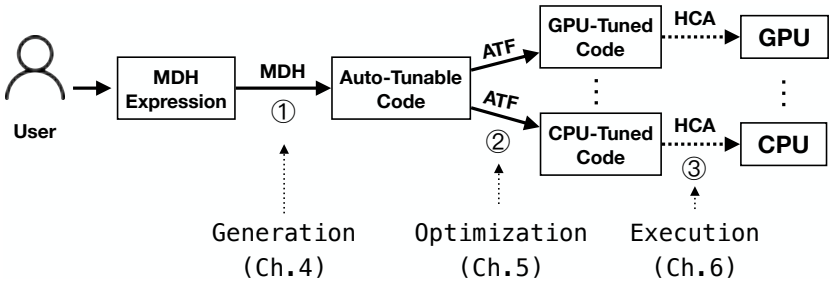


Figure 45: Overview of our `MDH+ATF+HCA` approach.

Figure 45 shows how we *generate*, *optimize*, and *execute* program code for our DSL programs (e.g., the program in Listing 38) using the MDH+ATF+HCA approach introduced in Chapters 4-6. In the figure, the DSL program – which is a straightforward, technical representation of its corresponding MDH expression (as discussed above) – is first used in step ① to *generate* auto-tunable program code, according to the MDH approach (Chapter 4). Afterward, in step ②, we use our ATF approach (Chapter 5) to *optimize* the auto-tunable code, generated in the previous step, for the particular target architecture and characteristics of the input and output data. The generated and optimized program code is finally *executed*, in step ③, on the target device using our HCA approach (Chapter 6). The scalar type `T` and sizes `I,K`, as well as configuration options (such as the user-desired tuning time in step ② and the particular target device in step ③) are set straightforwardly via command line parameters in our implementation of Figure 45. We aim to describe our technical implementation of MDH+ATF+HCA thoroughly in future work (see Part III).

[1] Our DSL implementation is currently a proof of concept that has technical limitations: it is implemented for combine operators *concatenation* and *point-wise combination* only (Examples 1 and 2 in Chapter 4), and it allows using for view functions (Definition 4.2.3) only index functions that are affine. Our future work aims to introduce a fully functional DSL language that implements all features of our MDH formalism presented in Chapter 4 (as outlined in Chapter 8).

7.2 AUTO-PARALLELIZATION VIA MDH+ATF+HCA

Achieving performance, portability, and productivity is challenging when programming modern parallel systems, e.g., via OpenCL or CUDA, because advanced mechanisms for generating, optimizing, and executing the code are required (as we thoroughly discuss in Chapters 4-6 of this thesis). Even when relying on high-level *Domain-Specific Languages (DSLs)* which hide from the user the complexity of hardware and optimization details, e.g., the DSL that we introduce in Chapter 7.1, non-expert users may feel overwhelmed due to the non-traditional nature of DSLs. Also, users with extensive practice in mainstream programming languages, e.g., the C language, experience programming in these traditional, well-known approaches as more intuitive and less challenging than DSL programming. A further obstacle for the use of DSLs is that large code bases are already implemented in C – rewriting these bases to DSL programs requires considerable effort.

Polyhedral compilers, such as *PPCG* [219] and *Pluto* [262], simplify parallel programming by automatically parallelizing sequential program code, e.g., implemented in the C programming language. For this, a polyhedral compiler extracts from the sequential program code the so-called *polyhedral model* – a formal representation of the code, based on concepts from mathematical geometry, which captures important information, such as the number of loop iterations and memory access relations (`read` and/or `write`). The extracted model is then optimized by the polyhedral compiler via so-called *affine transformations* which enable important optimizations, such as tiling.

State-of-the-art polyhedral compilers have weaknesses[2]: they are usually designed and optimized toward only a single particular architecture (e.g., only GPU), thereby limiting their applicability for other kinds of architectures. Moreover, we discuss and demonstrate experimentally later in Section 7.2.5 that polyhedral compilers sometimes even fail to achieve the full performance potential of their target architecture.

This section presents `md_poly` – a novel compiler that fully automatically generates portable, high-performance code for modern parallel architectures, such as GPUs and CPUs, from straightforward, sequential C code. For this, `md_poly` combines polyhedral techniques with our MDH+ATF+HCA approach for code generation, optimization, and execution (introduced in Chapters 4-6). In particular, we demonstrate that the formal program representation used in polyhedral compilers (polyhedral model) can be automatically transformed to an equivalent MDH expression (as introduced in Chapter 4) for MDH-supported computations, such as linear algebra routines and stencil computations. Consequently, the transformation allows using our MDH+ATF+HCA compilation pipeline from Figure 45 for generating, optimizing, and executing code for state-of-the-art parallel architectures for `md_poly`'s C-based input programs.

[2] Polyhedral compilers are thoroughly discussed in terms of performance, portability, and productivity in Chapter 4 of this thesis.

From a theoretical perspective, our findings show that regarding code generation, our MDH formalism is more expressive than the state-of-the-art polyhedral approach for MDH-supported computations, because the MDH representation captures more information relevant for optimization (as we discuss in this section).

We present experimental results on GPU and CPU for two popular case studies used in deep learning – *Multi-Channel Convolution (MCC)* and *Matrix Multiplication (MatMul)*. Our experiments confirm that `md_poly` achieves encouraging result as compared to the polyhedral compilers PPCG and Pluto which also fully automatically parallelize and optimize straightforward, sequential C code. We also present and discuss experimental results for approaches *OpenACC* [355] and *OpenMP* [357] which rely on user-defined code annotations for parallelizing and optimizing C programs.

7.2.1 *The `md_poly` Compiler: Overview*

Figure 46 depicts the overview of `md_poly`'s internal design. Starting from a sequential C program (requirements on the C program are discussed in Section 7.2.4), we first extract in step Ⓐ in the figure the polyhedral model – this is usually the same step in all C-based polyhedral compilers (e.g., PPCG) – using the existing *Polyhedral Extraction Tool (`pet`)* [235]. In the next step Ⓑ, the extracted polyhedral model is transformed into an equivalent MDH expression – this transformation step is the main scientific contribution of our `md_poly` compiler and discussed in the next section. Afterward, we use our DSL-based pipeline presented in Figure 45 to generate, optimize, and execute code from the obtained MDH expression, according to Chapter 7.1.

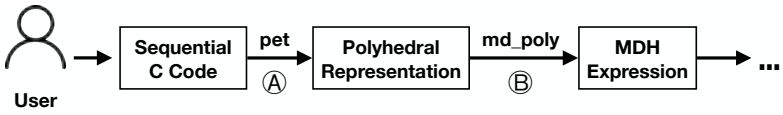


Figure 46: Overview of `md_poly`'s internal design.

7.2.2 *Transformation: Polyhedral Model to MDH Representation*

We show how the polyhedral model can be automatically transformed into an equivalent MDH expression (step Ⓑ in Figure 46). We present our transformation using the example of *Matrix-Vector Multiplication (`MatVec`)* (Listing 39).

```
1 for( int i = 0; i < I; ++i )
2   for( int k = 0; k < K; ++k )
3   {
4     w[i] += M[i][k] * v[k]; // Statement 1 (S1)
5   }
```

Listing 39: Sequential *Matrix-Vector Multiplication (MatVec)* in C.

7.2.2.1 *Polyhedral Model*

We extract the polyhedral model from the sequential C implementation of `MatVec` (Listing 39) straightforwardly using the *Polyhedral Extraction Tool (pet)* [235] (step Ⓐ in Figure 46). The model's two basic building blocks are the so-called *iteration domain* and *access relations*; we discuss both briefly in the following.

ITERATION DOMAIN An iteration domain represents the statements in the sequential C program, which is in the example of Listing 39 the single statement `S1` in line 4. Each statement is dependent on the particular values of the enclosing loop iterators – iterators `i` and `k` for statement `S1`. Consequently, the iteration domain of `MatVec` is (in polyhedral notation):

$$(I, K) \rightarrow \{\ S1(i,k) \ \mid 0 \leq i < I,\ 0 \leq k < K \ \}$$

ACCESS RELATION The access relation describes how data are accessed by statements – `read` and/or `write`. For example, for `MatVec`, matrix `M` and vector `v` represent the input data; both are read only in Listing 39 (line 4); vector `w` is read and written (line 4). Consequently, the access relation of `MatVec` is as follows (in polyhedral notation):

```
Read accesses:      S1(i,k) → w[i],M[i][k],v[k]
Write accesses:     S1(i,k) → w[i]
```

7.2.2.2 *MDH Representation*

We use the information captured within the polyhedral model (Section 7.2.2.1) to automatically generate a corresponding MDH expression (step Ⓑ in Figure 46) and consequently a DSL program for the MDH+ATF+HCA approach (discussed in Chapter 7.1). An MDH expression consists of higher-order functions `inp_view`, `out_view`, and `md_hom` (see Chapter 4); we instantiate the functions for `MatVec` based on `MatVec`'s polyhedral model (Section 7.2.2.1), as follows.

PATTERN `inp_view`: As input parameters of pattern `inp_view`, we have to extract from the polyhedral model the following information for `MatVec`:

1. *input buffers:* `w, M, v`
2. *index functions:* `w:(i,k)->(i), M:(i,k)->(i,k),v:(i,k)->(k)`

We can extract parameters 1.-2. straightforwardly from the polyhedral model's access relation – all data having `read` accesses represent input data.

PATTERN `out_view`: For pattern `out_view`, we need parameters:

3. *output buffers:* `w`
4. *index functions:* `w:(i,k)->(i)`

Analogously as for the parameters of pattern `inp_view`, we can extract parameters 3.-4. from the polyhedral model's access relation – all data having `write` accesses represent output data.

PATTERN `md_hom`: As parameters for pattern `md_hom`, we need:

5. *scalar function:* `f` (shown in Listing 40)
6. *combine operators:* $\circledast_1$ (for `i` loop) and $\circledast_2$ (for `k` loop)

which we define as follows.

Scalar Function Listing 40 shows the scalar function `f` of `MatVec`'s corresponding `md_hom` instance when generated automatically by `md_poly` according to `MatVec`'s polyhedral representation. The function's basic building block is statement `S1` (line 3 in Listing 40) taken from Listing 39, line 4; we extract the statement from the polyhedral model's iteration domain.

We set data with `read` accesses (for `MatVec`, these are: `w[i]`, `M[i][k]`, and `v[k]` – see Section 7.2.2.1) as the input arguments of function `f` (line 1 in Listing 40), and we return the value of variables with `write` accesses at the end of `f`'s function definition (line 5 in Listing 40). All information required for generating scalar function `f` is provided by polyhedral model's access relation (see Section 7.2.2.1).

```
1 T_OUT f( T_w w_i, T_M M_i_k, T_v v_k )
2 {
3   w_i += M_i_k * v_k; // Statement 1 (S1)
4
5   return w_i;
6 }
```

Listing 40: Scalar function `f` of `MatVec`.

Note that the automatically generated scalar function `f` for `MatVec` in Listing 40 is different from the hand-implemented scalar function for `MatVec` in Figure 39 (which is a straightforward multiplication $*$ only): i) the automatically generated function in Listing 40 takes as input also element `w_i` from the output vector `w`, according to line 4 of Listing 39, and ii) the function performs addition `+` (line 3) (whereas the hand-implemented scalar function in Figure 39 computes multiplication $*$ only), because combine operators different from concatenation ++ (e.g., combine operator `+`, as in the case of `MatVec` – see Figure 39) cannot be extracted automatically from the polyhedral model, as we discuss in the next paragraph.

Combine Operators Combine operators different from concatenation ++ (such as addition +) are not explicitly represented in the polyhedral model[3] [168, 178, 308] and thus cannot be extracted – automatically identifying such combine operators would require a complicated semantic analysis of the sequential code in Listing 39, which is impossible in general (Rice's theorem [274]). We provide two different solutions to circumvent this problem: 1) ignoring the parallelism potential in such dimensions (e.g., as in PPCG and Pluto); 2) requesting combine operators explicitly from the user: for example, in the case of `MatVec`, the user annotates the code in Listing 39 with the following (OpenMP-like [357]) directive `#pragma mdh ( w[i] : ++ , + )`, thereby explicitly stating combine operators ++ (for the `i`-dimension) and + (`k`-dimension) for `md_poly`.

7.2.3 *Polyhedral Model vs. MDH Representation*

Compared to a hand-implemented MDH expression, an MDH expression that is automatically generated from the polyhedral model, as described in Section 7.2.2.2, has restrictions: combine operators different from concatenation (e.g., addition `+`, as in the case of `MatVec`) cannot be used in the generated MDH expression, because such combine operators are not explicitly represented in the polyhedral model (as discussed in Section 7.2.2.2). This restriction is a limitation of the polyhedral model, because it prevents parallelization in non-concatenation dimensions and thus, it can negatively affect performance (as we confirm in our experiments in Section 7.2.5).

In contrast, every MDH expression (automatically generated as well as hand implemented) can be automatically transformed into an efficient polyhedral representation, because the MDH expression captures all information required for creating the polyhedral model.

Summarizing, the MDH representation captures more information relevant for high-performance code generation than the polyhedral model for MDH-supported computations.

7.2.4 *Current Limitations*

Our range of supported C input programs is currently limited: we require perfectly nested `for`-loops in the input program that have rectangular iteration domains, i.e., loops in the sequential C program need to start from 0, be incremented by 1 after each loop iteration, and have static loop bounds. Moreover, we require statically resolvable index functions for accessing arrays within the loop body, thereby being currently limited to computations on, e.g., dense data formats. Also, for buffers that are both read and written, we require that they are initialized by the neutral element of the corresponding combine op-

[3] We use standard polyhedral analysis [255] to check whether loop iterations are independent of each other (and thus having concatenation ++ as combine operator), or dependent of each other (requiring a combine operator different from concatenation that cannot be identified via polyhedral techniques in general [168, 178, 308]).

erator (e.g., by value 0 in the case of buffer w in Listing 39, which is the neutral element of addition +). However, we do not necessarily consider this requirement for read/write buffers as a limitation: for example, if the w output vector in Listing 39 is not 0-initialized, the program in Listing 39 would not compute classic MatVec, because the program in Listing 39 then would also include a vector addition (the result vector of MatVec is summed with the initial values of vector w). In our approach, we aim to express such computations – e.g., MatVec combined with vector addition – in a systematic way, as two separate building blocks (i.e., two C programs for md_poly, or two MDH expressions for our DSL-based interface in Chapter 7.1, respectively), for which we then generate one optimized low-level program code (so-called fusing, as discussed in more detail in Chapter 8).

Our future work aims to significantly widen the range of md_poly's target C input programs (as outlined in Chapter 8), e.g., by using *polyhedral transformations*, such as *loop skewing*, before step Ⓑ in Figure 46, to make iteration domains rectangular, and by extending the MDH approach toward supporting sparse data formats.

7.2.5 *Experimental Evaluation*

All experiments described in this section can be reproduced using our artifact implementation [54].

We experimentally evaluate our md_poly compiler on NVIDIA V100 GPU and Intel Skylake CPU by comparing md_poly to both:

1. *annotation-free* approaches (in Section 7.2.5.1): PPCG [219] and Pluto [262] which fully automatically parallelize and optimize straightforward, sequential C code;
2. *annotation-based* approaches (in Section 7.2.5.2): OpenACC [355] and OpenMP [357] which parallelize and optimize C code based on user-defined code annotations.

For completeness, we compare in Section 7.2.5.3 also md_poly when using code annotations against md_poly when not using annotations (see Section 7.2.2.2).

As case studies, we use two real-world computations that are popular in the area of deep learning: i) *Multi-Channel Convolution (MCC)*, and ii) *Matrix Multiplication (MatMul)* (both also discussed in Chapter 4 of this thesis).

In all our experiments, we use input data either taken from real-world deep learning neural networks (abbreviated by RW in the following) or sizes that are considered as preferable for our competitors, such as powers of two (abbreviated by PC). For our MCC study, these sizes are

1. RW: $(1 \times 7 \times 7 \times 512)$ images; $(512 \times 3 \times 3 \times 512)$ filters
2. PC: $(1 \times 4096 \times 4096 \times 1)$ images; $(1 \times 5 \times 5 \times 1)$ filters

and for `MatMul`

1. `RW`: (10×64) and (64×500) input matrices
2. `PC`: (1024×1024) and (1024×1024) input matrices

Experimental Setup

We use for experiments a system equipped with an Intel Xeon Skylake CPU Gold-6140 @ 2.30GHz and an NVIDIA Tesla V100-SXM2-16GB GPU. Our `md_poly` compiler generates CUDA code (for GPU) and OpenCL code (for CPU); we compile the generated codes using NVIDIA HPC SDK 22.1 for CUDA and Intel's OpenCL runtime version 18.1.0.0920 for OpenCL. We use frameworks `PPCG` version `0.08.2` and `Pluto` commit `12e075a`. The same as our approach, `PPCG` generates CUDA for GPUs (CPUs are not supported by `PPCG`); we compile the `PPCG`-generated CUDA code also using the CUDA compiler from NVIDIA HPC SDK 22.1, the same as for our approach. Compiler `Pluto` targets CPUs by generating OpenMP code; we compile the `Pluto`-generated OpenMP code using the compiler from Intel oneAPI Base Toolkit 2022.1.1. We use `OpenACC` code for GPUs, which we compile also using our system's NVIDIA HPC SDK 22.1. Our Pluto-independent `OpenMP` experiments are also compiled via Intel oneAPI Base Toolkit 2022.1.1.

Time measurements are made using the C++ chrono library and the CUDA/OpenCL profiling APIs, respectively.

The same as our `md_poly` compiler, approaches `PPCG` and `Pluto` allow auto-tuning the generated GPU and CPU code specifically for the particular target architecture and characteristics of the input and output data. We auto-tune each of these three frameworks (`md_poly`, as well as `PPCG` and `Pluto`) for each combination of computation and data characteristics generously for 12h, using the *Auto-Tuning Framework (ATF)* (introduced in Chapter 5 of this thesis). We use such generous tuning time to avoid auto-tuning issues in our experiments, because such issues are not a focus of our experiments (auto-tuning is thoroughly discussed and experimentally evaluated in Chapter 5 of this thesis).

7.2.5.1 *Annotation-Free Approaches*

Figure 47 reports the speedup of our `md_poly`-generated CUDA code over the CUDA code generated by `PPCG` (on GPU); the figure also reports the speedup of `md_poly`'s generated OpenCL code over the OpenMP code generated by `Pluto` (on CPU). Both `PPCG` and `Pluto` fully automatically parallelize C code, without requiring code annotations from the user. To fairly compare to `PPCG` and `Pluto`, we use for `md_poly` also annotation-free C code as input, thereby limiting `md_poly`'s performance potential: without code annotations, `md_poly` is not able to parallelize reduction dimensions (as discussed in Section 7.2.2.2).

Annotation Free	NVIDIA Volta GPU				Intel Skylake CPU			
	MCC		MatMul		MCC		MatMul	
	RW	PC	RW	PC	RW	PC	RW	PC
PPCG	483.58	17.82	1.00	1.12	–	–	–	–
Pluto	–	–	–	–	70.66	19.68	2.52	6.82

Figure 47: Speedup (higher is better) of md_poly's automatically generated, optimized, and executed code on GPU and CPU over PPCG and Pluto.

We observe from Figure 47 that md_poly often achieves significantly higher performance than PPCG and Pluto. This is because md_poly internally uses the MDH's optimization space (introduced and discussed in Chapter 4 of this thesis) which captures more optimization opportunities than the spaces' of PPCG and Pluto, which are both based on polyhedral transformations (the differences between MDH and polyhedral compilers are thoroughly discussed in Chapter 4). The MDH's large optimization space can be efficiently explored via the ATF auto-tuning framework (introduced in Chapter 5 of this thesis) which is used by md_poly for search space exploration (see Section 7.2.1). We achieve higher speedups over PPCG and Pluto for our MCC study than for MatMul, because MCC is more challenging to optimize than MatMul: MCC is implemented in C using a 7-layered loop nest, whereas the MatMul implementation relies on a nest of 3 loops only. Consequently, MCC requires advanced optimization, as introduced for MDH in Chapter 4 of this thesis, e.g., efficiently exploiting fast memory resources (such as GPU registers) and relying also on fine-grained parallelization instead of coarse grained-parallelization only as in PPCG and Pluto (as discussed in Chapter 4).

7.2.5.2 *Annotation-Based Approaches*

Figure 48 reports the speedup of md_poly's generated code over the code generated by OpenACC (for GPU) and OpenMP (for CPU) which both rely on user-provided code annotations for parallelization and optimization. In contrast to our experiments in Figure 47, we use in Figure 48 as input for md_poly C code that is annotated with directives (according to Section 7.2.2.2)[4], allowing md_poly to also parallelize reduction dimensions, the same as OpenACC and OpenMP.

[4]When using annotated C code as input for md_poly, we achieve with md_poly the same, encouraging experimental results as presented for the MDH approach in Chapter 4, because md_poly internally uses MDH for code generation (Section 7.2.1), and it generates well-performing MDH expressions when code annotations are provided by the user (as discussed in Section 7.2.2.2).

Annotation Based	NVIDIA Volta GPU				Intel Skylake CPU			
	MCC		MatMul		MCC		MatMul	
	RW	PC	RW	PC	RW	PC	RW	PC
OpenACC	1103.54	279.62	59.00	73.69	–	–	–	–
OpenMP	–	–	–	–	3475.44	196.58	4.98	62.61

Figure 48: Speedup (higher is better) of md_poly's automatically generated, optimized, and executed code on GPU and CPU over OpenACC and OpenMP.

We observe in Figure 48 better performance for our md_poly-generated code over the code generated and used by OpenACC and OpenMP. This is because OpenACC and OpenMP rely on only straightforward optimizations. Also, both approaches do not have an integrated auto-tuner for generating code that is optimized for the particular target architecture and characteristics of the input and output data. Thereby, both approaches avoid the overhead for auto-tuning, but at the cost of performance losses, which is in particular disadvantageous for deep learning computations, because auto-tuning is an only one-time overhead. The higher performance of md_poly for our MCC study than for MatMul is again because of the higher complexity of MCC compared to MatMul, thereby requiring advanced optimizations, e.g., as introduced and discussed for MDH in Chapter 4 of this thesis.

7.2.5.3 *md_poly: Annotation Free vs. Annotation Based*

Figures 49 and 50 report the performance of our md_poly-generated code for our case studies on GPU and CPU when using as input for md_poly: 1) plain C code that is not annotated and thus fully automatically optimized and parallelized (denoted as AF in the figures); 2) C code that is annotated with an MDH directive (denoted as AB) allowing advanced optimization (as discussed in Section 7.2.2.2).

We observe from the figures that code annotations by the user further speedup the performance of the md_poly-generated code. For example, for MCC on the RW input size on GPU (Figure 49), the annotation-based input AB for md_poly performs more than twice as good as the annotation-free input AF (note the logarithmic scale in the figure), and the AF input could already achieve a speedup > 400× as compared to PPCG (Figure 47).

Annotations are particularly important for reduction-heavy computations, such as MCC on the RW input sizes, which has large reduction dimensions, because annotations allow parallelizing such dimensions as required to achieve the full performance potential of modern parallel architectures, e.g., GPUs.

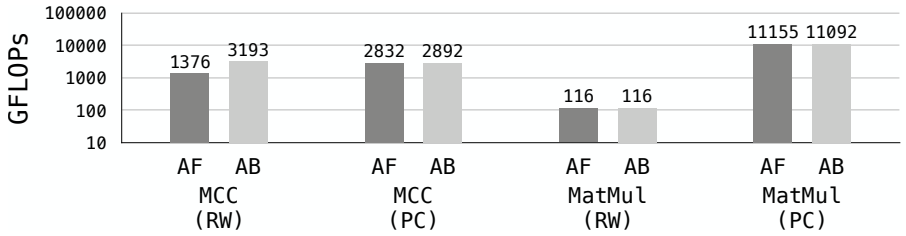


Figure 49: Performance (higher is better) of `md_poly` on NVIDIA Volta GPU when using as input C code that is *Annotation Free (`AF`)* vs. C code that is *Annotation Based (`AB`)*.

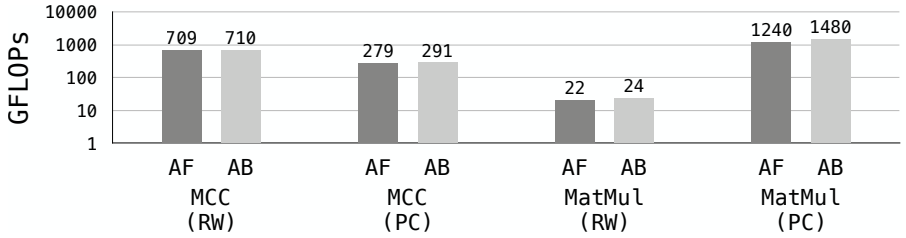


Figure 50: Performance (higher is better) of `md_poly` on Intel Skylake CPU when using as input C code that is *Annotation Free (`AF`)* vs. C code that is *Annotation Based (`AB`)*.

7.2.6 *Summary*

We present `md_poly` – a novel compiler that automatically generates portable high-performance code for GPU and CPU from straightforward, sequential C programs. For this, `md_poly` combines the MDH+ATF+HCA approach (introduced in Chapters 4-6 of this thesis) with techniques from polyhedral compilation – a formal approach based on concepts from mathematical geometry. We show that the polyhedral program representation can be automatically transformed into a corresponding MDH expression, allowing generating well-performing code for different kinds of target architectures using the MDH+ATF+HCA approach introduced in this thesis. From a theoretical perspective, our findings show that the MDH formalism is more expressive than the polyhedral approach regarding code generation and optimization for MDH-supported computations, because the MDH representation captures more information relevant for generating well-performing low-level program code (e.g., in CUDA and OpenCL).

Our experiments on GPU and CPU confirm for computations relevant in deep learning that `md_poly` achieves better performance than the PPCG and Pluto approaches which both also fully automatically parallelize and optimize straightforward, sequential C code. Our experiments also report better performance of `md_poly` over the popular frameworks OpenACC and OpenMP which parallelize and optimize programs based on user-provided code annotations.

Part III

FUTURE WORK

INTRODUCTORY REMARKS

This part outlines our work-in-progress results and ideas for further developing our concepts and methodologies introduced in Chapters 4-6, regarding code *Generation* (presented in Chapter 8), *Optimization* (Chapter 9), and *Execution* (Chapter 10).

Our intended future work is already funded by different institutions and projects:

1. DFG Project[5] (funding sum: 606,271€) for extending and optimizing our approach toward computations relevant in Deep Learning (DL).

2. HiPEAC Collaboration[6] (funding sum: 5,000€) ... to combine our approach with polyhedral compilation techniques, with the overall goal to achieve high user productivity.

Moreover, based on our work on automatic program optimization (Chapter 5), we were able to initiate and organize a workshop[7] that lead to interesting, ongoing collaborations, including [10].

Our work-in-progress results have also been presented and awarded at different venues, including:

1. `SC'21` – *Best Poster Finalist*: introducing MDH-based fusion optimization targeting Deep Learning (DL) computations

2. `CGO'20` – *SRC Gold Award*: combining our MHD+ATF+HCA approach with polyhedral compilation techniques toward higher user productivity

3. `PACT'20` – *SRC Gold Award*: using our MHD+ATF+HCA approach for stencil computations

4. `PUMPS+AI'19` – *Best Poster Award*: achieving performance, portability, and productivity via our MHD+ATF+HCA approach

5. `IHK'18` (German Chamber of Commerce and Industry): design and implementation of a performance portable BLAS library via our MDH+ATF+HCA approach

[5] German Research Foundation (DFG) – project (470527619): *Performance, Portability, and Productivity for Deep Learning Applications on Multi- and Many-Core Architectures (PPP-DL)*

[6] HiPEAC Collaboration Grant – project: *Productive Parallel Programming via Polyhedral Techniques and Multi-Dimensional Homomorphisms*, in collaboration with *Tobias Grosser (Univ. of Edinburgh, UK)*

[7] Lorentz Center Workshop, 2022, Leiden NL, titled: *Generic Autotuning Technology for GPU Applications*, organized by: Ben van Werkhoven, Jiří Filipovic, Ari Rasch, Gabriele Keller; https://www.lorentzcenter.nl/generic-autotuning-technology-for-gpu-applications.html

8 ENHANCE CODE GENERATION

We aim to advance our code generation approach (presented in Chapter 4) in our future work, as outlined in the following.

8.1 HIGH-LEVEL PROGRAM TRANSFORMATIONS

Our MDH high-level program representation (introduced in Section 4.2) captures semantic information that are hard (or often even impossible due to Rice's theorem [274]) to extract from executable program code (e.g., in OpenMP, CUDA, or OpenCL).

Our future work aims to exploit the semantic information captured in MDH's high-level representation to identify and express high-level program optimizations[1], via *functional program transformation rules* [316]. Such rules usually cannot be applied to executable program code, because extracting semantic information from executable low-level code is usually too challenging.

A simple, illustrative example is transforming two MDH high-level instances – both expressing embarrassingly parallel computations – to one instance (a.k.a. *map fusion*):

$$\texttt{md\_hom}(g,(+\!\!+,\ldots,+\!\!+)) \circ \texttt{md\_hom}(f,(+\!\!+,\ldots,+\!\!+)) = \texttt{md\_hom}(g \circ f,(+\!\!+,\ldots,+\!\!+))$$

We aim to identify and introduce more high-level program transformations (besides map fusion) in our future work. Our MDH approach offers a formalism for expressing and proving such transformations.

Encouraging work-in-progress results for MDH-based high-level program transformations are presented in [59].

8.2 TARGETING MULTIPLE DATA-PARALLEL COMPUTATIONS

Our MDH approach currently expresses individual data-parallel computations only, e.g., only one matrix multiplication or one convolution, correspondingly. Since real-world programs usually consist of multiple data-parallel computations, such as deep learning graphs [157] which often contain in the same program matrix multiplications and also convolutions, we aim to extend our MDH approach toward expressing simultaneously multiple data-parallel computations (instead of expressing each computation individually, as currently in MDH). Such an extension of MDH will enable applying MDH optimization across computations (a.k.a. *fusion* optimization [23, 206]), e.g., transferring data through fast memory resources between computations.

[1] In MDH, we refer to optimizations as *high-level* when they transform a program in MDH's high-level representation to another, semantically equal instance in the high-level representation of MDH.

Figure 51 illustrates the current MDH approach, as introduced in Chapter 4, for an example program (represented as a data-flow graph) consisting of computations *Matrix Multiplication (`MatMul`)* and *Convolution (`Conv`)*. For each of the two computations, the current MDH approach generates one individual kernel (e.g., in CUDA for GPU) that is specifically optimized for the corresponding computation, without taking into account other computations for optimization.

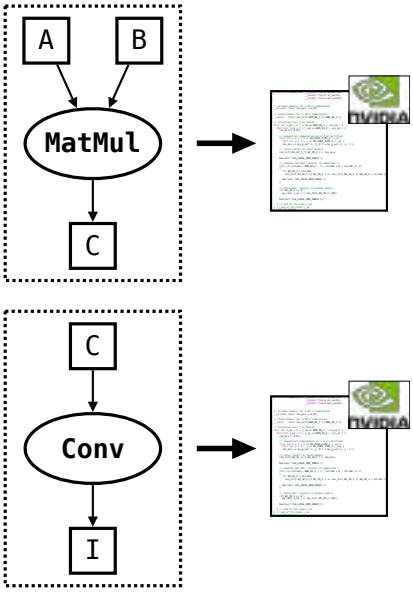


Figure 51: Current MDH code generation approach (introduced in Chapter 4): for each data-parallel computation (in this example, two computations: `MatMul` and `Conv`), MDH generates one individual kernel (e.g., in CUDA), thereby being unable to apply optimization across computations.

Figure 52 shows how our intended MDH code generation approach will work for the program[2] as in Figure 51 consisting of computations `MatMul` and `Conv`: our new, extended MDH approach will generate one single kernel for both computations. This enables applying optimization across the two computations, e.g., transferring the `C` matrix through a fast memory region and reducing the kernel launch overhead (to one launch, instead of two launches as in Figure 51).

[2] In MDH, we refer to an individual data-parallel computation (defined according to Chapter 4) as *MDH Computation* (such as one `MatMul` or one `Conv`), and we refer to a graph of MDH computations as *MDH Program* (e.g., as in Figure 52 which is a graph representing a program consisting of the two MDH computations `MatMul` and `Conv`).

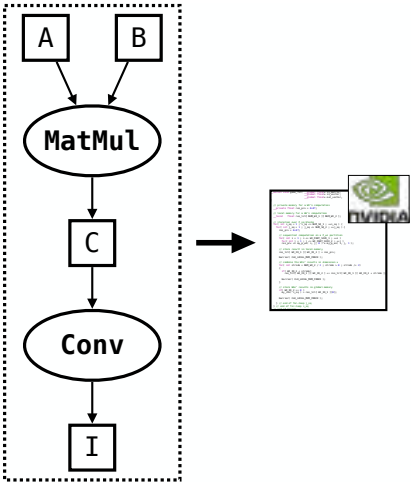


Figure 52: Extended MDH code generation approach, intended for our future work: in contrast to Figure 51, one optimized kernel is generated for multiple data-parallel computations, allowing applying optimization across the computations, such as transferring data between computations through fast memory resources (e.g., matrix C).

Figure 53 briefly recapitulates the internal design of our current MDH code generation approach introduced in Chapter 4 (using a notation similar to Figure 21, but simplified for illustration). Correspondingly, Figure 54 illustrates the internal design of our intended approach.

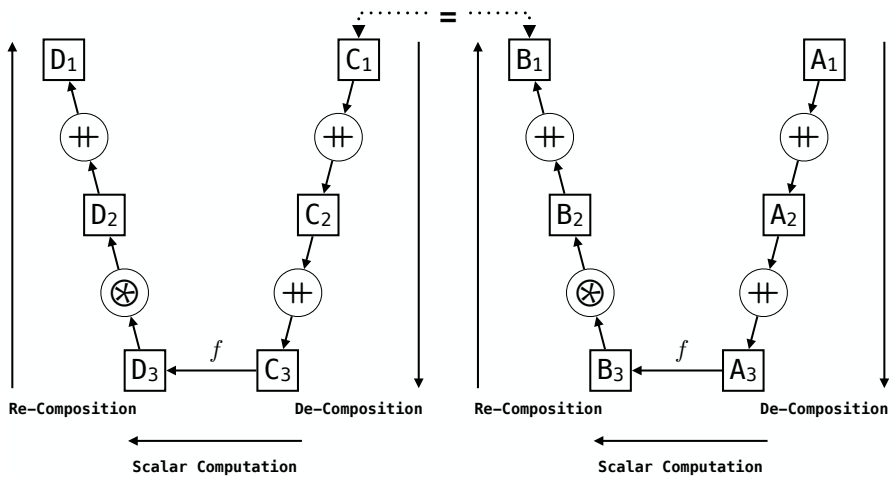


Figure 53: Internal design of the original MDH code generation approach (introduced in Chapter 4): data are transferred between computations *high* in the memory hierarchy, through *slow* memory regions.

In the current MDH approach (Figure 53), the result of the first computation (right part of the figure) is fully re-composed to the final result (denoted as $\mathtt{B}_1$ in the figure) – usually into a slow memory region that is high up in the memory hierarchy, e.g., *device memory* in the case of a CUDA computation. Afterward, the second computation (left part of the figure) operates on the final result of the first computation (denoted as $\mathtt{C}_1$ in the figure, which is equal to $\mathtt{B}_1$) and de-composes the result (denoted as $\mathtt{C}_2$), usually into a fast memory region (e.g., *shared memory* in CUDA).

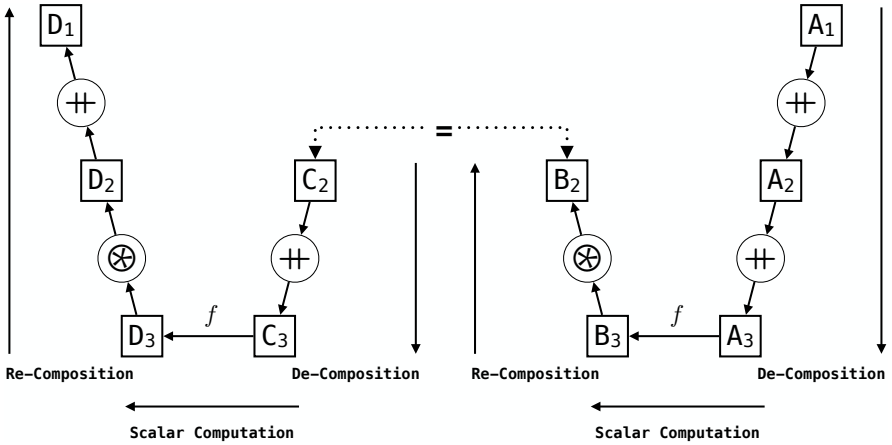


Figure 54: Internal design of MDH code generation approach, intended for our future work: data are transferred between computations *low* in the memory hierarchy, through *fast* memory regions.

In our intended approach (Figure 54), instead of fully re-composing the intermediate results of the first computation into a high memory region and de-composing the obtained result back down the memory hierarchy for the second computation (as in Figure 53), our intended approach keeps the intermediate results of the first computation (a.k.a. *tile*, see Chapter 4, and denoted as $\mathtt{B}_2$ in the figure) in the low, fast memory region and passes the data in this fast memory region to the second computation. In contrast to Figure 53, our intended approach in Figure 54 avoids costly write and read accesses to the slow memory region, thereby achieving higher performance.

We aim to design our intended MDH code generation approach for both interdependent computations (as in the left part of Figure 55, a.k.a. *vertical fusion* [23, 206]) and also computations that are independent of each other (right part of Figure 55, a.k.a. *horizontal fusion* [31]). Fusion optimization is usually more effective for interdependent computations: for such computations, fusion enables transferring data through fast memory regions, which is essential for achieving high performance on state-of-the-art computer systems [206]. In contrast, the fusion of computations that are independent allows only reducing the kernel launch overhead (e.g., in CUDA and CUDA-like approaches), and it often allows better exploiting the hardware, because a fused kernel executes more operations (e.g., of two, fused data-parallel computations instead of one data-parallel computation per kernel only).

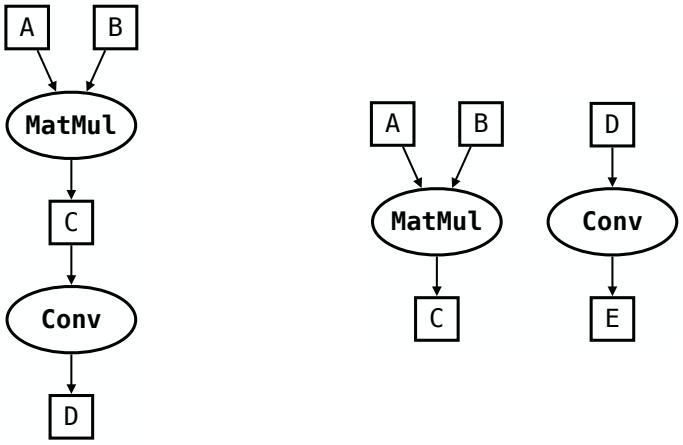


Figure 55: Computations `MatMul` and `Conv` in two different programs (left and right part of the figure): in the left part of the figure, both computations are interdependent in the program (the `Conv` computation uses the result matrix `C` of `MatMul` as input), and in the right part of the figure, the computations are independent of each other (no data dependencies).

8.3 COMPUTATIONS ON IRREGULAR INPUTS & OUTPUTS

In our MDH approach, input and output data are represented as multi-dimensional arrays (Definition 1). These arrays currently are assumed to have a regular structure: according to Definition 1, arrays in MDH always have the same size in a particular dimension, independent of the position in the array's other dimensions. However, many application areas, e.g., computations on sparse data formats [74], rely on arrays that have an irregular structure. Supporting in MDH irregularly shaped arrays is intended for our future work (inspired by [102]) and outlined in the following.

Figure 56 illustrates a 2-dimensional 4×4 array that has a regular structure, as currently supported in MDH. Any row i contains the same number of elements – the four elements $(i,1)$ – $(i,4)$ – and any column j also contains always the same number of elements – elements $(1,j)$ – $(4,j)$.

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

Figure 56: Example array that has a regular structure.

Figure 57 illustrates a 2-dimensional array that has an irregular structure, as intended to be supported by MDH in our future work. For example, row 1 consists of the 3 elements $(1,1)$ – $(1,3)$, but row 2 consist of two elements only – the elements $(2,1)$ and $(2,2)$.

(1,1)	(1,2)	(1,3)	
(2,1)	(2,2)		
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)			

Figure 57: Example array that has an irregular structure.

In our future work, instead of representing the structure of a D-dimensional array as a tuple of D natural numbers $(N_1, \ldots, N_D) \in \mathbb{N}^D$ (as currently done in MDH – see Definition 1), we aim to represent the structure of arrays as a tuple of D functions, $(\mathfrak{N}_1, \ldots, \mathfrak{N}_D) \in (\mathbb{N}^{D-1} \to \mathbb{N})^D$ each mapping $D-1$ natural numbers to a natural number. Our intention is that each function $\mathfrak{N}_d$, $d \in [1, D]_{\mathbb{N}}$, computes the size of the D-dimensional array in dimension d based on the position in the array $(i_1, \ldots, i_{d-1}, i_{d+1}, \ldots, i_D)$ in the $D-1$ other dimensions. For example, regarding Figure 57, the irregularly shaped example array in the figure will be represented in our intended MDH formalism as $(\mathfrak{N}_1, \mathfrak{N}_2)$ for $\mathfrak{N}_1 := 4$ (i.e., $\mathfrak{N}_1$ is the constant function 4) and $\mathfrak{N}_2(1) := 3, \mathfrak{N}_2(2) := 2, \mathfrak{N}_2(3) := 4, \mathfrak{N}_2(4) := 1$.

Our intended MDH approach particularly can represent regularly shaped arrays (as currently supported by MDH) by using constant functions only: for example, the array in Figure 56 is represented in our intended formalism as $(\mathfrak{N}_1, \mathfrak{N}_2)$ for $\mathfrak{N}_1 := 4$ and $\mathfrak{N}_2 := 4$, i.e., both functions are the constant function 4. Consequently, our current approach will be a straightforward, special case of our intended approach.

8.4 HETEROGENEOUS SYSTEM MODEL

In Definition 10, we introduce the formal representation of our target systems. Our definition is capable of representing any kind of system that is *homogeneous*, i.e., which has on each particular layer identical memory resources and cores. However, our current ASM definition cannot efficiently represent systems that are *heterogeneous*, i.e., consisting of different kinds of memory resources and/or cores on the same layer (such as a system containing an NVIDIA GPU and an AMD GPU).

Our future work aims to generalize our Definition 10 to capture also heterogeneous systems, as follows. Note that in the following, we refer to ASMs according to our current Definition 10 as *Homogeneous Abstract System Model (Hom-ASM)*, as our current ASMs can represent homogeneous systems only (as discussed above).

Definition 16. Heterogeneous Abstract System Model An L-*layered Heterogeneous Abstract System Model (Het-ASM)*, $L \in \mathbb{N}$, is a tuple

$$(S_1, \ldots, S_m), m \in \mathbb{N}$$

where each S_i is either a homogeneous or heterogeneous ASM instance.

We illustrate Definition 16 using the examples of heterogeneous multi-device and multi-node systems.

HETEROGENEOUS MULTI-DEVICE SYSTEM Using Definition 16, a multi-device system containing, e.g., a CPU (programmed via OpenMP), two NVIDIA GPUs (programmed via CUDA), and an AMD GPU (programmed via OpenCL) is of the following form:

$$ASM_{\text{multi-dev}} == (\texttt{OpenMP}, \texttt{CUDA}, \texttt{CUDA}, \texttt{OpenCL})$$

where `OpenMP`, `CUDA`, and `OpenCL` represent homogeneous device instances (see Example 11).

We present some encouraging work-in-progress results for MDH when used for heterogeneous multi-device system in [72, 164], inspired by dynamic load balancing approaches [248].

HETEROGENEOUS MULTI-NODE SYSTEM A multi-node system consisting of two nodes is represented using Definition 16 as follows:

$$ASM_{\text{multi-node}} = ((S_1, S_2))$$

Here, for example, S_1 represents a node equipped with an OpenMP device (e.g., CPU), a CUDA device (NVIDIA GPU), and an OpenCL device (AMD GPU), and S_2 another node equipped with an OpenMP device and a CUDA device:

$$S_1 = (\texttt{OpenMP}, \texttt{CUDA}, \texttt{OpenCL})$$
$$S_2 = (\texttt{OpenMP}, \texttt{CUDA})$$

8.5 POST-PROCESSING

Our MDH approach currently requires for correctness tile sizes (see Table 1) to evenly divide the input size – this requirement is ensured via constraints on tile size tuning parameters, as discussed in Section 4.4. For example, for an input size of N (e.g., N = 10), only divisors of N are valid tile sizes (e.g., tiles of sizes 2 or 5, in the case of N = 10). The same applies to tile sizes on different layers: a tile size for a lower layer must evenly divide a tile size for an upper layer (see Section 14).

This requirement on tile sizes is a current limitation of MDH and can degrade performance, as optimized sizes of tiles often also depend on the target hardware, and not only the input size.

Figure 58 illustrates how we aim to relax this constraint on tile size tuning parameters – a tile size for a low layer must evenly divide the tile size for an upper layer (including the input size) – by introducing a so-called *Post-Processing* for MDH. Our intended post-processing aims to allow arbitrary tile sizes: any *full tile* (e.g., of size 3 as in the lower part of Figure 58) is processed exactly as described in Chapter 4, and the remaining input elements (a.k.a. *partial tile*, consisting of one element in Figure 58) are processed separately, but also as described in Chapter 4 for an adapted tile size (of size 1 in Figure 58, instead of size 3 as for full tiles).

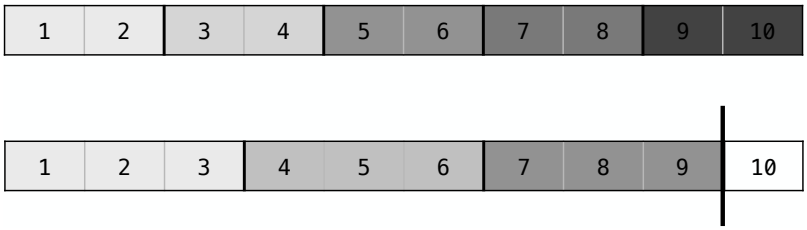


Figure 58: Example input of size 10 for a tile size of 2 (upper part of figure) and a tile size of 3 (lower part). The example in the lower part requires post-processing the border which consists of 1 element in this example.

Supporting post-processing is particularly important for input sizes with a low number of divisors (e.g., prime numbers). Our implementation of MDH supports post-processing, even though post-processing is not described in Chapter 4. We aim to thoroughly describe and evaluate our post-processing mechanism in our future work, and we aim to also compare our mechanism to other kinds of post-processing approaches, e.g., *padding*.

8.6 IMPROVING AUTOMATIC PARALLELIZATION

Chapter 7.2 presents our approach to automatic parallelization of sequential C code, based on polyhedral compilation techniques. As outlined in Section 7.2.4, our approach is currently restricted to a limited set of C programs supported as input by our system.

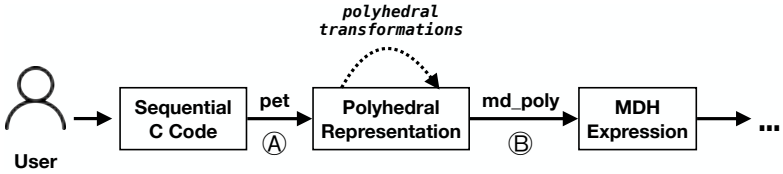


Figure 59: Our future works aims to extend our current automatic parallelization approach (depicted in Figure 46) by *polyhedral transformations* (highlighted via a dotted line in Figure 59), e.g., to make iteration spaces rectangular, thereby supporting a wider range of C input programs as input for our approach.

Figure 59 depicts how our future work aims to significantly widen the scope of C programs supported as input by our system. Currently (see Chapter 7.2), our system uses the polyhedral representation of the C input program to check whether the C program satisfies the limitation required to be taken as input by our system (see Figure 46), e.g., by checking if C program's iteration domain is rectangular and if loops in the program have static bounds (see Section 7.2.4).

Our future work (Figure 59) aims to use the polyhedral representation particularly also for applying *polyhedral transformations* [255], rather than just checking program characteristics (as done in Chapter 7.2). Using polyhedral transformations, such as *loop skewing* [317], we can transform C input programs not supported by our system (according to Section 7.2.4) to a form that can be used as input by our system, thereby widening the scope of C programs supported as input by our system.

8.7 SUPPORTING INDIRECT DATA ACCESSES

In Chapter 4, we introduce *view functions* (Definitions 6 and 8) which transform input and output buffers (referred to as `BUF` in Chapter 4) to our internal data representation (referred to as `MDA`). For high expressivity, our view functions are defined as arbitrary functions that satisfy a specific function specification only (Definitions 6 and 8). However, higher-order functions `inp_view` and `out_view` (Definitions 7 and 9), both introduced in Chapter 4 to conveniently compute view functions, are not capable of computing *any* potential view function. For example, our two higher-order functions cannot handle input and output buffers that are accessed *indirectly*, e.g., as required for expressing computations on sparse data formats [282] in MDH.

```
1  void spmv_csr(int *row_ptr, int *colind, float *val, int N, float *x,
       float *y)
2  {
3    int i, j;
4    float temp;
5    for(i = 0; i < N ; i++)
6    {
7      temp = y[i];
8      for(j = row_ptr[i]; j < row_ptr[i+1]; j++){
9        temp += val[j] * x[colind[j]];
10     }
11     y[i] = temp;
12   }
13 }
```

Listing 41: Sparse Matrix-Vector Multiplication (`SpMV`) in CSR format implemented in C (taken from [80]). While in line 9 the `val` buffer is accessed *directly* (via index `j`), buffer `x` is accessed *indirectly* using index function `colind`.

Listing 41 illustrates *direct* and *indirect* buffer accesses, using the example of *Sparse Matrix-Vector Multiplication (`SpMV`)* in the so-called *Compressed Sparse Row (CSR) Format* [282]. In line 9, we observe that buffer `val` is accessed via loop index `j`, which is known as a *direct* buffer access. In contrast, buffer `x` is accessed in line 9 using index function `colind` and thus *indirectly*.

Our future work aims to extend functions `inp_view` and `out_view` toward taking index functions as input. Thereby, we enable using `inp_view` and `out_view` for computing view functions that access buffers indirectly, as required, for example, for expressing `SpMV` (Listing 41) in MDH.

8.8 SUPPORTING DYNAMIC SHAPES

In Chapter 4, we introduce a type system for MDH that expresses and maintains data sizes (a.k.a. *shape*) – of the input and output data, as well as of intermediate results – carefully at the type level. Our code generation approach, as described in Chapter 4, currently requires data sizes to be statically known, i.e., the executable code that we eventually generate from high-level MDH expressions is specifically generated for a fixed size of the input and output data.

Being fixed in the data sizes has the advantage of enabling strong error checking at compile time, and particularly, it enables specifically optimizing the generated code for the particular data sizes, which is often essential for achieving high performance [155]. However, this (deliberate) limitation of our MDH approach may decrease user's productivity, e.g., when the user prefers using its generated code for multiple input sizes (rather than achieving the best possible performance for each individual input size).

Our MDH formalism already allows being flexible in data sizes, by using symbol $*$ instead of a particular size (Definition 23). Our implementation of MDH (Section .8) also supports flexible data sizes, even though not described in Chapter 4: instead of using particular data sizes in the generated code, the sizes are replaced by variables that are passed as input arguments to the generated code.

Our future work aims to enhance our code generation approach, by extending it for flexible data sizes. Note that flexible data sizes in particular may require *Post-Processing* (described above), because the dynamic sizes do not necessarily divide the tile sizes evenly.

8.9 COST MODEL

Our MDH approach offers a formalism for generating code that is generic in performance-critical parameters (Table 1). Currently, we rely on auto-tuning (see Chapter 5) for choosing parameter values that are optimized for a particular target architecture and characteristics of the input and output data.

While auto-tuning fully automatically finds well-performing parameter configurations for different kinds of architectures and data characteristics (see Sections 4.5 and 5.7), auto-tuning is costly in general: for example, we auto-tuned our implementations in Section 4.5 for 12h[3]. The time for auto-tuning is well-amortized in important application areas, e.g., deep learning, because the auto-tuned implementations are re-used many times and in each program run (see Section 4.5), and because auto-tuning is usually less costly and time intensive than relying on manual optimization by human experts. However, auto-tuning still may become a bottleneck for large applications containing many computations to tune.

Our future work aims to introduce an analytical cost model to minimize (or even avoid) the overhead for auto-tuning. For our intended cost model, we will introduce *operational semantics* [319] for MDH (which is currently introduced based on *denotational semantics* [326] in Chapter 4). Operational semantics allows reasoning about sequences of computational steps and consequently to reason about runtime and resource-related properties of our generated code. For this, we particularly need to extend our ASM system representation (Definition 10) by performance-related properties of systems, e.g., target system's numbers of cores and its sizes of fast memory resources[4].

Since usually not all kinds of optimizations are well-predictable via cost models (e.g., tile sizes), we aim to rely optionally on a hybrid approach for high-quality optimization results, e.g., determining tile sizes via auto-tuning and other optimizations via our intended cost model (such as choosing a well-performing data access pattern). We consider a hybrid approach as a promising trade-off between high-quality optimization results and tuning time.

We present and discuss further intended improvements of our optimization process, e.g., based on machine learning techniques, later in Chapter 9.

[3] Well-performing parameter configurations for MDH were usually found in less tuning time, as discussed in Section 4.5. We use such generous auto-tuning time to avoid auto-tuning issues in our experiments in Section 4.5 (in particular, auto-tuning issues for our competitors, to make evaluation more challenging for MDH). This is because auto-tuning issues are not the focus of experiments in Section 4.5 – auto-tuning is thoroughly discussed and experimentally evaluated in Chapter 5.

[4] An instance of our intended, new system model will represent a particular device (e.g., a specific NVIDIA GPU with its particular number of cores and memory resources). Consequently, we aim to call our new model *Concrete System Model (CSM)*, rather than *Abstract System Model (ASM)* (Definition 10) which represents a class of devices, e.g., any CUDA device (as our ASM captures only CUDA-specific properties of devices, see Definition 10 and Example 11, but no device-specific properties).

8.10 OPTIMIZATIONS EXPRESSED VIA EXPERT USERS

Our future work aims to introduce a convenient interface for expert users to (optionally) allow them incorporating their expert knowledge into the optimization process, e.g., to reduce (or even avoid) the overhead for auto-tuning. In contrast to the existing optimization languages (a.k.a. *scheduling languages* which are discussed in Section 4.6.1), we aim to design our optimization language such that any potential MDH optimization decision (Table 1) can optionally be left to the auto-tuning engine of our system. In contrast, the existing optimization languages usually allow leaving only some optimizations for auto-tuning (e.g., choosing optimized sizes of tiles) but not others (e.g., identifying an optimized data access pattern). Furthermore, our language will be fully based on our MDH formalism, thereby allowing strong error checking and precise error messages.

We present first results for our MDH-based optimization language in [10].

8.11 VISUALIZING (DE/RE)-COMPOSITIONS

To contribute to a better understanding of our generated code [367], e.g., for performance experts and educational purposes, we aim to exploit the systematic design of our low-level MDH representation (Section 4.3) to introduce an equivalent, graphical representation for programs expressed in our low-level representation. Such a graphical representation will visualize a program's (de/re)-composition that is expressed in MDH's low-level representation.

Figure 60 shows our intended, graphical representation for programs expressed in our low-level representation, using the example of matrix multiplication. Performance-critical parts are highlighted gray in the figure and correspond to tuning parameters (Table 1).

In the figure, each (de/re)-composition step is separated via dotted lines. The partition sizes of (de/re)-composition steps are denoted in the center part of the figure. Memory regions for input/output buffers are stated below buffers, and the buffers memory layouts are visualized in form of two-dimensional coordinate crosses. To each dimension of the iteration space (depicted as three-dimensional coordinate crosses in the center part of the figure), we assign a corresponding core layer. Formally, the MDH approach uses different iteration spaces for the input data (visualized in Figure 60 via the three-dimensional coordinate crosses below `INP MDA`) and output data (coordinate crosses below `OUT MDA`); the axes are annotated by our visualization tool with the corresponding combine operators: in MDH, always concatenation ++ for input data, and concatenations and addition for the output in the case of `MatMul` (as discussed in Chapter 4). The numbers `0-6` in boxes, which are also located in the center part of Figure 60, set the order in which (de/re)-composition steps should be processed in our generated code. The index functions used to access input and output buffers are stated below view functions in Figure 60.

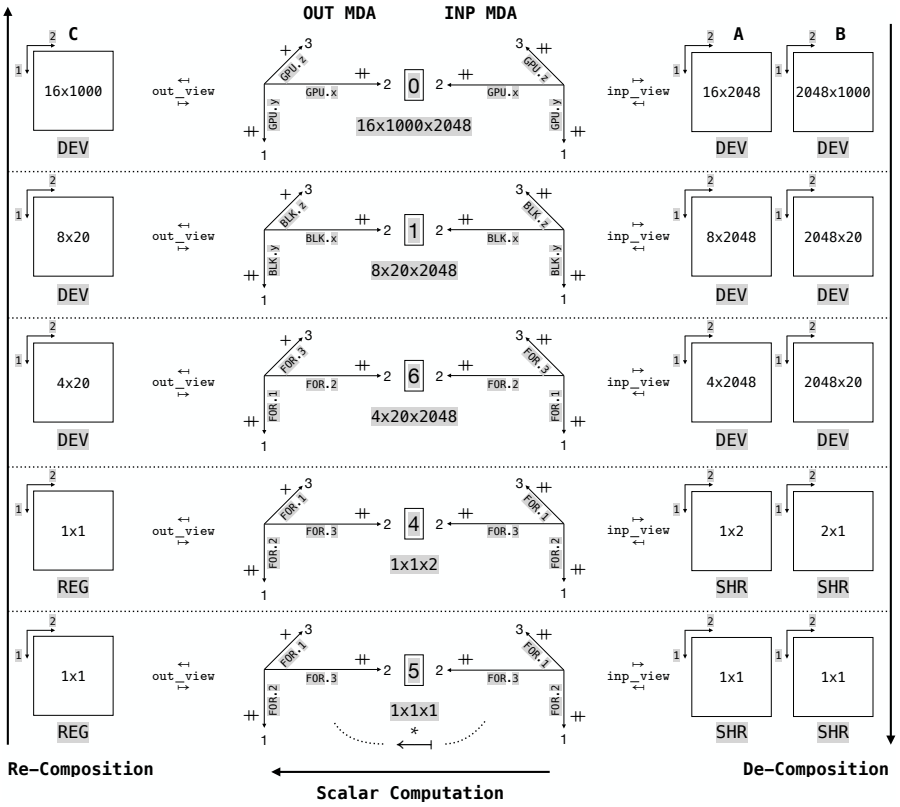


Figure 60: Intended visualization of MDH low-level program representation for the example of matrix multiplication.

8.12 CODE GENERATION

Our code generation approach is currently briefly outlined in Section .8, in the form of a formally inspired, imperative-style pseudo code notation. Our pseudocode can be straightforwardly transformed to executable program code (e.g., in OpenMP, CUDA, or OpenCL): all the major optimization decisions regarding parallelization and data movement optimizations are explicitly expressed in our pseudocode notation such that generating executable program code from our pseudocode becomes a straightforward, mechanical process only.

In our future work, we aim to thoroughly describe and illustrate our pseudocode, present its implementation (e.g., in Python [3]), and discuss how we generate executable program code in OpenMP, CUDA, and OpenCL from our pseudocode.

Our future work also aims to demonstrate that the structure of our pseudocode contributes significantly to defining code-level optimization passes in an efficient and systematic way (as also outlined in Section .9), such as loop fusion and buffer size reduction. Such passes require complex code heuristics when they are designed and applied to arbitrary program code (rather than MDH-generated code).

8.13 TARGETING DOMAIN-SPECIFIC HARDWARE EXTENSIONS

Our MDH formalism captures high-level semantic information for programs implemented in its high-level representation; we aim to exploit this high-level information in our future work to efficiently target domain-specific hardware extensions in our generated code. For example, NVIDIA offers so-called *tensor cores* [148] to multiply matrices of size 4×4 immediately in hardware. Consequently, tensor cores have the potential to significantly accelerate matrix multiplication on NVIDIA GPUs.

We can exploit tensor cores in our approach; for this, we aim to match the user-provided MDH high-level expression against the MDH-based matrix multiplication compute pattern (see Figure 20). Such matching allows de-composing the computation of matrix multiplication down to 4×4 matrices (instead of fully down to 1×1 matrices, as currently done in Section 4.4). The matrix multiplication on 4×4 matrices can then be mapped to a tensor core operation (instead of to standard CUDA code computing straightforwardly the multiplication, as currently done in our approach).

8.14 TARGETING ASSEMBLY-LEVEL PROGRAMMING MODELS

Our MDH approach offers a formalism for code generation in state-of-the-art parallel programming models. Even though our MDH formalism is not limited to code generation in particular programming models, MDH is currently implemented for code generation in OpenMP, CUDA, and OpenCL, only.

Our future work aims to extend our implementation of MDH toward code generation in programming models operating on the assembly level (rather than the higher OpenMP/CUDA/OpenCL level of abstraction). Targeting assembly level models, such as *PTX (Parallel Thread Execution)* [40] for GPUs or *LLVM (Low-Level Virtual Machine)* [281] for CPUs, offers more opportunities [214, 264], e.g., efficiently exploiting the vector intrinsic operations of assembly languages.

8.15 FURTHER EXPERIMENTAL RESULTS

Our future work aims to experimentally evaluate our approach for further case studies, against more competitors, and on more architectures.

We present encouraging work-in-progress experimental results for:

- Deep Learning Computations [363] on GPU and CPU, in [59, 113]
- MDH vs. Benchmark Suites `Parboil` [234] and `Rodinia` [256] on GPU and CPU, in [104]
- *Basic Linear Algebra Subprograms (BLAS)* [321] on GPU, CPU, and mobile device, in [133]
- *Ensemble Classifier Chains (ECC)* [244] on GPU and CPU, in [135, 136, 151, 162]
- MDH vs. `CUTLASS` [351], `PPCG` [218], and `CUB` [35] on GPU, in [89]
- MDH vs. `Kokkos` [196], `RAJA` [198], and `SYCL` [191, 343] on GPU and CPU, in [77]
- *Discrete Cosine Transform (DCT)* [325] and *Histogram* [271] on GPU, in [186]
- *Fast Fourier Transform (FFT)* [315] on GPU, in [180]

8.16 EMBEDDING MDH'S DOMAIN-SPECIFIC LANGUAGE

To further contribute to the productivity of our MDH+ATF+HCA approach, we aim to embed the MDH's *Domain-Specific Language (DSL)* (presented in Chapter 7.1) into mainstream programming languages (a.k.a. *embedded Domain-Specific Language (eDSL)* [123]), e.g., Python which is becoming increasingly important in both academia and industry [46].

Listing 42 shows as an example how we aim to embed the MDH's DSL into the Python programming language using the example of *Matrix-Vector Multiplication (MatVec)*. We observe that even though Listing 42 is implemented in the Python's syntax, it is very close to the MatVec example in our plain, un-embedded DSL language in Listing 38.

```
def matvec( T: ScalarType, I: int, K: int ):
    @mdh(outputs = {'w':Buffer[ T,[I] ]},
         inputs  = {'M':Buffer[ T,[I,K] ], 'v':Buffer[ T,[K] ]} )
    def matvec_mdh():
        def f_mul( out, inp ):
            out['w'] = inp['M'] * inp['v']

        def plus(res, lhs, rhs):
            res = lhs + rhs

        return (
            out_view[ T ]( {'w': [lambda i, k: (i)]} ),
              md_hom[ I,K ]( f_mul, (cc, pw(plus)) ),
                inp_view[ T,T ]( {'M': [lambda i, k: (i, k)],
                                  'v': [lambda i, k: (k)]   })
        )

    return matvec_mdh
```

Listing 42: MatVec expressed in MDH's Python Interface.

8.17 IMPROVING ACCESSABILITY

Our MDH approach is currently implemented as a standalone compiler, in the C++ programming language, that takes high-level MDH programs as input (e.g., as in Figure 38) and generates executable code from it (e.g., in OpenMP, CUDA, or OpenCL).

To make our work better accessible for the community, we aim to implement our approach also into the *MLIR* [56] framework which offers a reusable compiler infrastructure. By relying additionally also on MLIR (instead of our standalone compiler only), we allow other MLIR projects [7] conveniently using our approach for code generation. The contributions of this work give a precise, formal recipe, of how to implement our MDH approach into projects like MLIR.

Work-in-progress results for implementing MDH into MLIR are presented in [66].

9 ENHANCE CODE OPTIMIZATION

We aim to advance our code optimization approach (presented in Chapter 5) in our future work, as outlined in the following.

9.1 IMPROVING SPACE GENERATION & STORING & EXPLORATION

We aim to reduce search space generation time and memory footprint of constrained search spaces (discussed in Sections 5.3 and 5.4), by refining our notion of an interdependent parameter group. According to Section 5.3, two parameters are interdependent and thus in the same group iff one of them occurs in the syntax tree of the other parameter's constraint function. Currently, we consider interdependency as a transitive relation, i.e., when parameters p_1 and p_2 are interdependent as well as parameter p_1 and p_3, we consider p_1, p_2, p_3 to be all in the same group of interdependent parameters.

In our future work, we aim to refine our notion of an interdependent parameter group as follows: we put parameters in the same group iff they have interdependencies with any other parameters within the group. For example, regarding the example parameters p_1, p_2, p_3 above, we would split them in groups $\{p_1, p_2\}$ and $\{p_1, p_3\}$, rather than using a single group $\{p_1, p_2, p_3\}$ as we do currently according to Chapter 5.

We briefly illustrate how we aim to use our new notion of interdependent parameter groups for improving our processes of search space generation and storing, using a simple, illustrative, real-world example: auto-tuning OpenCL parameters *Local Tile Size (`LT`)*, *Number of Work-Groups (`WG`)*, *Private Tile Size (`PT`)*, and *Number of Work-Items (`WI`)* (see Chapter 5) for an arbitrary, fixed input size $N \in \mathbb{N}$. For simplicity, we use for all parameters the same ranges, consisting of the two values 2 and 4 only:

$$
\begin{aligned}
&\texttt{LT} := (\quad \texttt{"LT"}\ ,\ \texttt{LT\_range} := \{2,4\}\ ,\ \texttt{divides}(N) \qquad) \\
&\texttt{WG} := (\quad \texttt{"WG"}\ ,\ \texttt{WG\_range} := \{2,4\}\ ,\ \texttt{divides}(N/LT) \qquad) \\
&\texttt{PT} := (\quad \texttt{"PT"}\ ,\ \texttt{PT\_range} := \{2,4\}\ ,\ \texttt{divides}(LT) \qquad) \\
&\texttt{WI} := (\quad \texttt{"WI"}\ ,\ \texttt{WI\_range} := \{2,4\}\ ,\ \texttt{divides}(LT/PT) \qquad)
\end{aligned}
$$

Listing 43 shows how we aim to exploit our new notion of an interdependent parameter group for a new search space generation algorithm[1] – our new algorithm in Listing 43 is functionally equivalent to the original algorithm according to Section 5.3 (Listing 44), but our new algorithm has a lower runtime complexity: our new notion of an interdependent parameter group enables splitting the original example loop nest (Listing 44) into a nest that has a lower depth and consequently lower runtime complexity (Listing 43).

```
1  // new search space generation algorithm
2  for ( lt : LT_range )
3  {
4    for ( wg : WG_range )
5      if( /* LT divides N && WG divides N/LT */ )
6        search_space.add( ("LT",lt), ("WG",wg) );
7
8    for ( pt : PT_range )
9      for ( wi : WI_range )
10       if( /* PT divides LT && WI divides LT/PT */ )
11         search_space.add( ("PT",pt), ("WI",wi) );
12 }
```

Listing 43: Our intended, new search space generation algorithm has a lower runtime complexity than the original generation algorithm introduced in Chapter 5 (Listing 44) – in this example: 3-layered loops nest (new algorithm) vs. 4-layered nest (original algorithm) – by generating search space of parameters without interdepencies independent of each other.

```
1  // original algorithm (according to Section 5.3)
2  for ( lt : LT_range )
3    for ( wg : WG_range )
4      for ( pt : PT_range )
5        for ( wi : WI_range )
6        {
7          if( /* LT divides N  && WG divides N/LT  &&
8                 PT divides LT && WI divides LT/PT */ )
9            search_space.add( lt,wg,pt,wi );
10       }
```

Listing 44: Original search space generation algorithm introduced in Chapter 5 (simplified for better illustration).

Figure 61 shows how we intend to exploit our new group notion for reducing memory footprint of constrained search spaces. For this, we improve our CoT structure (introduced in Section 5.4), based on our new notion of an interdependent parameter group, toward avoiding redundancies as occurring in the original CoT (see Figure 62) introduced in Section 5.4.

[1] For simplicity, we do not express parallelization optimizations in Listing 43, which work for our new algorithm analogously as for our original algorithm in Listing 22.

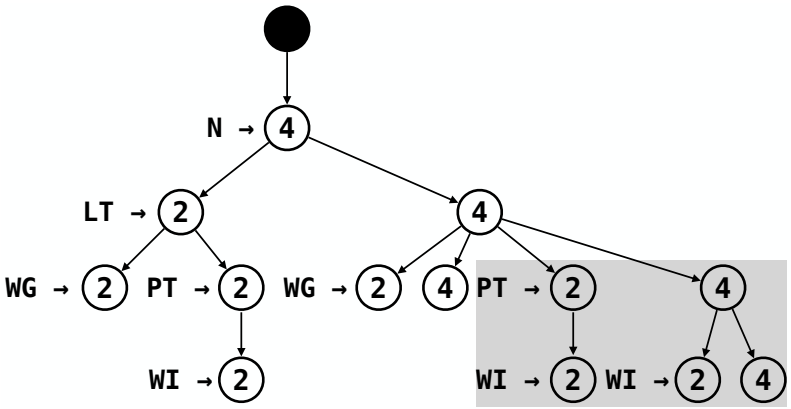


Figure 61: Our intended, new CoT structure has a lower memory footprint than the original CoT structure introduced in Chapter 5 (Figure 62) – in this example: 13 nodes (new CoT, as in this figure) nodes vs. 18 nodes (original CoT, as in Figure 62) – by avoiding redundancies (highlighted gray in the figures).

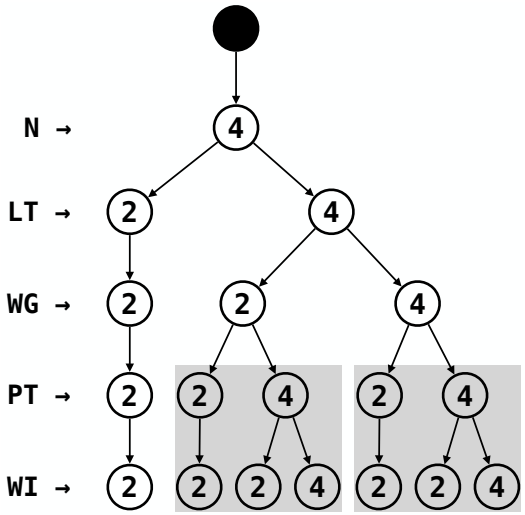


Figure 62: Original CoT structure introduced in Chapter 5 (redundancies highlighted gray in the figure).

9.2 CONSTRAINT SATISFACTION PROBLEMS (CSP)

Our processes to search space generation and storing are closely related to the area of *Constraint Satisfaction Problems (CSP)* [287]. In our future work, we aim to test our approach for CSP-related applications, e.g., graph coloring [273]. Our experiments in Section 5.7.4 report encouraging results as compared to a state-of-the-art constraint solver for representative auto-tuning applications. This is because the constraint solver does not introduce and exploit parallelization and parameter grouping optimizations, as we do in Section 5.3. Also, the solver does not rely on an efficient data structure for storing valid configurations, whereas we introduce the CoT structure, in Section 5.4, for storing valid configurations with a low memory footprint (as also reported experimentally in Section 5.7.4).

9.3 MULTI-OBJECTIVE AUTO-TUNING

Our approach supports multi-objective auto-tuning (as described in Section 5.6), but it prioritizes one objective over another, e.g., achieving the best possible runtime performance (highest priority) and then auto-tuning for low energy consumption (lower priority). This may sometimes be suboptimal, e.g., when neglectable losses in runtime performance allow significantly reducing energy consumption.

In our future work, we aim to improve our multi-objective auto-tuning process toward identifying the *Pareto-optimal* set of configurations, similarly as in [96], i.e., all configurations that cannot be optimized further for one objective (e.g., runtime performance) without worsening the quality for another objective (e.g., energy consumption). Hellsten et al. [2] show that the auto-tuning techniques introduced in this work can be greatly combined with identifying Pareto-optimal configurations.

9.4 NEW SEARCH TECHNIQUES & EXPLORATION STRATEGIES

We aim to implement more kinds of search techniques into our implementation introduced in Section 5.6, e.g., swarm-based techniques which have proven to be efficient for a broad range of applications [266]. By implementing new search techniques, we combine the advantages of the new techniques with our novel concepts to search space generation, storing, and exploration (introduced in Sections 5.3-5.5). Some early, work-in-progress results have been produced and presented in [118].

To improve our overall exploration strategy, we aim to combine multiple search techniques together. Such combination is important, because different parts of search spaces may differ significantly in characteristics [150], e.g., parts corresponding to *ratio* tuning parameters (like the number of threads and sizes of tiles) or *nominal/ordinal* parameters (like parameters for data layout optimizations and loop permutations). Previous work [100, 150] shows that different search techniques usually perform very differently on search space parts with different characteristics. However, the previous work is limited to unconstrained search spaces, while our approach, introduced in Chapter 5, targets in particular constrained search spaces, as required for auto-tuning recent parallel implementations for state-of-the-art architectures (see Section 5.7).

9.5 MACHINE LEARNING TECHNIQUES IN AUTO-TUNING

To reduce the overall auto-tuning time of our approach, we aim to rely on machine learning techniques, inspired by [155]: we aim to predict the performance of parameter configurations via machine learning techniques, rather than executing the configuration on a real machine which is costly. Early, work-in-progress results for combining our approach with machine learning techniques have been produced and presented in [118].

9.6 USER INTERFACE

We introduce in Section 5.6 an interface for auto-tuning C++ programs at runtime. In our future work, we aim to introduce interfaces for auto-tuning programs at runtime written in programming languages also different from C++, e.g., Python which is gaining increasing popularity [46]. Work-in-progress results for our Python-based auto-tuning interface have been produced in [67].

10 ENHANCE CODE EXECUTION

We aim to advance our code execution approach (presented in Chapter 6) in our future work, as outlined in the following.

10.1 SINGLE-MODEL SUPPORT

The user interface of our HCA approach, as presented in Chapter 6, currently relies on different programming models: a C/C++-based model for implementing host code and an OpenCL/CUDA-based model[1] for implementing device code. While such a distinction allows advantages like *Just-in-Time Compilation (JIT)*, e.g., in order to benefit from runtime values during device code compilation (such as the input size), such distinction increases complexity for the user of our system, because the user has to program in different programming models.

To simplify programming host and device code for the user of our system, we aim to unify host and device programming: in our future work, we aim to optionally support also a single-model approach, where device and host code are implemented both in the same programming language, inspired by [191, 197].

10.2 NEW TARGET MODELS

Our HCA approach is currently implemented as a high-level abstraction for OpenCL- and CUDA-compatible devices. While OpenCL and CUDA already cover a broad range of architectures (in particular OpenCL which can be used for programming different kinds of architectures: GPUs, CPUs, etc), we aim to broaden our scope of architectures, e.g., by supporting also the OpenMP [357] programming model, because OpenMP is widely used in the community and sometimes achieves higher performance than, e.g., OpenCL, when used for CPU programming [233].

[1] The user of our system can arbitrarily chose between implementing device code in OpenCL or CUDA, respectively; our system automatically translates from OpenCL to CUDA (or CUDA to OpenCL) when device code is implemented in OpenCL for a CUDA device (or in CUDA for an OpenCL device). For this, we rely internally on a proof-of-concept source-to-source translation engine [142] based on the *clang project* [4]. In our future work, we aim to improve our OpenCL-to-CUDA/CUDA-to-OpenCL translation engine, e.g., by supporting advanced C++ features, such as automatic type deduction and template meta programming.

10.3 NEW INTERFACE KINDS

HCA currently relies on C++ as host programming language. However, our approach is not C++ specific, and its user interface can also be integrated into other kinds of mainstream programming languages, e.g., Python which is gaining more attraction in both academia and industry [46].

10.4 HIGHER-LEVEL ABSTRACTIONS

We aim to optionally raise the abstraction level of HCA for the user, e.g., by offering an MDH-inspired kernel language, *DPCL (Data-Parallel Computing Language)*, capable of expressing data-parallel computations uniformly for different targets (OpenCL, CUDA, OpenMP, ...), and by offering automatic work distribution among devices for the HCA user.

Part IV

CONCLUSION

CONCLUSION

This thesis addresses the challenge of simultaneously achieving, in one combined approach, three major challenges – *Performance*, *Portability*, and *Productivity* – for data-parallel computations targeting state-of-the-art parallel architectures. The goal of this thesis has been identified as an important research challenge at major computer science conferences and workshops.

To achieve our goal, we introduce a holistic code *generation*, *optimization*, and *execution* approach consisting of three major sub-projects:

1. *Multi-Dimensional Homomorphisms (MDH)* – a code *generation* approach based on a novel algebraic formalism for expressing and optimizing data-parallel computations. Compared to existing code generation approaches, MDH is designed to target arbitrary combinations of data-parallel computations and parallel architectures, whereas existing approaches are often limited to particular sub-classes of computations and architectures only, e.g., only linear algebra routines on only GPU or only stencil computations on CPU. Furthermore, MDH's optimization process is designed to work fully automatically, without incorporating the user, and MDH is based on a fully formal, algebraic foundation. Our experiments confirm that even though MDH is designed as generic in the target computation and architecture, and relies on a fully automatic optimization processes, it achieves competitive and often better performance than state-of-the-art code generation approaches, on GPUs and CPUs, including hand-optimized solutions provided by vendors.

2. *Auto-Tuning Framework (ATF)* – a code *optimization* approach that fully automatically finds device- and data-optimized values of performance-critical program parameters (a.k.a. *tuning parameters*), such as numbers of threads and sizes of tiles. In contrast to existing auto-tuning approaches, ATF significantly improves *generating*, *storing*, and *exploring* the search spaces of modern parallel implementations whose tuning parameters often have so-called *interdependencies* among them, e.g., the value of one tuning parameter has to evenly divide the value of another tuning parameter. Also, ATF has a particular focus on usability, by providing an easy-to-use user interface in the form of both a domain-specific language as well as interfaces embedded in general-purpose programming languages (such as C++). Our experiments confirm that ATF achieves high auto-tuning efficiency for GPU- and CPU-oriented applications from various popular domains.

3. *Host-Code Abstraction (HCA)* – a code *execution* approach that offers a high-level programming interface for implementing so-called host code which is required in modern parallel programming approaches (such CUDA and OpenCL) for code execution. Our HCA approach combines major advantages over existing approaches, such as internally managing data transfers between devices and automatically managing device synchronization, but also being targeted to multi-node systems. Furthermore, HCA automatically performs host code optimizations, such as overlapping data transfers with device computations, as required for achieving high host code performance.

We show that each of our three sub-projects – even though developed independently of each other – complement each other to a holistic code generation, optimization, and execution approach. Our holistic approach is accessed by the end user via an easy-to-use high-level *Domain-Specific Language (DSL)*. Our DSL conveniently expresses data-parallel computations, agnostic of hardware and optimization details, and it is formally grounded by our MDH approach. As an alternative to DSL programming, our approach allows taking as input straightforward (annotated) sequential program code, thereby further contributing to the usability of our approach.

We conclude this thesis by arguing that we consider this work as a promising starting point for future research and development, by discussing our ideas and encouraging work-in-progress results for continuing our approach.

Part V

APPENDIX

Our appendix provides details for the interested reader that should not be required for understanding the basic concepts and ideas discussed in this thesis.

.1 MATHEMATICAL FOUNDATION

We rely on a set theoretical foundation, based on ZFC set theory [298]. We avoid class theory, such as NBG [327], by assuming, for example, that our universe of types contains all relevant representatives (`int`, `float`, `struct`, etc), but is not the "class of all types". Thereby, we avoid fundamental issues [75] which are not relevant for this work.

.1.1 *Family*

Definition 17 (Family)**.** Let I and A be two sets. A *family* F from I to A is any set

$$F := \{ (i, a) \mid i \in I \land a \in A \}$$

such that the following two properties are satisfied:

- *left-total*: $\forall i \in I : \exists a \in A : (i, a) \in F$
- *right-unique*: $(i, a) \in F \land (i, a') \in F \Rightarrow a = a'$

We refer to I also as *index set* of family F and to A as F's *image set*. If I has a strict total order $<$, we refer to F also as *ordered family*.

Notation 4 (Family)**.** Let F be a family from I to A.
We write:

- F_i for the unique $a \in A$ such that $(i, a) \in F$;
- $(F_i)_{i \in I}$ instead of F to explicitly state F's index and image sets in our notation;
- $(F_{i_1,\ldots,i_n})_{i_1 \in I_1,\ldots,i_n \in I_n}$ instead of $(\ldots(F_{i_1,\ldots,i_n})_{i_n \in I_n}\ldots)_{i_1 \in I_1}$.

Alternatively, depending on the context, we use the following notation:

- $F^{<i>}$ instead of F_i;
- $(F_i)^{<i \in I>}$ instead of $(F_i)_{i \in I}$;
- $(F_{i_1,\ldots,i_n})^{<i_1 \in I_1 \mid \ldots \mid i_n \in I_n>}$ instead of $(F_{i_1,\ldots,i_n})_{i_1 \in I_1,\ldots,i_n \in I_n}$.

For nested families, each index set I_k may depend on the earlier-defined values $i_1, \ldots, i_{k-1}$ (not explicitly stated above for brevity).

Definition 18 (Tuple)**.** We identify n-tuples as families that have index set $[1, n]_{\mathbb{N}}$.

Example 12 (Tuple)**.** A 2-tuple (a, b) (a.k.a *pair*) is a family $(F_i)_{i \in I := [1,2]_{\mathbb{N}}}$ for which $F_1 = a$ and $F_2 = b$.

.1.2 *Scalar Types*

We denote by

$$\texttt{TYPE} := \{\, \texttt{int}, \texttt{int8}, \texttt{int16}, \ldots, \texttt{float}, \texttt{double}, \ldots, \texttt{struct}, \ldots \,\}$$

our set of *scalar types*, where `int8` and `int16` represent 8-bit/16-bit integer numbers, `float` and `double` are the types of single/double precision floating point numbers (IEEE 754 standards), `structs` contain a fixed set of other scalar types, etc. For simplicity, we interpret integer types (`int`, `int8`, `int16`, `...`) uniquely as integers $\mathbb{Z}$, floating point number types (`float` and `double`) as rationale numbers $\mathbb{Q}$, etc.

For high flexibility, we avoid fixing `TYPE` to a particular set of scalar types, i.e., we assume that `TYPE` contains all practice-relevant types. This is legal, because our formalism makes no assumptions on the number and kinds of scalar types.

We consider operations on scalar types (addition, multiplication, etc) to be: 1) *atomic*: we do not aim at parallelizing or otherwise optimizing operations on scalar values in this work; 2) *size preserving*: we assume that all values of a scalar type have the same arbitrary but fixed size.

Note that we can potentially also define, for example, the set of arbitrarily sized matrices $\{T^{m\times n} \mid m, n \in \mathbb{N}, T \in \texttt{TYPE}\}$ as scalar type in our approach. However, this would prevent any kind of formal reasoning about type correctness and performance of matrix-related operations (e.g., matrix multiplication), such as parallelization (due to our atomic assumption above) or type correctness (e.g., assuring in matrix multiplication that number of columns of the first input matrix coincides with and number of rows of the second matrix: due to our size preservation assumption above, we would not be able to distinguish matrices based on their sizes).

.1.3 *Functions*

Definition 19 (Function)**.** Let $A \in \texttt{TYPE}$ and $B \in \texttt{TYPE}$ be two scalar types.

A *(total) function* f is a tuple of the form

$$f \in \{\, (\underbrace{(A, B)}_{\texttt{function type}}, \underbrace{G_f}_{\substack{\texttt{function}\\ \texttt{graph}}}) \mid G_f \subseteq \{\, (a, b) \mid a \in A \wedge b \in B \,\} \,\}$$

that satisfies the following two properties:

- *left-total*: $\forall a \in A : \exists b \in B : (a, b) \in G_f$;
- *right-unique*: $(a, b) \in G_f \wedge (a, b') \in G_f \Rightarrow b = b'$.

We write $f(a)$ for the unique $b \in B$ such that $(a, b) \in G_f$. Moreover, we denote f's function type as $A \to B$, and we write $f : A \to B$ to state that f has function type $A \to B$.

We refer to:

- A as the *domain* of f
- B as the *co-domain* (or *range*) of f
- (A, B) as the *type* of f
- G_f as the *graph* of f

If f does not satisfy the left total property, we say f is *partial*, and we denote f's type as $f : A \to_p B$.

We allow functions to have so-called *dependent types* [291] for expressive typing. For example, dependent types enable encoding the sizes of families into the type system, which contributes to better error checking. We refer to dependently typed functions as *meta-functions*, as outlined in the following.

Definition 20 (Meta-Function)**.** We refer to any family of functions

$$(f^{<i>} : A^{<i>} \to B^{<i>})^{<i \in I>}$$

as *meta-function*. In the context of meta-functions, we refer to index $i \in I$ also as *meta-parameter*, to index set I as *meta-parameter type*, to $A^{<i \in I>}$ and $B^{<i \in I>}$ as meta-types (as both are generic in meta-parameter $i \in I$), and to $A^{<i>} \to B^{<i>}$ for concrete i as meta-function f's ordinary function type.
In the following, we often write:

- $f^{<i \in I>} : A^{<i>} \to B^{<i>}$ instead of $(f^{<i>} : A^{<i>} \to B^{<i>})^{<i \in I>}$;
- $f^{<i>} : A' \to B^{<I>}$ (or $f^{<i>} : A^{<i>} \to B'$) iff $A^{<i>} = A'$ (or $B^{<i>} = B'$) for all $i \in I$.

We use *multi-stage meta-functions* as a concept analogous to *multi staging* [302] in programming and similar to *currying* in mathematics.

Definition 21 (Multi-Stage Meta-Function)**.** A *multi-stage meta-function* is a nested family of functions:

$$f^{\overbrace{< i_1 \in I_1^{<>} >}^{\text{stage 1}} \dots \overbrace{< i_S \in I_S^{<i_1,\dots,i_{S-1}>} >}^{\text{stage } S}} : \underbrace{A^{<i_1,\dots,i_S>} \to B^{<i_1,\dots,i_S>}}_{\texttt{function instance}}$$

Here, $I_s^{<i_1,\dots,i_{s-1}>}$, $s \in [1, S]_{\mathbb{N}}$, is the meta-parameter type on stage s, which may depend on all meta-parameters of the previous stages $i_1, \dots, i_{s-1}$. We refer to such meta-functions also as *S-stage meta-functions*, and we denote their type also as

$$f^{<i_1 \in I_1^{<>} \mid \dots \mid i_S \in I_S^{<i_1,\dots,i_{S-1}>}>} : A^{<i_1,\dots,i_S>} \to B^{<i_1,\dots,i_S>}$$

and access to them as

$$f^{<i_1 \mid \dots \mid i_S>}(x)$$

where different stages are separated by vertical bars.

We allow partially applying parameters (meta and ordinary) of meta-functions.

Definition 22 (Partial Meta-Function Application)**.** Let

$$f^{<i_1 \in I_1 | \ldots | i_S \in I_S>} : A^{<i_1,\ldots,i_S>} \to B^{<i_1,\ldots,i_S>}$$

be a meta-function (meta-parameters of meta-types $I_1, \ldots, I_S$ omitted for brevity).

- The *partial application* of meta-function f on stage s to meta-parameter $\hat{i}_s$ is the meta-function

$$f'^{<i_1 \in \hat{I}_1 | \ldots | i_{s-1} \in \hat{I}_{s-1} | i_{s+1} \in I_{s+1} | \ldots | i_S \in I_S>} : A^{<i_1,\ldots,i_{s-1},\hat{i}_s,i_{s+1}\ldots,i_S>} \to B^{<i_1,\ldots,i_{s-1},\hat{i}_s,i_{s+1},\ldots,i_S>}$$

  where $\hat{I}_1 \subseteq I_1, \ldots, \hat{I}_{s-1} \subseteq I_{s-1}$ are the largest sets such that $\hat{i}_s \in I_s^{<i_1,\ldots,i_{s-1}>}$ for all $i_1 \in \hat{I}_1, \ldots, i_{s-1} \in \hat{I}_{s-1}$. The function is defined as:

$$f'^{<i_1|\ldots|i_{s-1}|i_{s+1}|\ldots|i_S>}(x) := f^{<i_1|\ldots|i_{s-1}|\hat{i}_s|i_{s+1}|\ldots|i_S>}(x)$$

  We write for f''s type also

$$f^{<i_1 \in \hat{I}_1 | \ldots | i_{s-1} \in \hat{I}_{s-1} | \hat{i}_s | i_{s+1} \in I_{s+1} | \ldots | i_S \in I_S>} : A^{<i_1,\ldots,i_{s-1},\hat{i}_s,i_{s+1}\ldots,i_S>} \to B^{<i_1,\ldots,i_{s-1},\hat{i}_s,i_{s+1},\ldots,i_S>}$$

  where f' is replaced by f and $i_s \in I_s$ is replaced by the concrete value $\hat{i}_s$.

- The *partial application* of meta-function f to ordinary parameter x is the meta-function

$$f'^{<i_1 \in \hat{I}_1 | \ldots | i_S \in \hat{I}_S>} : \underbrace{B_1^{<i_1,\ldots,i_S>} \to B_2^{<i_1,\ldots,i_S>}}_{= B^{<i_1,\ldots,i_S>}}$$

  where $\hat{I}_1 \subseteq I_1, \ldots, \hat{I}_S \subseteq I_S$ are the largest sets such that $x \in A^{<i_1,\ldots,i_S>}$ for all $i_1 \in \hat{I}_1, \ldots, i_S \in \hat{I}_S$. The function is defined as:

$$f'^{<i_1|\ldots|i_S>}(\underbrace{x'}_{\in B_1^{<\ldots>}}) := f^{<i_1|\ldots|i_S>}(\underbrace{x}_{\in A^{<\ldots>}})(\underbrace{x'}_{\in B_1^{<\ldots>}})$$

We allow *generalizing* meta-parameters. For example, when generalizing meta-parameters that express input sizes, we allow using the corresponding meta-function on arbitrarily sized inputs (a.k.a. *dynamic size* in programming). In our generated code, meta-parameters are available at compile time such that concrete meta-parameter values can be exploited for generating well-performing code (e.g., for setting static loop boundaries). Consequently, meta-parameter generalization increases the expressivity of the generated code (e.g., by being able to process differently sized inputs without re-compilation for unseen input sizes), but usually at the cost of performance, because generalized meta-parameters cannot be exploited during code generation.

Definition 23 (Generalized Meta-Parameters)**.** Let

$$f^{<i_1 \in I_1 | \ldots | i_s \in I_s | \ldots | i_S \in I_S>} : A^{<i_1,\ldots,i_s,\ldots,i_S>} \to B^{<i_1,\ldots,i_s,\ldots,i_S>}$$

be a meta-function (meta-parameters of $I_1, \ldots, I_S$ omitted for brevity) such that

$$f^{<i_1|\ldots|i_s|\ldots|i_S>}(x) = f^{<i_1|\ldots|i'_s|\ldots|i_S>}(x)$$

i.e., f's behavior is invariant under different values of meta-parameter i_s in stage s.

The *generalization* of f in meta-parameter $s \in [1, S]_{\mathbb{N}}$ is the meta-function

$$f'^{<i_1 \in I_1 \mid \ldots \mid i_{s-1} \in I_{s-1} \mid i_{s+1} \in I_{s+1} \mid \ldots \mid i_S \in I_S>} : \bigcup_{i_s \in I_s^{<i_1,\ldots,i_{s-1}>}} A^{<i_1 \mid \ldots \mid i_{s-1} \mid i_s \mid i_{s+1} \mid \ldots \mid i_S>} \to \bigcup_{i_s \in I_s^{<i_1,\ldots,i_{s-1}>}} B^{<i_1 \mid \ldots \mid i_{s-1} \mid i_s \mid i_{s+1} \mid \ldots \mid i_S>}$$

which is defined as:

$$f'^{<i_1 \mid \ldots \mid i_{s-1} \mid i_{s+1} \mid \ldots \mid i_S>}(x) := f'^{<i_1 \mid \ldots \mid i_s \mid i_{s+1} \mid \ldots \mid i_S>}(x)$$

for an arbitrary $i_s \in I_s$ such that $x \in A^{<i_1 \mid \ldots \mid i_{s-1} \mid i_s \mid i_{s+1} \mid \ldots \mid i_S>}$.

We write for f's type also

$$f^{<i_1 \in I_1 \mid \ldots \mid i_{s-1} \in I_{s-1} \mid * \in I_s \mid i_{s+1} \in I_{s+1} \mid \ldots \mid i_S \in I_S>} : A^{<i_1,\ldots,i_S>} \to B^{<i_1,\ldots,i_S>}$$

where i_s is replaced by $*$, and for access to f

$$f^{<i_1 \mid \ldots \mid i_{s-1} \mid * \mid i_{s+1} \mid \ldots \mid i_S>}(x)$$

We use *postponed meta-parameters* to change the order of meta-parameters of already defined meta-functions. For example, we use postponed meta-parameters in Definition 32 to compute the values of meta-parameters based on the particular meta-parameter values of later stages.

Definition 24 (Postponed Meta-Parameters)**.** Let

$$f^{<i_1 \in I_1 | \ldots | i_s \in I_s | \ldots | i_S \in I_S>} : A^{<i_1,\ldots,i_s,\ldots,i_S>} \to B^{<i_1,\ldots,i_s,\ldots,i_S>}$$

be a meta-function (meta-parameters of $I_1, \ldots, I_S$ omitted via ellipsis for brevity) such that for each $k \in (s, S]_{\mathbb{N}}$, it holds:

$$I_k^{<i_1 \mid \ldots \mid i_s \mid \ldots \mid i_{k-1}>} = I_k^{<i_1 \mid \ldots \mid i'_s \mid \ldots \mid i_{k-1}>}$$

i.e., the I_k are invariant under different values of meta-parameter i_s in stage s (i.e., i_s can be ignored in the parameter list of I_k).

Function f' is function f *postponed* on stage s to meta-type

$$\hat{I}_s^{<i_1 \mid \ldots \mid i_{s-1} \mid i_{s+1} \mid \ldots \mid i_S>} \subseteq I_s^{<i_1 \mid \ldots \mid i_{s-1}>}$$

which, in contrast to I_s, may also depend on meta-parameter values $i_{s+1}, \ldots, i_S$, iff f' is of type

$$f'^{<i_1 \in I_1^{<\ldots>} \mid \ldots \mid i_{s-1} \in I_{s-1}^{<\ldots>} \mid i_{s+1} \in I_{s+1}^{<\ldots>} \mid \ldots \mid i_S \in I_S^{<\ldots>} > < i_s \in \hat{I}_s^{<i_1,\ldots,i_S>} >} :$$
$$A^{<i_1,\ldots,i_s,\ldots,i_S>} \to B^{<i_1,\ldots,i_s,\ldots,i_S>}$$

and defined as:

$$f'^{<i_1 \mid \ldots \mid i_{s-1} \mid i_{s+1} \mid \ldots \mid i_S > < i_s >}(a) = f^{<i_1 \mid \ldots \mid i_{s-1} \mid i_s \mid i_{s+1} \mid \ldots \mid i_S >}(a)$$

We write for f''s type also

$$f^{<i_1 \in I_1^{<\ldots>} \mid \ldots \mid i_{s-1} \in I_{s-1}^{<\ldots>} \mid \to \mid i_{s+1} \in I_{s+1}^{<\ldots>} \mid \ldots \mid i_S \in I_S^{<\ldots>} > < i_s \in \hat{I}_s^{<\ldots>} >} :$$
$$A^{<i_1,\ldots,i_s,\ldots,i_S>} \to B^{<i_1,\ldots,i_s,\ldots,i_S>}$$

where f' is replaced by f and i_s by symbol "$\to$". For access to f', we write

$$f^{<i_1 \mid \ldots \mid i_{s-1} \mid \to \mid i_{s+1} \mid \ldots \mid i_S > < i_s >}(x)$$

When using a binary function for combining a family of elements, we often use the following notation.

Notation 5 (Iterative Function Application)**.** Let $\circledast : T \times T \to T$ be an arbitrary associative and commutative binary function on scalar type $T \in \texttt{TYPE}$. Let further x be an arbitrary family that has index set $I := \{i_1, \ldots, i_N\}$ and image set $\{x_i\}_{i \in I} \subseteq T$.

We write $\underset{i \in I}{\circledast} x_i$ instead of $x_{i_1} \circledast \ldots \circledast x_{i_N}$ (infix notation).

.1.4 *MatVec Expressed in MDH DSL*

Our MatVec example from Figure 64 is expressed in MDH's Python-based high-level *Domain-Specific Language (DSL)*, used as input by our *MDH compiler* [3], as follows:

```
1  def matvec(T: ScalarType, I: int, K: int):
2      @mdh( out( w = Buffer[T, [I]]                        ) ,
3            inp( M = Buffer[T, [I, K]], v = Buffer[T, [K]] ) )
4      def mdh_matvec():
5          def mul(out, inp):
6              out['w'] = inp['M'] * inp['v']
7
8          def scalar_plus(res, lhs, rhs):
9              res['w'] = lhs['w'] + rhs['w']
10
11         return (
12             out_view[T]( w = [lambda i, k: (i)] ),
13               md_hom[I, K]( mul, ( CC, PW(scalar_plus) ) ),
14                 inp_view[T, T]( M = [lambda i, k: (i, k)] ,
15                                 v = [lambda i, k: (k)   ] )
16         )
```

Listing 45: MatVec expressed in MDH's Python DSL.

Our MDH compiler takes an expression as in Listing 45 as input, and it fully automatically generates auto-tuned program code from it, according to the methodologies presented in this thesis (particularly in Chapter 4).

We rely on a Python-based DSL, because Python is becoming increasingly popular in both academia and industry [46]. Our future work aims to offer our MDH-based DSL in further approaches, e.g., the *MLIR* compiler framework [56] to make our approach better accessible to the community.

.2 FULL VERSION: SECTION 4.2

We introduce functional building blocks, in the form of higher-order functions, that express data-parallel computations on a high abstraction level. The goal of our high-level abstraction is to express computations agnostic of hardware and optimization details, and thus in a user-productive manner, while still capturing all information relevant for generating high-performance program code. We introduce the building blocks of our abstraction based on our novel algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)* which formalizes data parallelism.

Figure 63 shows a basic overview of our high-level representation. We express data-parallel computations using exactly three higher-order functions only (a.k.a. *patterns* or *skeletons* [243] in programming terminology): 1) `inp_view` transforms the domain-specific input data (e.g., a matrix and a vector in the case of matrix-vector multiplication) to a *Multi-Dimensional Array (MDA)* which is our internal data representation and defined later in this section; 2) `md_hom` expresses the data-parallel computation; 3) `out_view` transforms the computed MDA back to the domain-specific data representation.

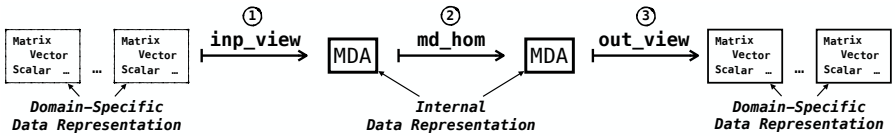


Figure 63: High-level representation (overview).

In the following, after informally discussing an introductory example in Section .2.1, we formally define and discuss each higher-order function in detail in Section .2.2 (function `md_hom`) and Section .2.3 (functions `inp_view` and `out_view`). Note that Section .2.2 and Section .2.3 introduce and present the internals and formal details of our approach, which are not relevant for the end user of our system – the user only needs to operate on the abstraction level discussed in Section .2.1.

.2.1 *Introductory Example*

Figure 64 shows how our high-level representation is used for expressing the example of matrix-vector multiplication `MatVec`[2] (Figure 7). Computation `MatVec` takes as input a matrix $M \in T^{I \times K}$ and vector $v \in T^{K}$ of arbitrary scalar type[3] T and sizes $I \times K$ (matrix) and K (vector), for arbitrary but fixed positive natural numbers $I, K \in \mathbb{N}$[4]. In the figure, based on index function $(i,k) \to (i,k)$ and $(i,k) \to (k)$, high-level function `inp_view` computes a function that takes M and v as input and maps them to a 2-dimensional array of size $I \times K$ (referred to as *input MDA* in the following and defined formally in the next subsection). The MDA contains at each point (i,k) the pair $(M_{i,k}, v_k) \in T \times T$ comprising element $M_{i,k}$ within matrix M (first component) and element v_k within vector v (second component). The input MDA is then mapped via function `md_hom` to an output MDA of size $I \times 1$, by applying multiplication $*$ to each pair $(M_{i,k}, v_k)$ within the input MDA, and combining the obtained intermediate results within the MDA's first dimension via ++ (concatenation – also defined formally in the next subsection) and in second dimension via + (point-wise addition). Finally, function `out_view` computes a function that straightforwardly maps the output MDA, of size $I \times 1$, to `MatVec`'s result vector $w \in T^{I}$, which has scalar type T and is of size I. For the example of `MatVec`, the output view is trivial, but it can be used in other computations (such as matrix multiplication) to conveniently express more advanced variants of computations (e.g., computing the result matrix of matrix multiplication as transposed, as we demonstrate later).

```
MatVec<T∈TYPE|I,K∈ℕ> :=
        out_view<T>( w:(i,k)↦(i) ) ∘
            md_hom<I,K>( ∗, (++,+) ) ∘
                inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

Figure 64: High-level expression for Matrix-Vector Multiplication (`MatVec`).[5]

[2] The expression in Figure 64 can also be extracted from straightforward, annotated sequential code [81, 82].

[3] We consider as *scalar types* integers $\mathbb{Z}$ (a.k.a. `int` in programming), floating point numbers $\mathbb{Q}$ (a.k.a. `float` or `double`), any fixed collection of types (a.k.a. *record* or *struct*), etc. We denote the set of scalar types as `TYPE` in the following. Details on scalar types are provided in the Appendix, Section .1.2, for the interested reader.

[4] We denote by $\mathbb{N}$ the set of positive natural number $\{1, 2, \dots\}$, and we use $\mathbb{N}_0$ for the set of natural numbers including 0.

.2.2 *Function* `md_hom`

We introduce higher-order function `md_hom` to express *Multi-Dimensional Homomorphisms (MDHs)* – our formal representation of data-parallel computations – in a convenient and structured way. In the following, we recapitulate the definition of MDHs and function `md_hom`, but in a more general and formally more precise setting than done in the original MDH work.

In order to define MDH functions, we first need to introduce two central building blocks used in the definition of MDHs: i) *Multi-Dimensional Arrays (MDAs)* – the data type on which MDHs operate and which uniformly represent domain-specific input and output data (scalar, vectors, matrices, ...), and ii) *Combine Operators* which we use to combine elements within a particular dimension of an MDA.

Multi-Dimensional Arrays

Definition 25 (Multi-Dimensional Array)**.** Let $\texttt{MDA-IDX-SETs} := \{I \subset \mathbb{N}_0 \mid |I| < \infty\}$ be the set of all finite subsets of natural numbers; in the context of MDAs, we refer to the subsets also as *MDA index sets*. Let further $T \in \texttt{TYPE}$ be an arbitrary scalar type, $D \in \mathbb{N}$ a natural number, $I := (I_1, \ldots, I_D) \in \texttt{MDA-IDX-SETs}^D$ a tuple of D-many MDA index sets, and $N := (N_1, \ldots, N_D) := (|I_1|, \ldots, |I_D|)$ the tuple of index sets' sizes.

A *Multi-Dimensional Array (MDA)* a that has *dimensionality* D, *size* N, *index sets* I, and *scalar type* T is a function with the following signature:

$$a : I_1 \times \ldots \times I_D \to T$$

We refer to $I_1 \times \ldots \times I_D \to T$ as the *type* of MDA a.

Notation 6. For better readability, we denote MDAs' types and accesses to them using a notation close to programming. We often write:

- $a \in T[\, I_1, \ldots, I_D \,]$ instead of $a : I_1 \times \ldots \times I_D \to T$ to denote the type of MDA a;
- $a \in T[\, N_1, \ldots, N_D \,]$ instead of $a : [0, N_1)_{\mathbb{N}_0} \times \cdots \times [0, N_D)_{\mathbb{N}_0} \to T$;[6]
- $a[\, i_1, \ldots, i_D \,]$ instead of $a(i_1, \ldots, i_D)$ to access MDA a at position $(i_1, \ldots, i_D)$.

[5] Our technical implementation takes as input a representation that is equivalent to Figure 64, expressed via straightforward program code (see Appendix, Section .1.4)

[6] We denote by $[L, U)_{\mathbb{N}_0} := \{\, n \in \mathbb{N}_0 \mid L \leq n < U \,\}$ the half-open interval of natural numbers (including 0) between L (incl.) and U (excl.).

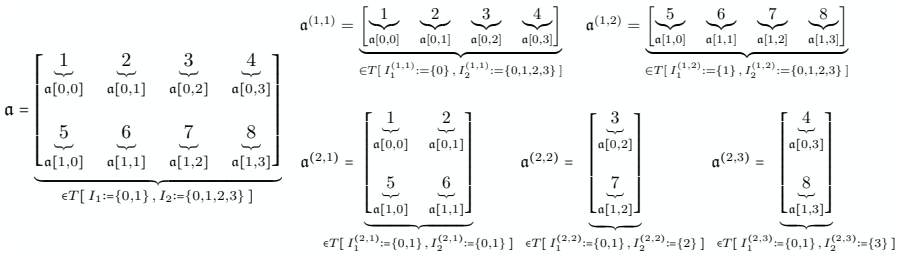


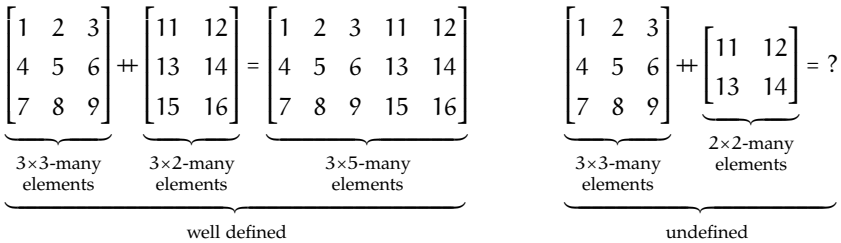
Figure 65: MDA examples.

Figure 65 shows six MDAs for illustration. The left part of the figure shows MDA $\mathfrak{a}$ which is of type $\mathfrak{a} : I_1 \times I_2 \to T$, for $I_1 = \{0,1\}$, $I_2 = \{0,1,2,3\}$, and $T = \mathbb{Z}$ (integer numbers). On the right side, five MDAs are shown, named $\mathfrak{a}^{(1,1)}$, $\mathfrak{a}^{(1,2)}$, $\mathfrak{a}^{(2,1)}$, $\mathfrak{a}^{(2,2)}$, $\mathfrak{a}^{(2,3)}$ – the superscripts are part of the names and represent a two-dimensional numbering of the five MDAs. The MDAs $\mathfrak{a}^{(1,1)}$ and $\mathfrak{a}^{(1,2)}$ are of types $\mathfrak{a}^{(1,1)} : I_1^{(1,1)} \times I_2^{(1,1)} \to T$ and $\mathfrak{a}^{(1,2)} : I_1^{(1,2)} \times I_2^{(1,2)} \to T$, for $I_1^{(1,1)} = \{0\}$ and $I_1^{(1,2)} = \{1\}$, and coinciding second dimensions $I_2^{(1,1)} = I_2^{(1,2)} = \{0,1,2,3\}$. The MDAs $\mathfrak{a}^{(2,1)}$, $\mathfrak{a}^{(2,2)}$, and $\mathfrak{a}^{(2,3)}$ are of types $\mathfrak{a}^{(2,1)} : I_1^{(2,1)} \times I_2^{(2,1)} \to T$, $\mathfrak{a}^{(2,2)} : I_1^{(2,2)} \times I_2^{(2,2)} \to T$, and $\mathfrak{a}^{(2,3)} : I_1^{(2,3)} \times I_2^{(2,3)} \to T$, and they coincide in their first dimensions $I_1^{(2,1)} = I_1^{(2,2)} = I_1^{(2,3)} = \{0,1\}$; their second dimensions are $I_2^{(2,1)} = \{0,1\}$, $I_2^{(2,2)} = \{2\}$, and $I_2^{(2,2)} = \{3\}$. Note that MDAs $\mathfrak{a}^{(1,1)}, \mathfrak{a}^{(1,2)}, \mathfrak{a}^{(2,1)}, \mathfrak{a}^{(2,2)}, \mathfrak{a}^{(2,3)}$ can be considered as *parts* (a.k.a. *tiles* in programming) of MDA $\mathfrak{a}$. We formally define and use *partitionings* of MDAs in Section .4.

Combine Operators

A central building block in our definition of MDHs is a *combine operator*. Intuitively, we use a combine operator to combine all elements within a particular dimension of an MDA. For example, in Figure 7 (matrix-vector multiplication), we combine elements of the 2-dimensional MDA via combine operator *concatenation* in MDA's first dimension and via operator *point-wise addition* in the second dimension.

Technically, combine operators are functions that take as input two MDAs and yield a single MDA as their output (formal definition follows soon). By definition, we require that the index sets of the two input MDAs coincide in all dimensions except in the dimension to combine; thereby, we catch undefined cases already at the type level, e.g., trying to concatenate improperly sized MDAs:



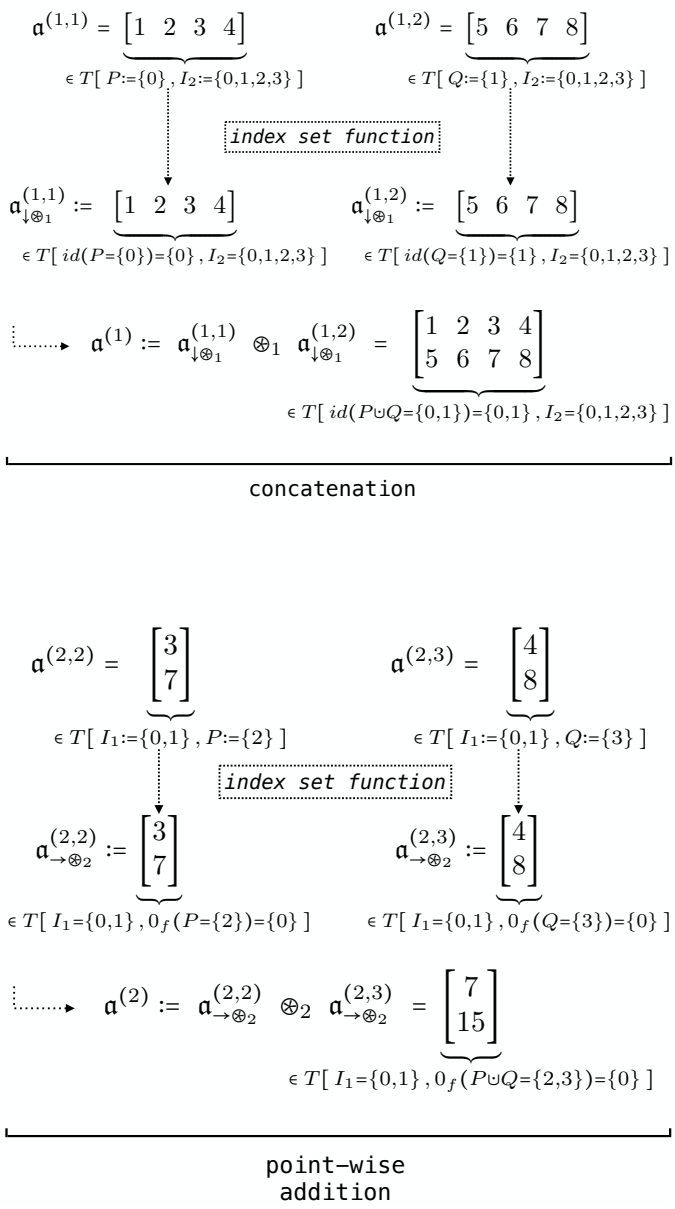


Figure 66: Illustration of *combine operators* using the examples *concatenation* (top part of figure) and *point-wise addition* (bottom part).

Here, on the left, we can reasonably define the concatenation of MDAs that contain 3×3-many elements and 3×2 elements. However, as indicated in the right part of the figure, it is not possible to intuitively concatenate MDAs of sizes 3×3 and 2×2, as the MDAs do not match in their number of elements in any of the two dimensions.

Figure 66 illustrates combine operators informally using the example operators *concatenation* (left part of the figure) and *point-wise addition* (right part). We illustrate concatenation using the example MDAs $\mathfrak{a}^{(1,1)}$ and $\mathfrak{a}^{(1,2)}$ from Figure 65; for point-wise addition, we use MDAs $\mathfrak{a}^{(2,2)}$ and $\mathfrak{a}^{(2,3)}$ from Figure 65 (all MDAs are chosen arbitrarily, and the example works the same for other MDAs).

In the case of concatenation (left part of Figure 66), MDAs $a^{(1,1)}$ and $a^{(1,2)}$ coincide in their second dimension $I_2 := \{0,1,2,3\}$, which is important, because we concatenate in the first dimension, thus requiring coinciding index sets in all other dimensions (as motivated above). In the case of the point-wise addition example (right part of Figure 66), the example MDAs $a^{(2,2)}$ and $a^{(2,3)}$ coincide in their first dimension $I_1 := \{0,1\}$, as required for combining the MDAs in the second dimension. The varying index sets of the four MDAs are denoted as P and Q in the figure, which are in the case of the concatenation example, index sets in the first dimension of MDAs $a^{(1,1)}$ and $a^{(1,2)}$; in the case of the point-wise addition example, the varying index sets of MDAs $a^{(2,1)}$ and $a^{(2,2)}$ belong to the second dimensions.

In the following, we assume w.l.o.g. that the varying index sets P and Q of MDAs to combine are disjoint. Our assumption will not be a limitation for our approach: we will apply combine operators always to parts of MDAs that belong to the same MDA, causing that the index sets of the parts are disjoint by construction. For example, in the case of the concatenation example in Figure 66, the parts $a^{(1,1)}$ and $a^{(1,2)}$ of MDA a correspond to the first and second row of the same MDA a in Figure 65 and thus have different index sets in their first dimension, and in the case of the point-wise addition example in Figure 66, the parts $a^{(2,2)}$ and $a^{(2,3)}$ represent the third and fourth column of MDA a and thus have different index sets in their second dimension.

We define combine operators based on *index set functions* (also defined formally soon). Index set functions precisely describe, on the type level, the index set of the combined output MDA and thus how an MDA's index set evolves during combination. For this, an index set function takes as input the input MDA's index set in the dimension to combine, and the function yields as its output the index set of the output MDA which is combined in this dimension. In the case of the concatenation example in Figure 66, the index set function is identity id and thus trivial. However, in the case of point-wise addition, the corresponding index set function is the constant function 0_f which maps any index set to the singleton set $\{0\}$ containing index 0 only. This is because when combining via point-wise addition, the MDA shrinks in the combined dimension to only one element which we aim to uniformly access via MDA index 0. In Figure 66, we denote MDAs $a^{(1,1)}$, $a^{(1,2)}$, $a^{(2,2)}$, $a^{(2,3)}$ after applying the corresponding index set function as: $a^{(1,1)}_{\downarrow\circledast_1}$, $a^{(1,2)}_{\downarrow\circledast_1}$, $a^{(2,2)}_{\rightarrow\circledast_2}$, $a^{(2,3)}_{\rightarrow\circledast_2}$; the combined MDAs are denoted as $a^{(1)}$ and $a^{(2)}$ in the figure. The concatenation operator is denoted in the figure generically as $\circledast_1$, and point-wise addition is denoted as $\circledast_2$, correspondingly.

We now define *combine operators* formally, and we illustrate this formal definition afterward using the example operators *concatenation* and *point-wise combination*. For the interested reader, details on some technical design decisions of combine operators are outlined in the Appendix, Section .3.1.

Definition 26 (Combine Operator)**.** Let $\texttt{MDA-IDX-SETs} \dot{\times} \texttt{MDA-IDX-SETs} := \{ (P, Q) \in \texttt{MDA-IDX-SETs} \times \texttt{MDA-IDX-SETs} \mid P \cap Q = \varnothing \}$ denote the set of all pairs of MDA index sets that are disjoint. Let further $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}} : \texttt{MDA-IDX-SETs} \to \texttt{MDA-IDX-SETs}$ be a function on MDA index sets, $T \in \texttt{TYPE}$ a scalar type, $D \in \mathbb{N}$ an MDA dimensionality, and $d \in [1, D]_{\mathbb{N}}$ an MDA dimension.

We refer to any binary function $\circledast$ of type (parameters in angle brackets are type parameters)

$$\circledast^{<(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)\in\texttt{MDA-IDX-SETs}^{D-1},(P,Q)\in\texttt{MDA-IDX-SETs}\dot{\times}\texttt{MDA-IDX-SETs}>} :$$
$$T[\, I_1, \ldots, \underbrace{\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P)}_{\uparrow d}, \ldots, I_D\,] \times T[\, I_1, \ldots, \underbrace{\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(Q)}_{\uparrow d}, \ldots, I_D\,]$$
$$\to T[\, I_1, \ldots, \underbrace{\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P \uplus Q)}_{\uparrow d}, \ldots, I_D\,]$$

as *combine operator* that has *index set function* $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}$, *scalar type* T, *dimensionality* D, and *operating dimension* d. We denote combine operator's type concisely as $\texttt{CO}^{<\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}} \mid T \mid D \mid d>}$.

Since function $\circledast$'s ordinary function type $T[\ldots] \times T[\ldots] \to T[\ldots]$ is generic in parameters $(I_1, \ldots, I_{d-1}, I_{d+1}, \ldots, I_D)$ and (P, Q) (these type parameters are denoted in angle brackets in Definition 26), we refer to function $\circledast$ as *meta-function*, to the type parameters in angle brackets as *meta-parameters*, and we say *meta-types* to $T[\, I_1, \ldots, \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P), \ldots, I_D\,]$ (first input MDA), $T[\, I_1, \ldots, \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(Q), \ldots, I_D\,]$ (second input MDA), and $T[\, I_1, \ldots, \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P \uplus Q), \ldots, I_D\,]$ (output MDA) as these types are generic in meta-parameters. Formal definitions and details about our meta-parameter concept are provided in Section .1 of our Appendix for the interested reader.

We use meta-functions as an analogous concept to *metaprogramming* in programming language theory to achieve high generality. For example, by defining combine operators as meta-functions, we can use the operators on input MDAs that operate on arbitrary index sets while still guaranteeing correctness, e.g., that index sets of the two input MDAs match in all dimensions except in the dimension to combine (as discussed above). For simplicity, we often refrain from explicitly stating meta-parameters when they are clear from the context; for example, when they can be deduced from the types of their particular inputs (a.k.a. *type deduction* in programming).

We now formally discuss the example operators *concatenation* and *point-wise combination*. For high flexibility, we define both operators generically in the scalar type T of their input and output MDAs, the MDAs' dimensionality D, as well as in the dimension d to combine.

Example 13 (Concatenation)**.** We define *concatenation* as function $+\!\!+$ of type

$$+\!\!+^{<T\in\texttt{TYPE}\,|\,D\in\mathbb{N}\,|\,d\in[1,D]_{\mathbb{N}}\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)\in\texttt{MDA-IDX-SETs}^{D-1},\ (P,Q)\in\texttt{MDA-IDX-SETs}\,\dot{\times}\,\texttt{MDA-IDX-SETs}>}:$$

$$T[\,I_1,\ldots,\underbrace{\mathrm{id}(P)}_{\uparrow d},\ldots,I_D\,]\ \times\ T[\,I_1,\ldots,\underbrace{\mathrm{id}(Q)}_{\uparrow d},\ldots,I_D\,]$$

$$\to\ T[\,I_1,\ldots,\underbrace{\mathrm{id}(P\cup Q)}_{\uparrow d},\ldots,I_D\,]$$

where $\mathrm{id}:\texttt{MDA-IDX-SETs}\to\texttt{MDA-IDX-SETs}$ is the identity function on MDA index sets.
The function is computed as:

$$+\!\!+^{<T\,|\,D\,|\,d\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D),(P,Q)>}(a_1,a_2)[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,]$$

$$:=\begin{cases} a_1[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] & ,\ i_d\in P \\ a_2[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] & ,\ i_d\in Q \end{cases}$$

The function is well defined, because P and Q are disjoint. We usually use an infix notation for $+\!\!+^{<\ldots>}$ (meta-parameters omitted via ellipsis), i.e., we write $a_1 +\!\!+^{<\ldots>} a_2$ instead of $+\!\!+^{<\ldots>}(a_1,a_2)$.

The vertical bar in the superscript of $+\!\!+$ denotes that function $+\!\!+$ can be partially evaluated (a.k.a. *Currying* [320] in math and *multi staging* [302] in programming) for particular values of meta-parameters: $T\in\texttt{TYPE}$ (first stage), $D\in\mathbb{N}$ (second stage), etc. Partial evaluation (formally defined in the Appendix, Definition 21) enables both: 1) expressive typing and thus better error elimination: for example, parameter $(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)\in\texttt{MDA-IDX-SETs}^{D-1}$ can depend on meta-parameter $D\in\mathbb{N}$, because D is defined in an earlier stage, which allows precisely limiting the length of the tuple $(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)$ to $D-1$ index sets; 2) generality: for example, we can instantiate $+\!\!+$ to $+\!\!+^{<T>}$ which is specific for a particular scalar type $T\in\texttt{TYPE}$, but still generic in meta-parameters $D\in\mathbb{N}$, $d\in[1,D]_{\mathbb{N}}$, ..., as these meta-parameters are defined in later stages. We specify stages and their order according to the recommendations in Haskell Wiki [211], e.g., using earlier stages for meta-parameters that are expected to change less frequently than other meta-parameters.

It is easy to see that $+\!\!+^{<T\,|\,D\,|\,d>}$ is a combine operator of type $\texttt{CO}^{<i\,d\,|\,T\,|\,D\,|\,d>}$ for any particular choice of meta-parameters $T\in\texttt{TYPE}$, $D\in\mathbb{N}$, and $d\in[1,D]_{\mathbb{N}}$.

Example 14 (Point-Wise Combination)**.** We define *point-wise combination*, according to a binary function $\oplus : T \times T \to T$ (e.g. addition), as function $\vec{\bullet}$ of type

$$\vec{\bullet}^{<T \in \texttt{TYPE} \mid D \in \mathbb{N} \mid d \in [1,D]_{\mathbb{N}} \mid (I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D) \in \texttt{MDA-IDX-SETs}^{D-1},\ (P,Q) \in \texttt{MDA-IDX-SETs} \dot{\times} \texttt{MDA-IDX-SETs}>} :$$

$$\underbrace{T \times T \to T}_{\oplus} \ \to\ T[I_1,\ldots,\underbrace{0_f(P)}_{\uparrow d},\ldots,I_D] \times T[I_1,\ldots,\underbrace{0_f(Q)}_{\uparrow d},\ldots,I_D]$$

$$\to T[I_1,\ldots,\underbrace{0_f(P \uplus Q)}_{\uparrow d},\ldots,I_D]$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\texttt{point-wise combination (according to } \oplus\texttt{)}}$$

where $0_f : \texttt{MDA-IDX-SETs} \to \texttt{MDA-IDX-SETs}$, $I \mapsto \{0\}$ is the constant MDA index set function that maps any index set I to the index set containing MDA index 0 only. The function is computed as:

$$\vec{\bullet}^{<T \mid D \mid d \mid (I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D),(P,Q)>}(\oplus)(a_1,a_2)[i_1,\ldots,\underbrace{0}_{\uparrow d},\ldots,i_D] :=$$

$$a_1[i_1,\ldots,\underbrace{0}_{\uparrow d},\ldots,i_D] \oplus a_2[i_1,\ldots,\underbrace{0}_{\uparrow d},\ldots,i_D]$$

We often write $\oplus$ only, instead of $\vec{\bullet}(\oplus)$, and we usually use an infix notation for $\oplus$.

Function $\vec{\bullet}^{<T \mid D \mid d>}(\oplus)$ (meaning: $\vec{\bullet}$ is partially applied to ordinary function parameter $\oplus$ and thus still generic in parameters $(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)$ and (P,Q) – formal details provided in the Appendix, Definition 22) is a combine operator of type $\texttt{CO}^{<0_f \mid T \mid D \mid d>}$ for any binary operator $\oplus : T \times T \to T$.

Multi-Dimensional Homomorphisms

Now that we have defined MDAs (Definition 25) and combine operators (Definition 26), we can define *Multi-Dimensional Homomorphisms (MDHs)*. Intuitively, a function h operating on MDAs is an MDH iff we can apply the function independently to parts of its input MDA and combine the obtained intermediate results to the final result using combine operators; this can be imagined as a typical divide-and-conquer pattern. Compared to classic approaches, e.g., *list homomorphisms* [289, 306, 313], a major characteristic of MDH functions is that they allow (de/re)-composing computations in multiple dimensions (e.g., in Figure 7, in both the concatenation dimension as well as in the point-wise addition dimensions), rather than being limited to a particular dimension only (e.g., only the concatenation dimension or only point-wise addition dimension, respectively). We will see later that a multi-dimensional (de/re)-composition approach is essential to efficiently exploit the hardware of modern architectures which require fine-grained cache blocking and parallelization strategies to achieve their full performance potential.

Figure 67 illustrates the MDH property informally on a simple, two-dimensional input MDA. In the left part of the figure, we split the input MDA in dimension 1 (i.e., horizontally) into two parts a_1 and a_2, apply the MDH function h independently to each part, and combine the obtained intermediate results to the final result using the MDH function h's combine operator $\circledast_1$. Similarly, in the right part of Figure 67, we split the input MDA in dimension 2 (i.e., vertically) into parts and combine the results via MDH function h's second combine operator $\circledast_2$.

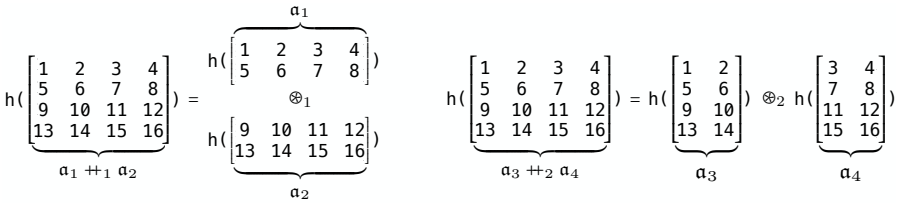


Figure 67: MDH property illustrated on a two-dimensional example computation.

Figure 68 shows an artificial example in which we apply the MDH property (illustrated in Figure 67) recursively. We refer in Figure 68 to the part above the horizontal dashed lines as *de-composition phase* and to the part below dashed lines as *re-composition* phase.

Definition 27 (Multi-Dimensional Homomorphism)**.** Let $T^{INP}, T^{OUT} \in$ `TYPE` be two arbitrary scalar types, $D \in \mathbb{N}$ a natural number, and $\overset{1}{\Rightarrow}_{MDA}, \ldots, \overset{D}{\Rightarrow}_{MDA} :$ `MDA-IDX-SETs` $\rightarrow$ `MDA-IDX-SETs` functions on MDA index sets. Let further $+\!\!+_d := +\!\!+^{<T^{INP} \mid D \mid d>} \in$ `CO`$^{<id \mid T^{INP} \mid D \mid d>}$ denote concatenation (Definition 13) in dimension $d \in [1, D]_{\mathbb{N}}$ on D-dimensional MDAs that have scalar type T^{INP}.

A function

$$h^{<I_1,\ldots,I_D \in \texttt{MDA-IDX-SETs}>} : T^{INP}[\, I_1, \ldots, I_D \,] \rightarrow T^{OUT}[\, \overset{1}{\Rightarrow}_{MDA}(I_1), \ldots, \overset{D}{\Rightarrow}_{MDA}(I_D) \,]$$

is a *Multi-Dimensional Homomorphism (MDH)* that has *input scalar type* T^{INP}, *output scalar type* T^{OUT}, *dimensionality* D, and *index set functions* $\overset{1}{\Rightarrow}_{MDA}, \ldots, \overset{D}{\Rightarrow}_{MDA}$, iff for each $d \in [1, D]_{\mathbb{N}}$, there exists a combine operator $\circledast_d \in$ `CO`$^{<\overset{d}{\Rightarrow}_{MDA} \mid T^{OUT} \mid D \mid d>}$ (Definition 26), such that for any concatenated input MDA $a_1 +\!\!+_d a_2$ in dimension d, the *homomorphic property* is satisfied:

$$h(\, a_1 +\!\!+_d a_2 \,) = h(a_1) \circledast_d h(a_2)$$

We denote the type of MDHs concisely as $\texttt{MDH}^{<T^{INP}, T^{OUT} \mid D \mid (\overset{d}{\Rightarrow}_{MDA})_{d \in [1,D]_{\mathbb{N}}}>}$.

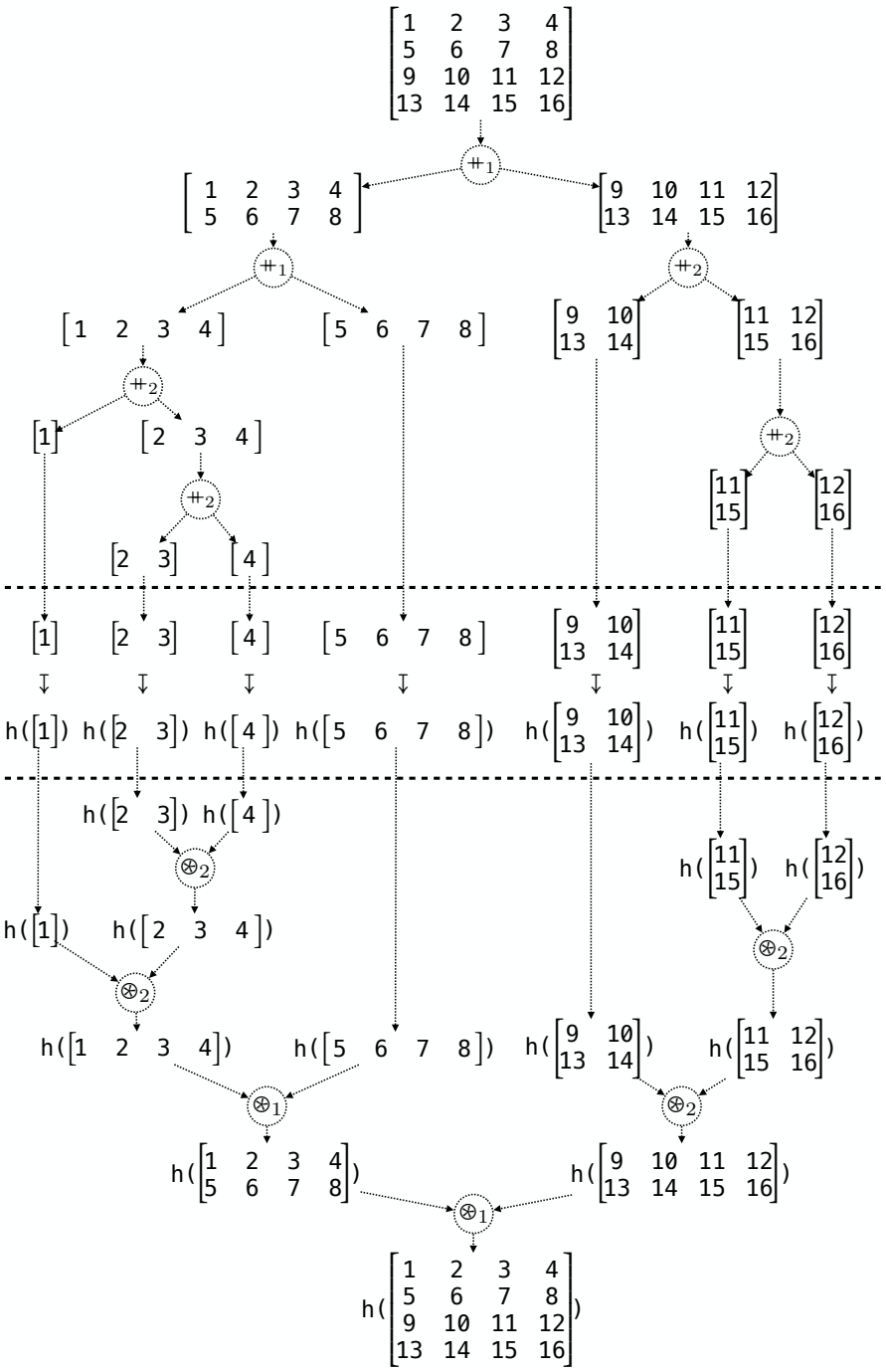


Figure 68: MDH property recursively applied to a two-dimensional example computation.

MDHs are defined such that applying them to a concatenated MDA in dimension d can be computed by applying the MDH h independently to the MDA's parts a_1 and a_2 and combining the intermediate results afterward by using its combine operator $\circledast_d$, as also informally discussed above. Note that by definition of MDHs, their combine operators are associative and commutative (which follows from the associativity and commutativity of $+\!\!+_d$). Note further that for simplicity, Definition 27 is specialized to MDHs whose input algebraic structure relies on concatenation, as such kinds of MDHs already cover the currently practice-relevant data-parallel computations (as we will see later). We provide a generalized definition of MDHs in Section .3.2 of our Appendix, for the algebraically interested reader.

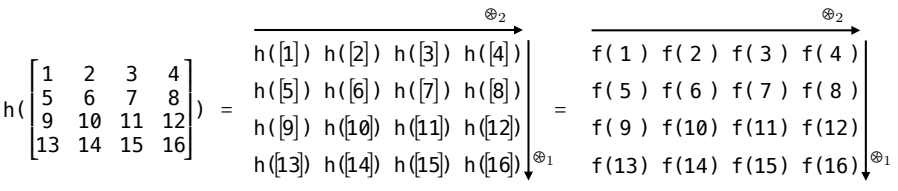
Example 15 (Function Mapping)**.** A simple example MDH is *function mapping* [251], expressed by higher-order function $\texttt{map}(f)(a)$, which applies a user-defined scalar function $f : T^{INP} \to T^{OUT}$ to each element within a D-dimensional MDA a. Function $\texttt{map}(f)$ is an MDH of type $\texttt{MDH}^{<T^{INP},T^{OUT}\,|\,D\,|\,id,\ldots,id>}$ whose combine operators are concatenation $+\!\!+$ in all of its D dimensions (Example 13). Function $\texttt{id}$ is the index set function of $+\!\!+$ (see Example 13) and consequently also of MDH $\texttt{map}(f)$. Formal details and definitions for *function mapping* can be found in the Appendix, Section .3.3.

Example 16 (Reduction)**.** A further MDH function is *reduction* [251], expressed by higher-order function $\texttt{red}(\oplus)(a)$, which combines all elements within a D-dimensional MDA a using a user-defined binary function $\oplus : T \times T \to T$. Reduction is an MDH of type $\texttt{MDH}^{<T,T\,|\,D\,|\,0_f,\ldots,0_f>}$, and its combine operators are point-wise combination $\overrightarrow{\bullet}(\oplus)$ in all dimensions (Example 14), which have 0_f as index set function. Formal details and definitions for *reduction* can be found in the Appendix, Section .3.3.

We show how Examples 15 and 16 (and particularly also more advanced examples) are expressed in our high-level representation in Section .2.5, based on higher-order functions `md_hom`, `inp_view`, and `out_view` (Figure 63) which we introduce in the following.

Higher-Order Function `md_hom`

We define higher-order function `md_hom` which conveniently expresses MDH functions in a uniform and structured manner. For this, we exploit that any MDH function is uniquely determined by its combine operators and its behavior on singleton MDAs, as informally illustrated in the following figure:



Here, f is the function on scalar values that behaves the same as h when restricted to singleton MDAs: $f(a[i_1,\ldots,i_D]) := h(a)$, for any MDA $a \in T[\{i_1\},\ldots,\{i_D\}]$ consisting of only one element that is accessed by (arbitrary) indices $i_1,\ldots,i_D \in \mathbb{N}_0$. For singleton MDAs, we usually use f instead of h, because f can be defined more conveniently by the user as h (which needs to handle MDAs of arbitrary sizes, and not only singleton MDAs as f). Also, since f takes as input a scalar value (rather than a singleton MDA, as h), the type of f also becomes simpler, which further contributes to simplicity.

We now formally introduce function `md_hom` which uniformly expresses any MDH function, by using only the MDH's behavior f on scalar values and the MDH's combine operators.

Definition 28 (Higher-Order Function `md_hom`)**.** The higher-order function `md_hom` is of type

$$\texttt{md\_hom}^{<T^{INP},T^{OUT}\in TYPE \mid D\in\mathbb{N} \mid (\overset{d}{\Rightarrow}_{MDA}^{MDA}:\text{MDA-IDX-SETs}\to\text{MDA-IDX-SETs})_{d\in[1,D]_{\mathbb{N}}}>}:$$

$$\underbrace{SF^{<T^{INP},T^{OUT}>}}_{f} \times \underbrace{\left(CO^{<\overset{1}{\Rightarrow}_{MDA}^{MDA} \mid T^{OUT} \mid D \mid 1>} \times \ldots \times CO^{<\overset{D}{\Rightarrow}_{MDA}^{MDA} \mid T^{OUT} \mid D \mid D>} \right)}_{\circledast_1,\ldots,\circledast_D}$$

$$\to_p \underbrace{MDH^{<T^{INP},T^{OUT} \mid D \mid (\overset{d}{\Rightarrow}_{MDA}^{MDA})_{d\in[1,D]_{\mathbb{N}}}>}}_{\texttt{md\_hom}(\ f\ ,\ (\circledast_1,\ldots,\circledast_D)\ )}$$

where $SF^{<T^{INP},T^{OUT}>}$ denotes the set of scalar functions of type $T^{INP} \to T^{OUT}$. Function `md_hom` is partial (indicated by $\to_p$ instead of $\to$), which we motivate after this definition. The function takes as input a scalar function f and a tuple of D-many combine operators $(\circledast_1,\ldots,\circledast_D)$, and it yields a function $\texttt{md\_hom}(\ f\ ,\ (\circledast_1,\ldots,\circledast_D)\ )$ which is defined as

$$\texttt{md\_hom}(\ f\ ,\ (\circledast_1,\ldots,\circledast_D)\ )(\ a\ ) := \underset{i_1\in I_1}{\circledast_1} \ \ldots \ \underset{i_D\in I_D}{\circledast_D} \ \vec{f}(\ a|_{\{i_1\}\times\ldots\times\{i_D\}}\ )$$

The combine operators' underset notation denotes straightforward iteration (explained formally in the Appendix, Notation 5), and the MDA $a|_{\{i_1\}\times\ldots\times\{i_D\}}$ is the restriction of a to the MDA containing the single element accessed via MDA indices $(i_1,\ldots,i_D)$. Function $\vec{f}$ behaves like scalar function f, but $\vec{f}$ operates on singleton MDAs (rather than scalars). Function $\vec{f}$ is of type

$$\vec{f}^{<i_1,\ldots,i_D\in\mathbb{N}_0>}:$$

$$T^{INP}[\ \{i_1\},\ldots,\{i_D\}\ ] \to T^{OUT}[\ \overset{1}{\Rightarrow}_{MDA}^{MDA}(\ \{i_1\}\ ),\ldots,\overset{D}{\Rightarrow}_{MDA}^{MDA}(\ \{i_D\}\ )\ ]$$

and defined as

$$[\vec{f}(x)[\ j_1,\ldots,j_D\ ] := f(\ x[\ i_1,\ldots,i_D\ ]\ )^7$$

For $\texttt{md\_hom}(\texttt{f}, (\circledast_1,\ldots,\circledast_D))$, we require by definition the homomorphic property (Definition 27), i.e., for each $d \in [1,D]_{\mathbb{N}}$, it must hold:

$$\texttt{md\_hom}(\texttt{f}, (\circledast_1,\ldots,\circledast_D))(a_1 +\!\!+_d a_2) =$$

$$\texttt{md\_hom}(\texttt{f}, (\circledast_1,\ldots,\circledast_D))(a_1) \;\circledast_d\; \texttt{md\_hom}(\texttt{f}, (\circledast_1,\ldots,\circledast_D))(a_2)$$

Using Definition 28, we express any MDH function uniformly via higher-order function `md_hom` using only the MDH's behavior `f` on scalar values[8] and its combine operators $\circledast_1,\ldots,\circledast_D$. The other direction also holds: each function expressed via `md_hom` is an MDH function, because we require the homomorphic property for `md_hom`.

Note that we can potentially allow in Definition 28 the case $D = 0$ in which we would define the `md_hom` instance equal to the scalar function `f`:

$$\texttt{md\_hom}(\texttt{f}, ()) := \texttt{f}$$

Note further that function `md_hom` is defined as partial function, because the homomorphic property is not met for all potential combinations of combine operators, e.g., $\circledast_1 = +$ (point-wise addition) and $\circledast_2 = *$ (point-wise multiplication). However, in many real-world examples, an MDH's combine operators are a mix of concatenations and point-wise combinations according to the same binary function. The following lemma proves that any instance of the `md_hom` higher-order function for such a mix of combine operators is a well-defined MDH function.

Lemma 3. Let $\oplus : T \to T$ be an arbitrary but fixed associative and commutative binary function on scalar type $T \in \texttt{TYPE}$. Let further $\circledast_1,\ldots,\circledast_D$ be combine operators of which any is either concatenation (Example 13) or point-wise combination according to binary function $\oplus$ (Example 14).

It holds that $\texttt{md\_hom}(\texttt{f}, (\circledast_1,\ldots,\circledast_D))$ is well defined.

Proof. See Appendix, Section .3.5. □

MDH functions are defined (Definition 27) such that they uniformly operate on MDAs (Figure 63). We introduce higher-order function `inp_view` to prepare domain-specific inputs (e.g., a matrix and a vector in the case of matrix-vector multiplication) as an MDA, and we use function `out_view` to transform the output MDA back to the domain-specific data requirements (like storing it as a transposed matrix in the case of matrix multiplication, or splitting it into multiple outputs as we will see later with examples). We introduce both higher-order functions in the following.

[8] For simplicity, we ignore that the scalar functions of some MDHs (such as Mandelbrot) also take as input MDA indices, which requires slight, straightforward extension of function `md_hom`, as outlined in the Appendix, Section .3.4.

.2.3 *View Functions*

We start, in Section .2.3.1, by formally introducing *Buffers (BUF)* and *Index Functions* – both concepts are central building blocks in our definition of higher-order functions `inp_view` and `out_view`. In our approach, we use BUFs to represent domain-specific input and output data (scalars, vectors, matrices, etc), and *index functions* are used by the user to conveniently instantiate higher-order functions `inp_view` and `out_view`, e.g., index function $(i,k) \mapsto (i,k)$ and $(i,k) \mapsto (k)$ used in Figure 64 to instantiate function `inp_view`, and $(i,k) \mapsto (i)$ is used for `out_view`.

Sections .2.3.2 and .2.3.3 introduce *input views* and *output views* which are central concepts in our approach. We define *input views* as arbitrary functions that map a collection of BUFs to an MDA (Figure 63); higher-order function `inp_view` is then defined to conveniently compute an important class of input view functions that are relevant for expressing real-world computations. Correspondingly, Section .2.3.3 defines *output views* as functions that transform an MDA to a collection of BUFs, and higher-order function `out_view` is defined to conveniently compute important output views.

Finally, we discuss in Section .2.3.4 the relationship between higher-order function `inp_view` and `out_view`: we prove that both functions are inversely related to each other, allowing arbitrarily switching between our internal MDA representation and our domain-specific BUF representation (as required for our code generation process discussed later).

.2.3.1 *Preparation*

We formally introduce *Buffers (BUF)* and *Index Functions* in the following.

Definition 29 (Buffer)**.** Let $T \in \texttt{TYPE}$ be an arbitrary scalar type, $D \in \mathbb{N}_0$ a natural number[9], and $N := (N_1, \ldots, N_D) \in \mathbb{N}^D$ a tuple of natural numbers.

A *Buffer (BUF)* b that has *dimensionality* D, *size* N, and *scalar type* T is a function with the following signature:

$$b : [0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\bot\}$$

Here, we use $\bot$ to denote the *undefined value*. We refer to $[0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\bot\}$ as the *type* of BUF b, which we also denote as $T^{N_1 \times \ldots \times N_D}$, and we refer to the set $\texttt{BUF-IDX-SETs} := \{[0, N)_{\mathbb{N}_0} \mid N \in \mathbb{N}\}$ as *BUF index sets*. Analogously to Notation 6, we write $b[i_1, \ldots, i_D]$ instead of $b(i_1, \ldots, i_D)$ to avoid a too heavy usage of parentheses.

[9] We use the case $D = 0$ to represent scalar values (formal details provided in the Appendix, Section .3.7).

In contrast to MDAs (Definition 25), a BUF always operates on a contiguous range of natural numbers starting from 0, and a BUF may contain undefined values. These two differences allow straightforwardly transforming BUFs to data structures provided by low-level programming languages (e.g., *C arrays*, as used in OpenMP, CUDA, and OpenCL).

Note that in our generated program code (discussed later in Section .4), we implement MDAs on top of BUFs, as straightforward aliases that access BUFs, so that we do not need to transform MDAs to low-level data structures and/or store them otherwise physically in memory.

Definition 30 (Index Function)**.** Let $D \in \mathbb{N}$ be a natural number (representing an MDA's dimensionality) and $D_b \in \mathbb{N}_0$ (representing a BUF's dimensionality).

An *index function* $\mathfrak{idx}$ from D-dimensional MDA indices to D_b-dimensional BUF indices is any meta-function of type

$$\mathfrak{idx}^{<I_1^{\texttt{MDA}},\ldots,I_D^{\texttt{MDA}} \in \texttt{MDA-IDX-SETs}^D>} : I_1^{\texttt{MDA}} \times \ldots \times I_D^{\texttt{MDA}} \rightarrow I_1^{\texttt{BUF}} \times \ldots \times I_{D_b}^{\texttt{BUF}}$$

for $(I_1^{\texttt{BUF}},\ldots,I_{D_b}^{\texttt{BUF}}) := \Rightarrow_{\texttt{BUF}}^{\texttt{MDA}}(I_1^{\texttt{MDA}},\ldots,I_D^{\texttt{MDA}})$ where $\Rightarrow_{\texttt{BUF}}^{\texttt{MDA}} : \texttt{MDA-IDX-SETs}^D \rightarrow \texttt{BUF-IDX-SETs}^{D_b}$ is an arbitrary but fixed function that maps D-many MDA index sets to D_b-many BUF index sets. We denote the type of index functions as $\texttt{MDA-IDX-to-BUF-IDX}^{<D,D_b \mid \Rightarrow_{\texttt{BUF}}^{\texttt{MDA}}>}$.

In words: Index functions have to be capable of operating on any potential MDA index set. This generality will be required later for using index functions also on parts of MDAs whose index sets are subsets of the original MDA's index sets.

We will use index functions to access BUFs. For example, in the case of `MatVec` (Figure 7), we access its input matrix using index function $(i,k) \mapsto (i,k)$ which is of type

$$\texttt{MDA-IDX-to-BUF-IDX}^{<D:=2,D_b:=2 \mid \Rightarrow_{\texttt{BUF}}^{\texttt{MDA}}(I_1^{\texttt{MDA}},I_2^{\texttt{MDA}}) := [0,\max(I_1^{\texttt{MDA}})]_{\mathbb{N}_0},[0,\max(I_2^{\texttt{MDA}})]_{\mathbb{N}_0}>}$$

and we use index function $(i,k) \mapsto (k)$ to access `MatVec`'s input vector, which is of type

$$\texttt{MDA-IDX-to-BUF-IDX}^{<D:=2,D_b:=1 \mid \Rightarrow_{\texttt{BUF}}^{\texttt{MDA}}(I_1^{\texttt{MDA}},I_2^{\texttt{MDA}}) := [0,\max(I_2^{\texttt{MDA}})]_{\mathbb{N}_0}>}$$

Further examples of index function, e.g., for stencil computation `Jacobi1D`, are presented in the Appendix, Section .3.6, for the interested reader.

.2.3.2 *Input Views*

We define *input views* as any function that compute an MDA from a collection of (user-defined) BUFs. For example, in the case of `MatVec`, its input view takes as input two BUFs – a matrix and a vector – and it yields a two-dimensional MDA containing pairs of matrix and vector elements (illustrated in Figure 7). In contrast, the input view of `Jacobi1D` takes as input a single BUF (representing a vector) only, and it computes an MDA containing triples of BUF elements (Figure 8).

Definition 31 (Input View)**.** An *input view* from B-many BUFs, $B \in \mathbb{N}$, of arbitrary but fixed types $T_b^{N_1^b \times \ldots \times N_{D_b}^b}$, $b \in [1,B]_{\mathbb{N}}$, to an MDA of arbitrary but fixed type $T[I_1,\ldots,I_D]$ is any function iv of type:

$$iv : \underbrace{\bigtimes_{b=1}^{B} T_b^{N_1^b \times \ldots \times N_{D_b}^b}}_{\text{BUFs}} \to_p \underbrace{T[\, I_1, \ldots, I_D \,]}_{\text{MDA}}$$

We denote the type of iv as:

$$\texttt{IV}^{\overbrace{<B \mid (D_b)_{b\in[1,B]_{\mathbb{N}}} \mid (N_1^b,\ldots,N_{D_b}^b)_{b\in[1,B]_{\mathbb{N}}} \mid (T_b)_{b\in[1,B]_{\mathbb{N}}}}^{\text{BUFs' Meta-Parameters}} \mid \overbrace{D \mid I_1,\ldots,I_D \mid T}^{\text{MDA's Meta-Parameters}} >}$$

Example 17 (Input View – `MatVec`)**.** The input view of `MatVec` on a 1024×512 matrix and 512-sized vector (sizes chosen arbitrarily), both of integers $\mathbb{Z}$, is of type

$$\texttt{IV}^{\overbrace{<B=2 \mid D_1=2, D_2=1 \mid (N_1^1=1024, N_2^1=512), (N_1^2=512) \mid T_1=\mathbb{Z}, T_2=\mathbb{Z} \mid}^{\text{BUFs' meta-parameters}} \underbrace{D=2 \mid I_1=[0,1024)_{\mathbb{N}_0}, I_2=[0,512)_{\mathbb{N}_0} \mid T=\mathbb{Z}\times\mathbb{Z}>}_{\text{MDA's meta-parameters}}}$$

and defined as

$$\underbrace{[\, M(i,k) \,]_{i\in[0,1024)_{\mathbb{N}_0}, k\in[0,512)_{\mathbb{N}_0}}}_{\text{BUF (Matrix)}}, \underbrace{[\, v(k) \,]_{k\in[0,512)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}} \mapsto \underbrace{[\, M(i,k), v(k) \,]_{i\in[0,1024)_{\mathbb{N}_0}, k\in[0,512)_{\mathbb{N}_0}}}_{\text{MDA}}$$

Here, the BUFs' meta-parameters are as follows: $B = 2$ is the number of BUFs (matrix and vector); $D_1 = 2$ is dimensionality of the matrix and $D_2 = 1$ the vector's dimensionality; $(N_1^1, N_2^1) = (1024, 512)$ is the matrix size and $N_1^2 = 512$ the vector's size; $T_1, T_2 = \mathbb{Z}$ are the scalar types of the matrix and vector. The MDA's meta-parameters are: $D = 2$ is the computed MDA's dimensionality; I_1, I_2 are the MDA's index sets; parameter $T = \mathbb{Z} \times \mathbb{Z}$ is MDA's scalar type (pairs of matrix/vector elements – see Figure 7).

Example 18 (Input View – `Jacobi1D`)**.** The input view of `Jacobi1D` on a 512-sized vector of integers is of type

$$\texttt{IV}^{\overbrace{<B=1 \mid D_1=1 \mid (N_1^1=512) \mid T_1=\mathbb{Z}}^{\text{BUFs' meta-parameters}} \mid \overbrace{D=1 \mid I_1=[0,512-2)_{\mathbb{N}_0} \mid T=\mathbb{Z}\times\mathbb{Z}\times\mathbb{Z}>}^{\text{MDA's meta-parameters}}}$$

and defined as

$$\underbrace{[\, v(i) \,]_{i\in[0,512)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}} \mapsto \underbrace{[\, v(i+0), v(i+1), v(i+2) \,]_{i\in[0,512-2)_{\mathbb{N}_0}}}_{\text{MDA}}$$

We introduce higher-order function `inp_view` which computes important input views conveniently and in a structured manner from user-defined index functions $(\mathfrak{id}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}$ (Definition 30). Here, $B \in \mathbb{N}$ represents the number of BUFs that the computed input view will take as input, and A_b represents the number of accesses to the b-th BUF required for computing an individual MDA element.

In the case of `MatVec` (Figure 7), we use $B := 2$ because `MatVec` has two input BUFs: a matrix M (the first input of `MatVec` and thus identified by $b = 1$) and a vector v (identified by $b = 2$). For the number of accesses, we use for the matrix $A_1 := 1$, as one element is accessed within matrix M to compute an individual MDA element – matrix element $M[i,k]$ for computing MDA element at position (i,k). For the vector, we use $A_2 := 1$, as the single element $v[k]$ is accessed within the vector. The index functions of `MatVec` are: $\mathfrak{id}_{1,1}(i,k) := (i,k)$ (used to access the matrix) and $\mathfrak{id}_{2,1}(i,k) := (k)$ (used for the vector).

In contrast, for `Jacobi1D` (Figure 8), we use $B := 1$ because `Jacobi1D` has vector v as its only input, and we use $A_1 := 3$ because the vector is accessed three times to compute an individual MDA element at arbitrary position i: first access $v[i+0]$, second access $v[i+1]$, and third access $v[i+2]$. The index functions of `Jacobi1D` are: $\mathfrak{id}_{1,1}(i) := (i+0)$, $\mathfrak{id}_{1,2}(i) := (i+1)$, and $\mathfrak{id}_{1,3}(i) := (i+2)$.

More generally, higher-order function `inp_view` uses the index functions $\mathfrak{idx}_{b,a}$ to compute an input view that maps BUFs $\mathfrak{b}_1,\ldots,\mathfrak{b}_B$ to an MDA $\mathfrak{a}$ that contains at position $i_1,\ldots,i_D$ the following element:

$$
\begin{aligned}
&\mathfrak{a}[i_1,\ldots,i_D] := \\
&\underbrace{\Big(\underbrace{\big(\underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,1}(i_1,\ldots,i_D)\,]\in T_1}_{a=1},\ \ldots,\ \underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,A_1}(i_1,\ldots,i_D)\,]\in T_1}_{a=A_1}\big)}_{b=1}\Big.}\\
&\qquad\qquad\vdots\\
&\underbrace{\Big.\big(\underbrace{\mathfrak{b}_B[\,\mathfrak{idx}_{B,1}(i_1,\ldots,i_D)\,]\in T_B}_{a=1},\ \ldots,\ \underbrace{\mathfrak{b}_B[\,\mathfrak{idx}_{B,A_B}(i_1,\ldots,i_D)\,]\in T_B}_{a=A_B}\big)\Big)}_{b=B}
\end{aligned}
$$

The element consists of B-many tuples – one per BUF – and each such tuple contains A_b-many elements – one element per access to the b-th BUF. For `MatVec`, the element is of the form

$$
\begin{aligned}
\mathfrak{a}[i_1,i_2] := \ &\underbrace{\Big(\underbrace{\big(\underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,1}(i_1,i_2):=(i_1,i_2)\,]\in T_1}_{a=1}\big),}_{b=1}\Big.}\\
&\underbrace{\Big.\big(\underbrace{\mathfrak{b}_2[\,\mathfrak{idx}_{2,1}(i_1,i_2):=(i_2)\,]\in T_2}_{a=1}\big)\Big)}_{b=2}
\end{aligned}
$$

and for `Jacobi1D`, the element is

$$
\begin{aligned}
\mathfrak{a}[i_1] := \ &\underbrace{\Big(\underbrace{\big(\underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,1}(i_1):=(i_1+0)\,]\in T_1,}_{a=1}\Big.}\\
&\underbrace{\Big.\underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,2}(i_1):=(i_1+1)\,]\in T_1,}_{a=2}}\\
&\underbrace{\Big.\underbrace{\mathfrak{b}_1[\,\mathfrak{idx}_{1,3}(i_1):=(i_1+2)\,]\in T_1}_{a=3}\big)\Big)}_{b=1}
\end{aligned}
$$

In the following, we introduce higher-order function `inp_view` which computes important input views conveniently and in a uniform, structured manner. Function `inp_view` takes as input a collection of index functions (Definition 30), and it uses these index functions to compute a corresponding input view (Definition 31), as outlined above and described in detail in the following.

Figures 69 and 70 use the examples `MatVec` and `Jacobi1D` to informally illustrate how function `inp_view` uses index functions to compute input views. In the two figures, we use domain-specific identifiers for better clarity: in the case of `MatVec`, we use for its two input BUFs the identifiers M and v instead of $\mathfrak{b}_1$ and $\mathfrak{b}_2$, as well as identifiers i and j instead of $\mathfrak{i}_1$ and $\mathfrak{i}_2$ for index variables; for `Jacobi1D`, we use identifier v instead of $\mathfrak{b}_1$, and i instead of $\mathfrak{i}_1$.

We now formally define higher-order function `inp_view`. For high flexibility and formal correctness, function `inp_view` relies on a type that involves many meta-parameters. The high number of meta-parameters, and the resulting complex type of function `inp_view`, might appear daunting to the user. However, Notation 7 confirms that despite the complex type of function `inp_view`, the function can be conveniently expressed by the user (as also illustrated in Figure 64), because meta-parameters can be automatically deduced from `inp_view`'s input parameters.

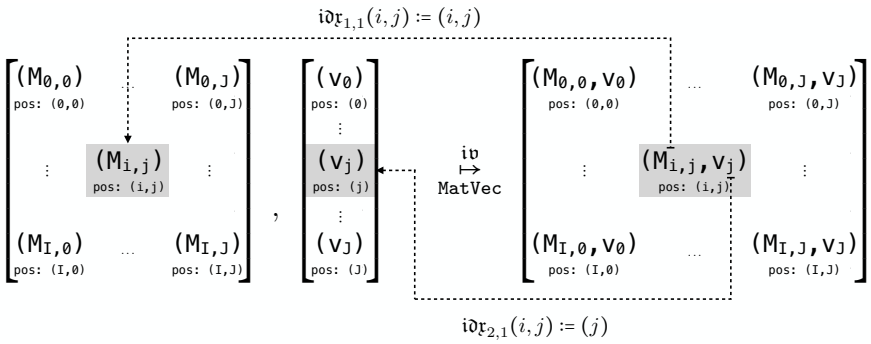


Figure 69: Input view illustrated using the example `MatVec`.

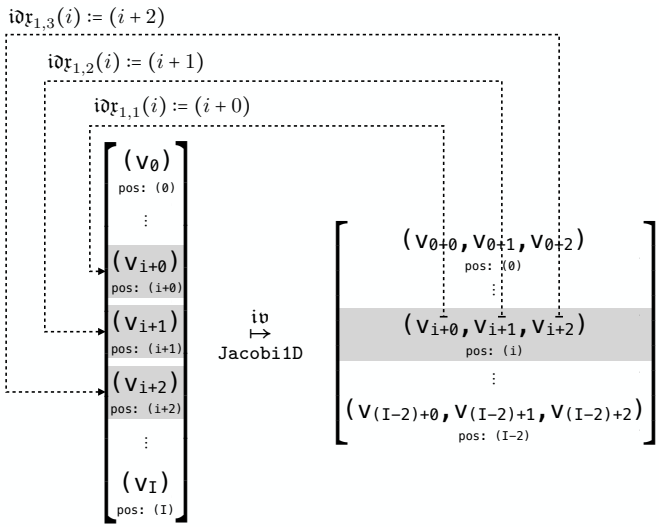


Figure 70: Input view illustrated using the example `Jacobi1D`.

Definition 32 (Higher-Order Function `inp_view`)**.** Function `inp_view` is of type

$$\texttt{inp\_view}^{<} \underbrace{B\in\mathbb{N}}_{\text{Number BUFs}} \mid \underbrace{A_1,\ldots,A_B\in\mathbb{N}}_{\text{Number BUFs' Accesses}} \mid \underbrace{D_1,\ldots,D_B\in\mathbb{N}_0}_{\text{BUFs' Dimensionalities}} \mid \underbrace{D\in\mathbb{N}}_{\text{MDA Dimensionality}} \mid \underbrace{(\Rightarrow_{\text{BUF}}^{\text{MDA}\,b,a}:\texttt{MDA-IDX-SETs}^{D}\to\texttt{BUF-IDX-SETs}^{D_b})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Set Functions (MDA indices to BUF indices)}} >:$$

$$\underbrace{\underbrace{\underbrace{\overset{B}{\underset{b=1}{\times}}}_{\text{Buffer}}\underbrace{\overset{A_b}{\underset{a=1}{\times}}}_{\text{Access}}\underbrace{\texttt{MDA-IDX-to-BUF-IDX}^{<D,D_b\mid\Rightarrow_{\text{BUF}}^{\text{MDA}\,b,a}>}}_{\text{Index Function: }\mathfrak{idx}_{b,a}}}_{\text{Index Functions: }\mathfrak{idx}_{1,1},\ldots,\mathfrak{idx}_{B,A_B}}}_{\rightarrow\ \texttt{IV}^{<B\mid D_1,\ldots,D_B\mid\to\mid T_1,\ldots,T_B\ \in\ \texttt{TYPE}\mid \atop D\mid I_1,\ldots,I_D\ \in\ \texttt{MDA-IDX-SETs}\mid\to><\ N_1^1,\ldots,N_{D_B}^B\mid T>}}$$

BUFs' Meta-Parameters; MDA's Meta-Parameters; Postponed Parameters; Input View: $\mathfrak{iv}$

for $N_d^b := 1 + \max(\ \cup_{a\in[1,A_b]_{\mathbb{N}}} \overset{d}{\Rightarrow}_{\text{BUF}}^{\text{MDA}\,b,a}(I_1,\ldots,I_D)\ )$ and $T := \times_{b=1}^{B}\times_{a=1}^{A_b} T_b$, and it is defined as:

$$\underbrace{(\mathfrak{idx}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Functions}} \mapsto \underbrace{(\underbrace{\mathfrak{b}_1,\ldots,\mathfrak{b}_B}_{\text{BUFs}})\overset{\mathfrak{iv}}{\mapsto}\underbrace{\mathfrak{a}}_{\text{MDA}}}_{\text{Input View}}$$

for

$$\mathfrak{a}[i_1,\ldots,i_D] := (\,\mathfrak{a}_{b,a}[i_1,\ldots,i_D]\,)_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}$$

and

$$\mathfrak{a}_{b,a}[i_1,\ldots,i_D] \;:=\; \mathfrak{b}_b[\,\mathfrak{idx}_{b,a}(i_1,\ldots,i_D)\,]$$

Higher-order function `inp_view` takes as input a collection of index functions that are of types `MDA-IDX-to-BUF-IDX` (Definition 30), and it computes an input view of type `IV` (Definition 31) based on the index functions, as illustrated in Figures 69 and 70.

As concrete meta-parameter values of type `MDA-IDX-to-BUF-IDX` (listed in angle brackets), we use straightforwardly the values of meta-parameters passed to function `inp_view`. Similarly, we use the particular meta-parameter values of function `inp_view` also for type `IV`'s meta-parameters B, $D_1,\ldots,D_B$, and D

To be able using the computed input view on arbitrarily typed input buffers and letting the input view compute MDAs that have arbitrary index sets, we keep `IV`'s meta-parameters $T_1, \ldots, T_B$ (scalar types of the computed view's input buffers) and $I_1, \ldots, I_D$ (index sets of the view's returned MDA) flexible. Being flexible in the BUFs' scalar types and MDA's index sets is important for convenience: for example, in the case of `MatVec`, this flexibility allows using the computed input view generically for matrices and vectors that have arbitrary scalar types (e.g., either `int` or `float`) and sizes (I, J) (matrix) and J (vector), for arbitrary $I, J \in \mathbb{N}$, without needing to re-compute a new input view every time again when BUFs' scalar types and/or sizes change.

We automatically compute the sizes N_d^b of BUFs in `IV`'s meta-parameter list (e.g., in the case of `MatVec`, the size of the input matrix (I, J) and vector size J), according to the formula in Definition 32, based on the flexible MDA's index sets (e.g., sets $[0, I)_{\mathbb{N}_0}$ and $[0, J)_{\mathbb{N}_0}$ for `MatVec`). By computing BUF sizes from MDA index sets (rather than requesting the sizes explicitly from the user), we achieve strong error checking: for example, for `MatVec`, we can ensure – already on the type level – that the number of columns of its input matrix and the size of its input vector match. To compute the BUF sizes, we *postpone* via $\rightarrow$ (defined formally in the Appendix, Definition 24) the sizes in `IV`'s meta-parameter list to later meta-parameter stages; this is because the sizes are defined in early stages and thus have no access to the MDA's index sets which are defined in later stages. Our formula in Definition 32 then works as follows: for each BUF b, its size N_d^b has to be well-defined in each of its dimensions d, for all accesses a, which is checked by using the BUF's index functions on all indices within the MDA index sets $I_1, \ldots, I_D$. Here, in the computation of N_d^b, function $\overset{d}{\Rightarrow}^{b,a}_{\text{BUF}}$ computes the d-th component of the D_b-sized output tuple of $\Rightarrow^{b,a}_{\text{BUF}}$ (the computed component is the index set of BUF b in dimension d for the a-th index function used to access the BUF).

We automatically compute also MDA's scalar type T using the formula presented in Definition 32. The formula computes T as a tuple that consists of the BUFs' scalar types, as each MDA element consist of BUF elements (illustrated in Figures 69 and 70). Postponing T in `IV`'s meta-parameter list is done (analogously as for N_d^b), but is actually not required, because the BUFs' scalar types $T_1, \ldots, T_B$ are already defined in earlier meta-parameter stages than T. However, we will see that postponing T is required later in the Definition 34 of higher-order function `out_view`; therefore, we postpone T also in our definition of `inp_view` to increase consistency between our definitions of `inp_view` (Definition 32) and `out_view` (Definition 34).

Note that function `inp_view` is not capable of computing every kind of input view function (Definition 31). For example, `inp_view` cannot be used for computing MDAs that are required for expressing computations on sparse data formats [70], because such MDAs need dynamically accessing BUFs. This limitation of `inp_view` can be relaxed by generalizing our index functions toward taking additional, dynamic input arguments, which we consider as future work (as outlined in Chapter 8).

Notation 7 (Input Views). Let $\texttt{inp\_view}^{\langle\cdots\rangle}(\ (\ i\mathfrak{dx}_{b,a}\ )_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}\ )$ be a particular instance of higher-order function `inp_view` (meta-parameters omitted via ellipsis for simplicity) for an arbitrary but fixed choice of index functions. Let further $\texttt{ID}_1,\ldots,\texttt{ID}_B \in \Sigma^*$ be arbitrary, user-defined BUF identifiers (e.g., $\texttt{ID}_1$ = `"M"` and $\texttt{ID}_2$ = `"v"` in the case of `MatVec`), for an arbitrary, fixed collection of letters $\Sigma = \{A, B, C, \ldots, a, b, c, \ldots, 1, 2, 3, \ldots\}$.

For better readability, we use the following notation for the 2-dimensional structure of index functions taken as input by function `inp_view`, inspired by Lattner et al. [56]:

$$\texttt{inp\_view}(\ \texttt{ID}_1 : i\mathfrak{dx}_{1,1},\ldots,i\mathfrak{dx}_{1,A_1}\ ,\ \ldots\ ,\ \texttt{ID}_B : i\mathfrak{dx}_{B,1},\ldots,i\mathfrak{dx}_{B,A_B}\ )$$

We refrain from stating `inp_view`'s meta-parameters in our notation, as the parameters can be automatically deduced from the number and types of index functions.

Example 19. Function `inp_view` is used for `MatVec` and `Jacobi1D` (in Notation 7) as follows:

$$\underline{\texttt{MatVec:}}\quad \texttt{inp\_view}(\ \underbrace{\underbrace{\texttt{M:}\ \underbrace{(i,k)\mapsto(i,k)}_{a=1}}_{b=1}\ ,\ \underbrace{\texttt{v:}\ \underbrace{(i,k)\mapsto(k)}_{a=1}}_{b=2}}\ )$$

$$\underline{\texttt{Jacobi1D:}}\quad \texttt{inp\_view}(\ \underbrace{\texttt{v:}\ \underbrace{(i)\mapsto(i+0)}_{a=1}\ ,\ \underbrace{(i)\mapsto(i+1)}_{a=2}\ ,\ \underbrace{(i)\mapsto(i+2)}_{a=3}}_{b=1}\ )$$

.2.3.3 *Output Views*

An *output view* is the counterpart of an input view: in contrast to an input view which maps BUFs to an MDA, an output view maps an MDA to a collection of BUFs. In the following, we define output views, and we introduce higher-order function `out_view` which computes output views in a structured manner (analogously to function `inp_view` for input views).

Figures 71 and 72 illustrate output views informally using the examples *transposed Matrix Multiplication* and *Double Reduction*.

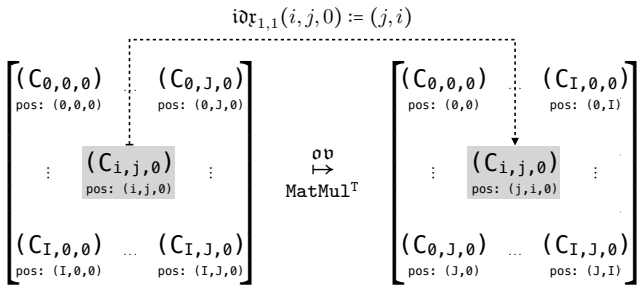


Figure 71: Output view illustrated using the example *transposed Matrix Multiplication*.

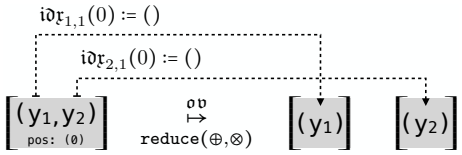


Figure 72: Output view illustrated using the example *Double Reduction*.

In the case of transposed matrix multiplication (Figure 71), the computed output MDA (the computation of matrix multiplication is presented later and not relevant for our following considerations) is stored via an output view as a matrix in a transposed fashion, using index function $(i, j, 0) \mapsto (j, i)$. Here, the MDA's third dimension (accessed via index 0) represents the so-called reduction dimension of matrix multiplication, and it contains only one element after the computation, as all elements in this dimension are combined via addition.

For double reduction (Figures 72), we combine the elements within the vector twice – once using operator $\oplus$ (e.g., $\oplus$ = + addition) and once using operator $\otimes$ (e.g, $\otimes$ = $*$ multiplication). The final outcome of double reduction is a singleton MDA containing a pair of two elements that represent the combined vector elements (e.g., the elements' sum and product). We store this MDA via an output view as two individual scalar values, using index functions $(0) \mapsto ()$[10] for both pair elements.

Definition 33 (Output View)**.** An *output view* from an MDA of arbitrary but fixed type $T[I_1, \ldots, I_D]$ to B-many BUFs, $B \in \mathbb{N}$, of arbitrary but fixed types $T_b^{N_1^b \times \ldots \times N_{D_b}^b}$, $b \in [1, B]_{\mathbb{N}}$, is any function $\mathfrak{ov}$ of type:

$$\mathfrak{ov} : \underbrace{T[\, I_1, \ldots, I_D \,]}_{\text{MDA}} \to_p \underbrace{\bigtimes_{b=1}^{B} T_b^{N_1^b \times \ldots \times N_{D_b}^b}}_{\text{BUFs}}$$

We denote the type of $\mathfrak{ov}$ as:

$$\mathfrak{ov}^{< D \mid I_1,\ldots,I_D \mid T \mid B \mid (D_b)_{b\in[1,B]_{\mathbb{N}}} \mid (N_1^b,\ldots,N_{D_b}^b)_{b\in[1,B]_{\mathbb{N}}} \mid (T_b)_{b\in[1,B]_{\mathbb{N}}} >}$$

(MDA's Meta-Parameters; BUFs' Meta-Parameters)

[10] The empty braces denote accessing a scalar value (formal details provided in the Appendix, Section .3.7).

Example 20 (Output View – `MatVec`)**.** The output view of `MatVec` computing a 1024-sized vector (size is chosen arbitrarily), of integers $\mathbb{Z}$, is of type

$$\texttt{OV}^{<\overbrace{D=2\,|\,I_1=[0,1024)_{\mathbb{N}_0},I_2=\{0\}\,|\,T=\mathbb{Z}}^{\text{MDA's meta-parameters}}\,|\,\overbrace{B=1\,|\,D_1=1\,|\,(N^1_1=1024)\,|\,T_1=\mathbb{Z}}^{\text{BUFs' meta-parameters}}>}$$

and defined as

$$\underbrace{[\,w(i)\,]_{i\in[0,1024)_{\mathbb{N}_0},k\in\{0\}}}_{\text{MDA}} \mapsto \underbrace{[\,w(i)\,]_{i\in[0,1024)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}}$$

Example 21 (Output View – `Jacobi1D`)**.** The output view of `Jacobi1D` computing a $(512-2)$-sized vector of integers is of type

$$\texttt{OV}^{<\overbrace{D=1\,|\,I_1=[0,512-2)_{\mathbb{N}_0}\,|\,T=\mathbb{Z}}^{\text{MDA's meta-parameters}}\,|\,\overbrace{B=1\,|\,D_1=1\,|\,(N^1_1=(512-2))\,|\,T_1=\mathbb{Z}}^{\text{BUFs' meta-parameters}}>}$$

and defined as

$$\underbrace{[\,w(i)\,]_{i\in[0,512-2)_{\mathbb{N}_0},k\in\{0\}}}_{\text{MDA}} \mapsto \underbrace{[\,w(i)\,]_{i\in[0,512-2)_{\mathbb{N}_0}}}_{\text{BUF (Vector)}}$$

We define higher-order function `out_view` formally as follows.

Definition 34 (Higher-Order Function `out_view`)**.** Function `out_view` is of type

$$\texttt{out\_view}^{<\underbrace{B\in\mathbb{N}}_{\texttt{Number BUFs}}\;|\;\underbrace{A_1,\dots,A_B\in\mathbb{N}}_{\texttt{Number BUFs' Accesses}}\;|\;\underbrace{D_1,\dots,D_B\in\mathbb{N}_0}_{\texttt{BUFs' Dimensionalities}}\;|\;\underbrace{D\in\mathbb{N}}_{\texttt{MDA Dimensionality}}\;|\;\underbrace{(\Rightarrow^{\texttt{MDA}}_{\texttt{BUF}}{}^{b,a}:\texttt{MDA-IDX-SETs}^{D}\to\texttt{BUF-IDX-SETs}^{D_b})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\texttt{Index Set Functions (MDA indices to BUF indices)}}>}:$$

$$\underbrace{\underbrace{\underset{b=1}{\overset{B}{\times}}\;\underset{a=1}{\overset{A_b}{\times}}}_{\text{Buffer Access}}\;\underbrace{\texttt{MDA-IDX-to-BUF-IDX}^{<D,D_b\,|\,\Rightarrow^{\texttt{MDA}}_{\texttt{BUF}}{}^{b,a}>}}_{\text{Index Function: }\mathfrak{idx}_{b,a}}}_{\text{Index Functions: }\mathfrak{idx}_{1,1},\dots,\mathfrak{idx}_{B,A_B}}$$

$$\to\;\underbrace{\texttt{OV}^{<\overbrace{D\,|\,I_1,\dots,I_D\in\texttt{MDA-IDX-SETs}\,|\to|}^{\text{MDA's Meta-Parameters}}\;\underbrace{B\,|\,D_1,\dots,D_B\,|\to|\,T_1,\dots,T_B\in\texttt{TYPE}}_{\text{BUFs' Meta-Parameters}}><\underbrace{N^1_1,\dots,N^B_{D_B}\,|\,T}_{\text{Postponed Parameters}}>}}_{\text{Output View: }\mathfrak{ov}}$$

which differs from `inp_view`'s type only in mapping index functions to `OV` (Definition 33), rather than `IV` (Definition 31). Function `out_view` is defined as:

$$\underbrace{(\mathfrak{id}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Functions}} \mapsto \underbrace{\underbrace{\mathfrak{a}}_{\text{MDA}} \overset{\mathfrak{ov}}{\mapsto} (\underbrace{\mathfrak{b}_1,\ldots,\mathfrak{b}_B}_{\text{BUFs}})}_{\text{Output View}}$$

for

$$\mathfrak{b}_b[\ \mathfrak{id}_{b,a}(i_1,\ldots,i_D)\ ] \ := \ \mathfrak{a}_{b,a}[i_1,\ldots,i_D]$$

and

$$(\mathfrak{a}_{b,a}[i_1,\ldots,i_D])_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}} := \mathfrak{a}[i_1,\ldots,i_D]$$

i.e., $\mathfrak{a}_{b,a}[i_1,\ldots,i_D]$ is the element at point $i_1,\ldots,i_D$ within MDA $\mathfrak{a}$ that belongs to the a-th access of the b-th BUF. We set $\mathfrak{b}_b[j_1,\ldots,j_{D_b}] := \perp$ (symbol $\perp$ denotes the undefined value) for all BUF indices $(j_1,\ldots,j_{D_b}) \in [0,N_1^b)_{\mathbb{N}_0} \times \ldots \times [0,N_D^b)_{\mathbb{N}_0} \setminus \bigcup_{a\in[1,A_b]_{\mathbb{N}}} \overset{d}{\Rightarrow}^{\texttt{MDA}\,b,a}_{\texttt{BUF}}(I_1,\ldots,I_D)$ which are not in the function range of the index functions.

Note that the computed output view $\mathfrak{ov}$ is partial (indicated by $\rightarrow_p$ in Definition 33), because for non-injective index functions, it must hold $\mathfrak{id}_{b,a}(i_1,\ldots,i_D) = \mathfrak{id}_{b,a'}(i'_1,\ldots,i'_D) \Rightarrow \mathfrak{a}_{b,a}[i_1,\ldots,i_D] = \mathfrak{a}_{b,a'}[i'_1,\ldots,i'_D]$ which may not be satisfied for each potential input MDA of the computed view.

Notation 8 (Output Views)**.** Analogously to Notation 7, we denote `out_view` for a particular choice of index functions as:

$$\texttt{out\_view}(\ \texttt{ID}_1 : \mathfrak{id}_{1,1},\ldots,\mathfrak{id}_{1,A_1} \ ,\ldots,\ \texttt{ID}_B : \mathfrak{id}_{B,1},\ldots,\mathfrak{id}_{B,A_B}\ )$$

Example 22. Function `out_view` is used for `MatVec` and `Jacobi1D` (in Notation 8) as follows:

$$\underline{\texttt{MatVec:}} \quad \texttt{out\_view}(\ \underbrace{\texttt{w:}\ \underbrace{(i,k)\mapsto(i)}_{\text{a=1}}}_{\text{b=1}}\ )$$

$$\underline{\texttt{Jacobi1D:}} \quad \texttt{out\_view}(\ \underbrace{\texttt{w:}\ \underbrace{(i)\mapsto(i)}_{\text{a=1}}}_{\text{b=1}}\ )$$

.2.3.4 *Relation between View Functions*

We use view functions to transform data from their domain-specific representation (represented in our formalism as BUFs, Definition 29) to our internal, MDA-based representation (via input views) and back (via output views), as also illustrated in Figure 63. In our implementation presented later, we aim to access data uniformly in the form of MDAs, thereby being independent of domain-specific data representations. However, we aim to store the data physically in the domain-specific format, as such format is usually the more efficient representation. For example, we aim to store the input data of `MatVec` in the domain-specific matrix and vector format, rather than as an MDA, because the input MDA of `MatVec` contains many redundancies – each vector element once per row of the input matrix (as illustrated in Figure 69).

The following lemma proves that functions `inp_view` and `out_view` are invertible and that they are each others inverses. Consequently, the lemma shows how we can arbitrarily switch between the domain-specific and our MDA-based representation, and consequently also that we can implicitly identify MDAs with the domain-specific data representation. For example, for computing `MatVec`, we will specify the computations via pattern `md_hom` which operates on MDAs (see Figure 63), but we use the view functions in our implementation to implicitly forward the MDA accesses to accesses to the physically stored BUF representation.

Lemma 4. Let

$$\texttt{inp\_view}(\ \texttt{ID}_1 : i\mathfrak{d}\mathfrak{x}_{1,1}, \ldots, i\mathfrak{d}\mathfrak{x}_{1,A_1}\ , \ldots ,\ \texttt{ID}_B : i\mathfrak{d}\mathfrak{x}_{B,1}, \ldots, i\mathfrak{d}\mathfrak{x}_{B,A_B}\ )$$

and

$$\texttt{out\_view}(\ \texttt{ID}_1 : i\mathfrak{d}\mathfrak{x}_{1,1}, \ldots, i\mathfrak{d}\mathfrak{x}_{1,A_1}\ , \ldots ,\ \texttt{ID}_B : i\mathfrak{d}\mathfrak{x}_{B,1}, \ldots, i\mathfrak{d}\mathfrak{x}_{B,A_B}\ )$$

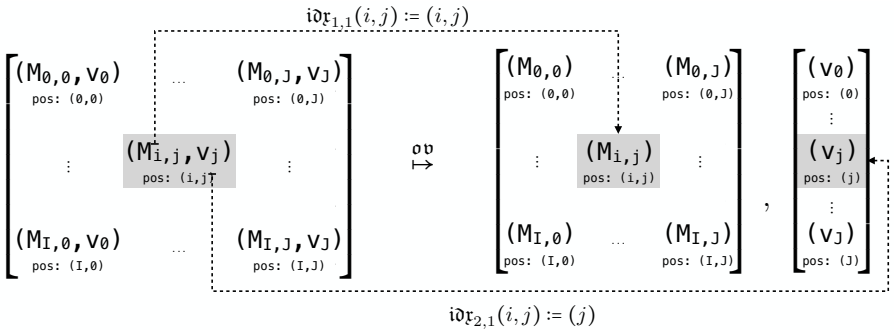
be two arbitrary instances of functions `inp_view` and `out_view` (in Notations 7 and 8), both using the same index functions $i\mathfrak{d}\mathfrak{x}_{1,1}, \ldots, i\mathfrak{d}\mathfrak{x}_{B,A_B}$.

It holds (index functions omitted via ellipsis for brevity):

$$\begin{aligned} \texttt{inp\_view}(\ldots) \circ \texttt{out\_view}(\ldots) &= \\ \texttt{out\_view}(\ldots) \circ \texttt{inp\_view}(\ldots) &= \texttt{id} \end{aligned}$$

Proof. Follows immediately from Definitions 32 and 34. □

The following figure illustrates the lemma using as example the inverse of `MatVec`'s input view (shown in Figure 69):



.2.4 *Generic High-Level Expression*

Figure 73 shows an expression in our high-level representation – consisting of higher-order functions `inp_view`, `md_hom`, and `out_view` (Figure 63) – that is generic in an arbitrary but fixed choice of index functions, scalar function, and combine operators. We express data-parallel computations using a particular instance of this generic expression in Figure 73.

Note that meta-parameters of higher-order function `inp_view`, `out_view`, and `md_hom` are omitted in Figure 73, because all parameters can be automatically deduced from the particular numbers and types of their inputs (index functions in the case of `inp_view` and `out_view`, and scalar function and combine operators for `md_hom`).

$$\texttt{out\_view}\langle T_1^{OB},\ldots,T_{B^{OB}}^{OB}\rangle(\ OB_1 : \ \mathfrak{id}_{1,1}^{OUT}\ ,\ldots,\ \mathfrak{id}_{1,A_1^{OB}}^{OUT}\ ,$$
$$\vdots$$
$$OB_{B^{OB}} : \mathfrak{id}_{B^{OB},1}^{OUT},\ldots,\mathfrak{id}_{B^{OB},A_{B^{OB}}^{OB}}^{OUT}\ )\ \circ$$

$$\texttt{md\_hom}\langle N_1,\ldots,N_D\rangle(\ f,\ (\circledast_1,\ldots,\circledast_D)\ )\ \circ$$

$$\texttt{inp\_view}\langle T_1^{IB},\ldots,T_{B^{IB}}^{IB}\rangle(\ IB_1 : \ \mathfrak{id}_{1,1}^{INP}\ ,\ldots,\ \mathfrak{id}_{1,A_1^{IB}}^{INP}\ ,$$
$$\vdots$$
$$IB_{B^{IB}} : \mathfrak{id}_{B^{IB},1}^{INP},\ldots,\mathfrak{id}_{B^{IB},A_{B^{IB}}^{IB}}^{INP}\ )$$

Figure 73: Generic high-level expression for data-parallel computations.

The concrete instance of `md_hom(...)` (i.e., the MDH function returned by `md_hom` for the particular input scalar function and combine operators) has as meta-parameter the MDH function's index sets (see Definition 27) for high flexibility. We use as index sets straightforwardly the input size $(N_1, \ldots, N_D) \in \mathbb{N}^D$ (which abbreviates $([0, N_1)_{\mathbb{N}_0}, \ldots, [0, N_D)_{\mathbb{N}_0})$ – see Notation 6)[11]. Instances of `inp_view(...)` and `out_view(...)` (i.e., the input and output view returned by `inp_view` and `out_view` for concrete index functions) have as meta-parameters the MDA's index sets and the scalar types of BUFs. We explicitly state only the meta-parameter for the BUFs' scalar types in our generic high-level expression (Figure 73), and we avoid explicitly stating the MDA's index sets for simplicity and to avoid redundancies, because the sets can be taken from the `md_hom`'s meta-parameter list.

Note that for better readability of our high-level expressions, we list meta-parameters before parentheses, i.e., instead of writing `inp_view(...)`$^{<\cdots>}$, `out_view(...)`$^{<\cdots>}$, and `md_hom(...)`$^{<\cdots>}$ for the particular instances of higher-order functions, where meta-parameters are listed at the end, we write `inp_view<...>(...)`, `out_view<...>(...)`, and `md_hom<...>(...)`.

.2.5 *Examples*

Figure 74 shows how our high-level representation is used for expressing different kinds of popular data-parallel computations. For brevity, we state only the index functions, scalar function, and combine operators of the higher-order functions; the expression in Figure 73 is then obtained by straightforwardly inserting these building blocks into the higher-order functions.

[11] Our formalism allows dynamic shapes, by using symbol $*$ instead of a particular natural number for N_i (formal details provided in the Appendix, Definition 23), which we aim to discuss thoroughly in future work.

md_hom	f	$\otimes_1$	$\otimes_2$	$\otimes_3$	$\otimes_4$
Dot	*	+			
MatVec	*	++	+		
MatMul	*	++	++	+	
MatMul^T	*	++	++	+	
bMatMul	*	++	++	++	+

Views	inp_view		out_view
	A	B	C
Dot	(k) ↦ (k)	(k) ↦ (k)	(k) ↦ ()
MatVec	(i,k) ↦ (i,k)	(i,k) ↦ (k)	(i,k) ↦ (i)
MatMul	(i,j,k) ↦ (i,k)	(i,j,k) ↦ (k,j)	(i,j,k) ↦ (i,j)
MatMul^T	(i,j,k) ↦ (k,i)	(i,j,k) ↦ (j,k)	(i,j,k) ↦ (j,i)
bMatMul	(b,i,j,k) ↦ (b,i,k)	(b,i,j,k) ↦ (b,k,j)	(b,i,j,k) ↦ (b,i,j)

1) Linear Algebra Routines

md_hom	f	$\otimes_1$	$\otimes_2$	$\otimes_3$	$\otimes_4$	$\otimes_5$	$\otimes_6$	$\otimes_7$	$\otimes_8$	$\otimes_9$	$\otimes_{10}$
Conv2D	*	++	++	+	+						
MCC	*	++	++	++	++	+	+	+			
MCC_Capsule	*	++	++	++	++	+	+	+	++	++	+

Views	inp_view		out_view
	I	F	O
Conv2D	(p,q,r,s) ↦ (p+r,q+s)	(p,q,r,s) ↦ (r,s)	(p,q,r,s) ↦ (p,q)
MCC	(n,p,...) ↦ (n,p+r,q+s,c)	(n,p,...) ↦ (k,r,s,c)	(n,p,...) ↦ (n,p,q,k)
MCC_Capsule	(n,p,...) ↦ (n,p+r,q+s,c,mi,mk)	(n,p,...) ↦ (k,r,s,c,mk,mj)	(n,p,...) ↦ (n,p,q,k,mi,mj)

2) Convolution Stencils

md_hom	f	$\otimes_1$	...	$\otimes_6$	$\otimes_7$
CCSD(T)	*	++	...	++	+

Views	inp_view		out_view
	A	B	C
I1	(a,...,g) ↦ (g,d,a,b)	(a,...,g) ↦ (e,f,g,c)	(a,...,g) ↦ (a,...,f)
I2	(a,...,g) ↦ (g,d,a,c)	(a,...,g) ↦ (e,f,g,b)	(a,...,g) ↦ (a,...,f)

3) Quantum Chemistry

md_hom	f	$\otimes_1$	$\otimes_2$	$\otimes_3$
Jacobi1D	J_{1D}	++		
Jacobi2D	J_{2D}	++	++	
Jacobi3D	J_{3D}	++	++	++

Views	inp_view	out_view
	I	O
Jacobi1D	(i1) ↦ (i1+0) , (i1) ↦ (i1+1) , ...	(i1) ↦ (i1)
Jacobi2D	(i1,i2) ↦ (i1+0,i2+1) , ...	(i1,i2) ↦ (i1,i2)
Jacobi3D	(i1,i2,i3) ↦ (i1+0,i2+1,i3+1) , ...	(i1,i2,i3) ↦ (i1,i2,i3)

4) Jacobi Stencils

md_hom	f	$\otimes_1$	$\otimes_2$
PRL	wght	++	$\max_{PRL}$

Views	inp_view		out_view
	N	E	M
PRL	(i,j) ↦ (i)	(i,j) ↦ (j)	(i,j) ↦ (i)

5) Probabilistic Record Linkage

md_hom	f	$\otimes_1$	$\otimes_2$
Histo	f_{Histo}	+	++
GenHisto	f	$\oplus$	++

Views	inp_view		out_view
	Elems	Bins	Out
Histo	(e,b) ↦ (e)	(e,b) ↦ (b)	(e,b) ↦ (b)
GenHisto	(e,b) ↦ (e)	(e,b) ↦ (b)	(e,b) ↦ (b)

6) Histogram

md_hom	f	$\otimes_1$
map(f)	f	++
reduce($\oplus$)	id	$\oplus$
reduce($\oplus$,$\otimes$)	(x) ↦ (x,x)	($\oplus$,$\otimes$)

Views	inp_view	out_view	
	I	O_1	O_2
map(f)	(i) ↦ (i)	(i) ↦ (i)	
reduce($\oplus$)	(i) ↦ (i)	(i) ↦ ()	
reduce($\oplus$,$\otimes$)	(i) ↦ (i)	(i) ↦ ()	(i) ↦ ()

7) Map/Reduce Patterns

md_hom	f	$\otimes_1$	$\otimes_2$
scan($\oplus$)	id	++$_{\text{prefix-sum}}$($\oplus$)	
MBBS	id	++$_{\text{prefix-sum}}$(+)	+

Views	inp_view	out_view
	A	Out
scan($\oplus$)	(i) ↦ (i)	(i) ↦ (i)
MBBS	(i,j) ↦ (i,j)	(i,j) ↦ (i)

8) Prefix Sum Computations

Figure 74: Data-parallel computations expressed in our high-level representation.

SUBFIGURE 1 We show how our high-level representation is used for expressing linear algebra routines: 1) `Dot` (*Dot Product*); 2) `MatVec` (*Matrix-Vector Multiplication*); 3) `MatMul` (*Matrix Multiplication*); 4) $\texttt{MatMul}^\intercal$ (*Transposed Matrix Multiplication*) which computes matrix multiplication on transposed input and output matrices; 5) `bMatMul` (*batched Matrix Multiplication*) where multiple matrix multiplications are computed using matrices of the same sizes.

We can observe from the subfigure that our high-level expressions for the routines naturally evolve from each other. For example, the `md_hom` instance for `MatVec` differs from the `md_hom` instance for `Dot` by only containing a further concatenation dimension $+\!\!+$ for its `i` dimension. We consider this close relation between the high-level expressions of `MatVec` and `Dot` in our approach as natural and favorable, as `MatVec` can be considered as computing multiple times `Dot` – one computation of `Dot` for each value of `MatVec's` `i` dimension. Similarly, the `md_hom` instance for `MatMul` is very similar to the expression of `MatVec`, by containing the further concatenation dimension `j` for `MatMul`'s `j` dimension. The same applies to `bMatMul`: its `md_hom` instance is the expression of `MatMul` augmented with one further concatenation dimension.

Regarding $\texttt{MatMul}^\intercal$, the basic computation part of $\texttt{MatMul}^\intercal$ and `MatMul` are the same, which is exactly reflected in our formalisms: both $\texttt{MatMul}^\intercal$ and `MatMul` are expressed using exactly the same `md_hom` instances. The differences between $\texttt{MatMul}^\intercal$ and `MatMul` lies only in the data accesses – transposed accesses in the case of $\texttt{MatMul}^\intercal$ and non-transposed accesses in the case of `MatMul`. Data accesses are expressed in our formalism, in a structured way, via view functions (as discussed in Section .2.3): for example, for $\texttt{MatMul}^\intercal$, we use for its first input matrix A the index function $(i, j, k) \mapsto (k, i)$ for transposed access, instead of using index function $(i, j, k) \mapsto (i, k)$ as for `MatMul`'s non-transposed accesses.

Note that all `md_hom` instances in the subfigure are well defined according to Lemma 3.

SUBFIGURE 2 We show how convolution-style stencil computations are expressed in our high-level representation: 1) `Conv2D` expresses a standard convolution that uses a 2D sliding window [270]; 2) `MCC` expresses a so-called *Multi-Channel Convolution* [121] – a generalization of `Conv2D` that is heavily used in the area of deep learning; 3) `MCC_Capsule` is a recent generalization of `MCC` [126] which attracted high attention due to its relevance for advanced deep learning neural networks [84].

While our `md_hom` instances for convolutions are quite similar to those of linear algebra routines (they all use multiplication $*$ as scalar function, and a mix of concatenations $+\!\!+$ and point-wise additions $+$ as combine operators), the index functions used for the view functions of convolutions are notably different from those used for linear algebra routines: the index functions of convolutions contain arithmetic expressions (e.g., `p+r` and `q+s`), thereby access neighboring elements in their input – a typical access pattern in stencil computations

that requires special optimizations [124]. Moreover, convolution-style computations are often high-dimensional (e.g., 10 dimensions in the case of `MCC_Capsule`), whereas linear algebra routines usually rely on less dimensions. Our experiments in Section 4.5 confirm that respecting the data access patterns and the high dimensionality of convolutions in the optimization process (as in our approach, which we discuss later) often achieves significantly higher performance than using optimizations chosen toward linear algebra routines, as in vendor libraries provided by NVIDIA and Intel for convolutions [165].

SUBFIGURE 3 We show how quantum chemistry computation *Coupled Cluster* (`CCSD(T)`) [91] is expressed in our high-level representation. The computation of `CCSD(T)` notably differs from those of linear algebra routines and convolution-style stencils, by accessing its high-dimensional input data in sophisticated transposed fashions: for example, the view function of `CCSD(T)`'s *instance one* (denoted as `I1` in the subfigure) uses indices `a` and `b` to access the last two dimensions of its A input tensor (rather than the first two dimensions of the tensor, as would be the case for non-transposed accesses).

For brevity, the subfigure presents only two `CCSD(T)` instances – in our experiments in Section 4.5, we present experimental results for nine different real-world `CCSD(T)` instances.

SUBFIGURES 4-6 The subfigures present computations whose scalar functions and combine operators are different from those used in Subfigures 1-3 (which are in Subfigures 1-3 straightforward multiplications $*$, concatenation $+\!\!+$, and point-wise additions $+$ only). For example, Jacobi stencils (Subfigure 4) use as scalar function the Jacobi-specific computation $\mathtt{J_{nD}}$ [220], and *Probabilistic Record Linkage (PRL)* [221], which is heavily used in data mining to identify duplicate entries in a data base, uses a PRL-specific both scalar function `wght` and combine operator $\mathtt{max_{PRL}}$ (point-wise combination via the PRL-specific binary operator $\mathtt{max_{PRL}}$) [116]. Histograms, in their generalized version [71] (denoted as `GenHisto` in Subfigure 6), use an arbitrary, user-defined scalar function `f` and a user-defined associative and commutative combine operator $\oplus$; the standard histogram variant `Histo` is then a particular instance of `GenHist`, for $\oplus = +$ (point-wise addition) and $\mathtt{f} = \mathtt{f_{Histo}}$, where $\mathtt{f_{Histo}}(e, b) = 1$ iff $e = b$ and $\mathtt{f_{Histo}}(e, b) = 0$ otherwise. Histogram's are often analyzed regarding their runtime complexity [71]; we provide such a discussion for our MDH-based Histogram implementation in the Appendix, Section .3.8, for the interested reader.

SUBFIGURE 7 We show how typical `map` and `reduce` patterns [251] are implemented in our high-level representation. Examples `map(f)` and `reduce(⊕)` (discussed in Examples 15 and 16) are simple and thus straightforwardly expressed in our representation. In contrast, example `reduce(⊕,⊗)` is more complex and shows how `reduce(⊕)` is extended to combine the input vector simultaneously twice – once combining vector elements via operator ⊕ and once using operator ⊗. The outcome of `reduce(⊕,⊗)` are two scalars – one representing the result of combination via ⊕ and the other of combination via ⊗ – which we map via the output view to output elements $\mathtt{O}_1$ (result of ⊕) and $\mathtt{O}_2$ (result of ⊗), correspondingly; this is also illustrated in Figure 72.

SUBFIGURE 8 We present *prefix-sum computations* [312] which differ from the computations in Subfigures 1-7 in terms of their combine operators: the operator used for expressing computations in Subfigure 8 is different from concatenation (Example 13) and point-wise combinations (Example 14). Computation `scan(⊕)` uses as combine operator $+\!\!+_{\texttt{prefix-sum}}(\oplus)$ which computes prefix-sum [299] (formally defined in the Appendix, Section .3.9) according to binary operator ⊕, and `MBBS` (Maximum Bottom Box Sum) [86] uses a particular instance of prefix-sum for ⊕ = + (addition).

.3 ADDENDUM SECTION 4.2

.3.1 *Design Decisions: Combine Operators*

We list some design decisions for combine operators (Definition 26).

Note 1.

- We deliberately restrict index set function $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}$ to compute the index set in the particular dimension d only, and not of all D Dimensions (i.e., the function's output is in $\texttt{MDA-IDX-SETs}$ and not $\texttt{MDA-IDX-SETs}^D$), because this enables applying combine operator $\circledast$ iteratively:

  $$(\ldots((a_1 \circledast^{<(P,Q)>} a_2) \circledast^{<(P\cup Q,R)>} a_3) \circledast^{<(P\cup Q\cup R,\ldots)>} \ldots$$

  for MDAs $a_1, a_2, a_3, \ldots$ that have index sets $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P), \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(Q), \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(R), \ldots$ in dimension d. This is because the index set of the output MDA changes only in dimension d, to the new index set $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P \cup Q)$, $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P \cup Q) \cup R)$, $\ldots$, so that the output MDA can be used as input for a new application of $\circledast$.

- It is a design decision that a combine operator's index set function $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}$ takes as input the MDA index set P or Q in the particular dimension d only, rather than the all sets $(I_1, \ldots, I_D)$. Our approach can be easily extended to index set functions $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}} : \texttt{MDA-IDX-SETs}^D \to \texttt{MDA-IDX-SETs}$ that take the entire MDA's index sets as input. However, we avoid this additional complexity, because we are currently not aware of any real-world application that would benefit from such extension.

- For better convenience, we could potentially define the meta-type of combine operators (Definition 26) such that meta-parameter $(I_1, \ldots, I_{d-1}, I_{d+1}, \ldots, I_D)$ is separated from parameter (P, Q) in a distinct, earlier stage (Definition 21). This would allow automatically deducing $(I_1, \ldots, I_{d-1}, I_{d+1}, \ldots, I_D)$ from the input MDAs' types, whereas for meta-parameter (P, Q), automatic deduction is usually not possible: function $\Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}$ has to be either invertible for automatically deducing P and Q from the input MDAs or invariant under different values of P and Q. Consequently, separating parameter $(I_1, \ldots, I_{d-1}, I_{d+1}, \ldots, I_D)$ in a distinct, earlier stage would allow avoiding explicitly stating this parameter, by deducing it from the input MDAs' type, and only explicitly stating parameter (P, Q), e.g., $\circledast_2^{<(P,Q)>}(a, b)$ instead of $\circledast_2^{<(I_1)|(P,Q)>}(a, b)$ for $a \in T[I_1, \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(P)]$ and $b \in T[I_1, \Rightarrow^{\texttt{MDA}}_{\texttt{MDA}}(Q)]$. We avoid separating $(I_1, \ldots, I_{d-1}, I_{d+1}, \ldots, I_D)$ and (P, Q) in this work, as we focus on concatenation (Example 13), prefix-sum (Example 25), and point-wise combination (Example 14) only, which have invertible or P/Q-invariant index set functions, respectively. Consequently, for the practice-relevant combine operators considered in this work, we can deduce all meta-parameters automatically.

.3.2 *Generalized Notion of MDHs*

The MDH Definition 27 can be generalized to have an arbitrary algebraic structure as input.

Definition 35 (Multi-Dimensional Homomorphism)**.** Let

$$\mathcal{A}^{\downarrow} := (\; \mathsf{T}^{\mathsf{INP}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(*), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(*)] \; , \; ({}^{\downarrow}\circledast_d)_{d \in [1,D]_{\mathbb{N}}} \;)$$

and

$$\mathcal{A}^{\uparrow} := (\; \mathsf{T}^{\mathsf{OUT}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(*), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(*)] \; , \; ({}^{\uparrow}\circledast_d)_{d \in [1,D]_{\mathbb{N}}} \;)$$

be two algebraic structures, where

$$({}^{\downarrow}\circledast_d \in \mathsf{CO}^{<\overset{d}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow} \mid \mathsf{T}^{\mathsf{INP}} \mid D \mid d>})_{d \in [1,D]_{\mathbb{N}}}$$

and

$$({}^{\uparrow}\circledast_d \in \mathsf{CO}^{<\overset{d}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow} \mid \mathsf{T}^{\mathsf{OUT}} \mid D \mid d>})_{d \in [1,D]_{\mathbb{N}}}$$

are tuples of combine operators, for $D \in \mathbb{N}$, $\mathsf{T}^{\mathsf{INP}}, \mathsf{T}^{\mathsf{OUT}} \in \mathtt{TYPE}$, $\overset{d}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}, \overset{d}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow} : \mathtt{MDA\text{-}IDX\text{-}SETs} \to \mathtt{MDA\text{-}IDX\text{-}SETs}$, and the two structures' carrier sets

$$\mathsf{T}^{\mathsf{INP}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(*), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(*)]$$

and

$$\mathsf{T}^{\mathsf{OUT}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(*), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(*)]$$

denote the set of MDAs that are in the function domain of combine operators (the star symbol is used for indicating the function range of index functions).

A *Multi-Dimensional Homomorphism (MDH)* from the algebraic structure $\mathcal{A}^{\downarrow}$ to the structure $\mathcal{A}^{\uparrow}$ is any function

$$h^{<I_1,\ldots,I_D \in \mathtt{MDA\text{-}IDX\text{-}SETs}>} :$$
$$\mathsf{T}^{\mathsf{INP}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(I_1), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\downarrow}(I_D)] \to \mathsf{T}^{\mathsf{OUT}}[\overset{1}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(I_1), \ldots, \overset{D}{\Rightarrow}_{\mathsf{MDA}}^{\mathsf{MDA}\uparrow}(I_D)]$$

that satisfies the *homomorphic property*:

$$h(\; a_1 \, {}^{\downarrow}\circledast_d \, a_2 \;) \; = \; h(a_1) \, {}^{\uparrow}\circledast_d \, h(a_2)$$

The MDH Definition 27 is a special case of our generalized MDH Definition 35, for ${}^{\downarrow}\circledast_d = +\!\!+^{<\mathsf{T}^{\mathsf{INP}} \mid D \mid d>}$ (Example 13).

Higher-order function `md_hom` (originally introduced in Definition 28) is defined for the generalized MDH Definition 35 as follows.

Definition 36 (Higher-Order Function `md_hom`)**.** The higher-order function `md_hom` is of type

$$\texttt{md\_hom}^{<\texttt{T}^{\texttt{INP}},\texttt{T}^{\texttt{OUT}}\in\texttt{TYPE}\mid \texttt{D}\in\mathbb{N}\mid (\overset{\texttt{d}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}:\texttt{MDA-IDX-SETs}\to\texttt{MDA-IDX-SETs})_{\texttt{d}\in[1,\texttt{D}]_{\mathbb{N}}},\ (\overset{\texttt{d}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}:\texttt{MDA-IDX-SETs}\to\texttt{MDA-IDX-SETs})_{\texttt{d}\in[1,\texttt{D}]_{\mathbb{N}}}>}:$$

$$\underbrace{(\ \texttt{CO}^{<\overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}\mid \texttt{T}^{\texttt{OUT}}\mid \texttt{D}\mid 1>}\times\ldots\times\texttt{CO}^{<\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}\mid \texttt{T}^{\texttt{OUT}}\mid \texttt{D}\mid \texttt{D}>}\ )}_{{}^{\downarrow}\otimes_1,\ldots,{}^{\downarrow}\otimes_{\texttt{D}}}\ \times$$

$$\underbrace{\texttt{SF}^{<\texttt{T}^{\texttt{INP}},\texttt{T}^{\texttt{OUT}}>}}_{\texttt{f}}\ \times$$

$$\underbrace{(\ \texttt{CO}^{<\overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}\mid \texttt{T}^{\texttt{OUT}}\mid \texttt{D}\mid 1>}\times\ldots\times\texttt{CO}^{<\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}\mid \texttt{T}^{\texttt{OUT}}\mid \texttt{D}\mid \texttt{D}>}\ )}_{{}^{\uparrow}\otimes_1,\ldots,{}^{\uparrow}\otimes_{\texttt{D}}}$$

$$\to_{\texttt{p}}\ \underbrace{\texttt{MDH}^{<\texttt{T}^{\texttt{INP}},\texttt{T}^{\texttt{OUT}}\mid \texttt{D}\mid (\overset{\texttt{d}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow})_{\texttt{d}\in[1,\texttt{D}]_{\mathbb{N}}},(\overset{\texttt{d}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow})_{\texttt{d}\in[1,\texttt{D}]_{\mathbb{N}}}>}}_{\texttt{md\_hom}(\ (\otimes_1^{\downarrow},\ldots,\otimes_{\texttt{D}}^{\downarrow})\ ,\ \texttt{f}\ ,\ (\otimes_1^{\uparrow},\ldots,\otimes_{\texttt{D}}^{\uparrow})\ )}$$

The function takes as input a scalar function f and two tuples of D-many combine operators $({}^{\downarrow}\otimes_1,\ldots,{}^{\downarrow}\otimes_{\texttt{D}})$ and $({}^{\uparrow}\otimes_1,\ldots,{}^{\uparrow}\otimes_{\texttt{D}})$, and it yields a function $\texttt{md\_hom}(\ ({}^{\downarrow}\otimes_1,\ldots,{}^{\downarrow}\otimes_{\texttt{D}})\ ,\ \texttt{f}\ ,\ ({}^{\uparrow}\otimes_1,\ldots,{}^{\uparrow}\otimes_{\texttt{D}})\ )$ which is defined as:

$$\begin{aligned}
&{}^{\downarrow}a && \in \texttt{T}^{\texttt{INP}}[\ \overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}(\ I_1\ ),\ldots,\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}(\ I_{\texttt{D}}\ )\ ]\\
=:&\\
&\underset{i_1\in I_1}{{}^{\downarrow}\otimes_1}\ldots\underset{i_{\texttt{D}}\in I_{\texttt{D}}}{{}^{\downarrow}\otimes_{\texttt{D}}}\ {}^{\downarrow}a^{<i_1,\ldots,i_d>} && \in \texttt{T}^{\texttt{INP}}[\ \overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}(\ \{i_1\}\ ),\ldots,\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\downarrow}(\ \{i_{\texttt{D}}\}\ )\ ]\\
\mapsto&\\
&\underset{i_1\in I_1}{+_1}\ldots\underset{i_{\texttt{D}}\in I_{\texttt{D}}}{+_{\texttt{D}}}\ {}^{\downarrow}a_f^{<i_1,\ldots,i_d>} && \in \texttt{T}^{\texttt{INP}}[\ \{i_1\},\ldots,\{i_{\texttt{D}}\}\ ]\\
\overset{\vec{f}}{\mapsto}&\\
&\underset{i_1\in I_1}{+_1}\ldots\underset{i_{\texttt{D}}\in I_{\texttt{D}}}{+_{\texttt{D}}}\ {}^{\uparrow}a_f^{<i_1,\ldots,i_d>} && \in \texttt{T}^{\texttt{OUT}}[\ \{i_1\},\ldots,\{i_{\texttt{D}}\}\ ]\\
\mapsto&\\
&\underset{i_1\in I_1}{{}^{\uparrow}\otimes_1}\ldots\underset{i_{\texttt{D}}\in I_{\texttt{D}}}{{}^{\uparrow}\otimes_{\texttt{D}}}\ {}^{\uparrow}a^{<i_1,\ldots,i_d>} && \in \texttt{T}^{\texttt{OUT}}[\ \overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}(\ \{i_1\}\ ),\ldots,\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}(\ \{i_{\texttt{D}}\}\ )\ ]\\
=:&\\
&{}^{\uparrow}a && \in \texttt{T}^{\texttt{OUT}}[\ \overset{1}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}(\ I_1\ ),\ldots,\overset{\texttt{D}}{\Rightarrow}_{\texttt{MDA}}^{\texttt{MDA}\uparrow}(\ I_{\texttt{D}}\ )\ ]
\end{aligned}$$

Here, $\vec{f}$ denotes the adaption of function f to operate on MDAs comprising a single scalar value only – the function is of type

$$\vec{f}^{<i_1,\ldots,i_{\texttt{D}}\in\mathbb{N}>}:\texttt{T}^{\texttt{INP}}[\ \{i_1\},\ldots,\{i_{\texttt{D}}\}\ ]\to\texttt{T}^{\texttt{OUT}}[\ \{i_1\},\ldots,\{i_{\texttt{D}}\}\ ]$$

and defined as

$$\vec{f}(x)[\ i_1,\ldots,i_{\texttt{D}}\ ]:=f(x[\ i_1,\ldots,i_{\texttt{D}}\ ])$$

We refer to the first application of $\mapsto$ as *de-composition*, to the application of $\overset{\vec{f}}{\mapsto}$ as *scalar function application*, and to the second application of $\mapsto$ as *re-composition*.
For $\texttt{md\_hom}(\ (\circledast_1^{\downarrow},\ldots,\circledast_d^{\downarrow}),\ f,\ (\circledast_1^{\uparrow},\ldots,\circledast_d^{\uparrow})\ )$, we require by definition the homomorphic property (Definition 35), i.e., for each $d \in [1,D]_{\mathbb{N}}$, it must hold:

$$\begin{aligned}\texttt{md\_hom}(\ ({}^{\downarrow}\circledast_1,\ldots,{}^{\downarrow}\circledast_D)\ ,\ f\ ,\ ({}^{\uparrow}\circledast_1,\ldots,{}^{\uparrow}\circledast_D)\ )(\ a_1\ {}^{\downarrow}\circledast_d\ a_2\ )\ = \\ \texttt{md\_hom}(\ ({}^{\downarrow}\circledast_1,\ldots,{}^{\downarrow}\circledast_D)\ ,\ f\ ,\ ({}^{\uparrow}\circledast_1,\ldots,{}^{\uparrow}\circledast_D)\ )(\ a_1\ ) \\ {}^{\uparrow}\circledast_d \\ \texttt{md\_hom}(\ ({}^{\downarrow}\circledast_1,\ldots,{}^{\downarrow}\circledast_D)\ ,\ f\ ,\ ({}^{\uparrow}\circledast_1,\ldots,{}^{\uparrow}\circledast_D)\ )(\ a_2\ )\end{aligned}$$

.3.3 *Simple MDH Examples*

FUNCTION MAPPING Function $\texttt{map}^{<T^{INP},T^{OUT}\,|\,D\,|\,(I_1,\ldots,I_D)>}(f)$ maps a function $f : T^{INP} \to T^{OUT}$ to each element of an MDA that has scalar type $T^{INP} \in \texttt{TYPE}$, dimensionality $D \in \mathbb{N}$, and index sets $I := (I_1,\ldots,I_D) \in \texttt{MDA-IDX-SETs}^D$. The function is of type

$$\texttt{map}^{<T^{INP},T^{OUT}\in\texttt{TYPE}\,|\,D\in\mathbb{N}\,|\,(I_1,\ldots,I_D)\in\texttt{MDA-IDX-SETs}^D>}:$$
$$\underbrace{T^{INP}\to T^{OUT}}_{f}\ \to\ \underbrace{T^{INP}[\ I_1\ ,\ldots,\ I_D\ ]\ \to\ T^{OUT}[\ I_1\ ,\ldots,\ I_D\ ]}_{\texttt{map}^{<T^{INP},T^{OUT}\,|\,D\,|\,(I_1,\ldots,I_D)>}(f)}$$

and it is computed as:

$$a \overset{\texttt{map}^{<T^{INP},T^{OUT}\,|\,D\,|\,(I_1,\ldots,I_D)>}(f)}{\mapsto} \mathrel{+\!\!+}_1{}_{i_1\in I_1}\ \ldots\ \mathrel{+\!\!+}_D{}_{i_D\in I_D}\ \vec{f}_{\texttt{map}}(\ a|_{\{i_1\}\times\ldots\times\{i_D\}}\ )$$

Here, $\mathrel{+\!\!+}_d := \mathrel{+\!\!+}^{<T^{OUT}\,|\,D\,|\,d>}$ denotes concatenation (Example 13) in dimension d, MDA $a|_{\{i_1\}\times\ldots\times\{i_D\}}$ is the restriction of a to the single scalar element accessed via indices $(i_1,\ldots,i_D)$, and $\vec{f}_{\texttt{map}}$ denotes the adaption of function f to operate on MDAs comprising a single value only: it is of type

$$\vec{f}_{\texttt{map}}^{<i_1,\ldots,i_D\in\mathbb{N}>} : T^{INP}[\ \{i_1\},\ldots,\{i_D\}\ ] \to T^{OUT}[\ \{i_1\},\ldots,\{i_D\}\ ]$$

and defined as

$$\vec{f}_{\texttt{map}}(x)[\ i_1,\ldots,i_D\ ] := f(x[\ i_1,\ldots,i_D\ ])$$

It is easy to see that $\texttt{map}^{<T^{INP},T^{OUT}\,|\,D>}(f)$ is an MDH of type $\texttt{MDH}^{<T^{INP},T^{OUT}\,|\,D\,|\,id,\ldots,id>}$ whose combine operators are concatenation $\mathrel{+\!\!+}_d \in \texttt{CO}^{<id\,|\,T^{OUT}\,|\,D\,|\,d>}$ in all dimensions $d \in [1,D]_{\mathbb{N}}$.

We have chosen `map` function's order of stages – $T^{INP}, T^{OUT} \in \texttt{TYPE}$ (stage 1), $D \in \mathbb{N}$ (stage 2), and $(I_1,\ldots,I_D)\texttt{MDA-IDX-SETs}^D$ (stage 3) – according to the recommendations in Haskell Wiki [211], i.e., earlier stages (such as the scalar types T^{INP}, T^{OUT}) are expected to change less frequently than later stages (e.g., the MDAs' index sets $I_1,\ldots,I_D$).

REDUCTION Function $\texttt{red}^{<T\,|\,D\,|\,(I_1,\ldots,I_D)>}(\oplus)$ combines all elements within an MDA that has scalar type $T \in \texttt{TYPE}$, dimensionality $D \in \mathbb{N}$, and index sets $I := (I_1,\ldots,I_D) \in \texttt{MDA-IDX-SETs}^D$, using an associative and commutative binary function $\oplus : T \times T \to T$. The function is of type

$$\texttt{red}^{<T\in\texttt{TYPE}\,|\,D\in\mathbb{N}\,|\,(I_1,\ldots,I_D)\in\texttt{MDA-IDX-SETs}^D>}:$$
$$\underbrace{T \times T \to T}_{\oplus} \;\to\; \underbrace{T[\,I_1\,,\ldots,\,I_D\,] \;\to\; T[\,1,\ldots,1\,]}_{\texttt{red}^{<T\,|\,D\,|\,(I_1,\ldots,I_D)>}(\oplus)}$$

and it is computed as:

$$a \overset{\texttt{red}^{<T\,|\,D\,|\,(I_1,\ldots,I_D)>}(\oplus)}{\mapsto} \underset{i_1\in I_1}{\overrightarrow{\bullet}_1}(\oplus) \;\ldots\; \underset{i_D\in I_D}{\overrightarrow{\bullet}_D}(\oplus)\; \vec{f}_{\texttt{red}}(\, a|_{\{i_1\}\times\ldots\times\{i_D\}}\,)$$

Here, $\overrightarrow{\bullet}_d(\oplus) := \overrightarrow{\bullet}^{<T\,|\,D\,|\,d>}(\oplus)$ denotes point-wise combination (Example 14) in dimension d, MDA $a|_{\{i_1\}\times\ldots\times\{i_D\}}$ is defined as above, and $\vec{f}_{\texttt{red}}$ is the function of type

$$\vec{f}_{\texttt{red}}^{<i_1,\ldots,i_D\in\mathbb{N}>} : T^{\texttt{INP}}[\,\{i_1\},\ldots,\{i_D\}\,] \to T^{\texttt{OUT}}[\,\{0\},\ldots,\{0\}\,]$$

that is defined as

$$\vec{f}_{\texttt{red}}(x)[\,0,\ldots,0\,] := x[\,i_1,\ldots,i_D\,]$$

It is easy to see that $\texttt{red}^{<T\,|\,D>}(\oplus)$ is an MDH of type $\texttt{MDH}^{<T,T\,|\,D\,|\,0_f,\ldots,0_f>}$ whose combine operators are point-wise addition $\overrightarrow{\bullet}_d(\oplus) \in \texttt{CO}^{<id\,|\,T\,|\,D\,|\,d>}$ in all dimensions $d \in [1,D]_{\mathbb{N}}$. The same as for function `map`, function `red`'s order of meta-parameter stages are chosen according to [211].

.3.4 *Design Decisions: `md_hom`*

We list some design decisions for function `md_hom` (Definition 28).

Note 2. For some MDHs (such as Mandelbrot), the scalar function f (Definition 28) is dependent on the position in the input MDA, i.e., it takes as arguments, in addition to $a[i_1,\ldots,i_D]$, also the indices $i_1,\ldots,i_D$. Such MDHs can be easily expressed via `md_hom` after a straightforward type adjustment: type $SF^{<T^{INP},T^{OUT}>}$ has to be defined as the set of functions $f : T^{INP} \times \texttt{MDA-IDX-SETs}^D \to T^{OUT}$ (rather than of functions $f : T^{INP} \to T^{OUT}$, as in Definition 28).

Since we do not aim at forcing scalar functions to always take MDA indices as input arguments – for expressing most computations, this is not required (Figure 74) and only causes additional complexity – we assume in the following two different definitions of pattern `md_hom`: one variant exactly as in Definition 28, and one variant with the adjusted type for scalar functions and that passes automatically indices $i_1,\ldots,i_D$ to f. The two variants can be easily differentiated, via an additional, boolean meta-parameter `USE_MDA_INDICES`: first variant iff `USE_MDA_INDICES = false` and second variant iff `USE_MDA_INDICES = true`.

For simplicity, we focus in this thesis on the first variant (as in Definition 28), because it is the more common variant, and because all insights presented in this work apply to both variants.

.3.5 *Proof md_hom Lemma 3*

Proof. Let $a_1 \in T[I_1^{a_1}, \ldots, I_D^{a_2}]$ and $a_2 \in T[I_1^{a_2}, \ldots, I_D^{a_2}]$ be two arbitrary MDAs that are concatenable in dimension d.

According to Definition 28, we have to show that

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_1 +\!\!+_d a_2\,) =$$

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_1\,) \circledast_d \texttt{md\_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_2\,)$$

For this, we first show for arbitrary $k \in [1, D)_{\mathbb{N}}$ that

$$\ldots \underset{i_k \in I_k}{\circledast_k} \underset{i_{k+1} \in I_{k+1}}{\circledast_{k+1}} \ldots \chi|_{\ldots,\{i_k\},\{i_{k+1}\},\ldots} = \ldots \underset{i_{k+1} \in I_{k+1}}{\circledast_{k+1}} \underset{i_k \in I_k}{\circledast_k} \ldots \chi|_{\ldots,\{i_k\},\{i_{k+1}\},\ldots}$$

from which follows

$$\underset{i_1 \in I_1}{\circledast_1} \ldots \underset{i_D \in I_D}{\circledast_D} \chi|_{\{i_1\},\ldots,\{i_D\}} = \underset{i_{\sigma(1)} \in I_{\sigma(1)}}{\circledast_{\sigma(1)}} \cdots \underset{i_{\sigma(D)} \in I_{\sigma(D)}}{\circledast_{\sigma(D)}} \chi|_{\{i_1\},\ldots,\{i_D\}}$$

for any permutation $\sigma : \{1, \ldots, D\} \twoheadrightarrow \{1, \ldots, D\}$. Afterward, in our assumption above, we can assume w.l.o.g. that $d = 1$.

> Case 1: $[\circledast_k = \circledast_{k+1}]$ Follows immediately from the commutativity of $+\!\!+$ or $\vec{\bullet}(\oplus)$ for commutative $\oplus$, respectively. ✓
>
> Case 2: $[\circledast_k \neq \circledast_{k+1}]$ Trivial, as it is either $\circledast_k = +\!\!+$ or $\circledast_{k+1} = +\!\!+$, and
>
> $$(\underset{i_d \in I_d}{+\!\!+} \chi|_{\ldots,\{i_d\},\ldots})[i_1, \ldots, i_D] = (\chi|_{\ldots,\{i_d\},\ldots})[i_1, \ldots, i_D]$$
>
> according to the definition of MDA concatenation $+\!\!+$ (Example 13). ✓

Let now be $d = 1$ (see assumption above), it holds:

$$\begin{aligned}
&\texttt{md_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_1 +\!\!+_1 a_2\,) \\
&\quad = \underset{i_1 \in I_1}{\circledast_1} \ldots \underset{i_D \in I_D}{\circledast_D} \vec{f}(\, (a_1 +\!\!+_1 a_2)|_{\{i_1\} \times \ldots \times \{i_D\}}\,) \\
&\quad = \underset{i_1 \in I_1^{a_1}}{\circledast_1} \ldots \underset{i_D \in I_D}{\circledast_D} \vec{f}(\, a_1|_{\{i_1\} \times \ldots \times \{i_D\}}\,) \\
&\qquad\quad \circledast_1 \underset{i_1 \in I_1^{a_2}}{\circledast_1} \ldots \underset{i_D \in I_D}{\circledast_D} \vec{f}(\, a_2|_{\{i_1\} \times \ldots \times \{i_D\}}\,) \\
&\quad = \texttt{md_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_1\,) \\
&\qquad\quad \circledast_1 \texttt{md_hom}(\, f\,,\, (\circledast_1, \ldots, \circledast_D)\,)(\, a_2\,) \quad ✓
\end{aligned}$$

□

.3.6 *Examples of Index Functions*

We present examples of index functions (Definition 30).

Example 23 (Matrix-Vector Multiplication)**.** The index functions we use for expressing Matrix-Vector Multiplication (`MatVec`) are:

- Input Matrix:

$$\mathfrak{idx}(i,k) := (i,k) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=2,D_b=2\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}, I_2^{MDA}) := [0, \max(I_1^{MDA})]_{\mathbb{N}_0}, [0, \max(I_2^{MDA})]_{\mathbb{N}_0}$$

- Input Vector:

$$\mathfrak{idx}(i,k) := (k) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=2,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}, I_2^{MDA}) := [0, \max(I_2^{MDA})]_{\mathbb{N}_0}$$

- Output Vector:

$$\mathfrak{idx}(i,k) := (i) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=2,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}, I_2^{MDA}) := [0, \max(I_1^{MDA})]_{\mathbb{N}_0}$$

Example 24 (Jacobi 1D)**.** The index functions we use for expressing Jacobi 1D (`Jacobi1D`) are:

- Input Buffer, 1. Access:

$$\mathfrak{idx}(i) := (i+0) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=1,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}) := [0, \max(I_1^{MDA}) + 0]_{\mathbb{N}_0}$$

- Input Buffer, 2. Access:

$$\mathfrak{idx}(i) := (i+1) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=1,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}) := [0, \max(I_1^{MDA}) + 1]_{\mathbb{N}_0}$$

- Input Buffer, 3. Access:

$$\mathfrak{idx}(i) := (i+2) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=1,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}) := [0, \max(I_1^{MDA}) + 2]_{\mathbb{N}_0}$$

- Output Buffer:

$$\mathfrak{idx}(i) := (i) \in \texttt{MDA-IDX-to-BUF-IDX}^{<D=1,D_b=1\,|\,\Rightarrow^{MDA}_{BUF}>}$$

$$\text{for } \Rightarrow^{MDA}_{BUF}(I_1^{MDA}) := [0, \max(I_1^{MDA})]_{\mathbb{N}_0}$$

.3.7 *Representation of Scalar Values*

Scalar values can be considered as 0-dimensional BUFs (Definition 29). Consequently, in Definition 29, the cartesian product $[0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0}$ is empty for $D = 0$, and thus results in the neutral element of the cartesian product. As any singleton set can be considered as neutral element of cartesian product (up to bijection), we define the set $\{\epsilon\}$ containing the dedicated symbol epsilon only, as the uniquely determined neutral element of cartesian product (inspired by the notation of the *empty word*).

We often refrain from explicitly stating symbol ϵ, e.g., by writing b instead of $b[\epsilon]$ for accessing a BUF, or $(i_1, \ldots, i_D) \to ()$ instead of $(i_1, \ldots, i_D) \to (\epsilon)$ for index functions.

Note that alternatively, scalar values can be considered as any multi-dimensional BUF containing a single element only. For example, a scalar value s can be represented as 1-dimensional BUF $b_{1D}[0] := s$, or a 2-dimensional BUF $b_{2D}[0, 0] := s$, or a 3-dimensional BUF $b_{3D}[0, 0, 0] := s$, etc. However, this results in an ambiguous representation of scalar values, which we aim to avoid by considering scalars as 0-dimensional BUFs, as described above.

.3.8 *Runtime Complexity of Histograms*

Our implementation of Histograms (Subfigure 5 in Figure 74) has a *work complexity* of $\mathcal{O}(E * B)$, where E is the number of elements to check and B the number of bins, i.e., our MDH Histogram implementation is not work efficient. However, our Histograms' *step complexity* [269] is $\mathcal{O}(\log(E))$: step complexity is often used for parallel algorithms and assumes an infinite number of cores, i.e., we can ignore in our implementation of Histogram the concatenation dimension B (which has a step complexity of $\mathcal{O}(1)$) and take into account its reduction dimension B only, which has a step complexity of $\log(B)$ (parallel reduction [269]). In contrast, related approaches [71] are often work efficient, by having a work complexity of $\mathcal{O}(B)$; however, their high work efficiency is at the cost of their step complexity which is also $\mathcal{O}(B)$, rather than $\mathcal{O}(\log(B))$ as for our implementation in Subfigure 5, thereby being asymptotically less efficient for parallel machines consisting of many cores. Our future work will show that the work-efficient Histogram implementation introduced in Henriksen et al. [71] can also be expressed in our approach, by using for scalar function f an optimized micro kernel for Histogram computation, similarly as done in the related work.

.3.9 *Combine Operator of Prefix-Sum Computations*

We define *prefix-sum* which is the combine operator of compute pattern `scan` and example `MBBS` in Section .2.5.

Example 25 (Prefix-Sum)**.** We define *prefix-sum*, according to a binary function $\oplus : T \times T \to T$ (e.g. addition), as function $\circledast_{\texttt{prefix-sum}}$ of type

$$\circledast_{\texttt{prefix-sum}}^{<T\in\texttt{TYPE}\,|\,D\in\mathbb{N}\,|\,d\in[1,D]_{\mathbb{N}}\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D)\in\texttt{MDA-IDX-SETs}^{D-1},\ (P,Q)\in\texttt{MDA-IDX-SETs}\,\dot{\times}\,\texttt{MDA-IDX-SETs}>} :$$

$$\underbrace{T\times T\to T}_{\oplus} \;\to\; T[I_1,\ldots,\underbrace{id(P)}_{\uparrow d},\ldots,I_D]\times T[I_1,\ldots,\underbrace{id(Q)}_{\uparrow d},\ldots,I_D] \;\to$$

$$\underbrace{T[I_1,\ldots,\underbrace{id(P\uplus Q)}_{\uparrow d},\ldots,I_D]}_{\texttt{prefix-sum (according to }\oplus\texttt{)}}$$

where $id : \texttt{MDA-IDX-SETs} \to \texttt{MDA-IDX-SETs}$ is the identity function on MDA index sets. The function is computed as (w.l.o.g., we assume $max(P) < max(Q)$ for commutativity):

$$\circledast_{\texttt{prefix-sum}}^{<T\,|\,D\,|\,d\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D),(P,Q)>}(\oplus)(a_1,a_2)[i_1,\ldots,i_d,\ldots,i_D]$$

$$:= \begin{cases} a_1[i_1,\ldots,i_d,\ldots,i_D] & , i_d \in P \\ a_1[i_1,\ldots,max(P),\ldots,i_D] \oplus & \\ a_2[i_1,\ldots,i_d,\ldots,i_D] & , i_d \in Q \end{cases}$$

Function $\circledast_{\texttt{prefix-sum}}^{<T\,|\,D\,|\,d>}(\oplus)$ (meaning: $\circledast_{\texttt{prefix-sum}}$ is partially applied to ordinary function parameter $\oplus$; formal details provided in the Appendix, Definition 22) is a combine operator of type $\texttt{CO}^{<id\,|\,T\,|\,D\,|\,d>}$ for any binary operator $\oplus : T \times T \to T$.

.4 FULL VERSION: SECTION 4.3

We introduce our low-level representation for expressing data-parallel computations. In contrast to our high-level representation, our low-level representation explicitly expresses the de-composition and re-composition of computations (informally illustrated in Figure 9). Moreover, our low-level representation is designed such that it can be straightforwardly transformed to executable program code, because it explicitly captures and expresses the optimizations for the memory and core hierarch of the target architecture.

In the following, after briefly discussing an introductory example in Section .4.1, we introduce in Section .4.2 our formal representation of computer systems, to which we refer to as *Abstract System Model (ASM)*. Based on this model, we define *low-level MDAs*, *low-level BUFs*, and *low-level combine operators* in Section .4.3, which are basic building blocks of our low-level representation.

Note that all details and concepts discussed in this section are not exposed to the end users of our system and therefore transparent for them: expressions in our low-level representation are generated fully automatically for the user, from expressions in our high-level representation (Figure 10), according to the methodologies presented later in Section .6 and auto-tuning [61].

.4.1 *Introductory Example*

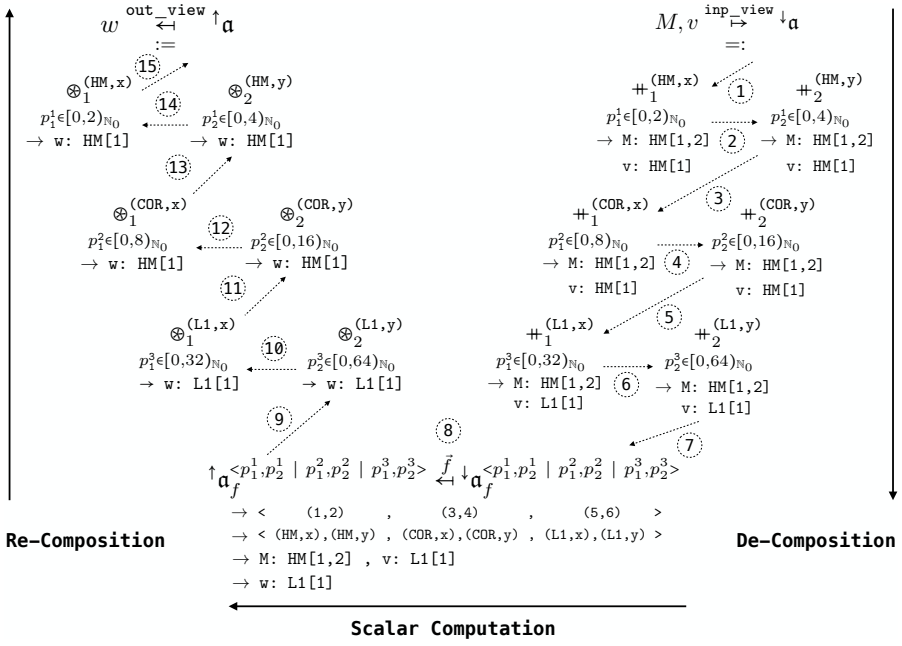


Figure 75: Low-level expression for straightforwardly computing Matrix-Vector Multiplication (`MatVec`) on a simple, artificial architecture with two memory layers (`HM` and `L1`) and one core layer (`COR`). Dotted lines indicate data flow.

Figure 75 illustrates our low-level representation by showing how `MatVec` (Matrix-Vector Multiplication) is expressed in our representation. In our example, we use an input matrix $M \in \mathsf{T}^{512\times4096}$ of size 512×4096 (size chosen arbitrarily) that has an arbitrary but fixed scalar type $\mathsf{T} \in$ `TYPE`; the input vector $v \in \mathsf{T}^{4096}$ is of size 4096, correspondingly.

For better illustration, we consider for this introductory example a straightforward, artificial target architecture that has only two memory layers – *Host Memory (`HM`)* and *Cache Memory (`L1`)* – and one *Core Layer* (`COR`) only; our examples presented and discussed later in this section target real-world architectures (e.g., CUDA-capable NVIDIA GPUs). The particular values of tuning parameters (discussed in detail later in this section), such as the number of threads and the order of combine operators, are chosen by hand for this example and as straightforward for simplicity.

Our low-level representation works in three phases: 1) *de-composition* (steps 1-7, in the right part of Figure 75), 2) *scalar* (step 8, bottom part of the figure), 3) *re-composition* (steps 9-15, left part). Steps are arranged from right to left, inspired by the application order of function composition.

1. DE-COMPOSITION PHASE: The de-composition phase (steps 1-7 in Figure 75) partitions input MDA $^{\downarrow}a$ (in the top right of Figure 75) to the structure $^{\downarrow}a_f^{<\cdots>}$ (bottom right) to which we refer to as *low-level MDA* and define formally in the next subsection. The low-level MDA represents a partitioning of MDA $^{\downarrow}a$ (a.k.a *hierarchical, multi-dimensional tiling* in programming), where each particular choice of indices $p_1^1 \in [0,2)_{\mathbb{N}_0}$, $p_2^1 \in [0,4)_{\mathbb{N}_0}$, $p_1^2 \in [0,8)_{\mathbb{N}_0}$, $p_2^2 \in [0,16)_{\mathbb{N}_0}$, $p_1^3 \in [0,32)_{\mathbb{N}_0}$, $p_2^3 \in [0,64)_{\mathbb{N}_0}$ refers to an MDA that represents an individual part of MDA $^{\downarrow}a$ (a.k.a. *tile* in programming – informally illustrated in Figure 65). The partitions are arranged on multiple layers (indicated by the p's superscripts) and in multiple dimensions (indicated by subscripts) – as illustrated in Figure 76 – according to the memory/core layers of the target architecture and dimensions of the MDH computation: we partition for each of the target architecture's three layers (`HM`, `L1`, `COR`) and in each of the two dimensions of the MDH (dimensions 1 and 2, as we use example `MatVec` in Figure 75, which represents a two-dimensional MDH computation). Consequently, our partitioning approach allows efficiently exploiting each particular layer of the target architecture (both memory and core layers), and also optimizing for both dimensions of the target computation (in the case of `MatVec`, the i-dimension and also the k-dimension – see Figure 7), allowing fine-grained optimizations.

We compute the partitionings of MDAs by applying the concatenation operator (Example 13) inversely[12] (indicated by using =: instead of := in the top right part of Figure 75). For example, we partition in Figure 75 MDA $^{\downarrow}a$ first via the inverse of $+\!\!+_1^{(\mathtt{HM},x)}$ in dimension 1

[12] It is easy to see that operator *concatenation* (Example 13) is invertible for any particular choice of meta-parameters (formally proved in the Appendix, Section .5.2).

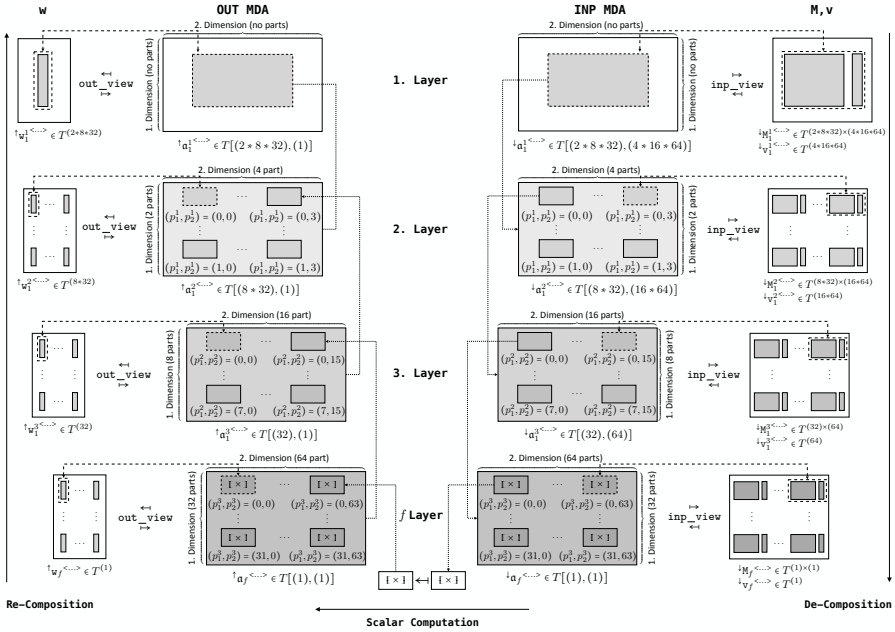


Figure 76: Illustration of multi-layered, multi-dimensional MDA partitioning using the example MDA from Figure 75. In this example, we use three layers and two dimensions, according to Figure 75.

(indicated by the subscript 1 of $+\!\!+_1^{(\texttt{HM},\texttt{x})}$; the superscript `(HM,x)` is explained later) into 2 parts, as p_1^1 iterates over interval $[0,2)_{\mathbb{N}_0} = \{0,1\}$ which consists of two elements (0 and 1) – the interval is chosen arbitrarily for this example. Afterward, each of the obtained parts is further partitioned, in the second dimension, via $+\!\!+_2^{(\texttt{HM},\texttt{y})}$ into 4 parts (p_2^1 iterates over $[0,4)_{\mathbb{N}_0} = \{0,1,2,3\}$ which consists of four elements). The $(2 * 4)$-many `HM` parts are then each further partitioned in both dimensions for the `COR` layer into $(8 * 16)$ parts, and each individual `COR` part is again partitioned for the `L1` layer into $(32 * 64)$ parts, resulting in $(2 * 8 * 32) * (4 * 16 * 64) = 512 * 4096$ parts in total.

We always use a *full partitioning* in our low-level expressions[13], i.e., each particular choice of indices p_1^1, p_2^1, p_1^2, p_2^2, p_1^3, p_2^3 points to an MDA that contains a single element only (in Figure 76, the individual elements are denoted via symbol $\times$, in the bottom part of the figure). By relying on a full partitioning, we can apply scalar function f to the fully partitioned MDAs later in the scalar phase (described in the next paragraph). This is because function f is defined on scalar values (Definition 28) to make defining scalar functions more convenient for the user (as discussed in Section .2.2).

The superscript of combine operators, e.g., `(COR,x)` of operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$, is a so-called *operator tag* (formal definition given in the next subsection). A tag indicates to our code generator whether its combine operator is assigned to a memory layer (and thus computed sequentially in our generated code) or to a core layer (and thus computed in parallel). For example, tag `(COR,x)` indicates that

[13] Our future work (outlined in Chapter 8) aims to additionally allow coarser-grained partitioning schemas, e.g., to target domain-specific hardware extensions (such as *NVIDIA Tensor Cores* [148] which compute 4×4 matrices immediately in hardware, rather than 1×1 matrices as obtained in the case of a full partitioning).

parts processed by operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$ should be computed by cores `COR`, and thus in parallel; the dimension tag `x` indicates that `COR` layer's `x` dimension should be used for computing the operator (we use dimension `x` for our example architecture as an analogous concept to CUDA's thread/block dimensions `x,y,z` for GPU architectures [38]), as we also discuss in the next subsection. In contrast, tag `(HM,x)` refers to a memory layer (host memory `HM`) and thus, operator $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ is computed sequentially. Since the current state-of-practice programming approaches (OpenMP, CUDA, OpenCL, ...) have no explicit notion of memory tiles (e.g., by offering the potential variables `tileIdx.x/tileIdx.y/tileIdx.z`, as analogous concepts to CUDA variables `threadIdx.x/threadIdx.y/threadIdx.z`), the dimensions tag `x` in `(HM,x)` is currently ignored by our code generator, because `HM` refers to a memory layer.

Note that the number of parts (2 parts on layer 1 in dimension 1; 4 parts on layer 1 in dimension 2; ...), the combine operators' tags, and our partition order (e.g. first partitioning in MDA's dimension 1 and afterward in dimension 2) are chosen arbitrarily for this example. These choices are critical for performance and should be optimized[14] for a particular target architecture and characteristics of the input and output data (size, memory layouts, etc.), as we discuss in detail later in this section.

2. SCALAR PHASE: In the scalar phase (step 8 in Figure 75), we apply MDH's scalar function f to the individual MDA elements

$$\downarrow_{a_f}^{<p_1^1,p_2^1 \mid p_1^2,p_2^2 \mid p_1^3,p_2^3>}$$

for each particular choice of indices p_1^1, p_2^1, p_1^2, p_2^2, p_1^3, p_2^3, which results in

$$\uparrow_{a_f}^{<p_1^1,p_2^1 \mid p_1^2,p_2^3 \mid p_1^3,p_2^3>}$$

In the figure, $\vec{f}$ (introduced in Definition 28) is the slight adaption of function f that operates on a singleton MDA, rather than a scalar.

Annotation → `< (1,2) , ... >` indicates the application order of applying scalar function (in this example, first iterating over p_1^1, then over p_2^1, etc), and we use annotation → `< (HM,x) , ... >` to indicate how the scalar computation is assigned to the target architecture (this is described in detail later in this section). Annotations → `M: HM , v: L1` and → `w: L1` (in the bottom part of Figure 75) indicate the memory regions to be used for reading and writing the input scalar of function f (also described later in detail).

[14] We currently rely on auto-tuning [61] for choosing optimized values of performance-critical parameters, as we discuss in Section 4.5.

3. RE-COMPOSITION PHASE: Finally, the re-composition phase (steps 9-15 in Figure 75) combines the computed parts $\uparrow a_f^{<p_1^1,p_2^1 \mid p_1^2,p_2^2 \mid p_1^3,p_2^3>}$ (bottom left in the figure) to the final result $\uparrow a$ (top left) via MDH's combine operators, which are in the case of matrix-vector multiplication $\circledast_1 := ++$ (concatenation) and $\circledast_2 := +$ (point-wise addition). In this example, we first combine the `L1` parts in dimension 2 and then in dimension 1; afterward, we combine the `COR` parts in both dimensions, and finally the `HM` parts. Analogously to before, this order of combine operators and their tags are chosen arbitrarily for this example and should be auto-tuned for high performance.

In the de- and re-composition phases, the arrow notation below combine operators allow efficiently exploiting architecture's memory hierarchy, by indicating the memory region to read from (de-composition phase) or to write to (re-composition phase); the annotations also indicate the memory layouts to use. We exploit these memory and layout information in both: i) our code generation process to assign combine operators' input and output data to memory regions and to chose memory layouts for the data (row major, column major, etc); ii) our formalism to specify constraints of programming models, e.g., that in CUDA, results of GPU cores can only be combined in designated memory regions [37]. For example, annotation → `M: HM[1,2] , v: L1[1]` below an operator in the de-composition phase indicates to our code generator that the parts (a.k.a tiles) of matrix M used for this computation step should be read from host memory `HM` and that parts of vector v should be copied to and accessed from fast `L1` memory. The annotation also indicates that `M` should be stored using a row-major memory layout (as we use `[1,2]` and not `[2,1]`). The memory regions and layouts are chosen arbitrarily for this example and should be chosen as optimized (auto-tuned) for the particular target architecture and characteristics of the input and output data. Formally, the arrow notation of combination operators is a concise notation to hide MDAs and BUFs for intermediate results (discussed in the Appendix, Section .5.3, for the interested reader).

Excursion: Code Generation[15]

Our low-level expressions can be straightforwardly transformed to executable program code in imperative-style programming languages (such as OpenMP, CUDA, and OpenCL). As code generation is not the focus of this work, we outline our code generation approach briefly using the example of Figure 75. Details about our code generation process are provided in Section .8 of our Appendix, and will be presented and illustrated in detail in our future work.

[15] Our implementation of MDH is open source: `https://mdh-lang.org`

We implement combine operators as sequential or parallel loops. For example, the operator $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ is assigned to memory layer `HM` and thus implemented as a sequential loop (loop range indicated by $[0,2)_{\mathbb{N}_0}$), and operator $+\!\!+_1^{(\texttt{COR},\texttt{x})}$ is assigned to core layer `COR` and thus implemented as a parallel loop (e.g., a loop annotated with `#pragma omp parallel for` in OpenMP [44], or variable `threadIdx.x` in CUDA [38]). Correspondingly, our three phases (de-composition, scalar, and re-composition) each correspond to an individual loop nest; we generate the nests as fused when the tags of combine operators have the same order in phases, as in Figure 75. Note that our currently targeted programming models (OpenMP, CUDA, and OpenCL) have no explicit notion of *tiles*, e.g., by offering the potential variable `tileIdx.x` for managing tiles automatically in the programming model (similarly as variable `threadIdx.x` automatically manages threads in CUDA). Consequently, when the operator tag refers to a memory layer, the dimension information within tags are currently ignored by our code generator (such as dimension `x` in tag `(HM,x)` which refers to memory layer `HM`).

Operators' memory regions correspond to straightforward allocations (e.g., in CUDA's `device`, `shared`, or `register` memory [38], according to the arrow annotations in our low-level expression). Memory layouts are implemented as aliases, e.g., *preprocessor directives* such as `#define M(i,k) M[k][i]` for storing `MatVec`'s input matrix M as transposed.

We implement MDAs also as aliases (according to Definition 32), e.g., `#define inp_mda(i,k) M[i][k],v[k]` for `MatVec`'s input MDA.

Code optimizations that are applied on a lower abstraction level than proposed by our representation in Example 75 are beyond the scope of this work and outlined in Section .9 of our Appendix e.g., loop fusion and loop unrolling which are applied on the loop-based abstraction level.

We provide an open source *MDH compiler* for code generation [3]. Our compiler takes as input a high-level MDH expression (as in Figure 64), in the form of a Python program (see Appendix, Section .1.4), and it fully automatically generates auto-tuned program code from it.

In the following, we introduce in Section .4.2 our formal representation of a computer system (which can be a single device, but also a multi-device or a multi-node system, as we discuss soon), and we illustrate our formal system representation using the example architectures targeted by programming models OpenMP, CUDA, and OpenCL. Afterward, in Section .4.3, we formally define the basic building blocks of our low-level representation – *low-level MDAs*, *low-level BUFs*, and *low-level combine operators* – based on our formal system representation.

.4.2 *Abstract System Model (ASM)*

Definition 37 (Abstract System Model)**.** An L-*Layered Abstract System Model (ASM)*, $L \in \mathbb{N}$, is any pair of two positive natural numbers

$$(\,\texttt{NUM\_MEM\_LYRs}\,,\,\texttt{NUM\_COR\_LYRs}\,) \in \mathbb{N} \times \mathbb{N}$$

for which $\texttt{NUM\_MEM\_LYRs} + \texttt{NUM\_COR\_LYRs} = L$.

Our ASM representation is capable of modeling architectures with arbitrarily deep memory and core hierarchies[16]: `NUM_MEM_LYRs` denotes the target architecture's number of memory layers and `NUM_COR_LYRs` the architecture's number of core layers, correspondingly. For example, the artificial architecture we use in Figure 75 is represented as an ASM instance as follows (bar symbols denote set cardinality):

$$\texttt{ASM}_{\texttt{artif.}} := (\ |\{\texttt{HM},\texttt{L1}\}|\ ,\ |\{\texttt{COR}\}|\ ) = (2,1)$$

The instance is a pair consisting of the numbers 2 and 1 which represent the artificial architecture's two memory layers (`HM` and `L1`) and its single core layers (`COR`).

Example 26. We show particular ASM instances that represent the device models of the state-of-practice approaches OpenMP, CUDA, and OpenCL[17]:

$$\texttt{ASM}_{\texttt{OpenMP}} :=$$
$$(\ |\{\texttt{MM},\texttt{L2},\texttt{L1}\}|\ ,\ |\{\texttt{COR}\}|\ ) = (3,1)$$

$$\texttt{ASM}_{\texttt{OpenMP+L3}} :=$$
$$(\ |\{\texttt{MM},\texttt{L3},\texttt{L2},\texttt{L1}\}|\ ,\ |\{\texttt{COR}\}|\ ) = (4,1)$$

$$\texttt{ASM}_{\texttt{OpenMP+L3+SIMD}} :=$$
$$(\ |\{\texttt{MM},\texttt{L3},\texttt{L2},\texttt{L1}\}|\ ,\ |\{\texttt{COR},\texttt{SIMD}\}|\ ) = (4,2)$$

$$\texttt{ASM}_{\texttt{CUDA}} :=$$
$$(\ |\{\texttt{DM},\texttt{SM},\texttt{RM}\}|\ ,\ |\{\texttt{SMX},\texttt{CC}\}|\ ) = (3,2)$$

$$\texttt{ASM}_{\texttt{CUDA+WRP}} :=$$
$$(\ |\{\texttt{DM},\texttt{SM},\texttt{RM}\}|\ ,\ |\{\texttt{SMX},\texttt{WRP},\texttt{CC}\}|\ ) = (3,3)$$

$$\texttt{ASM}_{\texttt{OpenCL}} :=$$
$$(\ |\{\texttt{GM},\texttt{LM},\texttt{PM}\}|\ ,\ |\{\texttt{CU},\texttt{PE}\}|\ ) = (3,2)$$

[16] We deliberately do not model into our ASM representation an architecture's particular number of cores and/or sizes of memory regions, because our optimization process is designed to be generic in these numbers and sizes, for high flexibility.

[17] Differences between Example 26 and Figure 4 are outlined in the Appendix, Section .5.4.

OpenMP is often used to target $(3+1)$-layered architectures which rely on 3 memory regions (main memory `MM`, and caches `L2` and `L1`) and 1 core layer (`COR`). OpenMP-compatible architectures sometimes also contain the `L3` memory region, and they may allow exploiting SIMD parallelization (a.k.a. *vectorization* [227]), which are expressed in our ASM representation as a further memory or core layer, respectively.

CUDA's target architectures are $(3+2)$-layered: they consist of *Device Memory (`DM`)*, *Shared Memory (`SM`)*, and *Register Memory (`RM`)*, and they offer as cores so-called *Streaming Multiprocessors (`SMX`)* which themselves consist of *Cuda Cores (`CC`)*. CUDA also has an implicit notion of so-called *Warps (`WRP`)* which are not explicitly represented in the CUDA programming model [38], but often exploited by programmers – via special intrinsics (e.g., *shuffle* and *tensor core intrinsics* [129, 148]) – to achieve highest performance.

OpenCL-compatible architectures are designed analogously to those targeted by the CUDA programming model; consequently, both OpenCL- and CUDA-compatible architectures are represented by the same ASM instance in our formalism. Apart from straightforward syntactical differences between OpenCL and CUDA [172], we see as the main differences between the two programming models (from our ASM-based abstraction level) that OpenCL has no notion of warps, and it uses a different terminology – *Global/Local/Private Memory (`GM`/`LM`/`PM`)* instead of device/shared/register memory, and *Compute Unit (`CU`)* and *Processing Element (`PE`)*, rather than `SMX` and `CC`.

In the following, we consider memory regions and cores of ASM-represented architectures as arrangeable in an arbitrary number of dimensions. Programming models for such architectures often have native support for such arrangements. For example, in the CUDA model, memory is accessed via arrays which can be arbitrary-dimensional (a.k.a *multi-dimensional C arrays*), and cores are programmed in CUDA via threads which are arranged in CUDA's so-called dimensions `x`, `y`, `z`; further thread dimensions can be explicitly programmed in CUDA, e.g., by embedding them in the last dimension `z`. Details on our arrangements of memory and cores are provided in the Appendix, Section .5.5.

We express constraints of programming models – for example, that in CUDA, `SMX` can combine their results in `DM` only [37] – via so-called *tuning parameter constraints*, which we discuss later in this section.

Note that we call our abstraction *Abstract System Model* (rather than *Abstract Architecture Model*, or the like), because it can also represent systems consisting of multiple devices and/or nodes, etc. For example, our ASM representation of a multi-GPU system is:

$$\texttt{ASM}_{\texttt{Multi-GPU}} := \left(\left|\{\texttt{HM}, \texttt{DM}, \texttt{SM}, \texttt{RM}\}\right| , \left|\{\texttt{GPU}, \texttt{SMX}, \texttt{CC}\}\right| \right) = (4, 3)$$

It extends our ASM-based representation of CUDA devices (Example 26) by *Host Memory (HM)* which represents the memory region of the system containing the GPUs (and in which the intermediate results of different GPUs are combined), and it introduces the further core layer GPU representing the system's GPUs. Analogously, our ASM representation of a multi-node, multi-GPU system is:

$$\texttt{ASM}_{\texttt{Multi-Node/GPU}} := (\,|\{\texttt{NM},\texttt{HM},\texttt{DM},\texttt{SM},\texttt{RM}\}|\,,\,|\{\texttt{NOD},\texttt{GPU},\texttt{SMX},\texttt{CC}\}|\,) = (5,4)$$

It adds to $\texttt{ASM}_{\texttt{Multi-GPU}}$ the memory layer *Node Memory (NM)* which represents the memory region of the host node, and it adds core layer *Node* (NOD) which represents the compute nodes. Our approach is currently designed for *homogeneous systems*, i.e., all devices/nodes/... are assumed to be identical. We aim to extend our approach to *heterogeneous systems* (which may consist of different devices/nodes/...) as future work, inspired by dynamic load balancing approaches [248].

.4.3 *Basic Building Blocks*

We introduce the three main basic building blocks of our low-level representation: 1) *low-level MDAs* which we use to partition MDAs and that represent multi-layered, multi-dimensionally arranged collection of ordinary MDAs (Definition 25) – one ordinary MDA per memory/core layer of their target ASM and for each dimension of the MDH computation (as illustrated in Figure 76); 2) *low-level BUFs* which are a collection of ordinary BUFs (Definition 29) and that are augmented with a *memory region* and a *memory layout*; 3) *low-level combine operators* which represent combine operators (Definition 26) to which the layer and dimension of their target ASM is assigned to be used to compute the operator in our generated code (e.g., a core layer to compute the operator in parallel).

Definition 38 (Low-Level MDA)**.** Let be $L \in \mathbb{N}$ (representing an ASM's number of layers) and $D \in \mathbb{N}$ (representing an MDH's number of dimensions). Let further be $P = (\,(P_1^1,\ldots,P_D^1)\,,\ldots,(P_1^L,\ldots,P_D^L)\,) \in \mathbb{N}^{L\times D}$ an arbitrary tuple of L-many D-tuples of positive natural numbers, $T \in \texttt{TYPE}$ a scalar type, and $I := (\,(I_d^{<p_1^1,\ldots,p_D^1|\ldots|p_1^L,\ldots,p_D^L>} \in \texttt{MDA-IDX-SETs})_{d\in[1,D]_{\mathbb{N}}}\,)^{<(p_1^1,\ldots,p_D^1)\in P_1^1\times\ldots\times P_D^1\;|\;\ldots\;|\;(p_1^L,\ldots,p_D^L)\in P_1^L\times\ldots\times P_D^L>}$ an arbitrary collection of D-many MDA index sets (Definition 25) for each particular choice of indices $(p_1^1,\ldots,p_D^1) \in P_1^1\times\ldots\times P_D^1$, ... , $(p_D^L,\ldots,p_D^L) \in P_1^L\times\ldots\times P_D^L$ [18] (illustrated in Figure 76).

An L-layered, D-dimensional, P-partitioned *low-level MDA* that has scalar type T and index sets I is any function a_{ll} of type:

$$a_{ll}^{\overbrace{<(p_1^1,\ldots,p_D^1)\in P_1^1\times\ldots\times P_D^1}^{\text{Partitioning: Layer 1}}\;|\;\ldots\;|\;\overbrace{(p_1^L,\ldots,p_D^L)\in P_1^L\times\ldots\times P_D^L>}^{\text{Partitioning: Layer L}}}:$$

$$I_1^{<p_1^1,\ldots,p_D^1|\ldots|p_1^L,\ldots,p_D^L>}\times\ldots\times I_D^{<p_1^1,\ldots,p_D^1|\ldots|p_1^L,\ldots,p_D^L>}\to T$$

[18] Analogously to Notation 6, we identify each $P_d^l \in \mathbb{N}$ implicitly also with the interval $[0,P_d^l)_{\mathbb{N}_0}$ (inspired by set theory).

We use low-level MDAs in the following to represent partitionings of MDAs (as illustrated soon and formally discussed the Appendix, Section .5.8).

Next, we introduce *low-level BUFs* which work similarly as BUFs (Definition 29), but are tagged with a memory region and a memory layout. While these tags have no effect on the operators' semantics, they indicate later to our code generator in which memory region the BUF should be stored and accessed, and which memory layout to chose for storing the BUF. Moreover, we use these tags to formally define constraints of programming models, e.g., that according to the CUDA specification [37], `SMX` cores can combine their results in memory region `DM` only.

Definition 39 (Low-Level BUF)**.** Let be $L \in \mathbb{N}$ (representing an ASM's number of layers) and $D \in \mathbb{N}$ (representing an MDH's number of dimensions). Let further $P = ((P_1^1,\ldots,P_D^1) , \ldots , (P_1^L,\ldots,P_D^L)) \in \mathbb{N}^{L\times D}$ be an arbitrary tuple of L-many D-tuples of positive natural numbers, $T \in \texttt{TYPE}$ a scalar type, and $N := ((N_d^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>} \in \mathbb{N})_{d\in[1,D]_{\mathbb{N}}})^{<(p_1^1,\ldots,p_D^1)\in P_1^1\times\ldots\times P_D^1 \mid \ldots \mid (p_1^L,\ldots,p_D^L)\in P_1^L\times\ldots\times P_D^L>}$ be a BUF's size (Definition 29) for each particular choice of $p_1^1,\ldots,p_D^L$.

An L-layered, D-dimensional, P-partitioned *low-level BUF* that has scalar type T and size N is any function b_{ll} of type ($\twoheadrightarrow$ denotes bijection):

$$b_{ll}^{\overbrace{<\texttt{MEM}\in[1,\texttt{NUM\_MEM\_LYRs}]_{\mathbb{N}}}^{\text{Memory Region}} \mid \overbrace{\sigma:[1,D]_{\mathbb{N}}\twoheadrightarrow[1,D]_{\mathbb{N}}}^{\text{Memory Layout}}>< \ldots \mid \overbrace{(p_1^l,\ldots,p_D^l)\in P_1^l\times\ldots\times P_D^l}^{\text{Partitioning: Layer } l} \mid \ldots >} :$$
$$[0,N_1^{<\ldots\mid p_1^l,\ldots,p_D^l\mid\ldots>})_{\mathbb{N}_0} \times \ldots \times [0,N_D^{<\ldots\mid p_1^l,\ldots,p_D^l\mid\ldots>})_{\mathbb{N}_0} \to T$$

We refer to `MEM` as low-level BUF's *memory region* and to σ as its *memory layout*, and we refer to the function

$$b_{ll}^{\texttt{trans}<\overbrace{\ldots}^{\text{Memory Region}} \mid \overbrace{\ldots}^{\text{Memory Layout}} >< \ldots \mid \overbrace{\ldots}^{\text{Partitioning: Layer } l} \mid \ldots >} :$$
$$[0,N_{\sigma(1)}^{<\ldots>})_{\mathbb{N}_0} \times \ldots \times [0,N_{\sigma(D)}^{<\ldots>})_{\mathbb{N}_0} \to T$$

that is defined as

$$b_{ll}^{\texttt{trans}<\ldots><\ldots>}(i_{\sigma(1)},\ldots,i_{\sigma(D)}) := b_{ll}^{<\ldots><\ldots>}(i_1,\ldots,i_D)$$

as b_{ll}'s *transposed function representation* (which we use to store the buffer in our generated code).

Finally, we introduce *low-level combine operators*. We define such operators to behave the same as ordinary combine operators (Definition 26), but we additionally tag them with a layer of their target ASM. Similarly as for low-level BUFs, the tag has no effect on semantics, but it is used in our code generation process to assign the computation to the hardware (e.g., indicating that the operator is computed by either an `SMX`, `WRP`, or `CC` when targeting CUDA – see Example 26). Also, we use the tags to define model-specific constraints in our formalism (as

also discussed for low-level BUFs). We also tag the combine operator with a dimension of the ASM layer, enabling later in our optimization process to express advanced data access patterns (a.k.a. *swizzles* [101]). For example, when targeting CUDA, flexibly mapping ASM dimensions on `CC` layer (in CUDA terminology, the dimensions are called `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`) to array dimensions enables the well-performing *coalesced global memory accesses* [37] for both transposed and non-transposed data layouts, by only using different dimension tags.

Definition 40 (ASM Level)**.** We refer to pairs (l_{ASM}, d_{ASM}) – consisting of an ASM layer $l_{ASM} \in [1, L]_{\mathbb{N}}$ and an ASM dimension $d_{ASM} \in [1, D]_{\mathbb{N}}$ – as *ASM Levels* (`ASM-LVL`)[19] (terminology motivated in the Appendix, Section .5.6):

$$\texttt{ASM-LVL} := \{\ (l_{ASM}, d_{ASM}) \mid l_{ASM} \in [1, L]_{\mathbb{N}}, d_{ASM} \in [1, D]_{\mathbb{N}}\}$$

Definition 41 (Low-Level Combine Operator)**.** Let be $L \in \mathbb{N}$ (representing an ASM's number of layers) and $D \in \mathbb{N}$ (representing an MDH's number of dimensions).

A *low-level combine operator*

$$\circledast^{<(l_{ASM}, d_{ASM}) \in \texttt{ASM-LVL} = \{\ (l,d)\ \mid\ l \in [1,L]_{\mathbb{N}},\ d \in [1,D]_{\mathbb{N}}\ \}>}$$

is a function for which $\circledast^{<(l_{ASM}, d_{ASM})>}$ is an ordinary combine operator (Definition 26), for each $(l_{ASM}, d_{ASM}) \in \texttt{ASM-LVL}$.

Note that in Figure 75, for better readability, we use domain-specific identifiers for ASM layers: `HM:=1` as an alias for the ASM layer that has id 1, `L1:=2` for the layer with id 2, and `COR:=3` for the layer with id 3. For dimensions, we use aliases $x := 1$ for ASM dimension 1 and $y := 2$ for ASM dimension 2, correspondingly.

.4.4 *Generic Low-Level Expression*

Figure 77 shows a generic expression in our low-level representation: it targets an arbitrary but fixed L-layered ASM instance, and it implements – on low level – the generic instance of our high-level expression in Figure 73. Inserting into the low-level expression a particular value for ASM's numbers of layer L, as well as particular values for the generic parameters of the high-level expression in Figure 73 (dimensionality D, combine operators $\circledast_1, \ldots, \circledast_d$, and input/output views) results in an instance of the expression in Figure 77 that remains generic in tuning parameters only; this auto-tunable instance will be the focus of our discussion in the remainder of this section.

In Section .6, we show that we fully automatically compute the auto-tunable low-level expression for a concrete ASM instance and high-level expression, and we automatically optimize this tunable expression for a particular target architecture and characteristics of the

[19] For simplicity, we refrain from annotating identifier `ASM-LVL` with values L and D (e.g., $\texttt{ASM-LVL}^{<L,D>}$), because both values will usually be clear from the context.

$$\mathfrak{b}^{\texttt{IB}}_1,\dots,\mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} \overset{\texttt{inp\_view}}{\mapsto} {}^{\downarrow}\mathfrak{a} \; =:$$

$$\underset{p_1^1\in\texttt{\#PRT(1,1)}}{\#_1^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{1,1})}} \quad \dots \quad \underset{p_D^1\in\texttt{\#PRT(1,D)}}{\#_D^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{1,D})}}$$

$$\underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{IB}}_1 : \downarrow\texttt{-mem}(1,1)[\sigma_{\downarrow\texttt{-mem}}(1,1)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(1,1)(D^{\texttt{IB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} : \downarrow\texttt{-mem}(1,1)[\sigma_{\downarrow\texttt{-mem}}(1,1)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(1,1)(D^{\texttt{IB}}_{B^{\texttt{IB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\downarrow\texttt{-ord}}(\texttt{1,1})} \quad \underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{IB}}_1 : \downarrow\texttt{-mem}(1,D)[\sigma_{\downarrow\texttt{-mem}}(1,D)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(1,D)(D^{\texttt{IB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} : \downarrow\texttt{-mem}(1,D)[\sigma_{\downarrow\texttt{-mem}}(1,D)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(1,D)(D^{\texttt{IB}}_{B^{\texttt{IB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\downarrow\texttt{-ord}}(\texttt{1,D})}$$

$$\vdots \qquad\qquad \vdots$$

$$\underset{p_1^L\in\texttt{\#PRT(L,1)}}{\#_1^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{L,1})}} \quad \dots \quad \underset{p_D^L\in\texttt{\#PRT(L,D)}}{\#_D^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{L,D})}}$$

$$\underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{IB}}_1 : \downarrow\texttt{-mem}(L,1)[\sigma_{\downarrow\texttt{-mem}}(L,1)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(L,1)(D^{\texttt{IB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} : \downarrow\texttt{-mem}(L,1)[\sigma_{\downarrow\texttt{-mem}}(L,1)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(L,1)(D^{\texttt{IB}}_{B^{\texttt{IB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\downarrow\texttt{-ord}}(\texttt{L,1})} \quad \underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{IB}}_1 : \downarrow\texttt{-mem}(L,D)[\sigma_{\downarrow\texttt{-mem}}(L,D)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(L,D)(D^{\texttt{IB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} : \downarrow\texttt{-mem}(L,D)[\sigma_{\downarrow\texttt{-mem}}(L,D)(1),\dots,\sigma_{\downarrow\texttt{-mem}}(L,D)(D^{\texttt{IB}}_{B^{\texttt{IB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\downarrow\texttt{-ord}}(\texttt{L,D})}$$

$${}^{\downarrow}\mathfrak{a}_f^{<p_1^1,\dots,\,p_D^1\ |\ \dots\ |\ p_1^L,\dots,\,p_D^L>}$$

(a) De-Composition Phase

$${}^{\downarrow}\mathfrak{a}_f^{<p_1^1,\dots,\,p_D^1\ |\ \dots\ |\ p_1^L,\dots,\,p_D^L>} \overset{\vec{f}}{\mapsto} {}^{\uparrow}\mathfrak{a}_f^{<p_1^1,\dots,\,p_D^1\ |\ \dots\ |\ p_1^L,\dots,\,p_D^L>}$$

$$\rightarrow <\ \sigma_{f\texttt{-ord}}(\texttt{1,1})\ ,\ \dots\ ,\ \sigma_{f\texttt{-ord}}(\texttt{L,D})\ >$$

$$\rightarrow <\leftrightarrow_{f\texttt{-ass}}(\texttt{1,1})\ ,\ \dots\ ,\ \leftrightarrow_{f\texttt{-ord}}(\texttt{L,D})>$$

$$\rightarrow\quad \mathfrak{b}^{\texttt{IB}}_1 :\ f^{\downarrow}\texttt{-mem}[\sigma_{f^{\downarrow}\texttt{-mem}}(1),\dots,\sigma_{f^{\downarrow}\texttt{-mem}}(D^{\texttt{IB}}_1)]\ ,\ \dots\ ,\ \mathfrak{b}^{\texttt{IB}}_{B^{\texttt{IB}}} :\ f^{\downarrow}\texttt{-mem}[\sigma_{f^{\downarrow}\texttt{-mem}}(1),\dots,\sigma_{f^{\downarrow}\texttt{-mem}}(D^{\texttt{IB}}_{B^{\texttt{IB}}})]$$

$$\rightarrow\quad \mathfrak{b}^{\texttt{OB}}_1 :\ f^{\uparrow}\texttt{-mem}[\sigma_{f^{\uparrow}\texttt{-mem}}(1),\dots,\sigma_{f^{\uparrow}\texttt{-mem}}(D^{\texttt{OB}}_1)]\ ,\ \dots\ ,\ \mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} :\ f^{\uparrow}\texttt{-mem}[\sigma_{f^{\uparrow}\texttt{-mem}}(1),\dots,\sigma_{f^{\uparrow}\texttt{-mem}}(D^{\texttt{OB}}_{B^{\texttt{OB}}})]$$

(b) Scalar Phase

$$\mathfrak{b}^{\texttt{OB}}_1,\dots,\mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} \overset{\texttt{out\_view}}{\hookleftarrow} {}^{\uparrow}\mathfrak{a} \; :=$$

$$\underset{p_1^1\in\texttt{\#PRT(1,1)}}{\circledast_1^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{1,1})}} \quad \dots \quad \underset{p_D^1\in\texttt{\#PRT(1,D)}}{\circledast_D^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{1,D})}}$$

$$\underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{OB}}_1 : \uparrow\texttt{-mem}(1,1)[\sigma_{\uparrow\texttt{-mem}}(1,1)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(1,1)(D^{\texttt{OB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} : \uparrow\texttt{-mem}(1,1)[\sigma_{\uparrow\texttt{-mem}}(1,1)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(1,1)(D^{\texttt{OB}}_{B^{\texttt{OB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\uparrow\texttt{-ord}}(\texttt{1,1})} \quad \underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{OB}}_1 : \uparrow\texttt{-mem}(1,D)[\sigma_{\uparrow\texttt{-mem}}(1,D)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(1,D)(D^{\texttt{OB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} : \uparrow\texttt{-mem}(1,D)[\sigma_{\uparrow\texttt{-mem}}(1,D)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(1,D)(D^{\texttt{OB}}_{B^{\texttt{OB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\uparrow\texttt{-ord}}(\texttt{1,D})}$$

$$\vdots \qquad\qquad \vdots$$

$$\underset{p_1^L\in\texttt{\#PRT(L,1)}}{\circledast_1^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{L,1})}} \quad \dots \quad \underset{p_D^L\in\texttt{\#PRT(L,D)}}{\circledast_D^{\leftrightarrow_{\texttt{prt-ass}}(\texttt{L,D})}}$$

$$\underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{OB}}_1 : \uparrow\texttt{-mem}(L,1)[\sigma_{\uparrow\texttt{-mem}}(L,1)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(L,1)(D^{\texttt{OB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} : \uparrow\texttt{-mem}(L,1)[\sigma_{\uparrow\texttt{-mem}}(L,1)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(L,1)(D^{\texttt{OB}}_{B^{\texttt{OB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\uparrow\texttt{-ord}}(\texttt{L,1})} \quad \underbrace{\begin{array}{l}\rightarrow\ \mathfrak{b}^{\texttt{OB}}_1 : \uparrow\texttt{-mem}(L,D)[\sigma_{\uparrow\texttt{-mem}}(L,D)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(L,D)(D^{\texttt{OB}}_1)] \\ \quad\vdots \\ \quad \mathfrak{b}^{\texttt{OB}}_{B^{\texttt{OB}}} : \uparrow\texttt{-mem}(L,D)[\sigma_{\uparrow\texttt{-mem}}(L,D)(1),\dots,\sigma_{\uparrow\texttt{-mem}}(L,D)(D^{\texttt{OB}}_{B^{\texttt{OB}}})]\end{array}}_{\hookrightarrow\ \sigma_{\uparrow\texttt{-ord}}(\texttt{L,D})}$$

$${}^{\uparrow}\mathfrak{a}_f^{<p_1^1,\dots,\,p_D^1\ |\ \dots\ |\ p_1^L,\dots,\,p_D^L>}$$

(c) Re-Composition Phase

Figure 77: Generic low-level expression for data-parallel computations.

No.	Name	Range	Description
0	#PRT	MDH-LVL $\to \mathbb{N}$	number of parts
D1	$\sigma_{\downarrow\text{-ord}}$	MDH-LVL $\nrightarrow$ MDH-LVL	de-composition order
D2	$\leftrightarrow_{\downarrow\text{-ass}}$	MDH-LVL $\nrightarrow$ ASM-LVL	ASM assignment (de-composition)
D3	$\downarrow\text{-mem}^{\langle ib\rangle}$	MDH-LVL $\to$ MR	memory regions of input BUFs (ib)
D4	$\sigma^{\langle ib\rangle}_{\downarrow\text{-mem}}$	MDH-LVL $\to [1,\ldots,\mathrm{D}^{\mathtt{IB}}_{ib}]_{\mathcal{S}}$	memory layouts of input BUFs (ib)
S1	$\sigma_{\text{f-ord}}$	MDH-LVL $\nrightarrow$ MDH-LVL	scalar function order
S2	$\leftrightarrow_{\text{f-ass}}$	MDH-LVL $\nrightarrow$ ASM-LVL	ASM assignment (scalar function)
S3	$\text{f}^{\downarrow}\text{-mem}^{\langle ib\rangle}$	MR	memory region of input BUF (ib)
S4	$\sigma^{\langle ib\rangle}_{\text{f}^{\downarrow}\text{-mem}}$	$[1,\ldots,\mathrm{D}^{\mathtt{IB}}_{ib}]_{\mathcal{S}}$	memory layout of input BUF (ib)
S5	$\text{f}^{\uparrow}\text{-mem}^{\langle ob\rangle}$	MR	memory region of output BUF (ob)
S6	$\sigma^{\langle ob\rangle}_{\text{f}^{\uparrow}\text{-mem}}$	$[1,\ldots,\mathrm{D}^{\mathtt{OB}}_{ob}]_{\mathcal{S}}$	memory layout of output BUF (ob)
R1	$\sigma_{\uparrow\text{-ord}}$	MDH-LVL $\nrightarrow$ MDH-LVL	re-composition order
R2	$\leftrightarrow_{\uparrow\text{-ass}}$	MDH-LVL $\nrightarrow$ ASM-LVL	ASM assignment (re-composition)
R3	$\uparrow\text{-mem}^{\langle ob\rangle}$	MDH-LVL $\to$ MR	memory regions of output BUFs (ob)
R4	$\sigma^{\langle ob\rangle}_{\uparrow\text{-mem}}$	MDH-LVL $\to [1,\ldots,\mathrm{D}^{\mathtt{OB}}_{ob}]_{\mathcal{S}}$	memory layouts of output BUFs (ob)

Table 8: Tuning parameters of our low-level expressions.

input and output data using auto-tuning [61]. Our final outcome is a concrete (non-generic) low-level expression (as in Figure 75) that is auto-tuned for the particular target architecture (represented via an ASM instance, e.g., ASM instance $\mathtt{ASM}_{\mathtt{CUDA}}$ when targeting an NVIDIA Ampere GPU) and high-level MDH expression. From this auto-tuned low-level expression, we can straightforwardly generate executable program code, because all the major optimization decisions have already been made in the previous auto-tuning step. Our overall approach is illustrated in Figure 10.

Auto-Tunable Parameters

Table 8 lists the tuning parameters of our auto-tunable low-level expressions – different values of tuning parameters lead to semantically equal variants of the auto-tunable low-level expression (which we prove formally in Section .6), but the variants will be translated to differently optimized code variants.

In the following, we explain the 15 tuning parameters in Table 8. We give our explanations in a general, formal setting that is independent of a particular computation and programming model; the parameters are discussed afterward for the concrete example computation *matrix multiplication* in the models OpenMP, CUDA, and OpenCL.

Our tuning parameters in Table 8 have constraints: 1) *algorithmic constraints* which have to be satisfied by all target programming models, and 2) *model constraints* which are specific for particular programming models only (CUDA-specific constraints, OpenCL-specific constraints, etc), e.g., that the results of CUDA's thread blocks can be combined in designated memory regions only [37]. We discuss algorithmic constraints in the following, together with our tuning parameters; model constraints are discussed in our Appendix, Section .5.1, for the interested reader.

In the following, we present our 15 tuning parameters in Table 8. Dotted lines separate parameters for different phases: parameters `D1-D4` customize the de-composition phase, parameters `S1-S6` the scalar phase, and parameters `R1-R4` the re-composition phase, correspondingly; the parameter `0` impacts all three phases (separated by a straight line in the table).

Note that our parameters do not aim to introduce novel optimization techniques, but to unify, generalize, and combine together well-proven optimizations, based on a formal foundation, toward an efficient, overall optimization process that applies to various combinations of data-parallel computations, architectures, and characteristics of input and output data (e.g., their size and memory layout).

In Table 8, we point to combine operators in Figure 75 using pairs (l, d) to which we refer as *MDH Levels* (terminology motivated in the Appendix, Section .5.7). We use the pairs as enumeration for operators in the de-composition and re-composition phases.

Definition 42 (MDH Level)**.** We refer to pairs $(l_{\texttt{MDH}}, d_{\texttt{MDH}})$ – consisting of a layer $l_{\texttt{MDH}} \in [1, L]_{\mathbb{N}}$ and dimension $d_{\texttt{MDH}} \in [1, D]_{\mathbb{N}}$ – as *MDH Levels* (`MDH-LVL`):

$$\texttt{MDH-LVL} := \{\ (l_{\texttt{MDH}}, d_{\texttt{MDH}}) \mid l_{\texttt{MDH}} \in [1, L]_{\mathbb{N}}, d_{\texttt{MDH}} \in [1, D]_{\mathbb{N}} \}^{20}$$

For example, in the de-composition phase of Figure 75 (right part of the figure), pair $(1, 1) \in$ `MDH-LVL` points to the first combine operator, as the operator operates on the first layer $l = 1$ and in the first dimension $d = 1$ (discussed in Section .4.1). Analogously, pairs $(1, 2), (2, 1) \in$ `MDH-LVL` point to the second and third operator, etc. An operator's enumeration can be easily deduced from its corresponding p variable: the variable's superscript indicates the operator's corresponding layer l and the variable's subscript indicates its dimension d.

PARAMETER 0: Parameter #PRT is a function that maps pairs in MDH-LVL to natural numbers; the parameter determines *how much* data are grouped together into parts in our low-level expression in Figure 77 (and consequently also in our generated code later), by setting the particular number of parts (a.k.a. *tiles*) used in our expression. For example, in Figure 75, we use $\#\mathtt{PRT}(1,1) := 2$ which causes combine operators $+\!\!+_1^{(\mathtt{HM},\mathtt{x})}$ and $\circledast_1^{(\mathtt{HM},\mathtt{x})}$ to iterate over interval $[0,2)_{\mathbb{N}_0}$ (and thus partitioning the MDH computation on level $(1,1)$ into two parts), and we use $\#\mathtt{PRT}(1,2) := 4$ to let operators $+\!\!+_2^{(\mathtt{HM},\mathtt{y})}$ and $\circledast_2^{(\mathtt{HM},\mathtt{x})}$ iterate over interval $[0,4)_{\mathbb{N}_0}$ (partitioning into four parts on level $(1,2)$), etc.

To ensure a full partitioning (so that we obtain singleton MDAs to which scalar function f can be applied in the scalar phase, as discussed above), we require the following algorithmic constraint for the parameter (N_d denotes the input size in dimension d, see Figure 73):

$$\prod_{l \in [1,L]_{\mathbb{N}}} \#\mathtt{PRT}(l,d) \;=\; N_d, \quad \text{for all } d \in [1,D]_{\mathbb{N}}$$

In our generated code, the number of parts directly translates to the number of *tiles* which are computed either sequentially (a.k.a. *cache blocking* [310]) or in parallel, depending on the combine operators's tags (which are chosen via Parameters D2,S2,R2, as discussed soon). In our example from Figure 75, we process parts belonging to combine operators tagged with HM and L1 sequentially, via for-loops, because HM and L1 correspond to ASM's memory layers (note that Parameter 0 only chooses the number of tiles; the parameter has no effect on explicitly copying data into fast memory resources, which is the purpose of Parameters D3,R3,S1,S2). The COR parts are computed in parallel in our generated code, because COR corresponds to ASM's core layer, and thus, the number of COR parts determines the number of threads used in our code.

An optimized number of tiles is essential for achieving high performance [307], e.g., due to its impact for locality-aware data accesses (number of sequentially computed tiles) and efficiently exploiting parallelism (number of tiles computed in parallel, which corresponds to the number of threads in our generated code).

PARAMETERS D1,S1,R1: These three parameters are permutations on MDH-LVL (indicated by symbol $\twoheadrightarrow$ in Table 8), determining *when* data are accessed and combined. The parameters specify the order (indicated by symbol $\hookrightarrow$ in Figure 77) of combine operators in the de-composition and re-composition phases (parameters D1 and R1), and the order of applying scalar function f to parts (parameter S1). Thereby, the parameters specify when parts are processed during the computation.

In our generated code, combine operators are implemented as sequential/parallel loops such that the parameters enable optimizing loop orders (a.k.a. *loop permutation* [303]). For combine operators assigned (via parameter `R2`) to ASM's core layer and thus computed in parallel, parameter `R1` particularly determines when the computed results of threads are combined: if we used in the re-composition phase of Figure 75 combine operators tagged with `(COR,x)` and `(COR,y)` immediately after applying scalar function `f` (i.e., in steps ⑩ and ⑪, rather than steps ⑫ and ⑬), we would combine the computed intermediate results of threads multiple times, repeatedly after each individual computation step of threads, and using the two operators at the end of the re-composition phase (in steps ⑭ and ⑮) would combine the result of threads only once, at the end of the re-composition phase. Combining the results of threads early in the computation usually has the advantages of reduced memory footprint, because memory needs to be allocated for one thread only, but at the cost of more computations, because the results of threads need to be combined multiple times. In contrast, combing the results of threads late in the computation reduces the amount of computations, but at the cost of higher memory footprint. Our parameters make this trade-off decision generic in our approach such that the decision can be left to an auto-tuning system, for example.

Note that each phase corresponds to an individual loop nest which we fuse together when parameters `D1,S1,R1` (as well as parameters `D2,S2,R2`) coincide (as also outlined in our Appendix, Section .9).

PARAMETERS `D2,S2,R2`: These parameters (symbol $\twoheadrightarrow$ in the table denotes bijection) assign MDH levels to ASM levels, by setting the tags of low-level combine operators (Definition 41). Thereby, the parameters determine *by whom* data are processed (e.g., threads or `for`-loops), similar to the concept of `bind` in scheduling languages [16]. Consequently, the parameters determine which parts should be computed sequentially in our generated code and which parts in parallel. For example, in Figure 75, we use $\leftrightarrow_{\downarrow\texttt{-ass}}\ (2,1) := (\texttt{COR},\texttt{x})$ and $\leftrightarrow_{\downarrow\texttt{-ass}}\ (2,2) := (\texttt{COR},\texttt{y})$, thereby assigning the computation of MDA parts on layer 2 in both dimensions to ASM's `COR` layer in the decomposition phase, which causes processing the parts in parallel in our generated code. For multi-layered core architectures, the parameters particularly determine the thread layer to be used for the parallel computation (e.g., `block` or `thread` in CUDA).

Using these parameters, we are able to flexibly set data access patterns in our generated code. In Figure 75, we assign parts on layer 2 to `COR` layers, which results in a so-called *block access* pattern of cores: we start $8 * 16$ threads, according to the $8 * 16$ core parts, and each thread processes a part of the input MDA representing a block of 32×64 MDA elements within the input data. If we had assigned in the figure the first computation layer to ASM's `COR` layer (in the figure, this layer is assigned to ASM's `HM` layer), we would start $2 * 4$ threads and each thread would process MDA parts of size $(8 * 32) \times (16 * 64)$; assigning

the last MDH layer to `COR`s would result in $(2 * 8 * 32) \times (4 * 16 * 64)$ threads each processing a singleton MDA (a.k.a. *strided access*).

The parameters also enable expressing so-called *swizzle* access patterns [101]. For example, in CUDA, processing consecutive data elements in data dimension 1 by threads that are consecutive in thread dimension 2 (a.k.a `threadIdx.y` dimension in CUDA) can achieve higher performance due to the hardware design of fast memory resources in NVIDIA GPUs. Such swizzle patterns can be easily expressed and auto-tuned in our approach; for example, by interchanging in Figure 75 tags `(COR,x)` and `(COR,y)`. For memory layers (such as `HM` and `L1`), the dimension tags `x` and `y` currently have no effect on our generated code, as the programming models we target at the moment (OpenMP, CUDA, and OpenCL) have no explicit notion of tiles. However, this might change in the future when targeting new kinds of programming models, e.g., for upcoming architectures.

PARAMETERS `D3,R3` AND `S3,S5`: Parameters `D3` and `R3` set for each BUF the memory region to be used, thereby determining *where* data are read from or written to, respectively. In the table, we use $\texttt{ib} \in [1, \mathrm{B}^{\texttt{IB}}]_{\mathbb{N}}$ to refer to a particular input BUF (e.g., `ib=1` to refer to the input matrix of matrix-vector multiplication, and `ib=2` to refer to the input vector), and $\texttt{ob} \in [1, \mathrm{B}^{\texttt{OB}}]_{\mathbb{N}}$ refers to an output BUF, correspondingly. Parameter `D3` specifies the memory region to read from, and parameter `R3` the regions to write to. The set $\texttt{MR} := [1, \texttt{NUM\_MEM\_LYRs}]_{\mathbb{N}}$ denotes the ASM's memory regions.

Similarly to parameters `D3` and `R3`, parameters `S3` and `S5` set the memory regions for the input and output of scalar function `f`.

Exploiting fast memory resources of architectures is a fundamental optimization [9, 64, 203, 300], particularly due to the performance gap between processors' cores and their memory systems [57, 285].

PARAMETERS `D4,R4` AND `S4,S6`: These parameters set the memory layouts of BUFs, thereby determining *how* data are accessed in memory; for brevity in Table 8, we denote the set of all BUF permutations $[1, \mathrm{D}]_{\mathbb{N}} \twoheadrightarrow [1, \mathrm{D}]_{\mathbb{N}}$ (Definition 39) as $[1, \ldots, \mathrm{D}]_{\mathcal{S}}$ (symbol $\mathcal{S}$ is taken from the notation of *symmetric groups* [284]). In the case of our matrix-vector multiplication example in Figure 75, we use a standard memory layout for all matrices, which we express via the parameters by setting them to the identity function, e.g., $\sigma^{\texttt{<M>}}_{\downarrow\texttt{-mem}}(1,1) := \mathrm{id}$ (Parameter `D4`) for the matrix read by operator $+\!\!+^{(\texttt{HM},\texttt{x})}_{1}$.

An optimized memory layout is important to access data in a locality-aware and thus efficient manner.

.4.5 *Examples*

Figures 78-81 show how our low-level representation is used for expressing the (de/re)-compositions of concrete, state-of-the-art implementations. For this, we use the popular example of matrix multiplication (`MatMul`), on a real-world input size taken from the `ResNet-50` [182] deep learning neural network (training phase).

To challenge our formalism: i) we express implementations generated and optimized according to notably different approaches: scheduling approach TVM using its recent Ansor [78] optimization engine which is specifically designed and optimized toward optimizing deep learning computations (e.g., `MatMul`); polyhedral compilers PPCG and Pluto with auto-tuned tile sizes; ii) we consider optimizations for two fundamentally different kinds of architectures: NVIDIA Ampere GPU and Intel Skylake CPU. We consider our study as challenging for our formalism, because it needs to express – in the same formal framework – the (de/re)-compositions of implementations generated and optimized according to notably different approaches (scheduling-based and polyhedral-based) and for significantly different kinds of architectures (GPU and CPU). Experimental results for TVM, PPCG, and Pluto (including the `MatMul` study used in this section) are presented and discussed in Section 4.5, as the focus of this section is on analyzing and discussing the expressivity of our low-level representation, rather than on its performance potential (which is often higher than that of TVM, PPCG, and Pluto, as we will see in Section 4.5).

In Figures 78-81, we list our low-level representation's particular tuning parameter values for expressing the TVM- and PPCG/Pluto-generated implementations. The parameters concisely describe the concrete (de/re)-composition strategies used by TVM, PPCG and Pluto for `MatMul` on GPU or CPU using the ResNet-50's input size. Inserting these tuning parameter values into our generic low-level expression in Figure 77 results in the concrete formal representation of the (de/re)-composition strategies used by TVM, PPCG and Pluto (similarly as in Figure 75).

In the following, we describe the columns of the tables in Figures 78-81, each of which listing particular values of tuning parameters in Table 8: column `0` lists values of tuning parameter `0` in Table 8, column `D1` of tuning parameter `D1`, etc. As all four tables follow the same structure, we focus on describing the particular example table in Figure 78 (example chosen arbitrarily), which shows the (de/re)-composition used in TVM's generated CUDA code for `MatMul` on NVIDIA Ampere GPU using input matrices of sizes 16×2048 and 2048×1000 taken from ResNet-50[21]. Note that for clarity, we use in the figures domain-specific aliases, instead of numerical values, to better differentiate between different ASM layers and memory regions. For example, we use in Figure 78 as aliases `DEV` := 1, `SHR` := 2, and `REG` := 3 to refer to CUDA's three memory layers (device memory layer `DEV`,

[21] For the interested reader, TVM's corresponding, Ansor-generated scheduling program is presented in our Appendix, Section .5.9.

shared memory layer SHR, and register memory layer REG), and we use DM := 1, SM := 2, and RM := 3 to refer to CUDA's memory regions device DM, shared SM, and register RM; aliases BLK := 4 and THR := 5 refer to CUDA's two core layers which are programmed via blocks and threads in CUDA.

We differentiate between memory layers and memory regions for the following reason: for example, using tuning parameter 0 in Table 8, we partition input data hierarchically for each particular memory layer of the target architecture (sometime possibly into one part only, which is equivalent to not partitioning). However, depending on the value of tuning parameter D3, we do not necessarily copy the input's parts always into the corresponding memory regions (e.g., a part on SHR layer is not necessarily copied into shared memory SM), for example, when the architecture provides automatically managed memory regions (as caches in CPUs) or when only some parts of the input are accessed multiple times (e.g., the input vector in the case of matrix-vector multiplication, but not the input matrix), etc.

COLUMN 0 The column lists the particular number of parts (a.k.a. *tiles*) used in TVM's multi-layered, multi-dimensional partitioning strategy for MatMul on the ResNet-50's input matrices which have the sizes $(I, K) = 16 \times 2048$ and $(K, J) = 2048 \times 1000$. We can observe from this column that the input MDA, which is initially of size $(I, J, K) = (16, 1000, 2048)$ for the ResNet-50's input matrices, is partitioned into $(2 * 50 * 1)$-many parts (indicated by the first three rows in column 1) – 2 parts in the first dimension, 50 parts in the second dimension, and 1 part in the third dimension. Each of these parts is then further partitioned into $(2 * 1 * 8)$-many parts (rows 4,5,6), and these parts are again partitioned into $(4 * 20 * 1)$-many further parts (rows 7,8,9), etc.

COLUMNS D1, S1, R1 These 3 columns describe the order in which parts are processed in the different phases: de-composition (column D1), scalar phase (column S1), and re-composition (column R1). For example, we can observe from column R1 that TVM's generated CUDA code first starts to combine parts on layer 1 in dimensions 1, 2, 3 (indicated by $(1,1)$, $(1,2)$, $(1,3)$ in rows 1, 2, 3 of column R1); afterward, the code combine parts on layer 3 in dimensions 1, 2, 3 (indicated by $(2,1)$, $(2,2)$, $(2,3)$ in rows 7, 8, 9 of column R1), etc.

Note that TVM uses the same order in the three phases (i.e., columns D1, S1, R1 coincide). Most likely this is because in CUDA, iteration over memory tiles are programmed via for-loops such that columns D1, S1, R1 represent loop orders; using the same order of loops in columns D1, S1, R1 thus allows TVM to generate the loop nests as fused for the three different phases (rather than generating three individual nests), which usually achieves high performance in CUDA.

Note further that the order of parts that are processed in parallel (columns `D2`, `S2`, `R2` determine if parts are processed in parallel or not, as described in the next paragraph) effects when results of blocks and threads are combined (a.k.a. *parallel reduction* [269]), e.g., early in the computation and thus often (but thereby often requiring less memory) or late and thus only once (but at the cost of higher more memory consumption), etc.

COLUMNS `D2`, `S2`, `R2` The columns determine how computations are assigned to the target architecture. In our example in Figure 78, we have $(2 * 50 * 1)$-many parts in the MDA's first partitioning layer, and each of these parts is assigned to be computed by an individual CUDA block (`BLK`) in the de-composition phase (rows 1-3 in column `D2`), i.e., TVM uses a so-called *grid size* of `2,50,1` in its generated CUDA code for `MatMul`. The $(4 * 20 * 1)$-many parts in the third partitioning layer (rows 7-9) are processed by CUDA threads (`THR`), i.e., the CUDA *block size* in the TVM-generated code is `4,20,1`. All other parts, e.g, those belonging to the $(2 * 1 * 8)$-many parts in the second partitioning layer (rows 4-6), are assigned to CUDA's memory layers (denoted as `DEV`, `SHR`, `REG` in Figure 78) and thus processed sequentially, via `for`-loops.

COLUMNS `D3`, `S3`, `S5`, `R3` While column `0` shows the multi-layered, multi-dimensional partitioning strategy used in TVM's CUDA code (according to the CUDA model's multi-layered memory and core hierarchies, shown in Example 26), column `0` does not indicate how CUDA's fast memory regions are exploited in the TVM-generated CUDA code for `MatMul` – column `0` only describes TVM's partitioning of the input/output computations such that parts of the input and output data can potentially fit into fast memory resources.

The actual mapping of parts to memory regions is done via columns `D3` (memory regions to be used for input data), columns `S3` and `S5` (memory regions to be used for the scalar computations), and column `R3` (memory regions to be used for storing the computed output data). For example, column `D3` indicates that in TVM's CUDA code for `MatMul`, parts of the A and B input matrices are stored in CUDA's fast shared memory `SM` (column `D3`, rows 4-5 and 10-15), and column `R3` indicates that each thread computes its results within CUDA's faster register memory `RM` (column `R3`, rows 4-6 and 10-15).

Our flexibility of separating the tiling strategy (Parameter `0`) from the actual usage of memory regions (columns `D3`, `S3`, `S5`, `R3`) allows us, for example, to store parts belonging to one input buffer into fast memory resources (e.g., the input vector of matrix-vector multiplication, whose values are accessed multiple times), but not parts of other buffers (e.g., the input matrix of matrix-vector multiplication, whose values are accessed only once) or only subparts of buffers, etc.

Note that in the case of Figure 81 which shows Pluto's (de/re)-composition for OpenMP code, the memory tags in columns `D3`, `S3`, `S5`, `R3` have no effect on the generated code: OpenMP relies on its target architecture's implicit memory system (e.g., CPU caches), rather than exposing the memory hierarchy explicitly to the programmer. Consequently, the memory tags are ignored by our OpenMP code generator and only emphasize the implementer's intention, e.g., that in Figure 81, each of the $(2 * 962 * 218)$-many tiles in the MDA's third partitioning layer are intended by the implementer to be processed in `L2` memory (rows 7-9 in columns `D3` and `R3`), even though this decision is eventually made by the automatic cache engine of the CPU.

COLUMNS `D4, S4, S6, R4` These columns set the memory layout to be used for memory allocations in the CUDA code. TVM choses in all cases CUDA's standard transpositions layout (indicated by `[1,2]` which is called *row-major* layout, instead of `[2,1]` which is known as *column-major* layout). Since the same layout and memory region is used on consecutive layers, the same memory allocation is re-used in the CUDA code. For example, parameters `D3` and `D4` contain the same values in rows 4-5 and 10-15, and thus only one memory buffer is allocated in shared memory for input buffer `A`; the buffer is accessed in the computations of all `SHR` and `REG` tiles, as well as `DEV` tiles in dimensions x and z. Similarly, only one buffer is allocated in register memory for computing the results of `SHR`, `REG`, and `DEV` tiles, because the rows 4-6 and 10-15 in columns `R3` and `R4` coincide.

TVM's (de/re)-composition for MatMul in CUDA on NVIDIA Ampere GPU

	De-Comp. Phase					*Scalar Phase*							*Re-Comp. Phase*			
0	**D1**	**D2**	**D3**		**D4**	**S1**	**S2**	**S3**		**S4**	**S5**	**S6**	**R1**	**R2**	**R3**	**R4**
			A	**B**	**A,B**			**A**	**B**	**A,B**	**C**	**C**			**C**	**C**
2	(1,1)	BLK,y	DM	DM	[1,2]	(1,1)	BLK,y						(1,1)	BLK,y	DM	[1,2]
50	(1,2)	BLK,x	DM	DM	[1,2]	(1,2)	BLK,x						(1,2)	BLK,x	DM	[1,2]
1	(1,3)	BLK,z	DM	DM	[1,2]	(1,3)	BLK,z						(1,3)	BLK,z	DM	[1,2]
2	(5,2)	DEV,x	SM	SM	[1,2]	(5,2)	DEV,x						(5,2)	DEV,x	RM	[1,2]
1	(5,3)	DEV,y	SM	SM	[1,2]	(5,3)	DEV,y						(5,3)	DEV,y	RM	[1,2]
8	(3,1)	DEV,z	DM	DM	[1,2]	(3,1)	DEV,z						(3,1)	DEV,z	RM	[1,2]
4	(2,1)	THR,y	DM	DM	[1,2]	(2,1)	THR,y						(2,1)	THR,y	DM	[1,2]
20	(2,2)	THR,x	DM	DM	[1,2]	(2,2)	THR,x	RM	RM	[1,2]	RM	[1,2]	(2,2)	THR,x	DM	[1,2]
1	(2,3)	THR,z	DM	DM	[1,2]	(2,3)	THR,z						(2,3)	THR,z	DM	[1,2]
1	(3,3)	SHR,x	SM	SM	[1,2]	(3,3)	SHR,x						(3,3)	SHR,x	RM	[1,2]
1	(4,1)	SHR,y	SM	SM	[1,2]	(4,1)	SHR,y						(4,1)	SHR,y	RM	[1,2]
128	(3,2)	SHR,z	SM	SM	[1,2]	(3,2)	SHR,z						(3,2)	SHR,z	RM	[1,2]
1	(4,3)	REG,x	SM	SM	[1,2]	(4,3)	REG,x						(4,3)	REG,x	RM	[1,2]
1	(5,1)	REG,y	SM	SM	[1,2]	(5,1)	REG,y						(5,1)	REG,y	RM	[1,2]
2	(4,2)	REG,z	SM	SM	[1,2]	(4,2)	REG,z						(4,2)	REG,z	RM	[1,2]

Figure 78: TVM's (de/re)-composition for `MatMul` in CUDA on GPU expressed in our low-level representation.

TVM's (de/re)-composition for MatMul in OpenCL on Intel Skylake CPU

	De-Comp. Phase					*Scalar Phase*							*Re-Comp. Phase*			
0	**D1**	**D2**	**D3**		**D4**	**S1**	**S2**	**S3**		**S4**	**S5**	**S6**	**R1**	**R2**	**R3**	**R4**
			A	**B**	**A,B**			**A**	**B**	**A,B**	**C**	**C**			**C**	**C**
1	(1,1)	WG,1	GM	GM	[1,2]	(1,1)	WG,1						(1,1)	WG,1	GM	[1,2]
125	(1,2)	WG,0	GM	GM	[1,2]	(2,1)	WG,0						(2,1)	WG,0	GM	[1,2]
1	(1,3)	WG,2	GM	GM	[1,2]	(3,1)	WG,2						(3,1)	WG,2	GM	[1,2]
1	(5,2)	GLB,0	LM	LM	[1,2]	(5,2)	GLB,0						(5,2)	GLB,0	PM	[1,2]
1	(5,3)	GLB,1	LM	LM	[1,2]	(5,3)	GLB,1						(5,3)	GLB,1	PM	[1,2]
16	(3,1)	GLB,2	GM	GM	[1,2]	(3,1)	GLB,2						(3,1)	GLB,2	PM	[1,2]
16	(2,1)	WI,1	GM	GM	[1,2]	(2,1)	WI,1						(2,1)	WI,1	GM	[1,2]
1	(2,2)	WI,0	GM	GM	[1,2]	(2,2)	WI,0	PM	PM	[1,2]	PM	[1,2]	(2,2)	WI,0	GM	[1,2]
1	(2,3)	WI,2	GM	GM	[1,2]	(2,3)	WI,2						(2,3)	WI,2	GM	[1,2]
1	(3,3)	LCL,0	LM	LM	[1,2]	(3,3)	LCL,0						(3,3)	LCL,0	PM	[1,2]
1	(4,1)	LCL,1	LM	LM	[1,2]	(4,1)	LCL,1						(4,1)	LCL,1	PM	[1,2]
128	(3,2)	LCL,2	LM	LM	[1,2]	(3,2)	LCL,2						(3,2)	LCL,2	PM	[1,2]
1	(4,3)	PRV,0	LM	LM	[1,2]	(4,3)	PRV,0						(4,3)	PRV,0	PM	[1,2]
8	(5,1)	PRV,1	LM	LM	[1,2]	(5,1)	PRV,1						(5,1)	PRV,1	PM	[1,2]
1	(4,2)	PRV,2	LM	LM	[1,2]	(4,2)	PRV,2						(4,2)	PRV,2	PM	[1,2]

Figure 79: TVM's (de/re)-composition for `MatMul` in OpenCL on CPU expressed in our low-level representation.

PPCG's (de/re)-composition for MatMul in CUDA on NVIDIA Ampere GPU

	De-Comp. Phase					*Scalar Phase*							*Re-Comp. Phase*				
0	**D1**	**D2**	**D3**		**D4**	**S1**	**S2**	**S3**		**S4**	**S5**	**S6**	**R1**	**R2**	**R3**	**R4**	
			A	**B**	**A,B**			**A**	**B**	**A,B**	**C**	**C**			**C**	**C**	
16	(1,1)	BLK,y	DM	DM	[1,2]	(1,1)	BLK,y						(1,1)	BLK,y	DM	[1,2]	
8	(1,2)	BLK,x	DM	DM	[1,2]	(1,2)	BLK,x						(1,2)	BLK,x	DM	[1,2]	
1	(1,3)	BLK,z	DM	DM	[1,2]	(1,3)	BLK,z						(1,3)	BLK,z	DM	[1,2]	
1	(2,1)	DEV,x	DM	DM	[1,2]	(2,1)	DEV,x						(2,1)	DEV,x	DM	[1,2]	
1	(2,2)	DEV,y	DM	DM	[1,2]	(2,2)	DEV,y						(2,2)	DEV,y	DM	[1,2]	
1	(2,3)	DEV,z	DM	DM	[1,2]	(2,3)	DEV,z						(2,3)	DEV,z	DM	[1,2]	
1	(3,1)	THR,y	DM	DM	[1,2]	(3,1)	THR,y						(3,1)	THR,y	RM	[1,2]	
125	(3,2)	THR,x	DM	DM	[1,2]	(3,2)	THR,x	RM	RM	[1,2]	RM	[1,2]	(3,2)	THR,x	RM	[1,2]	
1	(3,3)	THR,z	DM	DM	[1,2]	(3,3)	THR,z						(3,3)	THR,z	RM	[1,2]	
1	(4,1)	SHR,x	SM	DM	[1,2]	(4,1)	SHR,x						(4,1)	SHR,x	RM	[1,2]	
1	(4,2)	SHR,y	SM	DM	[1,2]	(4,2)	SHR,y						(4,2)	SHR,y	RM	[1,2]	
1181	(4,3)	SHR,z	SM	DM	[1,2]	(4,3)	SHR,z						(4,3)	SHR,z	RM	[1,2]	
1	(5,1)	REG,x	SM	DM	[1,2]	(5,1)	REG,x						(5,1)	REG,x	RM	[1,2]	
1	(5,2)	REG,y	SM	DM	[1,2]	(5,2)	REG,y						(5,2)	REG,y	RM	[1,2]	
1	(5,3)	REG,z	SM	DM	[1,2]	(5,3)	REG,z						(5,3)	REG,z	RM	[1,2]	

Figure 80: PPCG's (de/re)-composition for `MatMul` in CUDA on GPU expressed in our low-level representation.

Pluto's (de/re)-composition for MatMul in OpenMP on Intel Skylake CPU

	De-Comp. Phase					*Scalar Phase*							*Re-Comp. Phase*				
0	**D1**	**D2**	**D3**		**D4**	**S1**	**S2**	**S3**		**S4**	**S5**	**S6**	**R1**	**R2**	**R3**	**R4**	
			A	**B**	**A,B**			**A**	**B**	**A,B**	**C**	**C**			**C**	**C**	
8	(1,1)	COR,0	MM	MM	[1,2]	(1,1)	COR,0						(1,1)	COR,0	MM	[1,2]	
1	(1,2)	COR,1	MM	MM	[1,2]	(1,2)	COR,1						(1,2)	COR,1	MM	[1,2]	
1	(1,3)	COR,2	MM	MM	[1,2]	(1,3)	COR,2						(1,3)	COR,2	MM	[1,2]	
1	(2,1)	MM,0	MM	MM	[1,2]	(2,1)	MM,0						(2,1)	MM,0	MM	[1,2]	
1	(2,2)	MM,1	MM	MM	[1,2]	(2,2)	MM,1						(2,2)	MM,1	MM	[1,2]	
9	(2,3)	MM,2	MM	MM	[1,2]	(2,3)	MM,2						(2,3)	MM,2	MM	[1,2]	
2	(3,1)	L2,0	L2	L2	[1,2]	(3,1)	L2,0						(3,1)	L2,0	L2	[1,2]	
962	(3,2)	L2,1	L2	L2	[1,2]	(3,2)	L2,1	L1	L1	[1,2]	L1	[1,2]	(3,2)	L2,1	L2	[1,2]	
218	(3,3)	L2,2	L2	L2	[1,2]	(3,3)	L2,2						(3,3)	L2,2	L2	[1,2]	
1	(4,1)	L1,0	L1	L1	[1,2]	(4,1)	L1,0						(4,1)	L1,0	L1	[1,2]	
1	(4,2)	L1,1	L1	L1	[1,2]	(4,2)	L1,1						(4,2)	L1,1	L1	[1,2]	
1	(4,3)	L1,2	L1	L1	[1,2]	(4,3)	L1,2						(4,3)	L1,2	L1	[1,2]	

Figure 81: Pluto's (de/re)-composition for `MatMul` in OpenMP on CPU expressed in our low-level representation.

.5 ADDENDUM SECTION 4.3

.5.1 *Constraints of Programming Models*

Constraints of programming models can be expressed in our formalism; we demonstrate this using the example models CUDA and OpenCL. For this, we add to the general, model-unspecific constraints (described in Section .4.4) the new, model-specific constraints for CUDA (in Table 9 or Table 10) or for OpenCL (in Table 11), respectively.

For brevity, we use in the following:

$$(\overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{l}_{\mathsf{ASM}}, \overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{d}_{\mathsf{ASM}}) := \leftrightarrow^{-1}_{\bullet\text{-ass}} (\mathsf{l}_{\mathsf{ASM}}, \mathsf{d}_{\mathsf{ASM}}) \ , \quad \bullet \in \{\downarrow, \mathsf{f}, \uparrow\}$$

In Tables 9 and 10 for CUDA, the constraint No. `0` (which constrains tuning parameter No. `0` in Table 8) limits the number of cuda cores (`CC`) to 1024, according to the CUDA specification [38]. The constraints on tuning parameter `R3` specify that the results of `SMX` can be combined in device memory (`DM`) only in CUDA, and the results of `CC`s/`WRP`s in only device memory (`DM`) or shared memory (`SM`). Note that in the case of Table 10, `CC`s are not constrained in parameter 14, as `CC`s within a `WRP` have access to all CUDA memory regions: `DM`, `SM`, as well as `RM` (via warp shuffles [129]).

No.	Constraint	
`0`	$\prod_{d \in [1,D]_N} \#\mathsf{PRT}(\overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{CC}, \overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{d}) \leq 1024$	(Number of CCs limited)
`R3`	$\#\mathsf{PRT}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{BLK}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \Rightarrow \uparrow\text{-}\mathsf{mem}^{\langle ob \rangle}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{BLK}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) \in \{\mathsf{DM}\}$	(SMXs combine in DM)
	$\#\mathsf{PRT}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{CC}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \Rightarrow \uparrow\text{-}\mathsf{mem}^{\langle ob \rangle}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{CC}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) \in \{\mathsf{DM}, \mathsf{SM}\}$	(CCs combine in DM/SM)

Table 9: CUDA model constraints on tuning parameters.

No.	Constraint	
`0`	$\prod_{d \in [1,D]_N} \#\mathsf{PRT}(\overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{CC}, \overset{\bullet\text{-MDH}}{\leftarrow}\mathsf{d}) \leq 1024$	(Number of CCs limited)
`R3`	$\#\mathsf{PRT}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{BLK}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \Rightarrow \uparrow\text{-}\mathsf{mem}^{\langle ob \rangle}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{BLK}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) \in \{\mathsf{DM}\}$	(SMXs combine in DM)
	$\#\mathsf{PRT}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{WRP}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}} \Rightarrow \uparrow\text{-}\mathsf{mem}^{\langle ob \rangle}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{WRP}, \overset{\uparrow\text{-MDH}}{\leftarrow}\mathsf{d}) \in \{\mathsf{DM}, \mathsf{SM}\}$	(WRPs combine in DM/SM)

Table 10: CUDA+WRP model constraints on tuning parameters.

In Table 11 for OpenCL, the constraints are similar to the CUDA's constraints in Tables 9: they limit the number of `PE`s to $\mathsf{C}_{\mathsf{DEV}}$ (which is a device-specific constant in OpenCL), and the constraints specify the valid memory regions for combining the results of cores, according to the OpenCL specification [30].

No.	Constraint	
0	$\prod_{d \in [1,D]_{\mathbb{N}}} \#PRT(\overset{\bullet\text{-MDH}}{\leftarrow}\mathtt{WI}, \mathtt{d}\overset{\bullet\text{-MDH}}{\leftarrow}) \leq C_{DEV}$	(Number of PEs limited)
R3	$\#PRT(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{WG}, \mathtt{d}\overset{\uparrow\text{-MDH}}{\leftarrow}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{d}} \Rightarrow \uparrow\text{-}\mathtt{mem}^{<ob>}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{WG}, \mathtt{d}\overset{\uparrow\text{-MDH}}{\leftarrow}) \in \{\mathtt{GM}\}$	(CUs combine in GM)
	$\#PRT(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{WI}, \mathtt{d}\overset{\uparrow\text{-MDH}}{\leftarrow}) > 1 \wedge \circledast_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{d}} \neq +\!\!+_{\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{d}} \Rightarrow \uparrow\text{-}\mathtt{mem}^{<ob>}(\overset{\uparrow\text{-MDH}}{\leftarrow}\mathtt{WI}, \mathtt{d}\overset{\uparrow\text{-MDH}}{\leftarrow}) \in \{\mathtt{GM}, \mathtt{LM}\}$	(PEs combine in GM/LM)

Table 11: OpenCL model constraints on tuning parameters.

Note that the tables present some important example constraints only and are not complete: for example, CUDA and OpenCL devices are also constrained regarding their memory sizes (shared/private memory), which is not considered in the tables for brevity.

.5.2 *Inverse Concatenation*

Definition 43 (Inverse Concatenation)**.** The inverse of operator *concatenation* (Example 13) is function $+\!\!+^{-1}$ which is of type

$$+\!\!+^{-1<T \in \mathtt{TYPE}\,|\,D \in \mathbb{N}\,|\,d \in [1,D]_{\mathbb{N}}\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D) \in \mathtt{MDA\text{-}IDX\text{-}SETs}^{D-1},\ (P,Q) \in \mathtt{MDA\text{-}IDX\text{-}SETs} \dot{\times} \mathtt{MDA\text{-}IDX\text{-}SETs}>} :$$

$$T[\,I_1,\ldots,\underbrace{id(P \uplus Q)}_{\uparrow d},\ldots,I_D\,] \rightarrow$$

$$T[\,I_1,\ldots,\underbrace{id(P)}_{\uparrow d},\ldots,I_D\,] \times T[\,I_1,\ldots,\underbrace{id(Q)}_{\uparrow d},\ldots,I_D\,]$$

where $id : \mathtt{MDA\text{-}IDX\text{-}SETs} \rightarrow \mathtt{MDA\text{-}IDX\text{-}SETs}$ is the identity function on MDA index sets. The function is computed as:

$$+\!\!+^{-1<T\,|\,D\,|\,d\,|\,(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D),(P,Q)>}(a) := (a_1, a_2)$$

for

$$a_1[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] := a[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,],\ i_d \in P$$

and

$$a_2[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] := a[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,],\ i_d \in Q$$

i.e., a_1 and a_2 behave exactly as MDA a on their restricted index sets P or Q, respectively.

We often write for $(a_1, a_2) := +\!\!+^{-1<\ldots>}(a)$ (meta-parameters omitted via ellipsis) also

$$a =: a_1 +\!\!+^{<\ldots>} a_2$$

Our notation is justified by the fact that the inverse of MDA a is uniquely determined, as the two MDAs a_1 and a_2 which are equal to MDA a when concatenating them.

.5.3 *Example 75 in Verbose Math Notation*

Figures 82-84 show our low-level representation from Example 75 in verbose math notation. The symbols $\blacksquare_\perp, \ldots, \blacksquare_f$ used in the figures are a textual abbreviation for:

$$
\begin{array}{lllll}
\blacksquare_\perp & := & *,* & | \quad *,* & | \quad *,* \\
\blacksquare_1^1 & := & *,* & | \quad *,* & | \quad *,* \\
\blacksquare_2^1 & := & p_1^1,* & | \quad *,* & | \quad *,* \\
\blacksquare_1^2 & := & p_1^1,p_2^1 & | \quad *,* & | \quad *,* \\
\blacksquare_2^2 & := & p_1^1,p_2^1 & | \quad p_1^2,* & | \quad *,* \\
\blacksquare_1^3 & := & p_1^1,p_2^1 & | \quad p_1^2,p_2^2 & | \quad *,* \\
\blacksquare_2^3 & := & p_1^1,p_2^1 & | \quad p_1^2,p_2^2 & | \quad p_1^3,* \\
\blacksquare_f & := & p_1^1,p_2^1 & | \quad p_1^2,p_2^2 & | \quad p_1^3,p_2^3
\end{array}
$$

where symbol $*$ indicates generalization in meta-parameters (Definition 23).

In Example 75, the arrow annotation of combine operators is formally an abbreviation. For example, operator $+\!\!+_2^{(\texttt{COR},y)}$ in Figure 75 is annotated with $\rightarrow$ `M: HM[1,2],v: HM[1]` which abbreviates

$$
\ldots \quad {}^{\downarrow}a_2^{2<p_1^1,p_2^1 \mid p_1^2,p_2^2:=* \mid p_1^3:=*,p_2^3:=*>} =: \quad \underset{p_2^2 \in [0,16)_{\mathbb{N}_0}}{+\!\!+_2^{(\texttt{COR},y)}} \quad \ldots
$$

Here, ${}^{\downarrow}a_2^2$ represents the low-level MDA (Definition 38) that is already partitioned for layer 1 in dimensions 1 and 2, and for layer 2 in dimension 1 (because in Figure 75, operators $+\!\!+_1^{(\texttt{HM},x)}$, $+\!\!+_2^{(\texttt{HM},y)}$, $+\!\!+_1^{(\texttt{COR},x)}$ appear before operator $+\!\!+_2^{(\texttt{COR},y)}$), but not yet for layer 2 in dimension 2 as well as for layer 3 in both dimensions (indicated by symbol $*$ which is described formally in Definition 23 of our Appendix). In our generated code (discussed in Section .8 of our Appendix), we store low-level MDAs, e.g., ${}^{\downarrow}a_2^2$, using their domain-specific data representation, as the domain-specific representation is usually more efficient: in the case of `MatVec`, we physically store matrix M and vector v for the input MDA, and vector w for the output MDA. For example, low-level MDA

$${}^{\downarrow}a_2^{2<p_1^1,p_2^1 \mid p_1^2,p_2^2:=* \mid p_1^3:=*,p_2^3:=*>}$$

can be transformed via view functions (Definitions 32 and 34) to *low-level BUFs* (Definition 39)

$$
M_2^{2<\texttt{HM}\mid id><p_1^1,p_2^1 \mid p_1^2,p_2^2:=* \mid p_1^3:=*,p_2^3:=*>}, \quad v_2^{2<\texttt{HM}\mid id><p_1^1,p_2^1 \mid p_1^2,p_2^2:=* \mid p_1^3:=*,p_2^3:=*>}
$$

and back (Lemma 4). Similarly as for data structures in low-level programming models (e.g., *C arrays* as in OpenMP, CUDA, and OpenCL),

low-level BUFs are defined to have an explicit notion of memory regions and memory layouts.

In Figure 82, we de-compose the input MDA ${}^{\downarrow}a$, step by step, for the MDH levels $(1,1),\ldots,(3,2)$:

$$
\begin{aligned}
{}^{\downarrow}a \;=:\; & {}^{\downarrow}a_{\perp}^{<\blacksquare_{\perp}>} \rightarrow \\
& {}^{\downarrow}a_1^{1\,<\blacksquare_1^1>} \rightarrow {}^{\downarrow}a_2^{1\,<\blacksquare_2^1>} \rightarrow {}^{\downarrow}a_1^{2\,<\blacksquare_1^2>} \rightarrow {}^{\downarrow}a_2^{2\,<\blacksquare_2^2>} \rightarrow {}^{\downarrow}a_1^{3\,<\blacksquare_1^3>} \rightarrow {}^{\downarrow}a_2^{3\,<\blacksquare_2^3>} \rightarrow \\
& {}^{\downarrow}a_f^{<\blacksquare_f>}
\end{aligned}
$$

The input MDAs $({}^{\downarrow}a_d^l)_{l\in[1,3]_{\mathbb{N}},d\in[1,2]_{\mathbb{N}}}$, as well as ${}^{\downarrow}a_{\perp}$ and ${}^{\downarrow}a_f$, are all low-level MDA representations (Definition 38). We use as partitioning schema P (Definition 38)

$$P := ((P_1^1,P_2^1),\,(P_1^2,P_2^2),\,(P_1^3,P_2^3)) = ((2,4),\,(8,16),\,(32,64))$$

and we use the index sets I_d from Definition 44 (which define a uniform index set partitioning):

$$
\Big(\;\underset{{+\!\!+}_d}{\overset{d}{\Rightarrow}}{}^{\mathtt{MDA}}_{\mathtt{MDA}}\big(I_d^{<p_1^1,p_2^1\,|\,p_1^2,p_2^2\,|\,p_1^3,p_2^3>}\big)^{<(p_1^1,p_2^1)\in P_1^1\times P_2^1\,|\,(p_1^2,p_2^2)\in P_1^2\times P_2^2\,|\,(p_1^3,p_2^3)\in P_1^3\times P_2^3>}\;\Big)_{d\in[1,D]_{\mathbb{N}}}
$$

Here, $\underset{{+\!\!+}_d}{\overset{d}{\Rightarrow}}{}^{\mathtt{MDA}}_{\mathtt{MDA}}$ denotes the index set function of combine operator concatenation (Example 13), which is the identity function and explicitly stated for the sake of completeness only. Note that in Figure 82, we access low-level MDAs ${}^{\downarrow}a_d^l$ as generalized in some partition sizes, via $*$ (Definition 23), according to the definitions of the $\blacksquare_d^l$.

Each MDA ${}^{\downarrow}a$ can be transformed to its domain-specific data representation matrix ${}^{\downarrow}M$ and vector ${}^{\downarrow}v$ and vice versa, using the view functions, as discussed above.

Figure 83 shows our scalar phase, which is formally trivial.

In Figure 84, we re-compose the computed data ${}^{\uparrow}a_f^{<\blacksquare_f>}$, step by step, to the final result ${}^{\uparrow}a$:

$$
\begin{aligned}
& & & & & & & & & {}^{\uparrow}a_f^{<\blacksquare_f>} \rightarrow \\
{}^{\uparrow}a_1^{1\,<\blacksquare_1^1>} & \rightarrow {}^{\uparrow}a_2^{1\,<\blacksquare_2^1>} \rightarrow {}^{\uparrow}a_2^{2\,<\blacksquare_2^2>} \rightarrow {}^{\uparrow}a_1^{3\,<\blacksquare_1^3>} \rightarrow {}^{\uparrow}a_2^{3\,<\blacksquare_2^3>} \rightarrow \\
& & & & & & & & & {}^{\uparrow}a_{\perp}^{<\blacksquare_{\perp}>} \;=:\; {}^{\uparrow}a
\end{aligned}
$$

Analogously to the de-composition phase, each output MDA $({}^{\uparrow}a_d^l)_{l\in[1,3]_{\mathbb{N}},d\in[1,2]_{\mathbb{N}}}$, as well as ${}^{\uparrow}a_f$ and ${}^{\uparrow}a_{\perp}$, are low-level MDA representations, for P as defined above and index sets

$$
\Big(\;\underset{\circledast_d}{\overset{d}{\Rightarrow}}{}^{\mathtt{MDA}}_{\mathtt{MDA}}\big(I_d^{<p_1^1,p_2^1\,|\,p_1^2,p_2^2\,|\,p_1^3,p_2^3>}\big)^{<(p_1^1,p_2^1)\in P_1^1\times P_2^1\,|\,(p_1^2,p_2^2)\in P_1^2\times P_2^2\,|\,(p_1^3,p_2^3)\in P_1^3\times P_2^3>}\;\Big)_{d\in[1,D]_{\mathbb{N}}}
$$

where $\overset{\mathrm{d}}{\underset{\circledast_{\mathrm{d}}}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}$ are the index set functions of the combine operators (Definition 26) used in the re-composition phase. The same as in the decomposition phase, we access the output low-level MDAs as generalized in some partition sizes, according to our definitions of the $\blacksquare_{\mathrm{d}}^{\mathrm{l}}$, and we identify each MDA with its domain-specific data representation (the output vector w).

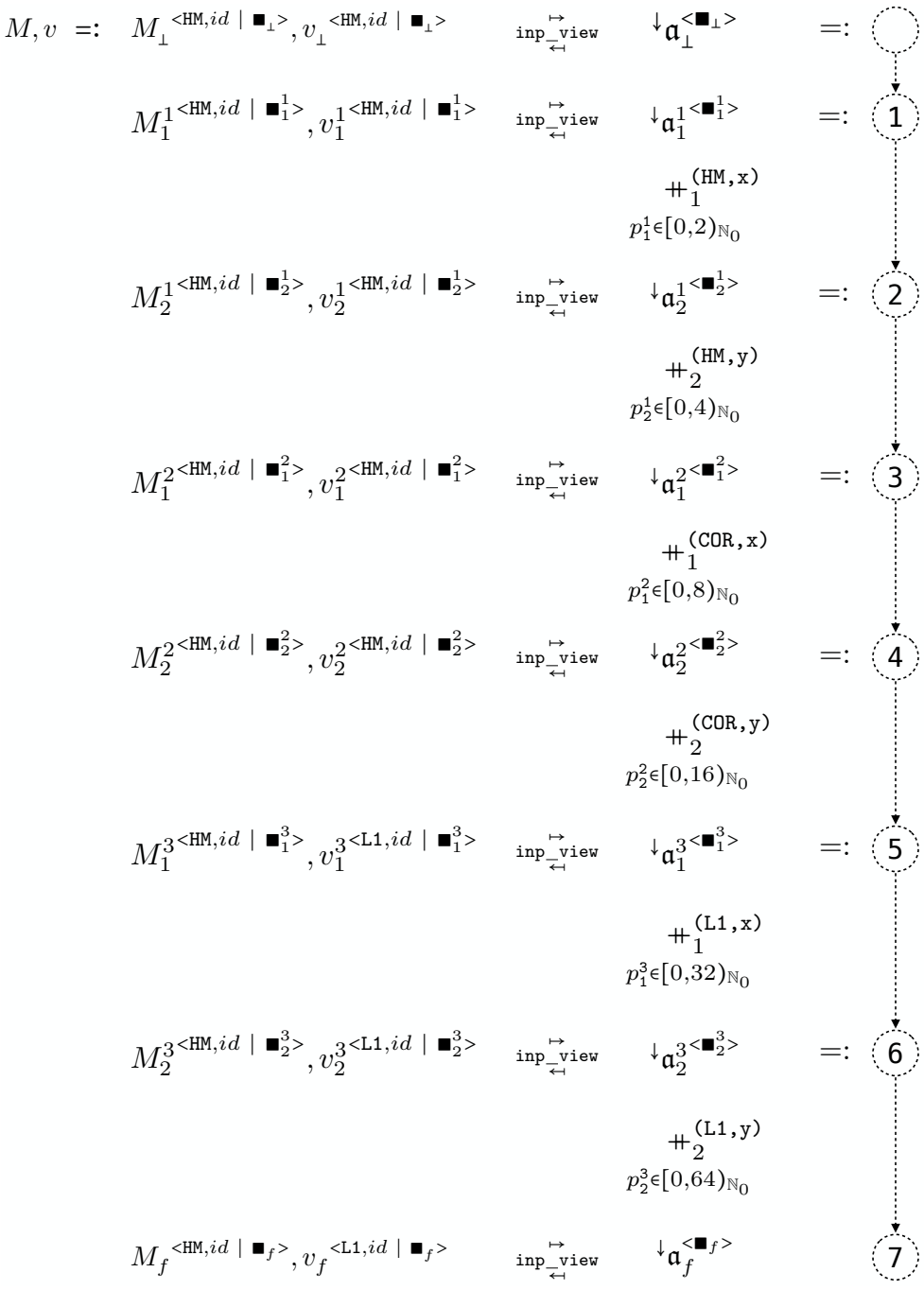
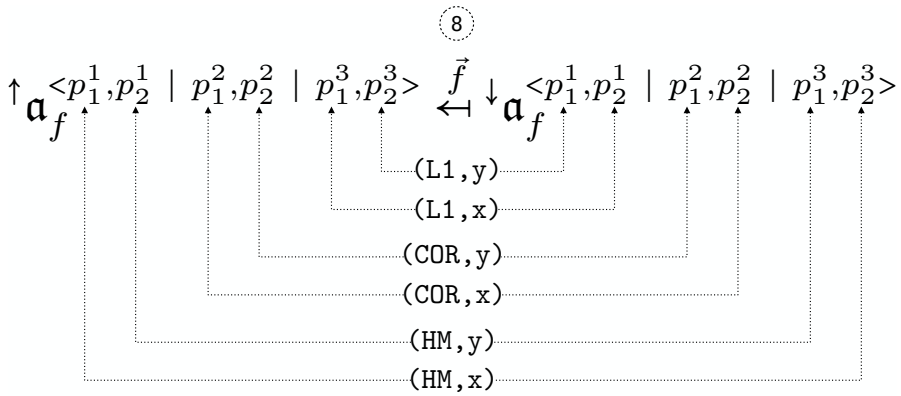


Figure 82: De-composition phase of Example 75 in verbose math notation.



$$\text{for all:} \quad p_1^1 \in P_1^1 \ , \ p_2^1 \in P_2^1 \ ,$$
$$p_1^2 \in P_1^2 \ , \ p_2^2 \in P_2^2 \ ,$$
$$p_1^3 \in P_1^3 \ , \ p_2^3 \in P_2^3$$

Figure 83: Scalar phase of Example 75 in verbose math notation.

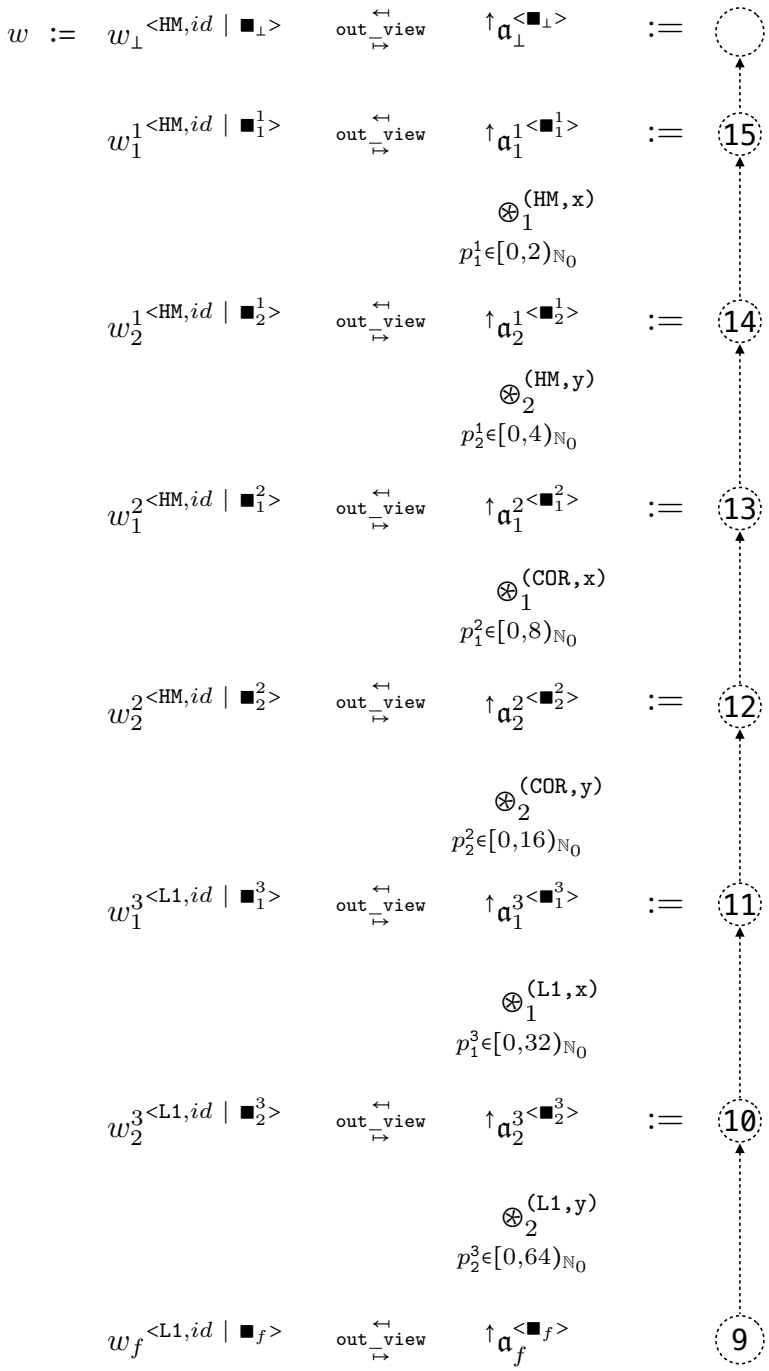


Figure 84: Re-composition phase of Example 75 in verbose math notation.

.5.4 *Counting Memory and Core Layers*

Figure 85 shows how we count the number of layers in memory and core hierarchies. While we represent memory hierarchies using a brick-like representation (left part of the figure), where counting starts from 1, we represent core hierarchies as trees (right part of the figure), where the root layer is numbered 0.

Note that for simplicity, Figure 4 depicts our brick- and tree-like hierarchy representations (as in Figure 85) in a more compact style, and Examples 11 and 26 represent the numbers of hierarchy layers only.

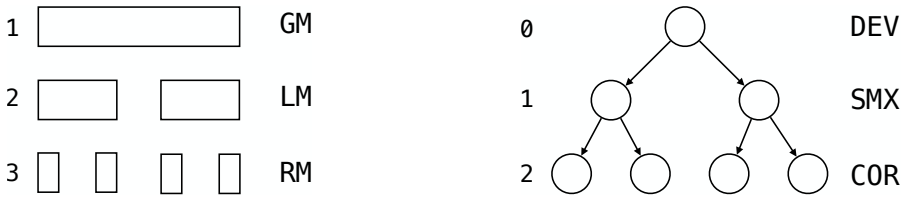


Figure 85: Counting the number of layers in memory and core hierarchies.

.5.5 *Multi-Dimensional ASM Arrangements*

We demonstrate how we arrange memory regions and cores of ASM-represented systems (Section .4.2) in multiple dimensions using the example of CUDA.

CORES (COR): In CUDA, `SMX` cores are programmed via so-called *CUDA Blocks*, and CUDA's `CC` cores are programmed via *CUDA Threads*. CUDA has native support for arranging its blocks and threads in up to three dimensions which are called `x`, `y`, and `z` in CUDA [37]. Consequently, even though the original CUDA specification [38] introduces `SMX` and `CC` without having an order, the CUDA programmer benefits from imagining `SMX` and `CC` as three-dimensionally arranged.

Additional dimensions can be explicitly programmed in CUDA. For example, to add a fourth dimension to CUDA, we can embed the additional dimension in the CUDA's `z` dimension, thereby splitting CUDA dimension `z` in the explicitly programmed dimensions `z_1` (third dimension) and `z_2` (fourth dimension), as follows:

```
z_1 := z % Z_1 and z_2 := z / Z_1
```

Here, `Z_1` represents the number of threads in the additional dimension, and symbol `%` denotes the modulo operator.

MEMORY (MEM): In CUDA, memory is managed via *C arrays* which may be multi-dimensional: to arrange (`DIM_1` × ... × `DIM_D`)-many memory regions, each of size `N`, we use a CUDA array of the following type (pseudocode):

```
array[ DIM_1 ]...[ DIM_D ][ N ]
```

Note that CUDA implicitly arranges its *shared* and *private* memory allocations in multiple dimensions, depending on the number of blocks and threads: a shared memory array of type `shared_array[ DIM_1 ]...[ DIM_D ][ N ]` is internally managed in CUDA as `shared_array[ blockIdx.x ][ blockIdx.y ][ blockIdx.z ][ DIM_1 ]...[ DIM_D ][ N ]`, i.e., each CUDA block has its own shared memory region. Analogously, a private memory array `private_array[ DIM_1 ]...[ DIM_D ][ N ]` is managed in CUDA as `private_array[ blockIdx.x ][ blockIdx.y ][ blockIdx.z ][ threadIdx.x ][ threadIdx.y ][ threadIdx.z ][ DIM_1 ]...[ DIM_D ][ N ]`, correspondingly. Our arrangement methodology continues the CUDA's approach by explicitly programming the additional arrangement dimensions `DIM_1,...,DIM_D`.

Figure 86 illustrates our multi-dimensional core and memory arrangement using the example of CUDA, for D = 2 (two-dimensional arrangement).

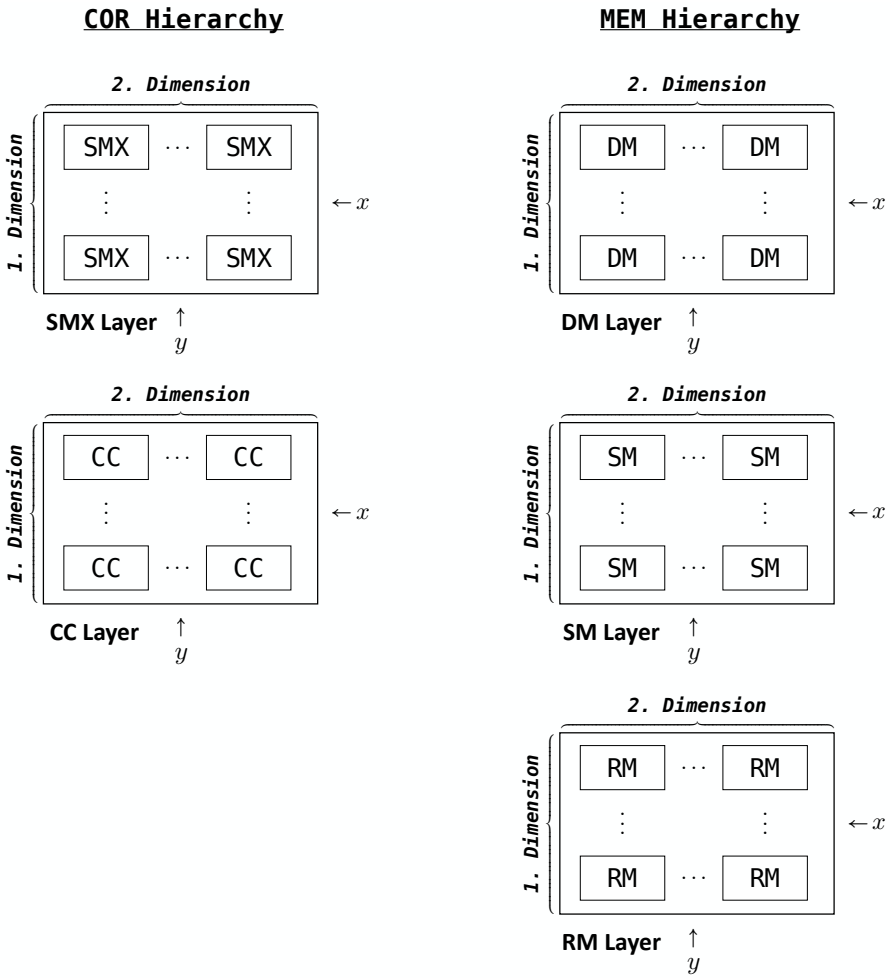


Figure 86: Multi-dimensional ASM arrangement illustrated using CUDA for the case D = 2 (two dimensions).

.5.6 *ASM Levels*

ASM levels are pairs (l_{ASM}, d_{ASM}) consisting of an ASM layer $l_{ASM} \in \mathbb{N}$ and ASM dimension $d_{ASM} \in \mathbb{N}$.

Figure 87 illustrates ASM levels using the example of CUDA's thread hierarchy. The figure shows that thread hierarchies can be considered as a tree in which each level is uniquely determined by a particular combination of a layer (`block` or `thread` in the case of CUDA) and dimension (`x`, `y`, or `z`). In the figure, we use `lvl` as an abbreviation for *level*, `l` for *layer*, and `d` for *dimension*.

For ASM layers and dimensions, we usually use their domain-specific identifiers, e.g., `BLK/CC` and `x/y/z` as aliases for numerical values of layers and dimensions.

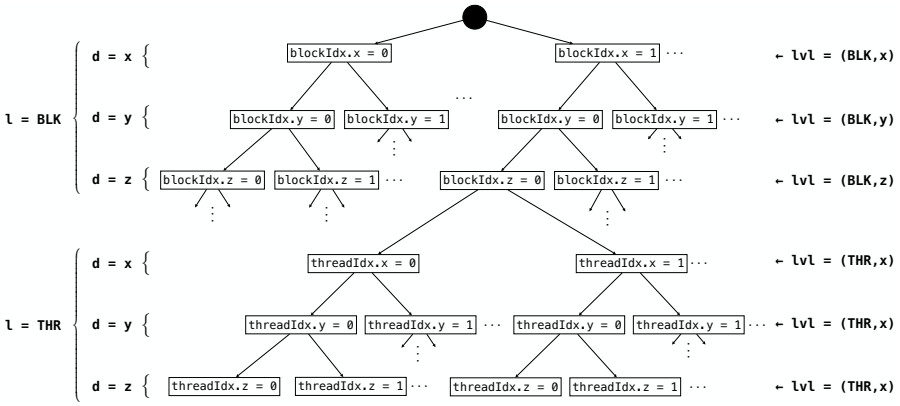


Figure 87: ASM levels illustrated using CUDA's thread hierarchy.

.5.7 *MDH Levels*

MDH levels are pairs (l_{MDH}, d_{MDH}) consisting of an MDH layer $l_{MDH} \in \mathbb{N}$ and MDH dimension $d_{MDH} \in \mathbb{N}$.

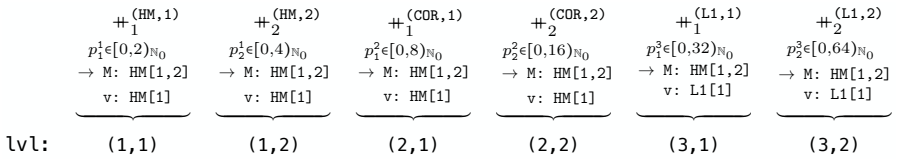


Figure 88: MDH levels illustrated using as example the de-composition phase in Figure 75.

Figure 88 illustrates MDH levels using as example the de-composition phase in Figure 75. The levels (l_{MDH}, d_{MDH}) can be derived from the super- and subscripts of combine operators' variables $p_{d_{MDH}}^{l_{MDH}}$.

.5.8 *MDA Partitioning*

We demonstrate how we partition MDAs into equally sized parts (a.k.a. *uniform partitioning*).

Definition 44 (MDA Partitioning)**.** Let $a \in T[I_1,\ldots,I_D]$ be an arbitrary MDA that has scalar type $T \in \texttt{TYPE}$, dimensionality $D \in \mathbb{N}$, index sets $I = (I_1,\ldots,I_D) \in \texttt{MDA-IDX-SETs}^D$, and size $N = \{|I_1|,\ldots,|I_D|\} \in \mathbb{N}^D$. We consider $I_d = \{i_1^d,\ldots,i_{N_d}^d\}$, $d \in [1,D]_{\mathbb{N}}$, such that $i_1^d < \ldots < i_{N_d}^d$ represents a sorted enumeration of the elements in I_d. Let further $P = ((P_1^1,\ldots,P_D^1),\ldots,(P_1^L,\ldots,P_D^L))$ be an arbitrary tuple of L-many D-tuples of positive natural numbers such that $\prod_{l\in[1,L]_{\mathbb{N}}} P_d^l$ divides N_d (the number of indices of MDA a in dimension d), for each $d \in \{1,\ldots,D\}$.

The L-*layered,* D-*dimensional,* P-*partitioning of MDA* a is the L-layered, D-dimensional, P-partitioned low-level MDA $a_{\texttt{prt}}$ (Definition 38) that has scalar type T and index sets

$$I_d^{<p_d^1,\ldots,p_d^L>} := \{ i_j \in I_d \mid j = OS + j',\ \text{for}\ \ OS := \sum_{l\in[1,L]_{\mathbb{N}}} p_d^l * \frac{N_d}{\prod_{l'\in[1,l]_{\mathbb{N}}} P_d^{l'}} \quad \text{and}\ \ j' \in PS := \frac{N_d}{\prod_{l'\in[1,L]_{\mathbb{N}}} P_d^{l'}} \ \}$$

i.e., set $I_d^{<p_d^1,\ldots,p_d^L>}$ denotes for each choice of parameters $p_d^1,\ldots,p_d^L$ a part of the uniform partitioning of the ordered index set I_d (OS in the formula above represents the OffSet to the part, and PS the Part's Size). The partitioned MDA $a_{\texttt{prt}}$ is defined as:

$$a =: \underbrace{\underset{p_1^1\in P_1^1}{+\!\!+_1} \ldots \underset{p_D^1\in P_D^1}{+\!\!+_D}}_{\text{Layer 1}} \ \ldots \ \underbrace{\underset{p_1^L\in P_1^L}{+\!\!+_1} \ldots \underset{p_D^L\in P_D^L}{+\!\!+_D}}_{\text{Layer L}} \ a_{\texttt{prt}}^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}$$

i.e., the parts $a_{\texttt{prt}}^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}$ are defined such that concatenating them results in the original MDA a.

.5.9 *TVM Schedule for* `MatMul`

Listing 46 shows TVM's Ansor-generated schedule program for `MatMul` on input matrices of sizes 16×2048 and 2048×1000 taken from ResNet-50's training phase, discussed in Section .4.5. Code formatting, such as names of variables and comments, have been shortened and adapted in the listing for brevity.

```
# exploiting fast memory resources for computed results
matmul_local, = s.cache_write([matmul], "local")
matmul_1, matmul_2, matmul_3 = tuple(matmul_local.op.axis) + tuple(
    matmul_local.op.reduce_axis)
SHR_1, REG_1 = s[matmul_local].split(matmul_1, factor=1)
THR_1, SHR_1 = s[matmul_local].split(SHR_1, factor=1)
DEV_1, THR_1 = s[matmul_local].split(THR_1, factor=4)
BLK_1, DEV_1 = s[matmul_local].split(DEV_1, factor=2)
SHR_2, REG_2 = s[matmul_local].split(matmul_2, factor=1)
THR_2, SHR_2 = s[matmul_local].split(SHR_2, factor=1)
DEV_2, THR_2 = s[matmul_local].split(THR_2, factor=20)
BLK_2, DEV_2 = s[matmul_local].split(DEV_2, factor=1)
SHR_3, REG_3 = s[matmul_local].split(matmul_3, factor=2)
DEV_3, SHR_3 = s[matmul_local].split(SHR_3, factor=128)
s[matmul_local].reorder(BLK_1, BLK_2, DEV_1, DEV_2, THR_1, THR_2,
    DEV_3, SHR_3, SHR_1, SHR_2, REG_3, REG_1, REG_2)

# low-level optimizations:
s[matmul_local].pragma(BLK_1, "auto_unroll_max_step", 512)
s[matmul_local].pragma(BLK_1, "unroll_explicit", True)

# tiling
matmul_1, matmul_2, matmul_3 = tuple(matmul.op.axis) + tuple(matmul.op
    .reduce_axis)
THR_1, SHR_REG_1 = s[matmul].split(matmul_1, factor=1)
DEV_1, THR_1 = s[matmul].split(THR_1, factor=4)
BLK_1, DEV_1 = s[matmul].split(DEV_1, factor=2)
THR_2, SHR_REG_2 = s[matmul].split(matmul_2, factor=1)
DEV_2, THR_2 = s[matmul].split(THR_2, factor=20)
BLK_2, DEV_2 = s[matmul].split(DEV_2, factor=1)
s[matmul].reorder(BLK_1, BLK_2, DEV_1, DEV_2, THR_1, THR_2, SHR_REG_1,
     SHR_REG_2)
s[matmul_local].compute_at(s[matmul], THR_2)

# block/thread assignments:
BLK_fused = s[matmul].fuse(BLK_1, BLK_2)
s[matmul].bind(BLK_fused, te.thread_axis("blockIdx.x"))
DEV_fused = s[matmul].fuse(DEV_1, DEV_2)
s[matmul].bind(DEV_fused, te.thread_axis("vthread"))
THR_fused = s[matmul].fuse(THR_1, THR_2)
s[matmul].bind(THR_fused, te.thread_axis("threadIdx.x"))

# exploiting fast memory resources for first input matrix:
A_shared = s.cache_read(A, "shared", [matmul_local])
A_shared_ax0, A_shared_ax1 = tuple(A_shared.op.axis)
A_shared_ax0_ax1_fused = s[A_shared].fuse(A_shared_ax0, A_shared_ax1)
A_shared_ax0_ax1_fused_o, A_shared_ax0_ax1_fused_i = s[A_shared].split
    (A_shared_ax0_ax1_fused, factor=1)
s[A_shared].vectorize(A_shared_ax0_ax1_fused_i)
A_shared_ax0_ax1_fused_o_o, A_shared_ax0_ax1_fused_o_i = s[A_shared].
    split(A_shared_ax0_ax1_fused_o, factor=80)
s[A_shared].bind(A_shared_ax0_ax1_fused_o_i, te.thread_axis("threadIdx
    .x"))
s[A_shared].compute_at(s[matmul_local], DEV_3)

# exploiting fast memory resources for second input matrix:
# ... (analogous to lines 40 – 47)
```

Listing 46: TVM schedule for Matrix Multiplication on NVIDIA Ampere GPU (variable names shortened for brevity).

.6 FULL VERSION: SECTION 4.4

We have designed our formalism such that an expression in our high-level representation (such as in Figure 64) can be *systematically lowered* to an expression in our low-level representation (as in Figure 75). We confirm this by parameterizing the generic high-level expression in Figure 73 – step-by-step – in the tuning parameters listed in Table 8, in a formally sound manner, which results exactly in the generic low-level expression in Figure 77.

Note that the tuning parameters in Table 8 can also be interpreted as parameters of the lowering process (instead of the low-level representation). This is because in practice, our lowering process takes as input a particular configuration of the tuning parameter in Table 8 (automatically chosen via auto-tuning), such that it lowers a particular instance in our high-level representation (i.e., for a concrete choice of: scalar function, combine operator, etc) straight to a particular instance in our low-level representation (instead of lowering first to the generic low-level instance in Figure 77 and then inserting tuning parameters in this generic instance).

Parameter `0`. Let $\downarrow a$ be the input MDA. Let further be $\downarrow a_f$ the L-layered, D-dimensional, P-partitioned low-level MDA (according to Definition 38), for

$$P := \Big(\underbrace{\underbrace{(\#PRT(1,1),\ldots,\#PRT(1,D))}_{\text{Dimension 1}} \ \underbrace{\ldots}_{\text{Dimension D}}}_{\text{Layer 1}} \ , \ldots , \ \underbrace{\underbrace{(\#PRT(L,1),\ldots,\#PRT(L,D))}_{\text{Dimension 1}} \ \underbrace{\ldots}_{\text{Dimension D}}}_{\text{Layer L}} \Big)$$

where #PRT denotes the number of partitions (Parameter `0` in Table 8), which is defined as:

$$\downarrow a \ =: \ \underbrace{\underbrace{\underset{p_1^1 \in P_1^1}{+\!\!+_1} \ \ldots \ \underset{p_D^1 \in P_D^1}{+\!\!+_D}}_{\text{Dimension 1}} \ \underbrace{\ldots}_{\text{Dimension D}}}_{\text{Layer 1}} \ \vdots \ \underbrace{\underbrace{\underset{p_1^L \in P_1^L}{+\!\!+_1} \ \ldots \ \underset{p_D^L \in P_D^L}{+\!\!+_D}}_{\text{Dimension 1}} \ \underbrace{\ldots}_{\text{Dimension D}}}_{\text{Layer L}} \quad \downarrow a_f^{<p_1^1,\ldots,p_D^1 \ | \ \ldots \ | \ p_1^L,\ldots,p_D^L>}$$

Applying $L * D$ times the homomorphic property (Definition 28), we get:

$$\uparrow a \;:=\; \texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\downarrow a) \;=$$

$$\underbrace{\underbrace{\overset{\circledast_1}{p_1^1 \in \#PRT(1,1)} \cdots \overset{\circledast_D}{p_D^1 \in \#PRT(1,D)}}_{\text{Dimension 1}} \cdots \underbrace{\overset{\circledast_1}{p_1^L \in \#PRT(L,1)} \cdots \overset{\circledast_D}{p_D^L \in \#PRT(L,D)}}_{\text{Dimension D}}}_{\text{Layer 1}} \qquad \underbrace{\qquad\qquad}_{\text{Layer L}}$$

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\, {}^{\downarrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}\,)$$

Since each part ${}^{\downarrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}$ contains a single scalar value only (according to the algorithmic constraint of Parameter 1, discussed in Section .4.4), it holds

$$\texttt{md\_hom}(\, f\,,\, (\circledast_1,\ldots,\circledast_D)\,)(\, {}^{\downarrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}\,) = {}^{\uparrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}$$

for

$${}^{\uparrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>} := \vec{f}(\, {}^{\downarrow}a_f^{<p_1^1,\ldots,p_D^1 \mid \ldots \mid p_1^L,\ldots,p_D^L>}\,)$$

and $\vec{f}$ defined as in Definition 28. □

Parameter `D1`. Parameter `D1` reorders concatenation operators $+\!\!+_1,\ldots,+\!\!+_D$ (Example 13). We prove our assumption w.l.o.g. for the case $D = 2$; the general case $D \in \mathbb{N}$ follows analogously.

Let $+\!\!+_{d_1} \in \texttt{CO}^{<id \mid T \mid D \mid d_1>}$ and $+\!\!+_{d_2} \in \texttt{CO}^{<id \mid T \mid D \mid d_2>}$ be two arbitrary concatenation operators that coincide in meta-parameters T and D, but may differ in their operating dimensions d_1 and d_2. We have to show

$$(a_1 +\!\!+_{d_1} a_2) +\!\!+_{d_2} (a_3 +\!\!+_{d_1} a_4) \;=\; (a_1 +\!\!+_{d_2} a_3) +\!\!+_{d_1} (a_2 +\!\!+_{d_2} a_4)$$

which follows from the definition of the concatenation operator $+\!\!+$ in Example 13. □

Parameters `D2`, `S2`, `R2`. These parameters replaces combine operators (Definition 26) by low-level combine operators (Definition 41), which has no effect on semantics. □

Parameters `D3`, `S3`, `S5`, `R3`. These parameters set the memory tags of low-level BUFs (Definition 39), which have no effect on semantics. □

Parameters `D4`, `S4`, `S6`, `R4`. The parameters change the memory layout of low-level BUFs (Definition 39), which does not affect extensional equality. □

Parameters `S1`. This parameter sets the order in which function f is applied to parts, which is trivially sound for any order. □

Parameters R1. Similarly to parameter D1, parameter R1 reorders combine operators $\circledast_1, \ldots, \circledast_D$, but the operators are not restricted to be concatenation. We prove our assumption by exploiting the MDH property (Definition 27) together with the proof of parameter D1, as follows:

$$
\begin{aligned}
& & (a_1 \;\circledast_{d_1}\; a_2) \;\circledast_{d_2}\; (a_3 \;\circledast_{d_1}\; a_4) & \\
&= \texttt{md\_hom}(\ldots)(& (a_1 \;+\!\!+_{d_1}\; a_2) \;+\!\!+_{d_2}\; (a_3 \;+\!\!+_{d_1}\; a_4) & \;) \\
&= \texttt{md\_hom}(\ldots)(& (a_1 \;+\!\!+_{d_2}\; a_3) \;+\!\!+_{d_1}\; (a_2 \;+\!\!+_{d_2}\; a_4) & \;) \\
&= & (a_1 \;\circledast_{d_2}\; a_3) \;\circledast_{d_1}\; (a_2 \;\circledast_{d_2}\; a_4) & \;\checkmark
\end{aligned}
$$

□

.7 ADDENDUM SECTION 4.5

.7.1 *Data Characteristics used in Deep Neural Networks*

Figure 89 shows the data characteristics used for the deep learning experiments in Figures 27 and 28 of Section 4.5. We use real-world characteristics taken from the neural networks ResNet-50, VGG-16, and MobileNet. For each network, we consider computations MCC and MatMul (Table 74), because these are the networks' most time-intensive building blocks. Each computation is called in each network on different data characteristics – we use for each combination of network and computation the two most time-intensive characteristics. Note that the MobileNet network does not use MatMul in its implementation.

The capsule variants MCC_Capsule in Figures 27 and 28 of Section 4.5 have the same characteristics as those listed for MCCs in Figure 89; the only difference is that MCC_Capsule, in addition to the dimensions N,H,W,K,R,S,C, uses three additional dimensions MI,MJ,MK, each with a fixed size of 4. This is because MCC_Capsule operates on 4×4 matrices, rather than scalars as MCC does.

Network	Phase	N	H	W	K	R	S	C	Stride H	Stride W	Padding	P	Q	Image Format	Filter Format	Output Format
ResNet-50	Training	16	230	230	64	7	7	3	2	2	VALID	112	112	NHWC	KRSC	NPQK
	Inference	1	230	230	64	7	7	3	2	2	VALID	112	112	NHWC	KRSC	NPQK
VGG-16	Training	16	224	224	64	3	3	3	1	1	VALID	224	224	NHWC	KRSC	NPQK
	Inference	1	224	224	64	3	3	3	1	1	VALID	224	224	NHWC	KRSC	NPQK
MobileNet	Training	16	225	225	32	3	3	3	2	2	VALID	112	112	NHWC	KRSC	NPQK
	Inference	1	225	225	32	3	3	3	2	2	VALID	112	112	NHWC	KRSC	NPQK

(a) Data characteristics used for MCC experiments.

Network	Phase	M	N	K	Transposition
ResNet-50	Training	16	1000	2048	NN
	Inference	1	1000	2048	NN
VGG-16	Training	16	4096	25088	NN
	Inference	1	4096	25088	NN

(b) Data characteristics used for MatMul experiments.

Figure 89: Data characteristics used for experiments in Section 4.5.

.7.2 *Runtime and Accuracy of* `cuBLASEx`

Listing 47 shows the runtime of `cuBLASEx` for its different *algorithm* variants. For demonstration, we use the example of matrix multiplication `MatMul` on `NVIDIA Volta GPU` for square input matrices of sizes 1024×1024. For each algorithm variant, we list both: 1) the runtime achieved by `cuBLASEx` (in nanoseconds `ns`), as well as 2) the maximum absolute deviation ($\texttt{delta}_{\texttt{max}}$ values) compared to a straightforward, sequential CPU computation. For example, the $\texttt{delta}_{\texttt{max}}$ value of algorithm `CUBLAS_GEMM_DEFAULT` is `3.14713e-05`, i.e., at least one value $c_{i,j}^{GPU}$ in the GPU-computed output matrix deviates by `3.14713e-05` from its corresponding, sequentially computed value $c_{i,j}^{seq}$ such that $|c_{i,j}^{GPU}| = |c_{i,j}^{seq}| + \texttt{3.14713e-05}$ (bar symbols $|\ldots|$ denote absolute value). All other GPU-computed values $c_{i',j}^{GPU}$, deviate from their sequentially computed CPU-variant by `3.14713e-05` or less.

Note that `cuBLASEx` offers 42 algorithm variants, but not all of them are supported for all potential characteristics of the input and output data (size, memory layout, ...). For our `MatMul` example, the list of unsupported variants includes: `CUBLAS_GEMM_ALGO1`, `CUBLAS_GEMM_ALGO12`, etc.

```
CUBLAS_GEMM_DEFAULT: 188416ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO2: 190464ns (delta_max: 6.86646e-05)
CUBLAS_GEMM_ALGO3: 186368ns (delta_max: 6.86646e-05)
CUBLAS_GEMM_ALGO4: 185344ns (delta_max: 6.86646e-05)
CUBLAS_GEMM_ALGO5: 181248ns (delta_max: 6.86646e-05)
CUBLAS_GEMM_ALGO6: 181248ns (delta_max: 6.86646e-05)
CUBLAS_GEMM_ALGO7: 178176ns (delta_max: 4.1008e-05)
CUBLAS_GEMM_ALGO8: 189440ns (delta_max: 4.1008e-05)
CUBLAS_GEMM_ALGO9: 171008ns (delta_max: 4.1008e-05)
CUBLAS_GEMM_ALGO10: 188416ns (delta_max: 4.1008e-05)
CUBLAS_GEMM_ALGO11: 191488ns (delta_max: 4.1008e-05)
CUBLAS_GEMM_ALGO18: 185344ns (delta_max: 2.67029e-05)
CUBLAS_GEMM_ALGO19: 172032ns (delta_max: 2.67029e-05)
CUBLAS_GEMM_ALGO20: 192512ns (delta_max: 2.67029e-05)
CUBLAS_GEMM_ALGO21: 201728ns (delta_max: 1.90735e-05)
CUBLAS_GEMM_ALGO22: 177152ns (delta_max: 1.90735e-05)
CUBLAS_GEMM_ALGO23: 194560ns (delta_max: 1.90735e-05)
CUBLAS_GEMM_DEFAULT_TENSOR_OP: 184320ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO0_TENSOR_OP:  62464ns (delta_max: 0.0131454)
CUBLAS_GEMM_ALGO1_TENSOR_OP:  52224ns (delta_max: 0.0131454)
CUBLAS_GEMM_ALGO2_TENSOR_OP:  190464ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO3_TENSOR_OP:  189440ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO4_TENSOR_OP:  183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO5_TENSOR_OP:  183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO6_TENSOR_OP:  183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO7_TENSOR_OP:  189440ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO8_TENSOR_OP:  183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO9_TENSOR_OP:  189440ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO10_TENSOR_OP: 188416ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO11_TENSOR_OP: 183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO12_TENSOR_OP: 183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO13_TENSOR_OP: 188416ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO14_TENSOR_OP: 183296ns (delta_max: 3.14713e-05)
CUBLAS_GEMM_ALGO15_TENSOR_OP: 189440ns (delta_max: 3.14713e-05)
```

Listing 47: Runtime of `cuBLASEx` for its different *algorithm* variants on `NVIDIA Volta GPU` when computing `MatMul` on square 1024×1024 input matrices.

.8 CODE GENERATION

This section outlines how imperative-style pseudocode is generated from our low-level program representation in Section .4. Optimizations that operate below the abstraction level of our low-level representation (e.g., loop unrolling) are beyond the scope of this section and outlined in Section .9. We aim to discuss and illustrate our code generation approach in detail in future work.

In the following, we highlight tuning parameters gray in our pseudocode, which are substituted by concrete, optimized values in our executable program code. Static parameters, such as scalar types and the number of input/output buffers, are denoted in math font and also substituted by concrete values in our executable code. We list meta-parameters in angle brackets `<...>`, and other static function annotations in double angle brackets `«...»`, e.g., `idx«OUT»«1,1»` for denoting index function $\mathfrak{id}^{\mathtt{OUT}}_{1,1}$ (used in Figure 73) in our pseudocode.

Overall Structure

Listing 48 shows the overall structure of our generated code. We implement a particular expression in our low-level representation (Figure 77) as a compute kernel that is structured in the following phases: 0) preparation (Section .8.0), 1) de-composition phase (Section .8.1), 2) scalar phase (Section .8.2), 3) re-composition phase (Section .8.3).

```
1  kernel mdh(
2    T_1^IB trans_ll_IB<<⊥>><<1>><*,...,*>,...,T_B^IB trans_ll_IB<<⊥>><<B^IB
         >><*,...,*> ,
3    T_1^OB trans_ll_OB<<⊥>><<1>><*,...,*>,...,T_B^OB trans_ll_OB<<⊥>><<B^OB
         >><*,...,*> )
4  {
5    // 0. preparation
6    ...
7    // 2. de-composition phase
8    ...
9    // 3. scalar phase
10   ...
11   // 4. re-composition phase
12   ...
13 }
```

Listing 48: Overall structure of our generated code.

.8.0 *Preparation*

Listing 49 shows the preparation phase. It prepares in five sub-phases the basic building blocks used in our low-level representation: 1) `md_hom` (Section .8.0.1), 2) `inp_view` (Section .8.0.2), 3) `out_view` (Section .8.0.3), 4) BUFs (Section .8.0.4), 5) MDAs (Section .8.0.5).

```
// 0. preparation
  // 0.1. md_hom
  ...
  // 0.2. inp_view
  ...
  // 0.3. out_view
  ...
  // 0.4. BUFs
  ...
  // 0.5. MDAs
  ...
```

Listing 49: Preparation Phase.

.8.0.1 *md_hom*

Listing 50 shows the user-defined scalar function and low-level combine operators (Definition 41) which are both provided by the user via higher-order function `md_hom` (Definition 28).

Listing 51 shows how we pre-implement for the user the two combine operators *concatenation* (Example 13) and *point-wise combination* (Example 14).

Listing 52 shows how we pre-implement the *inverse of concatenation* (Definition .5.2), which we use in the de-composition phase (via Definition 44).

```
// 0.1. md_hom

  // 0.1.1. scalar function
  f( T^INP inp ) -> T^OUT out
  {
    // ... (user defined)
  }

  // 0.1.2. combine operators
  ∀d ∈ [1,D]_ℕ :
  co<<d>><
       I_1,...,I_{d-1},I_{d+1},...,I_D ∈ MDA-IDX-SETs , (P,Q) ∈ MDA-IDX-SETs ×̇ MDA-IDX-SETs
       >(
    T^OUT[I_1,...,I_{d-1}, ⇒^d_MDA(P) ,I_{d+1},...,I_D] lhs ,
    T^OUT[I_1,...,I_{d-1}, ⇒^d_MDA(Q) ,I_{d+1},...,I_D] rhs ) ->
        T^OUT[I_1,...,I_{d-1},⇒^d_MDA(P ∪ Q),I_{d+1},...,I_D] res
  {
    // ... (user defined)
  }
```

Listing 50: Scalar Function & Combine Operators.

```
1  // 0.1.2. combine operators
2
3    // pre-implemented combine operators
4
5    // concatenation
6    ∀d ∈ ℕ:
7    cc<<d>><
         I_1,...,I_{d-1},I_{d+1},...,I_D ∈ MDA-IDX-SETs, (P,Q) ∈ MDA-IDX-SETs ×̇ MDA-IDX-SETs
         >(
8      T^OUT[I_1,...,I_{d-1}, id(P) ,I_{d+1},...,I_D] lhs ,
9      T^OUT[I_1,...,I_{d-1}, id(Q) ,I_{d+1},...,I_D] rhs ) ->
           T^OUT[I_1,...,I_{d-1},id(P ⊎ Q),I_{d+1},...,I_D] res
10   {
11     int i_1 ∈ I_1
12       ⋱
13         int i_{d-1} ∈ I_{d-1}
14           int i_{d+1} ∈ I_{d+1}
15             ⋱
16               int i_D ∈ I_D
17               {
18                 int i_d ∈ P
19                   res[ i_1,...,i_d,...,i_D] := lhs[ i_1,...,i_d,...,
                         i_D];
20                 int i_d ∈ Q
21                   res[ i_1,...,i_d,...,i_D] := rhs[ i_1,...,i_d,...,
                         i_D];
22               }
23   }
24
25   // point-wise combination
26   ∀d ∈ ℕ:
27   pw<<d>><
         I_1,...,I_{d-1},I_{d+1},...,I_D ∈ MDA-IDX-SETs, (P,Q) ∈ MDA-IDX-SETs ×̇ MDA-IDX-SETs
         >(⊕ : T^OUT × T^OUT → T^OUT)( T^OUT[I_1,...,I_{d-1}, 0_f(P) ,I_{d+1},...,I_D] lhs ,
28                                     T^OUT[I_1,...,I_{d-1}, 0_f(Q) ,I_{d+1},...,I_D] rhs )
29   -> T^OUT[I_1,...,I_{d-1},0_f(P ⊎ Q),I_{d+1},...,I_D] res
30   {
31     int i_1 ∈ I_1
32       ⋱
33         int i_{d-1} ∈ I_{d-1}
34           int i_{d+1} ∈ I_{d+1}
35             ⋱
36               int i_D ∈ I_D
37               {
38                 res[ i_1,...,i_{d-1} , 0 , i_{d+1},...,i_D] :=
39                       lhs[ i_1,...,i_{d-1} , 0 , i_{d+1},...,i_D]
40              atomic(⊕) rhs[ i_1,...,i_{d-1} , 0 , i_{d+1},...,i_D];
41               }
42   }
```

Listing 51: Pre-Implemented Combine Operators.

```
// 0.1.2. combine operators

  // pre-implemented combine operators

  // inverse concatenation
  ∀d ∈ ℕ:
  cc_inv<<d>><
       I_1,...,I_{d-1},I_{d+1},...,I_D ∈ MDA-IDX-SETs, (P,Q) ∈ MDA-IDX-SETs ẋ MDA-IDX-SETs
       >( T^INP[I_1,...,I_{d-1},id(P ⊎ Q),I_{d+1},...,I_D] res ) -> (
    T^INP[I_1,...,I_{d-1}, id(P) ,I_{d+1},...,I_D] lhs ,
    T^INP[I_1,...,I_{d-1}, id(Q) ,I_{d+1},...,I_D] rhs )
  {
    int i_1 ∈ I_1
      ⋱
        int i_{d-1} ∈ I_{d-1}
          int i_{d+1} ∈ I_{d+1}
            ⋱
              int i_D ∈ I_D
              {
                int i_d ∈ P
                  res[ i_1,...,i_d,...,i_D] =: lhs[ i_1,...,i_d,...,
                      i_D];
                int i_d ∈ Q
                  res[ i_1,...,i_d,...,i_D] =: rhs[ i_1,...,i_d,...,
                      i_D];
              }
  }
```

Listing 52: Pre-Implemented Combine Operators.

.8.0.2 *inp_view*

Listing 53 shows the user-defined index functions provided by the user via higher-order function `inp_view` (Definition 32).

```
// 0.2. inp_view

  // index functions
  ∀b ∈ [1,B^IB]_ℕ, a ∈ [1,A_b^IB]_ℕ: ∀d ∈ [1,D_b^IB]_ℕ:
  static
  idx<<INP>><<b,a>><<d>>(int i_MDA_1 , ... , i_MDA_D) -> int i_BUF_d
  {
      // ... (user defined)
  }
```

Listing 53: Index Functions (input).

.8.0.3 *out_view*

Listing 54 shows the user-defined index functions provided by the user via higher-order function `out_view` (Definition 34).

```
1  // 0.3. out_view
2
3    // index functions
4    ∀b ∈ [1,B^OB]_ℕ, a ∈ [1,A_b^OB]_ℕ: ∀d ∈ [1,D_b^OB]_ℕ:
5    static
6    idx<<OUT>>><b,a>>><d>>(int i_MDA_1 , ... , i_MDA_D) -> int i_BUF_d
7    {
8        // ... (user defined)
9    }
```

Listing 54: Index Functions (output).

.8.0.4 *BUFs*

Listing 55 shows our implementation of low-level BUFs (Definition 39). We compute BUFs' sizes using the ranges of their index functions (Definitions 32 and 34). Moreover, we partially evaluate BUFs' meta-parameters `MEM` (memory region) and σ (memory layout) immediately, as the same values are re-used for them during program runtime.

The BUFs in lines 30 and 45 as well as in lines 71 and 86 represent the BUFs' transposed function representation (Definition 39), and the BUFs in lines 23, 37, and 53 as well as in lines 64, 78, and 94 are the transposed BUFs' ordinary low-level BUF representation.

```
1  // 0.4. BUFs
2
3    // 0.4.1. compute BUF sizes
4    ∀IO ∈ {IB,OB}: ∀b ∈ [1,B^IO]_ℕ ∀d ∈ [1,D_b^IO]_ℕ:
5    static N<<IO>>><b>>><d>>(mda_idx_set I_1 , ... , I_D) -> int N_b_d
6    {
7      N_b_d := 0;
8
9      i_1 ∈ I_1
10       ⋱
11       i_D ∈ I_D
12       {
13         ∀a ∈ [1,A_b^IB]_ℕ:
14         N_b_d :=_max 1 + idx<<IO>>><b,a>>><d>>( i_1,...,i_D );
15       }
16   }
17
18   // 0.4.2. input BUFs
19
20   // initial BUFs
21   ∀b ∈ [1,B^IB]_ℕ:
22   static ll_IB<<⊥>>><b>>><▼_1^(⊥)> ∈ #PRT(1,1) ,...,<▼_D^(⊥)> ∈ #PRT(L,D) >(
23     int i_1,...,int i_D_b^IB ) -> T_b^IB a
24   {
25     a := trans_ll_IB<<⊥>>><b>>><▼_1^(⊥),...,▼_D^(⊥)>[ i_1 , ... , i_D_b^IB ];
26   }
```

```
27
28  // de-composition BUFs
29  ∀(l,d) ∈ MDH-LVL: ∀b ∈ [1,B^IB]_ℕ:
30  auto trans_ll_IB<<l,d>>><b>>< ▼^(l,d)_1 ∈ #PRT(1,1) ,...,  ▼^(l,d)_D ∈ #PRT(L,D)
        >
31    := ↓-mem^<b>(l,d)  T^IB_b[
32          N<<IB>>><b>>>< σ^<b>_↓-mem(l,d)(1)    >>( ( ⇒^d_MDA(N_d) )_d∈[1,D]_ℕ ) ,
33            ⋮
34          N<<IB>>><b>>>< σ^<b>_↓-mem(l,d)(D^IB_b) >>( ( ⇒^d_MDA(N_d) )_d∈[1,D]_ℕ ) ];
35
36  ∀(l,d) ∈ MDH-LVL: ∀b ∈ [1,B^IB]_ℕ:
37  static ll_IB<<l,d>>><b>>< ▼^(l,d)_1 ∈ #PRT(1,1) ,...,  ▼^(l,d)_D ∈ #PRT(L,D) >(
        int i_1,...,int i_D^IB_b ) -> T^IB_b a
38  {
39    a := trans_ll_IB<<l,d>>><b>>< ▼^(l,d)_1,...,  ▼^(l,d)_D>[
40       i_ σ^<b>_↓-mem(l,d)(1)  ,..., i_ σ^<b>_↓-mem(l,d)(D^IB_b)  ];
41  }
42
43  // scalar BUFs
44  ∀b ∈ [1,B^IB]_ℕ:
45  auto trans_ll_IB<f>>><b>>< ▼^(f)_1 ∈ #PRT(1,1) ,...,  ▼^(f)_D ∈ #PRT(L,D) >
46    := f^↓-mem^<b>  T^IB_b[
47          N<<IB>>><b>>>< σ^<b>_f↓-mem(1)    >>( ( ⇒^d_MDA(N_d) )_d∈[1,D]_ℕ ) ,
48            ⋮
49          N<<IB>>><b>>>< σ^<b>_f↓-mem(D^IB_b) >>( ( ⇒^d_MDA(N_d) )_d∈[1,D]_ℕ ) ];
50
51  ∀b ∈ [1,B^IB]_ℕ:
52  static ll_IB<<f>>><b>>< ▼^(f)_1 ∈ #PRT(1,1) ,...,  ▼^(f)_D ∈ #PRT(L,D) >(
53    int i_1 ,..., int i_D^IB_b ) -> T^IB_b a
54  {
55    a := trans_ll_IB<<f>>><b>>< ▼^(f)_1,...,  ▼^(f)_D>[
56          i_ σ^<b>_f↓-mem(1)  , ... , i_ σ^<b>_f↓-mem(D^IB_b)  ];
57  }
58
59  // 0.4.3. output BUFs
60
61  // initial BUFs
62  ∀b ∈ [1,B^OB]_ℕ:
63  static ll_OB<<⊥>>><b>>< ▲^(⊥)_1 ∈ #PRT(1,1) ,...,  ▲^(⊥)_D ∈ #PRT(L,D) >(
64    int i_1 ,..., int i_D^OB_b ) -> T^OB_b a
65  {
66    a := trans_ll_OB<<⊥>>><b>>< ▲^(⊥)_1,...,  ▲^(⊥)_D>[ i_1 , ... , i_D^OB_b ];
67  }
68
69  // re-composition BUFs
70  ∀(l,d) ∈ MDH-LVL: ∀b ∈ [1,B^OB]_ℕ:
71  auto trans_ll_OB<<l,d>>><b>>< ▲^(l,d)_1 ∈ #PRT(1,1) ,...,  ▲^(l,d)_D ∈ #PRT(L,D)
        >
72    := ↑-mem^<b>(l,d)  T^OB_b[
73          N<<OB>>><b>>>< σ^<b>_↑-mem(l,d)(1)    >>( ( ⇒^d_MDA(N_d) )_d∈[1,D]_ℕ ) ,
```

```
74          ⋮
75        N<<OB>><<<b>><<< σ_{↑-mem}^{<b>}(l,d)(D_b^{OB}) >>( ( ⇒_{MDA}^{d} MDA(N_d) )_{d∈[1,D]_ℕ} ) ];
76
77  ∀(l,d) ∈ MDH-LVL: ∀b ∈ [1, B^{OB}]_ℕ:
78  static ll_OB<<l,d>><<<b>>< ▲_1^{(l,d)} 1 ∈ #PRT(1,1) ,..., ▲_D^{(l,d)} L ∈ #PRT(L,D) >(
        int i_1 ,..., int i_D_b^{OB} ) -> T_b^{OB} a
79  {
80    a := trans_ll_OB<<l,d>><<<b>>< ▲_1^{(l,d)} 1,..., ▲_D^{(l,d)} L >[
81        i_ σ_{↑-mem}^{<b>}(l,d)(1) ,..., i_ σ_{↑-mem}^{<b>}(l,d)(D_b^{OB}) ];
82  }
83
84  // scalar BUFs
85  ∀b ∈ [1, B^{OB}]_ℕ:
86  auto trans_ll_OB<f>><<<b>>< ▲_1^{(f)} 1 ∈ #PRT(1,1) ,..., ▲_D^{(f)} L ∈ #PRT(L,D) >
87    := f^↑-mem^{<b>} T_b^{OB}[
88        N<<OB>><<<b>><<< σ_{f↑-mem}^{<b>}(1) >>( ( ⇒_{MDA}^{d} MDA(N_d) )_{d∈[1,D]_ℕ} ) ,
89          ⋮
90        N<<OB>><<<b>><<< σ_{f↑-mem}^{<b>}(D_b^{OB}) >>( ( ⇒_{MDA}^{d} MDA(N_d) )_{d∈[1,D]_ℕ} ) ];
91
92  ∀b ∈ [1, B^{OB}]_ℕ:
93  static ll_OB<<f>><<<b>>< ▲_1^{(f)} 1 ∈ #PRT(1,1) ,..., ▲_D^{(f)} L ∈ #PRT(L,D) >(
94      int i_1 ,..., int i_D_b^{OB} ) -> T_b^{OB} a
95  {
96    a := trans_ll_OB<<f>><<<b>>< ▲_1^{(f)} 1,..., ▲_D^{(f)} L >[
97        i_ σ_{f↑-mem}^{<b>}(1) , ... , i_ σ_{f↑-mem}^{<b>}(D_b^{OB}) ];
98  }
```

Listing 55: Low-Level BUFs.

where $\blacktriangledown_{\mathfrak{d}}^{(\bullet)\,l}$ and $\blacktriangle_{\mathfrak{d}}^{(\bullet)\,l}$, for $\bullet \in \{\bot\} \cup \texttt{MDH-LVL} \cup \{\texttt{f}\}$, are textually replaced by:

$$\blacktriangledown_{\mathfrak{d}}^{(\bullet)\,l} = \begin{cases} p_{\mathfrak{d}}^{l} & : \sigma_{\downarrow\texttt{-ord}}(l,\mathfrak{d}) < \bullet \\ * & : \text{else} \end{cases}$$

$$\blacktriangle_{\mathfrak{d}}^{(\bullet)\,l} = \begin{cases} p_{\mathfrak{d}}^{l} & : \sigma_{\uparrow\texttt{-ord}}(l,\mathfrak{d}) < \bullet \\ * & : \text{else} \end{cases}$$

(symbol $*$ is taken from Definition 23) where $<$ is defined according to the lexicographical order on $\texttt{MDH-LVL} = [1,L]_{\mathbb{N}} \times [1,D]_{\mathbb{N}}$, and:

$$\forall (l,d) \in \texttt{MDH-LVL}: \bot :< (l,d) :< \texttt{f}$$

Functions

$$\overset{1}{\underset{MDA}{\overset{\Rightarrow}{\#}}}, \cdots, \overset{D}{\underset{MDA}{\overset{\Rightarrow}{\#}}}$$

are the index set functions id of combine operator concatenation ++ (Example 13), and functions

$$\overset{1}{\underset{\circledast}{\Rightarrow}}\text{MDA}\,,\ \cdots\ ,\ \overset{D}{\underset{\circledast}{\Rightarrow}}\text{MDA}$$

are the index set functions of combine operators $\circledast_1,\ldots,\circledast_D$.

Note that we use generous BUFs sizes (lines 32-34, 47-49, 73-75, 88-90), as imperative-style programming models usually struggle with non-contiguous index ranges. We discuss optimizations targeting BUF sizes in Section .9.

Note further that we do not need to initialize output buffers with neutral elements of combine operators in lines 66, 81, and 97 of Listing 55, as the buffers are initialized implicitly in the re-composition phase (Section .8.3).

.8.0.5 *MDAs*

Listing 56 shows our implementation of low-level MDAs (Definitions 38 and 44).

Note that for a particular choice of meta-parameters, low-level BUFs (Definition 39) are ordinary BUFs (Definition 29), as required by the types of functions `inp_view` and `out_view` (Definitions 32 and 34).

```
1  // 0.5. MDAs
2
3    // 0.5.1. partitioned index sets
4    ∀d ∈ [1,D]_ℕ :
5    static I<<d>><p_d^1 ∈ #PRT(1,d) ,...,p_d^L ∈ #PRT(L,d) >( int j' ) -> int
         i_j
6    {
7      i_j := ( p_d^1 * ( N_d / ( #PRT(1,d) )             ) +
8               ⋮
9              p_d^L * ( N_d / ( #PRT(1,d) *...* #PRT(L,d) ) ) + j' );
10   }
11
12   // 0.5.2. input MDAs
13   ∀• ∈ {⊥} ∪ MDH-LVL ∪ {f}:
14   static ll_inp_mda<<•>><▼_1^(•) ∈ #PRT(1,1) ,...,▼_D^(•) ∈ #PRT(L,D) >( int
         i_1,...,
15     int i_D ) -> T_b^IB a
16   {
17     ∀b ∈ [1,B^IB]_ℕ, a ∈ [1,A_b^IB]_ℕ :
18       a := ll_IB<<•>><<b>><▼_1^(•),...,▼_D^(•)>(
19           idx<<INP>><<b,a>><< 1 >>( i_1,...,i_D ) ,
20             ⋮
21           idx<<INP>><<b,a>><<D_b^IB>>( i_1,...,i_D ) );
22   }
23
24   // 0.5.3. output MDAs
25   ∀• ∈ {⊥} ∪ MDH-LVL ∪ {f}:
26   static ll_out_mda<<•>><▲_1^(•) ∈ #PRT(1,1) ,...,▲_D^(•) ∈ #PRT(L,D) >(
27     int i_1 ,..., int i_D ) -> T_b^OB a
28   {
29
30
```

```
31      ∀b ∈ [1, B^OB]_ℕ, a ∈ [1, A_b^OB]_ℕ :
32        a := ll_OB<<•>><<b>><(•)_1, ..., (•)_L_D>(
33            idx<<OUT>><<b,a>><< 1 >>( i_1,...,i_D ) ,
34              ⋮
35            idx<<OUT>><<b,a>><<D_b^OB>>( i_1,...,i_D ) );
36  }
```

Listing 56: Low-Level MDAs.

For computing the partitioned index sets (lines 3-10), we exploit the following proposition.

Proposition 1. Let $a \in T[N_1, \ldots, N_D]$ be an arbitrary MDA that operates on contiguous index sets $[1, N_d)_{\mathbb{N}}$, $d \in [1, D]_{\mathbb{N}}$. Let further be

$$a^{<(p_1^1,\ldots,p_d^1)\in P_1^1\times\ldots\times P_D^1 \;\mid\; \ldots \;\mid\; (p_1^L,\ldots,p_D^L)\in P_1^L\times\ldots\times P_D^L>} : I_1^{<p_1^1 \cdots p_1^L>} \times \ldots \times I_D^{<p_D^1 \cdots p_D^L>} \to T$$

an arbitrary L-layered, D-dimensional, P-partitioning of MDA a.

It holds that j-th element within an MDA's part is accessed via index j:

$$I_d^{<p_d^1 \cdots p_d^L>} = \{ \underbrace{\sum_{l\in[1,L]_{\mathbb{N}}} p_d^l * \frac{N_d}{\prod_{l'\in[1,l]_{\mathbb{N}}} P_d^{l'}}}_{\text{OS}} + 0, \underbrace{\sum_{l\in[1,L]_{\mathbb{N}}} p_d^l * \frac{N_d}{\prod_{l'\in[1,l]_{\mathbb{N}}} P_d^{l'}}}_{\text{OS}} + 1, \ \ldots \ \}$$

Proof. Since MDA a's index sets are contiguous ranges of natural numbers, it holds the i_j – the index to access the j-th element within an MDA's part (Definition 44) – is equal to j itself. □

.8.1 *De-Composition Phase*

Listing 57 shows our implementation of the de-composition phase (Figure 77).

```
// 1. de-composition phase

  // 1.1. initialization
  ll_inp_mda<<⊥>> =: ll_inp_mda<< σ↓-ord(1,1) >>

  // 1.2. main
  int p_ σ↓-ord(1,1) ∈< ↔↓-ass (1,1) > #PRT ( σ↓-ord(1,1) )
  {
    ll_inp_mda<< σ↓-ord(1,1) >> =:cc< σ↓-ord(1,1) > inp_mda<< σ↓-ord(1,2) >>;

    int p_ σ↓-ord(1,2) ∈< ↔↓-ass (1,2) > #PRT ( σ↓-ord(1,2) )
    {
      ll_inp_mda<< σ↓-ord(1,2) >> =:cc< σ↓-ord(1,2) > inp_mda<< σ↓-ord(1,3) >>;
        ⋱
          int p_ σ↓-ord(L,D) ∈< ↔↓-ass (L,D) > #PRT ( σ↓-ord(L,D) )
          {
            ll_inp_mda<< σ↓-ord(L,D) >> =:cc< σ↓-ord(L,D) > inp_mda<<f>>;
          }
        ⋰
    }
  }
```

Listing 57: De-Composition Phase.

where

```
ll_inp_mda<<l,d>> =:cc<l,d> ll_inp_mda<<l′,d′>>
```

abbreviates

```
ll_inp_mda<<l′,d′>>< ▼^(l′,d′)^1_1 ,..., ▼^(l′,d′)^L_D >,
ll_inp_mda<<l ,d >>< ▼^(l,d)^1_1 ,..., ▼^(l,d)^L_D >
:=  cc_inv<<d>><⇒^1_MDA_MDA ( I<<1>>< ■^(l,d)^1_1,..., ■^(l,d)^L_1>(0) ),  // I_1
                 ⋮                                                        //...,I_{d−1},I_{d+1},...
               ⇒^D_MDA_MDA ( I<<D>>< ■^(l,d)^1_D,..., ■^(l,d)^L_D>(0) ),  //I_D

               ⇒^d_MDA_MDA ( I<<d>>< ⊞^(l,d)^1_d,..., ⊞^(l,d)^L_d>(0) ),  // P
               ⇒^d_MDA_MDA ( I<<d>>< ⊠^(l,d)^1_d,..., ⊠^(l,d)^L_d>(0) )   // Q
             >( ll_inp_mda<<l,d>>< ▼^(l,d)^1_1 ,..., ▼^(l,d)^L_D> )
```

Here, functions $\overset{1}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}, \ldots, \overset{D}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}$ are the index set functions id of combine operator concatenation $++_1, \ldots, ++_D$ (Example 13), and $\overset{(\bullet)}{\blacksquare}_{l}^{d}$, $\overset{(\bullet)}{\boxplus}_{l}^{d}$, $\overset{(\bullet)}{\boxtimes}_{l}^{d}$, for $\bullet \in$ MDH-LVL, are textually replaced by:

$$\overset{(\bullet)}{\blacksquare}_{l}^{d} := \begin{cases} p\_(l,d) & : \ \sigma_{\downarrow\text{-ord}}(l,d) < \bullet \\ [p\_(l,d), \ \#PRT(l,d)\ )_{\mathbb{N}_0} & : \ (l,d) = \bullet \\ [0, \ \#PRT(l,d)\ )_{\mathbb{N}_0} & : \ \sigma_{\downarrow\text{-ord}}(l,d) > \bullet \end{cases}$$

$$\overset{(\bullet)}{\boxplus}_{l}^{d} := \begin{cases} p\_(l,d) & : \ \sigma_{\downarrow\text{-ord}}(l,d) < \bullet \\ p\_(l,d) & : \ (l,d) = \bullet \\ [0, \ \#PRT(l,d)\ )_{\mathbb{N}_0} & : \ \sigma_{\downarrow\text{-ord}}(l,d) > \bullet \end{cases}$$

$$\overset{(\bullet)}{\boxtimes}_{l}^{d} := \begin{cases} p\_(l,d) & : \ \sigma_{\downarrow\text{-ord}}(l,d) < \bullet \\ (p\_(l,d), \ \#PRT(l,d)\ )_{\mathbb{N}_0} & : \ (l,d) = \bullet \\ [0, \ \#PRT(l,d)\ )_{\mathbb{N}_0} & : \ \sigma_{\downarrow\text{-ord}}(l,d) > \bullet \end{cases}$$

where $<$ is defined as lexicographical order, according to Section .8.0.4.
Note that we re-use `inp_mda«l,d»` for the intermediate results given by different iterations of variable `p_(l,d)`. Correctness is ensured, as it holds:

$$A \subseteq B \ \Rightarrow \ \overset{d}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}(A) \ \subseteq \ \overset{d}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}(B)$$

MDA `inp_mda«l,d»` has the following type when used for the intermediate result in a particular iteration of `p_(l,d)`:

$$\overset{1}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}\Big(I_1^{<\overset{(l,d)_1}{\blacksquare}_{1},\ldots,\overset{(l,d)_1}{\blacksquare}_{D}|\ldots|\overset{(l,d)_L}{\blacksquare}_{1},\ldots,\overset{(l,d)_L}{\blacksquare}_{D}>}\Big) \times \ldots \times \overset{D}{\underset{++}{\Rightarrow}}_{MDA}^{MDA}\Big(I_D^{<\overset{(l,d)_1}{\blacksquare}_{1},\ldots,\overset{(l,d)_1}{\blacksquare}_{D}|\ldots|\overset{(l,d)_L}{\blacksquare}_{1},\ldots,\overset{(l,d)_L}{\blacksquare}_{D}>}\Big) \ \to \ T^{INP}$$

Here, for a set $P \subseteq [0, \ \#PRT(l,d)\ )_{\mathbb{N}_0}$, index set $I_d^{<\ldots|\ldots P \ldots|\ldots>}$ denotes $\bigcup_{p_d^l \in P} I_d^{<\ldots|\ldots p_d^l \ldots|\ldots>}$.

.8.2 *Scalar Phase*

Listing 58 shows our implementation of the scalar phase (Figure 77).

```
// 2. scalar phase
int p_ σ_{f-ord}(1,1) ∈^{< ↔_{f-ass} (1,1) >} #PRT ( σ_{f-ord}(1,1) )
  ⋱
    int p_ σ_{f-ord}(L,D) ∈^{< ↔_{f-ass} (L,D) >} #PRT ( σ_{f-ord}(L,D) )
    {
      (
        ll_out_mda<<f>><<<
            p_(1,1) ,..., p_(1,D),
                      ...
            p_(L,1) ,..., p_(L,D)>><<<b,a>>(
                 1 ⇒^{MDA}_{MDA} ⊛ ( I<<1>><p_(1,1),...,p_(L,1)>(0) ) ,
                   ⋮
                 D ⇒^{MDA}_{MDA} ⊛ ( I<<D>><p_(1,D),...,p_(L,D)>(0) ) )
        )_{b∈[1,B^{0B}]_ℕ, a∈[1,A_b^{0B}]_ℕ} := f(
            ( ll_inp_mda<<f>><< p_(1,1) ,..., p_(1,D),
                                        ...
                                  p_(L,1) ,..., p_(L,D)>><<<b,a>>(
                      d ⇒^{MDA}_{MDA} ⋕ ( I<<1>><p_(1,1),...,p_(L,1)>(0) ) ,
                        ⋮
                      d ⇒^{MDA}_{MDA} ⋕ ( I<<D>><p_(1,D),...,p_(L,D)>(0) ) )
            )_{b∈[1,B^{IB}]_ℕ, a∈[1,A_b^{IB}]_ℕ} )
    }
```

Listing 58: Scalar Phase.

.8.3 *Re-Composition Phase*

Listing 59 shows our implementation of the re-composition phase (Figure 77).

```
1  // 3. re-composition phase
2
3    // 3.1. main
4    int p_ σ↑-ord(1,1) ∈< ↔↑-ass (1,1) > #PRT ( σ↑-ord(1,1) )
5    {
6      int p_ σ↑-ord(1,2) ∈< ↔↑-ass (1,2) > #PRT ( σ↑-ord(1,2) )
7      {
8        ⋱
9          int p_ σ↑-ord(L,D) ∈< ↔↑-ass (L,D) > #PRT ( σ↑-ord(L,D) )
10         {
11           ll_out_mda<< σ↑-ord(L,D) >> :=co< σ↑-ord(L,D) > out_mda<<f>>;
12         }
13       ∴
14       ll_out_mda<< σ↑-ord(1,2) >> :=co< σ↑-ord(1,2) > out_mda<< σ↑-ord(1,3) >>;
15     }
16     ll_out_mda<< σ↑-ord(1,1) >> :=co< σ↑-ord(1,1) > out_mda<< σ↑-ord(1,2) >>;
17   }
18
19   // 3.2. finalization
20   ll_out_mda<<⊥>> := ll_out_mda<< σ↑-ord(1,1) >>
```

Listing 59: Re-Composition Phase.

where

$$\texttt{ll\_out\_mda<<l,d>>} :=_{\texttt{co<l,d>}} \texttt{ll\_out\_mda<<l',d'>>}$$

abbreviates

$$\begin{aligned}
&\texttt{ll_out_mda<<l,d>>}\texttt{><}\blacktriangle^{(l,d)}_{1}\texttt{,...,}\blacktriangle^{(l,d)}_{D}\texttt{>}\\
&:= \texttt{co<<d>>}\overset{1}{\underset{\circledast}{\Rightarrow}}{}^{\text{MDA}}_{\text{MDA}}\texttt{(I<<1>><}\blacksquare^{(l,d)}_{1}\texttt{,...,}\blacksquare^{(l,d)}_{1}\texttt{>(0)),}\quad \texttt{//}\ I_1\\
&\qquad\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{//...,}I_{d-1}\texttt{,}I_{d+1}\texttt{,...}\\
&\qquad\overset{D}{\underset{\circledast}{\Rightarrow}}{}^{\text{MDA}}_{\text{MDA}}\texttt{(I<<D>><}\blacksquare^{(l,d)}_{D}\texttt{,...,}\blacksquare^{(l,d)}_{D}\texttt{>(0)),}\quad \texttt{//}\ I_D\\
&\qquad\overset{d}{\underset{\circledast}{\Rightarrow}}{}^{\text{MDA}}_{\text{MDA}}\texttt{(I<<d>><}\boxplus^{(l,d)}_{d}\texttt{,...,}\boxplus^{(l,d)}_{d}\texttt{>(0)),}\quad \texttt{//}\ P\\
&\qquad\overset{d}{\underset{\circledast}{\Rightarrow}}{}^{\text{MDA}}_{\text{MDA}}\texttt{(I<<d>><}\boxtimes^{(l,d)}_{d}\texttt{,...,}\boxtimes^{(l,d)}_{d}\texttt{>(0))}\quad \texttt{//}\ Q\\
&\qquad\texttt{>(ll_out_mda<<l,d>><}\blacktriangle^{(l,d)}_{1}\texttt{,...,}\blacktriangle^{(l,d)}_{D}\texttt{>,}\\
&\qquad\quad\texttt{ll_out_mda<<l',d'>><}\blacktriangle^{(l',d')}_{1}\texttt{,...,}\blacktriangle^{(l',d')}_{D}\texttt{>)}
\end{aligned}$$

Here, functions $\overset{1}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}, \ldots, \overset{D}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}$ are the index set function of combine operators $\circledast_1, \ldots, \circledast_D$ (Definition 26), and $\overset{(\bullet)_l}{\blacksquare_d}$, $\overset{(\bullet)_l}{\boxplus_d}$, $\overset{(\bullet)_l}{\boxtimes_d}$, for $\bullet \in$ `MDH-LVL`, are textually replaced by:

$$\overset{(\bullet)_l}{\blacksquare_d} := \begin{cases} p\_(l,d) & : \sigma_{\uparrow\texttt{-ord}}(l,d) < \bullet \\ [0, p\_(l,d)]_{\mathbb{N}_0} & : (l,d) = \bullet \\ [0, \texttt{\#PRT}(l,d))_{\mathbb{N}_0} & : \sigma_{\uparrow\texttt{-ord}}(l,d) > \bullet \end{cases}$$

$$\overset{(\bullet)_l}{\boxplus_d} := \begin{cases} p\_(l,d) & : \sigma_{\uparrow\texttt{-ord}}(l,d) < \bullet \\ [0, p\_(l,d))_{\mathbb{N}_0} & : (l,d) = \bullet \\ [0, \texttt{\#PRT}(l,d))_{\mathbb{N}_0} & : \sigma_{\uparrow\texttt{-ord}}(l,d) > \bullet \end{cases}$$

$$\overset{(\bullet)_l}{\boxtimes_d} := \begin{cases} p\_(l,d) & : \sigma_{\uparrow\texttt{-ord}}(l,d) < \bullet \\ p\_(l,d) & : (l,d) = \bullet \\ [0, \texttt{\#PRT}(l,d))_{\mathbb{N}_0} & : \sigma_{\uparrow\texttt{-ord}}(l,d) > \bullet \end{cases}$$

where $<$ is defined as lexicographical order, according to Section .8.0.4.

Note that we assume for index set functions $\overset{d}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}$ that

$$A \subseteq B \;\Rightarrow\; \overset{d}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}(A) \subseteq \overset{d}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}(B)$$

(which holds for all kinds of index set functions used in this thesis, e.g., in Examples 13 and 14) so that we can re-use `out_mda«l,d»` for the intermediate results given by different iterations of `p_(l,d)`. MDA `out_mda«l,d»` has the following type when used for the intermediate result in a particular iteration of variable `p_(l,d)`:

$$\overset{1}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}\big(\, I_1^{<\blacksquare_{1,\ldots,\blacksquare_D}^{(l,d)_1} \mid \ldots \mid \blacksquare_{1,\ldots,\blacksquare_D}^{(l,d)_L}>} \,\big) \times \ldots \times \overset{D}{\underset{\circledast}{\Rightarrow}}_{\texttt{MDA}}^{\texttt{MDA}}\big(\, I_D^{<\blacksquare_{1,\ldots,\blacksquare_D}^{(l,d)_1} \mid \ldots \mid \blacksquare_{1,\ldots,\blacksquare_D}^{(l,d)_L}>} \,\big) \to T^{\texttt{OUT}}$$

Note that in line 11 of Listing 59, we implicitly override the uninitialized value in `out_mda«l,d»` (not explicitly stated in the listing for brevity), thereby avoiding initializing output buffers with neutral elements of combine operators.

.9 CODE-LEVEL OPTIMIZATIONS

We consider optimizations that operate below the abstraction level of our low-level representation (Section .4) as *code-level optimizations*. For some code-level optimizations, e.g., loop fusion, we do not want to rely on the underlying compiler (e.g., the OpenMP/CUDA/OpenCL compiler): we exactly know the structure of our code presented in Section .8 and thus, we are able to implement code-level optimizations without requiring complex compiler analyses for optimizations.

We outline our conducted code-level optimizations, which our systems performs automatically for the underlying compiler (OpenMP, CUDA, OpenCL, etc). Our future work will thoroughly discuss our code-level optimizations and how we apply them to our generated program code. Some code-level optimizations, such as loop unrolling, are (currently) left to the underlying compiler, e.g., the OpenMP, CUDA, or OpenCL compiler. In our future work, we aim to incorporate code-level optimizations, as loop unrolling, into our auto-tunable optimization process.

Loop Fusion

In Listings 57, 58, 59, the lines containing symbol "$\in$" are mapped to `for`-loops. These loops can often be fused; for example, when parameters `D1`, `S1`, `R1` as well as parameters `D2`, `S2`, `R2` in Table 8 coincide (as in Figure 75). Besides reducing the overhead caused by loop control structures, loop fusion in particular allows significantly reducing the memory footprint: we can re-use the same memory region for each BUF partition (Definition 39), rather than allocating memory for all the partitions.

Buffer Elimination

In Listing 55, we allocate BUFs for each combination of a layer and dimension. However, when memory regions and memory layouts of BUFs coincide, we can avoid a new BUF allocation, by re-using the BUF of the upper level, thereby again reducing memory footprint.

Buffer Size Reduction

We can reduce the sizes of BUFs when specific classes of index functions are used for views (Definitions 32 and 34). For example, in the case of *Dot Product (`DOT`)* (Figure 74), when accessing its input in a strided fashion – via index function $k \mapsto (2 * k)$, instead of function $k \mapsto (k)$ (as in Figure 74) – we would have to allocate BUFs (Listing 55, lines 30 and 45) of size $2 * K$ for an input size of $K \in \mathbb{N}$; in these BUFs, each second value (accessed via indices $2 * k + 1$) would be undefined. We avoid this waste of memory, by using index function $k \mapsto (k)$ instead of $k \mapsto (2 * k)$ for allocated BUFs (Listing 56, lines 19-21 and 33-35, case "$\bullet \neq \bot$"), which avoids index functions' leading factors and potential constant additions. Thereby, we reduce the memory footprint from $2 * K$ to K. Furthermore, according to our

partitioning strategy (Listing 56, line 5, and Listings 57, 58, 59), we often access BUFs via offsets: $k \mapsto (2 * k)$ for $k \in \{OS + k' \mid k' \in \mathbb{N}\}$ and offset $OS \in \mathbb{N}$. We avoid such offset by using $k \mapsto (2 * (k - OS))$, thereby further reducing memory footprint.

Memory Operation Minimization

In our code, we access BUFs uniformly via MDAs (Listing 56), which may cause unnecessary memory operations. For example, in the de-composition phase (Listing 57) of, e.g., matrix multiplication (`MatMul`) (Figure 74), we iterate over all dimensions of the input (i.e., the i, j, k dimensions) for de-composition (Listing 52). However, the A input matrix of `MatMul` is accessed via MDA indices i and k only (Figure 74). We avoid these unnecessary memory operations (J-many in the case of an input MDA of size J in dimension j) by using index 0 only in dimension j for the de-composition of the A matrix. Analogously, we use index 0 only in the i dimension for the de-composition of `MatMul`'s B matrix which is accessed via MDA indices k and j only. Moreover, we exploit all available parallelism for memory copy operations. For example, for `MatMul`, we use also the threads intended for the j dimension when de-composing the A matrix, and we use the threads in the i dimension for the B matrix. For this, we flatten the thread ids over all dimensions i, j, k and re-structure them only in dimensions i, k (for the A matrix) or k, j (for the B matrix).

Constant Substitution

We use constants whenever possible. For example, in CUDA, variable `threadIdx.x` returns the thread id in dimension x. However, in our code, we use constant `0` instead of `threadIdx.x` when only one thread is started in dimension x, enabling the CUDA compiler to significantly simplify arithmetic expressions.

BIBLIOGRAPHY

[1] Gianpietro Consolaro et al. "PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler." In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2024, pp. 28–40. DOI: 10.1109/CGO57630.2024.10444791.

[2] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. "BaCO: A Fast and Portable Bayesian Compiler Optimization Framework." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, 2024, pp. 19–42. ISBN: 9798400703942. DOI: 10.1145/3623278.3624770. URL: https://doi.org/10.1145/3623278.3624770.

[3] MDH Project. *Multi-Dimensional Homomorphisms (MDH): An Algebraic Approach Toward Performance & Portability & Productivity for Data-Parallel Computations*. https://mdh-lang.org. 2024.

[4] *Clang: a C language family frontend for LLVM*. https://clang.llvm.org. 2023.

[5] Siyuan Feng et al. "TensorIR: An Abstraction for Automatic Tensorized Program Optimization." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, 804–817. ISBN: 9781450399166. DOI: 10.1145/3575693.3576933. URL: https://doi.org/10.1145/3575693.3576933.

[6] Intel. *Get Started with Energy Analysis*. https://www.intel.com/content/www/us/en/docs/socwatch/get-started-guide/2023-1/overview.html. 2023.

[7] MLIR – Multi-Level IR Compiler Framework. *Users of MLIR*. 2023. URL: https://mlir.llvm.org/users/.

[8] NVIDIA. *Warp Matrix Functions*. 2023. URL: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[9] Caio Salvador Rohwedder, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral. "To Pack or Not to Pack: A Generalized Packing Analysis and Transformation." In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2023. Montréal, QC, Canada: Association for Computing Machinery, 2023,

14–27. ISBN: 9798400701016. DOI: 10.1145/3579990.3580024. URL: https://doi.org/10.1145/3579990.3580024.

[10] **Ari Rasch**, Richard Schulze, Denys Shabalin, Anne Elster, Sergei Gorlatch, and Mary Hall. "(De/Re)-Compositions Expressed Systematically via MDH-Based Schedules." In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, 61–72. ISBN: 9798400700880. DOI: 10.1145/3578360.3580269. **Rank A**.

[11] Apache TVM Community. *Bind reduce axis to blocks*. https://discuss.tvm.apache.org/t/bind-reduce-axis-to-blocks/2907. 2022.

[12] Apache TVM Community. *Expressing nested reduce operations*. https://discuss.tvm.apache.org/t/expressing-nested-reduce-operations/8784. 2022.

[13] Apache TVM Community. *Implementing Array Packing via cache_read*. https://discuss.tvm.apache.org/t/implementing-array-packing-via-cache-read/13360. 2022.

[14] Apache TVM Community. *Invalid comm_reducer*. https://discuss.tvm.apache.org/t/invalid-comm-reducer/12788. 2022.

[15] Apache TVM Community. *Undetected parallelization issue*. https://discuss.tvm.apache.org/t/undetected-parallelization-issue/13224. 2022.

[16] Apache TVM Documentation. *Bind `ivar` to thread index `thread_ivar`*. https://tvm.apache.org/docs/reference/api/python/te.html?highlight=bind#tvm.te.Stage.bind. 2022.

[17] Apache TVM Documentation. *Tuning High Performance Convolution on NVIDIA GPUs*. https://tvm.apache.org/docs/how_to/tune_with_autotvm/tune_conv2d_cuda.html. 2022.

[18] Apache. *TVM: Open Deep Learning Compiler Stack*. https://github.com/apache/tvm. 2022.

[19] Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. "Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 313–324. DOI: 10.1109/CGO53902.2022.9741260.

[20] C++ reference. *Date and time utilities*. https://en.cppreference.com/w/cpp/chrono. 2022.

[21] Basile Clément and Albert Cohen. "End-to-End Translation Validation for the Halide Language." In: *OOPSLA 2022 - Conference on Object-Oriented Programming Systems, Languages, and Applications*. Vol. 6. Proceedings of the ACM on Programming Languages (PACMPL) No. OOPSLA1, Article 84. Auckland, New Zealand, Dec. 2022. DOI: 10.1145/3527328. URL: https://hal.inria.fr/hal-03653857.

[22] Facebook Research. *Tensor Comprehensions*. https://github.com/facebookresearch/TensorComprehensions. 2022.

[23] Junji Fukuhara and Munehiro Takimoto. "Automated Kernel Fusion for GPU Based on Code Motion." In: *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, 151–161. ISBN: 9781450392662. DOI: 10.1145/3519941.3535078. URL: https://doi.org/10.1145/3519941.3535078.

[24] GNU/Linux. *clock_gettime(3) – Linux man page*. https://linux.die.net/man/3/clock_gettime. 2022.

[25] Haskell.org. *Haskell: An advanced, purely functional programming language*. https://www.haskell.org. 2022.

[26] Intel. *oneAPI Math Kernel Library Link Line Advisor*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html. 2022.

[27] Intel. *oneDNN*. https://oneapi-src.github.io/oneDNN/group_dnnl_api.html. 2022.

[28] Intel. *oneMKL*. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl.html. 2022.

[29] Khronos. *Khronos Releases Vulkan SC 1.0 Open Standard for Safety-Critical Accelerated Graphics and Compute*. https://www.khronos.org/news/press/khronos-releases-vulkan-safety-critical-1.0-specification-to-deliver-safety-critical-graphics-compute. 2022.

[30] Khronos. *OpenCL: Open Standard For Parallel Programming of Heterogeneous Systems*. https://www.khronos.org/opencl/. 2022.

[31] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. "Automatic Horizontal Fusion for GPU Kernels." In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 14–27. DOI: 10.1109/CGO53902.2022.9741270.

[32] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. "Verified Tensor-Program Optimization via High-Level Scheduling Rewrites." In: *Proc. ACM Program. Lang.* 6.POPL (2022). DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[33] MDH Artifact Implementation. *https://gitlab.com/mdh-project/toplas22_artifact*. 2022.

[34] Michael Kruse. *Polyhedral Parallel Code Generation*. https://github.com/Meinersbur/ppcg, commit = 8a74e46, date = 19.11.2020. 2022.

[35] NVIDIA. *CUB*. https://docs.nvidia.com/cuda/cub/. 2022.

[36] NVIDIA. *CUDA Deep Neural Network library*. 2022. URL: https://developer.nvidia.com/cudnn.

[37] NVIDIA. *CUDA Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. 2022.

[38] NVIDIA. *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/. 2022.

[39] NVIDIA. *NVRTC*. https://docs.nvidia.com/cuda/nvrtc. 2022.

[40] NVIDIA. *Parallel Thread Execution ISA*. https://docs.nvidia.com/cuda/parallel-thread-execution. 2022.

[41] NVIDIA. *cuBLAS – BLAS-like Extension*. 2022. URL: https://docs.nvidia.com/cuda/cublas/index.html\#blas-like-extension.

[42] NVIDIA. *cuBLAS – Using the cuBLASLt API*. 2022. URL: https://docs.nvidia.com/cuda/cublas/index.html\#using-the-cublaslt-api.

[43] NVIDIA. *cuBLAS*. https://developer.nvidia.com/cublas. 2022.

[44] OpenMP. *The OpenMP API Specification for Parallel Programming*. https://www.openmp.org. 2022.

[45] Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. "RISE & Shine: Language-Oriented Compiler Design." In: *CoRR* abs/2201.03611 (2022). arXiv: 2201.03611. URL: https://arxiv.org/abs/2201.03611.

[46] TIOBE. *The Software Quality Company*. https://www.tiobe.com/tiobe-index/. 2022.

[47] TensorFlow. *MobileNet v1 models for Keras*. https://github.com/keras-team/keras/blob/master/keras/applications/mobilenet.py. 2022.

[48] TensorFlow. *ResNet models for Keras*. https://github.com/keras-team/keras/blob/master/keras/applications/resnet.py. 2022.

[49] TensorFlow. *VGG16 model for Keras*. https://github.com/keras-team/keras/blob/master/keras/applications/vgg16.py. 2022.

[50] Uday Bondhugula. *Pluto: An automatic polyhedral parallelizer and locality optimizer*. https://github.com/bondhugula/pluto, commit = 12e075a, date = 31.10.2021. 2022.

[51] Nicolas Vasilache et al. *Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction*. 2022. arXiv: 2202.03293 [cs.PL].

[52] Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. *LoopStack: a Lightweight Tensor Algebra Compiler Stack*. 2022. DOI: 10.48550/ARXIV.2205.00618. URL: https://arxiv.org/abs/2205.00618.

[53] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. "DISTAL: The Distributed Tensor Algebra Compiler." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, 286–300. ISBN: 9781450392655. DOI: 10.1145/3519939.3523437. URL: https://doi.org/10.1145/3519939.3523437.

[54] md_poly Artifact Implementation. *https://gitlab.com/mdh-project/benchmarks/2022-md_poly/*. 2022.

[55] Ajay Brahmakshatriya and Saman Amarasinghe. "BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 39–51. DOI: 10.1109/CGO51591.2021.9370333.

[56] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation." In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[57] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. "DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks." In: *IEEE Access* 9 (2021), pp. 134457–134502. DOI: 10.1109/ACCESS.2021.3110993.

[58] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. "Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming." In: *Proc. ACM Program. Lang.* 5.ICFP (2021). DOI: 10.1145/3473593. URL: https://doi.org/10.1145/3473593.

[59] Richard Schulze, **Ari Rasch**, and Sergei Gorlatch. "Code Generation and Optimization for Deep-Learning Computations on GPUs via Multi-Dimensional Homomorphisms *(Best Poster Finalist)*." In: *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, 2 pages. URL: https://sc21.supercomputing.org/proceedings/tech_poster/poster_files/rpost163s2-file3.pdf.

[60] Cambridge Yang, Eric Atkinson, and Michael Carbin. "Simplifying Dependent Reductions in the Polyhedral Model." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434301. URL: https://doi.org/10.1145/3434301.

[61] **Ari Rasch**, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)." In: *ACM Trans. Archit. Code Optim.* 18.1 (Jan. 2021), 26 pages. ISSN: 1544-3566. DOI: 10.1145/3427093. **Rank A**.

[62] ATF Artifact Implementation. *https://gitlab.com/mdh-project/taco2020-atf*. 2020.

[63] Apache TVM Community. *Non top-level reductions in compute statements*. https://discuss.tvm.apache.org/t/non-top-level-reductions-in-compute-statements/5693. 2020.

[64] Uday Bondhugula. *High Performance Code Generation in MLIR: An Early Case Study with GEMM*. 2020. arXiv: 2003.00532 [cs.PF].

[65] Cedric Nugteren. *CLTune Issue*. https://github.com/CNugteren/CLTune/blob/master/src/searchers/annealing.cc#L134 (commit: 2b49667). 2020.

[66] Google SIG MLIR Open Design Meeting. *Using MLIR for Multi-Dimensional Homomorphisms*. 2020. URL: https://www.youtube.com/watch?v=RQR_9tHscMI.

[67] Waldemar Gorus. "pyATF: Auto-Tuning Interdependent Tuning Parameters in Python." Bachelor's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2020.

[68] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies." In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408974. URL: https://doi.org/10.1145/3408974.

[69] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: A Data-Movement-Aware Scheduling Language for GPUs." In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. PACT '20. Virtual Event, GA, USA: Association for Computing Machinery, 2020, 71–82. ISBN: 9781450380751. DOI: 10.1145/3410463.3414632. URL: https://doi.org/10.1145/3410463.3414632.

[70] Mary Hall. "Research Challenges in Compiler Technology for Sparse Tensors." In: *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 2020, pp. viii–viii. DOI: 10.1109/IA351965.2020.00006.

[71] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. "Compiling Generalized Histograms for GPU." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00101.

[72] Sebastian Kock. "Evaluating the MDH Approach for Heterogeneous Multi-Device Systems." Master's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2020.

[73] NVIDIA. *CUDA C++ Best Practices Guide*. 2020.

[74] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. "Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR." In: *Proceedings of the 29th International Conference on Compiler Construction*. CC 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, 85–95. ISBN: 9781450371209. DOI: 10.1145/3377555.3377896. URL: https://doi.org/10.1145/3377555.3377896.

[75] Bertrand Russell. *The principles of mathematics*. Routledge, 2020.

[76] Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. "Meta-programming for cross-domain tensor optimizations." In: *SIGPLAN Not.* 53.9 (2020), 79–92. ISSN: 0362-1340. DOI: 10.1145/3393934.3278131. URL: https://doi.org/10.1145/3393934.3278131.

[77] Moritz Tätweiler. "Evaluating the MDH Approach Based on Frameworks Kokkos, Raja, Occa, and SYCL." Bachelors's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2020.

[78] Lianmin Zheng et al. "Ansor: Generating High-Performance Tensor Programs for Deep Learning." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 863–879. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zheng.

[79] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. "FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, 859–873. ISBN: 9781450371025. URL: https://doi.org/10.1145/3373376.3378508.

[80] pjots. *Sparse Matrix-Vector Multiplication (SpMV) in C*. https://github.com/Sable/fait-maison-spmv/blob/master/src/sequential/c/spmv_csr.c. 2020.

[81] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "md_poly: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms." In: *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. 2020, 4 pages.

[82] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "md_poly: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms." In: *ACM SRC Grand Finals Candidates, 2019 - 2020*. 2020, 5 pages.

[83] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code." In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197.

[84] Paul Barham and Michael Isard. "Machine Learning Systems Are Stuck in a Rut." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, 177–183. ISBN: 9781450367271. DOI: 10.1145/3317550.3321441. URL: https://doi.org/10.1145/3317550.3321441.

[85] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. "Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. 2019.

[86] Azadeh Farzan and Victor Nicolet. "Modular Divide-and-Conquer Parallelization of Nested Loops." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, 610–624. ISBN: 9781450367127. DOI: 10.1145/3314221.3314612. URL: https://doi.org/10.1145/3314221.3314612.

[87] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture." In: *Commun. ACM* 62.2 (2019), 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[88] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. "Incremental Flattening for Nested Data Parallelism." In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, 53–67. ISBN: 9781450362252. DOI: 10.1145/3293883.3295707. URL: https://doi.org/10.1145/3293883.3295707.

[89] Lars Hunloh. "Evaluating the MDH Approach using CUTLASS and PPCG." Bachelors's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2019.

[90] Intel. *Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM)*. https://www.intel.com/content/www/us/en/developer/articles/technical/onemkl-improved-small-matrix-performance-using-just-in-time-jit-code.html. 2019.

[91] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "A Code Generator for High-Performance Tensor Contractions on GPUs." In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 85–95. DOI: 10.1109/CGO.2019.8661182.

[92] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. "A Code Generator for High-performance Tensor Contractions on GPUs." In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 85–95. ISBN: 978-1-7281-1436-1. URL: http://dl.acm.org/citation.cfm?id=3314872.3314885.

[93] Bastian Köpcke, Michel Steuwer, and Sergei Gorlatch. "Generating Efficient FFT GPU Code with Lift." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2019. Berlin, Germany: ACM, 2019, pp. 1–13. ISBN: 978-1-4503-6814-8. DOI: 10.1145/3331553.3342613. URL: http://doi.acm.org/10.1145/3331553.3342613.

[94] Benoît Meister, Eric Papenhausen Akai Kaeru, and Benoît Pradelle Silexica. "Polyhedral Tensor Schedulers." In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 2019, pp. 504–512. DOI: 10.1109/HPCS48598.2019.9188233.

[95] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego R. Llanos. "Multi-device Controllers: A Library to Simplify Parallel Heterogeneous Programming." In: *International Journal of Parallel Programming* 47.1 (2019), pp. 94–113. DOI: 10.1007/s10766-017-0542-x. URL: https://doi.org/10.1007/s10766-017-0542-x.

[96] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. "HyperMapper: a Practical Design Space Exploration Framework." In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecom-*

munication Systems (MASCOTS). 2019, pp. 425–426. DOI: 10.1109/MASCOTS.2019.00053.

[97] OCAL Artifact Implementation. *https://gitlab.com/mdh-project/artifacts/OCAL*. 2019.

[98] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[99] S.J. Pennycook, J.D. Sewall, and V.W. Lee. "Implications of a metric for performance portability." In: *Future Generation Computer Systems* 92 (2019), pp. 947–958. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17300559.

[100] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. "Efficient Hierarchical Online-autotuning: A Case Study on Polyhedral Accelerator Mapping." In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: ACM, 2019, pp. 354–366. ISBN: 978-1-4503-6079-1. DOI: 10.1145/3330345.3330377. URL: http://doi.acm.org/10.1145/3330345.3330377.

[101] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, 65–78. ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. URL: https://doi.org/10.1145/3297858.3304059.

[102] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. "Position-Dependent Arrays and Their Application for High Performance Code Generation." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2019. Berlin, Germany: Association for Computing Machinery, 2019, 14–26. ISBN: 9781450368148. DOI: 10.1145/3331553.3342614. URL: https://doi.org/10.1145/3331553.3342614.

[103] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. "Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons." In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery,

2019, 1534–1543. ISBN: 9781450359337. DOI: 10.1145/3297280.3297434. URL: https://doi.org/10.1145/3297280.3297434.

[104] Lukas Rosendahl. "Evaluating the MDH Approach using Benchmark Suites Parboil and Rodinia." Master's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2019.

[105] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift." In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019). ISSN: 1544-3566. DOI: 10.1145/3368858. URL: https://doi.org/10.1145/3368858.

[106] Huihui Sun, Florian Fey, Jie Zhao, and Sergei Gorlatch. "WCCV: Improving the Vectorization of IF-statements with Warp-coherent Conditions." In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19. Phoenix, Arizona: ACM, 2019, pp. 319–329. ISBN: 978-1-4503-6079-1. DOI: 10.1145/3330345.3331059. URL: http://doi.acm.org/10.1145/3330345.3331059.

[107] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David R. Kaeli. "Summarizing CPU and GPU Design Trends with Product Data." In: *CoRR* abs/1911.11313 (2019). arXiv: 1911.11313. URL: http://arxiv.org/abs/1911.11313.

[108] Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. "Locus: A System and a Language for Program Optimization." In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 217–228. ISBN: 978-1-7281-1436-1.

[109] Thiago SFX Teixeira, William Gropp, and David Padua. "Managing code transformations for better performance portability." In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1290–1306.

[110] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically." In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566. DOI: 10.1145/3355606. URL: https://doi.org/10.1145/3355606.

[111] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically." In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019), 38:1–38:26. ISSN: 1544-3566. DOI: 10.1145/3355606. URL: http://doi.acm.org/10.1145/3355606.

[112] Ben van Werkhoven. "Kernel Tuner: A search-optimizing GPU code auto-tuner." In: *Future Generation Computer Systems* 90 (2019), pp. 347 –358. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18313359.

[113] Arne Wilp. "Accelerating Neural Networks using the MDH Approach." Master's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2019.

[114] **Ari Rasch** and Sergei Gorlatch. "ATF: A generic directive-based auto-tuning framework." In: *Concurrency and Computation: Practice and Experience* 31.5 (2019). e4423 cpe.4423, 14 pages. DOI: https://doi.org/10.1002/cpe.4423. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4423. **Rank A**.

[115] **Ari Rasch**, Richard Schulze, and Sergei Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms." In: *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2019, pp. 354–369. DOI: 10.1109/PACT.2019.00035. **Rank A**.

[116] **Ari Rasch**, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. "High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms." In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 526–533. ISBN: 9781450359337. DOI: 10.1145/3297280.3297330. **Rank B**.

[117] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. "Autotuning in High-Performance Computing Applications." In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083. DOI: 10.1109/JPROC.2018.2841200.

[118] Julian Bigge, Jan Ewald, Timo Hoth, Sebastian Kock, Leon Schöpfner, and Christian Wollny. "Automatic Program Optimization using Auto-Tuning and Machine Learning." Project Seminar, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2018.

[119] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "Learning to Optimize Tensor Programs." In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf.

[120] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[121] Vincent Dumoulin and Francesco Visin. *A guide to convolution arithmetic for deep learning*. 2018. arXiv: 1603.07285 [stat.ML].

[122] August Ernstsson, Lu Li, and Christoph Kessler. "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems." In: *International Journal of Parallel Programming* 46.1 (2018), pp. 62–80. DOI: 10.1007/s10766-017-0490-5. URL: https://doi.org/10.1007/s10766-017-0490-5.

[123] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. "A Programmable Programming Language." In: *Communications of the ACM* 61.3 (2018), pp. 62–71.

[124] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High Performance Stencil Code Generation with Lift." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, 100–112. ISBN: 9781450356176. DOI: 10.1145/3168824. URL: https://doi.org/10.1145/3168824.

[125] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. "High Performance Stencil Code Generation with Lift." In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: ACM, 2018, pp. 100–112. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168824. URL: http://doi.acm.org/10.1145/3168824.

[126] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. "Matrix capsules with EM routing." In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=HJWLfGWRb.

[127] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. "Autotune: A Derivative-Free Optimization Framework for Hyperparameter Tuning." In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '18. London, United Kingdom: Association for Computing Machinery, 2018, pp. 443–452. ISBN: 9781450355520. DOI: 10.1145/3219819.3219837. URL: https://doi.org/10.1145/3219819.3219837.

[128] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. "AnyDSL: a partial evaluation framework for programming high-performance libraries." In: *Proc. ACM*

Program. Lang. 2.OOPSLA (2018). DOI: 10.1145/3276489. URL: https://doi.org/10.1145/3276489.

[129] NVIDIA. *Warp-level Primitives*. 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[130] Gustavo Niemeyer. *python-constraint*. English. 2018. URL: https://pypi.org/project/python-constraint/.

[131] Cedric Nugteren. "CLBlast: A Tuned OpenCL BLAS Library." In: *Proceedings of the International Workshop on OpenCL*. ACM. 2018, 1–10.

[132] OpenTuner. *Interdependent Tuning Parameters (Issue 106)*. https://github.com/jansel/opentuner/issues/106. 2018.

[133] Richard Schulze. "Design and Implementation of a Performance-Portable BLAS Library via Multi-Dimensional Homomorphisms." Master's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2018.

[134] Akshitha Sriraman and Thomas F. Wenisch. "µTune: Auto-Tuned Threading for OLDI Microservices." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 177–194. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/sriraman.

[135] Luis Wetzel. "Evaluating Multi-Dimensional Homomorphisms via Ensemble Classifier Chains." Bachelor's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2018.

[136] Martin Wrodarczyk. "ECC Classification via Support Vector Machines and Multilayer Perceptron." Master's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2018.

[137] Vasileios Zois, Divya Gupta, Vassilis J. Tsotras, Walid A. Najjar, and Jean-Francois Roy. "Massively Parallel Skyline Computation for Processing-in-Memory Architectures." In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243187. URL: https://doi.org/10.1145/3243176.3243187.

[138] **Ari Rasch** and Sergei Gorlatch. "Multi-dimensional Homomorphisms and Their Implementation in OpenCL." In: *International Journal of Parallel Programming* 46.1 (2018), pp. 101–119. ISSN: 1573-7640. DOI: 10.1007/s10766-017-0508-z. **Rank A**.

[139] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. "Fastflow: high-level and efficient streaming on multi-core." In: *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

[140] David Beckingsale, Olga Pearce, Ignacio Laguna, and Todd Gamblin. "Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code." In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 307–316.

[141] Steffen Ernsting and Herbert Kuchen. "Data Parallel Algorithmic Skeletons with Accelerator Support." In: *International Journal of Parallel Programming* 45.2 (2017), pp. 283–299.

[142] Michael Gomulak. "Desing and Implementation of an OpenCL-to-CUDA Translator." Bachelor's Thesis, supervised by: Ari Rasch, Richard Schulze, Sergei Gorlatch. University of Muenster, 2017.

[143] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, 556–571. ISBN: 9781450349888. DOI: 10.1145/3062341.3062354. URL: https://doi.org/10.1145/3062341.3062354.

[144] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[145] Intel. *Educational Resources*. https://software.intel.com/en-us/intel-opencl-support/code-samples. 2017.

[146] Jason Ansel. *OpenTuner*. https://github.com/jansel/opentuner. 2017.

[147] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption." In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ARMS-CC '17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 1–6. ISBN: 9781450351164. DOI: 10.1145/3110355.3110356. URL: https://doi.org/10.1145/3110355.3110356.

[148] NVIDIA. *Programming Tensor Cores in CUDA 9*. 2017. URL: https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[149] NVIDIA. *Unified Memory for CUDA Beginners*. 2017. URL: https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[150] Philip Pfaffe, Martin Tillmann, Sigmar Walter, and Walter F. Tichy. "Online-Autotuning in the Presence of Algorithmic Choice." In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2017, pp. 1379–1388. DOI: 10.1109/IPDPSW.2017.28.

[151] Mona Riemenschneider, Alexander Herbst, **Ari Rasch**, Sergei Gorlatch, and Dominik Heider. "eccCL: parallelized GPU implementation of Ensemble Classifier Chains." In: *BMC Bioinformatics* 18.1 (2017), p. 371. DOI: 10.1186/s12859-017-1783-9. **Rank A**.

[152] Mohammed Sourouri, Espen Birger Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, and Per Gunnar Kjeldsberg. "Towards Fine-grained Dynamic Tuning of HPC Applications on Modern Multi-core Architectures." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, pp. 1–12.

[153] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "LIFT: A functional data-parallel IR for high-performance GPU code generation." In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 74–85. DOI: 10.1109/CGO.2017.7863730.

[154] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. "Controlled Kernel Launch for Dynamic Parallelism in GPUs." In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 649–660. DOI: 10.1109/HPCA.2017.14.

[155] Philippe Tillet and David Cox. "Input-Aware Auto-Tuning of Compute-Bound HPC Kernels." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126939. URL: https://doi.org/10.1145/3126908.3126939.

[156] Philippe Tillet and David Cox. "Input-Aware Auto-Tuning of Compute-bound HPC Kernels." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2017, pp. 1–12.

[157] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[158] M. Ahmad and O. Khan. "GPU concurrency choices in graph analytics." In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016, pp. 1–10.

[159] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. "Opening polyhedral compiler's black box." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: Association for Computing Machinery, 2016, 128–138. ISBN: 9781450337786. DOI: 10.1145/2854038.2854048. URL: https://doi.org/10.1145/2854038.2854048.

[160] David Castro, Kevin Hammond, and Susmit Sarkar. "Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes Using Types and Hylomorphisms." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 4–17. ISBN: 9781450342193. DOI: 10.1145/2951913.2951920. URL: https://doi.org/10.1145/2951913.2951920.

[161] Cedric Nugteren. *CLTune Issue 48*. 2016. URL: github.com/CNugteren/CLTune/issues/48.

[162] Alexander Herbst. "Parallelization of Ensemble Classifier Chains for GPUs." Master's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2016.

[163] Z. Jia, C. Xue, G. Chen, J. Zhan, L. Zhang, Y. Lin, and P. Hofstee. "Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading." In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2016, pp. 387–400.

[164] Fabian Kip. "A Multi-Device OpenCL Implementation for Matrix-Vector Multiplication on Heterogeneous Systems." Bachelor's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2016.

[165] Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, and Weimin Zheng. "Performance Analysis of GPU-Based Convolutional Neural Networks." In: *2016 45th International Conference on Parallel Processing (ICPP)*. 2016, pp. 67–76. DOI: 10.1109/ICPP.2016.15.

[166] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically Scheduling Halide Image Processing Pipelines." In: *ACM Trans. Graph.* 35.4 (2016). ISSN: 0730-0301. DOI: 10.1145/2897824.2925952. URL: https://doi.org/10.1145/2897824.2925952.

[167] Borja Pérez, José Luis Bosque, and Ramón Beivide. "Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU '16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 42–51. ISBN: 9781450341950. DOI: 10.1145/2884045.2884051. URL: https://doi.org/10.1145/2884045.2884051.

[168] Chandan Reddy, Michael Kruse, and Albert Cohen. "Reduction Drawing: Language Constructs and Polyhedral Compilation for Reductions on GPU." In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: Association for Computing Machinery, 2016, 87–97. ISBN: 9781450341219. DOI: 10.1145/2967938.2967950. URL: https://doi.org/10.1145/2967938.2967950.

[169] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. "Performance Portable GPU Code Generation for Matrix Multiplication." In: *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*. GPGPU '16. Barcelona, Spain: Association for Computing Machinery, 2016, 22–31. ISBN: 9781450341950. DOI: 10.1145/2884045.2884046. URL: https://doi.org/10.1145/2884045.2884046.

[170] Paul Springer and Paolo Bientinesi. "Design of a high-performance GEMM-like Tensor-Tensor Multiplication." In: *CoRR* (2016). arXiv: 1607.00145 [quant-ph]. URL: http://arxiv.org/abs/1607.00145.

[171] Steve Arnold. *CCCC Project Documentation*. http://sarnold.github.io/cccc/. 2016.

[172] StreamHPC. *Comparing Syntax for CUDA, OpenCL and HiP*. https://streamhpc.com/blog/2016-04-05/comparing-syntax-cuda-opencl-hip/. 2016.

[173] Jakub Szuppe. "Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL." In: *Proceedings of the 4th International Workshop on OpenCL*. IWOCL '16. Vienna, Austria: Association for Computing Machinery, 2016. ISBN: 9781450343381. DOI: 10.1145/2909437.2909454. URL: https://doi.org/10.1145/2909437.2909454.

[174] N. Vijaykumar, K. Hsieh, G. Pekhimenko, S. Khan, A. Shrestha, S. Ghose, A. Jog, P. B. Gibbons, and O. Mutlu. "Zorua: A holistic approach to resource virtualization in GPUs." In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–14.

[175] rharish100193. *Halstead Metrics Tool*. https://sourceforge.net/projects/halsteadmetricstool/. 2016.

[176] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Guilherme Peretti Pezzi, and Massimo Torquati. "The Loop-of-Stencil-Reduce Paradigm." In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 3. 2015, pp. 172–177. DOI: 10.1109/Trustcom.2015.628.

[177] Richard F. Barrett, Dylan T. Stark, Courtenay T. Vaughan, Ryan E. Grant, Stephen L. Olivier, and Kevin T. Pedretti. "Toward an Evolutionary Task Parallel Integrated MPI + X Programming Model." In: *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Many-*

cores. PMAM '15. San Francisco, California: Association for Computing Machinery, 2015, pp. 30–39. ISBN: 9781450334044. DOI: 10.1145/2712386.2712388. URL: https://doi.org/10.1145/2712386.2712388.

[178] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. *Polly's Polyhedral Scheduling in the Presence of Reductions*. 2015. arXiv: 1505.07716 [cs.PL].

[179] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. "Polly's Polyhedral Scheduling in the Presence of Reductions." In: *CoRR* abs/1505.07716 (2015). arXiv: 1505.07716. URL: http://arxiv.org/abs/1505.07716.

[180] Kevin Gehling. "Implementing Fast Fourier Transformation in SkelCL." Bachelors's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2015.

[181] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. "Polyhedral AST Generation Is More Than Scanning Polyhedra." In: *ACM Trans. Program. Lang. Syst.* 37.4 (2015). ISSN: 0164-0925. DOI: 10.1145/2743016. URL: https://doi.org/10.1145/2743016.

[182] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[183] Torsten Hoefler and Roberto Belli. "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807644. URL: https://doi.org/10.1145/2807591.2807644.

[184] B. Janßen et al. "Designing applications for heterogeneous many-core architectures with the FlexTiles Platform." In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 254–261.

[185] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image Recognition." In: *ICML deep learning workshop*. Vol. 2. 2015, pp. 1–30.

[186] Felix Krull. "Evaluating the SkelCL Library using Discrete Cosine Transform and Histograms." Bachelors's Thesis, supervised by: Ari Rasch, Sergei Gorlatch. University of Muenster, 2015.

[187] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[188] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. "PolyMage: Automatic Optimization for Image Processing Pipelines." In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, pp. 429–443. ISBN: 9781450328357. DOI: 10.1145/2694344.2694364. URL: https://doi.org/10.1145/2694344.2694364.

[189] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris. "Generating Efficient Tensor Contractions for GPUs." In: *2015 44th International Conference on Parallel Processing*. 2015, pp. 969–978.

[190] Cedric Nugteren and Valeriu Codreanu. "CLTune: A Generic Auto-Tuner for OpenCL Kernels." In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE. 2015, pp. 195–202.

[191] Ruyman Reyes et al. "SYCL: Single-source C++ accelerator programming." In: *PARCO*. 2015, pp. 673–682.

[192] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. "Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code." In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, 205–217. ISBN: 9781450336697. DOI: 10.1145/2784731.2784754. URL: https://doi.org/10.1145/2784731.2784754.

[193] Moisés Viñas, Basilio B. Fraguela, Zeki Bozkus, and Diego Andrade. "Improving OpenCL Programmability with the Heterogeneous Programming Library." In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 110–119. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.208. URL: https://www.sciencedirect.com/science/article/pii/S1877050915010169.

[194] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning." In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. Edmonton, AB, Canada: Association for Computing Machinery, 2014, 303–316. ISBN: 9781450328098. DOI: 10.1145/2628071.2628092. URL: https://doi.org/10.1145/2628071.2628092.

[195] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning." In: *Proceedings of the 23rd international*

conference on Parallel architectures and compilation. ACM. 2014, pp. 303–316.

[196] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731514001257.

[197] Michael Haidl and Sergei Gorlatch. "PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14." In: *2014 LLVM Compiler Infrastructure in HPC*. 2014, pp. 1–11. DOI: 10.1109/LLVM-HPC.2014.9.

[198] Richard D. Hornung and Jeffrey A. Keasler. "The RAJA Portability Layer: Overview and Status." In: (Sept. 2014). DOI: 10.2172/1169830. URL: https://www.osti.gov/biblio/1169830.

[199] Intel. *Ambient Occlusion Benchmark (AOBench)*. 2014. URL: http://code.google.com/p/aobench.

[200] Intel. *How to Increase Performance by Minimizing Buffer Copies on Intel Processor Graphics*. https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics. 2014.

[201] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding." In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[202] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding." In: *Proceedings of the 22nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: Association for Computing Machinery, 2014, pp. 675–678. ISBN: 9781450330633. DOI: 10.1145/2647868.2654889. URL: https://doi.org/10.1145/2647868.2654889.

[203] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. "Benchmarking the Memory Hierarchy of Modern GPUs." In: *Network and Parallel Computing*. Ed. by Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 144–156. ISBN: 978-3-662-44917-2.

[204] Saurav Muralidharan, Manu Shantharam, Mary Hall, Michael Garland, and Bryan Catanzaro. "Nitro: A Framework for Adaptive Code Variant Tuning." In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE. 2014, pp. 501–512.

[205] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556.

[206] Mohamed Wahib and Naoya Maruyama. "Scalable Kernel Fusion for Memory-Bound GPU Applications." In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 191–202. DOI: 10.1109/SC.2014.21.

[207] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. "Towards making autotuning mainstream." In: *The International Journal of High Performance Computing Applications* 27.4 (2013), pp. 379–393. DOI: 10.1177/1094342013493644.

[208] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Ricardo Nobre, Razvan Nane, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Koen Bertels. "Controlling a complete hardware synthesis toolchain with LARA aspects." In: *Microprocessors and Microsystems* 37.8, Part C (2013). Special Issue on European Projects in Embedded System Design: EPESD2012, pp. 1073 –1089. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2013.06.001. URL: http://www.sciencedirect.com/science/article/pii/S0141933113000859.

[209] Pascal Getreuer. "A Survey of Gaussian Convolution Algorithms." In: *Image Processing On Line* 3 (2013). https://doi.org/10.5201/ipol.2013.87, pp. 286–310.

[210] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. "LibWater: Heterogeneous Distributed Computing Made Easy." In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS '13. Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 161–172. ISBN: 9781450321303. DOI: 10.1145/2464996.2465008. URL: https://doi.org/10.1145/2464996.2465008.

[211] Haskell Wiki. *Parameter Order*. 2013. URL: https://wiki.haskell.org/Parameter_order.

[212] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. "A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code." In: *ACM Trans. Archit. Code Optim.* 9.4 (2013). ISSN: 1544-3566. DOI: 10.1145/2400682.2400690. URL: https://doi.org/10.1145/2400682.2400690.

[213] A. E. Kiasari, Z. Lu, and A. Jantsch. "An Analytical Latency Model for Networks-on-Chip." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.1 (2013), pp. 113–123.

[214] Junjie Lai and André Seznec. "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs." In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, pp. 1–10. DOI: 10.1109/CGO.2013.6494986.

[215] Alberto Magni, Dominik Grewe, and Nick Johnson. "Input-Aware Auto-Tuning for Directive-Based GPU Programming." In: *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. GPGPU-6. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 66–75. ISBN: 9781450320177. DOI: 10.1145/2458523.2458530. URL: https://doi.org/10.1145/2458523.2458530.

[216] NVIDIA. *Hyper-Q*. http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf. 2013.

[217] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[218] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral Parallel Code Generation for CUDA." In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013). ISSN: 1544-3566. DOI: 10.1145/2400682.2400713. URL: https://doi.org/10.1145/2400682.2400713.

[219] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA." In: *ACM Transactions on Architecture and Code Optimization* 9.4 (2013), 54:1–54:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400713.

[220] José María Cecilia, José Manuel García, and Manuel Ujaldón. "CUDA 2D Stencil Computations for the Jacobi Method." In: *Applied Parallel and Scientific Computing*. Ed. by Kristján Jónasson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 173–183. ISBN: 978-3-642-28151-8.

[221] Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642311636.

[222] Usman Dastgeer, Lu Li, and Christoph Kessler. "The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-Based Systems." In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012, pp. 711–720. DOI: 10.1109/SC.Companion.2012.97.

[223] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming." In: *Parallel Computing* 38.8 (2012). APPLICATION ACCELERATORS IN HPC, pp. 391–407. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0167819111001335.

[224] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation." In: *Parallel Processing Letters* 22.04 (2012), p. 1250010. DOI: 10.1142/S0129626412500107. eprint: https://doi.org/10.1142/S0129626412500107. URL: https://doi.org/10.1142/S0129626412500107.

[225] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. "dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. 2012, pp. 174–186. DOI: 10.1109/IPDPSW.2012.16.

[226] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters." In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, 2012, pp. 341–352. ISBN: 9781450313162. DOI: 10.1145/2304576.2304623. URL: https://doi.org/10.1145/2304576.2304623.

[227] Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. "Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures." In: *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*. IWOMP'12. Rome, Italy: Springer-Verlag, 2012, 59–72. ISBN: 9783642309601. DOI: 10.1007/978-3-642-30961-8_5. URL: https://doi.org/10.1007/978-3-642-30961-8_5.

[228] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation." In: *Parallel Computing* 38.3 (2012), pp. 157–174. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.

2011.09.001. URL: https://www.sciencedirect.com/science/article/pii/S0167819111001281.

[229] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[230] Junjie Lai and André Seznec. "Bound the Peak Performance of SGEMM on GPU with software-controlled fast memory." In: *[Research Report] RR-7923, 2012. hal-00686006v1* (2012).

[231] NVIDIA. *How to Optimize Data Transfers in CUDA C/C++*. https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/. 2012.

[232] NVIDIA. *How to Overlap Data Transfers in CUDA C/C++*. https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/. 2012.

[233] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. "Performance Gaps between OpenMP and OpenCL for Multi-core CPUs." In: *2012 41st International Conference on Parallel Processing Workshops*. 2012, pp. 116–125. DOI: 10.1109/ICPPW.2012.18.

[234] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. "Parboil: A revised benchmark suite for scientific and commercial throughput computing." In: *Center for Reliable and High-Performance Computing* 127 (2012).

[235] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool." In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France, 2012. URL: http://impact.gforge.inria.fr/impact2012/workshop_IMPACT/verdoolaege.pdf.

[236] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool." In: *International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*. Vol. 141. 2012.

[237] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198. DOI: https://doi.org/10.1002/cpe.1631. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631.

[238] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. "Accelerating Haskell Array Codes with Multicore GPUs." In: *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*. DAMP '11. Austin, Texas, USA: Association for Computing Machinery, 2011, 3–14. ISBN: 9781450304863. DOI: 10.1145/1926354.1926358. URL: https://doi.org/10.1145/1926354.1926358.

[239] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 676–687.

[240] ALEJANDRO DURAN, EDUARD AYGUADÉ, ROSA M. BADIA, JESÚS LABARTA, LUIS MARTINELL, XAVIER MARTORELL, and JUDIT PLANAS. "OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES." In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151. eprint: https://doi.org/10.1142/S0129626411000151. URL: https://doi.org/10.1142/S0129626411000151.

[241] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. ""Milepost GCC: Machine Learning Enabled Self-tuning Compiler"." In: *International journal of parallel programming* 39.3 (2011), pp. 296–327.

[242] Sergei Gorlatch and Murray Cole. "Parallel Skeletons." English. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer-Verlag GmbH, 2011, pp. 1417–1422. ISBN: 9780387097657. DOI: 10.1007/978-0-387-09766-4_24.

[243] Sergei Gorlatch and Murray Cole. "Parallel skeletons." In: *Encyclopedia of parallel computing*. Springer-Verlag GmbH, 2011, pp. 1417–1422.

[244] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. "Classifier chains for multi-label classification." In: *Machine Learning* 85.3 (2011), pp. 333–359. DOI: 10.1007/s10994-011-5256-5. URL: https://doi.org/10.1007/s10994-011-5256-5.

[245] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, pp. 1176–1182. DOI: 10.1109/IPDPS.2011.269.

[246] Joel Svensson, Mary Sheeran, and Koen Claessen. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors." In: *Implementation and Application of Functional Languages*. Ed. by Sven-Bodo Scholz and Olaf

Chitil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 156–173. ISBN: 978-3-642-24452-0.

[247] Enric Tejedor, Montse Farreras, David Grove, Rosa M. Badia, Gheorghe Almasi, and Jesus Labarta. "ClusterSs: A Task-Based Programming Model for Clusters." In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. HPDC '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 267–268. ISBN: 9781450305525. DOI: 10.1145/1996130.1996168. URL: https://doi.org/10.1145/1996130.1996168.

[248] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. "Dynamic load balancing on single- and multi-GPU systems." In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470413.

[249] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters." In: *2010 International Conference on High Performance Computing Simulation*. 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126.

[250] Johan Enmyren and Christoph W. Kessler. "SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems." In: *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*. HLPP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, 5–14. ISBN: 9781450302548. DOI: 10.1145/1863482.1863487. URL: https://doi.org/10.1145/1863482.1863487.

[251] Horacio González-Vélez and Mario Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers." In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160. DOI: https://doi.org/10.1002/spe.1026. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1026. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1026.

[252] Seyong Lee and Rudolf Eigenmann. "OpenMPC: Extended OpenMP Programming and Tuning for GPUs." In: *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11. DOI: 10.1109/SC.2010.36.

[253] Karl Rupp, Josef Weinbub, and Florian Rudolf. "Automatic Performance Optimization in ViennaCL for GPUs." In: *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*. POOSC '10. Reno, Nevada: Association for Computing Machinery, 2010. ISBN: 9781450305464. DOI: 10.1145/2039312.2039318. URL: https://doi.org/10.1145/2039312.2039318.

[254] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. "Maestro: Data Orchestration and Tuning for OpenCL Devices." In: *Euro-Par 2010 - Parallel Processing*. Ed. by Pasqua D'Ambra, Mario Guarracino, and Domenico Talia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 275–286. ISBN: 978-3-642-15291-7.

[255] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software – ICMS 2010*. Ed. by Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15582-6.

[256] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[257] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. "Annotation-Based Empirical Performance Tuning Using Orio." In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009, pp. 1–11.

[258] D. Schaa and D. Kaeli. "Exploring the multiple-GPU design space." In: *2009 IEEE International Symposium on Parallel Distributed Processing*. 2009, pp. 1–12.

[259] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. "Tuning parallel applications in parallel." In: *Parallel Computing* 35.8 (2009), pp. 475 –492. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2009.07.001. URL: http://www.sciencedirect.com/science/article/pii/S0167819109000805.

[260] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 101–113. ISBN: 9781595938602. DOI: 10.1145/1375581.1375595. URL: https://doi.org/10.1145/1375581.1375595.

[261] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model." In: *Compiler Construction*. Ed. by Laurie Hendren. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 132–146. ISBN: 978-3-540-78791-4.

[262] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. "Pluto: A practical and fully automatic polyhedral program optimization system." In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer. 2008.

[263] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A Framework for Composing High-Level Loop Transformations*. Tech. rep. Citeseer, 2008, pp. 0–27.

[264] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication." In: *ACM Trans. Math. Softw.* 34.3 (2008). ISSN: 0098-3500. DOI: 10.1145/1356052.1356053. URL: https://doi.org/10.1145/1356052.1356053.

[265] K Hentschel et al. "Das Krebsregister-Manual der Gesellschaft der epidemiologischen Krebsregister in Deutschland e.V." In: Zuckschwerdt Verlag, 2008.

[266] Riccardo Poli. "Analysis of the publications on the applications of particle swarm optimisation." In: *Journal of Artificial Evolution and Applications* 2008 (2008), pp. 1–10.

[267] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideovt, Ernest Bassous, and Andre R Leblanc. "Design of ion-implanted MOSFET's with very small physical dimensions." In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 38–50.

[268] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, and Olivier Temam. "Fast Compiler Optimisation Evaluation Using Code-feature Based Performance Prediction." In: *Proceedings of the 4th international conference on Computing frontiers*. ACM. 2007, pp. 131–142.

[269] Mark Harris et al. "Optimizing Parallel Reduction in CUDA." In: *NVIDIA Developer Technology* (2007).

[270] Victor Podlozhnyuk. "Image Convolution with CUDA." In: *NVIDIA Corporation White Paper* (2007).

[271] Ramtin Shams, RA Kennedy, et al. "Efficient histogram algorithms for NVIDIA CUDA compatible devices." In: *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*. Citeseer. 2007, pp. 418–422.

[272] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. "Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies." In: *International Journal of Parallel Programming* 34.3 (2006), pp. 261–317. DOI: 10.1007/s10766-006-0012-3. URL: https://doi.org/10.1007/s10766-006-0012-3.

[273] Pavol Hell. "From Graph Colouring to Constraint Satisfaction: There and Back Again." In: *Topics in Discrete Mathematics*. Ed. by Martin Klazar, Jan Kratochvíl, Martin Loebl, Jiří Matoušek, Pavel Valtr, and Robin Thomas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 407–432. ISBN: 978-3-540-33700-3.

[274] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[275] Gerald Baumgartner, Alexander Auer, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert J Harrison, So Hirata, Sriram Krishnamoorthy, et al. "Synthesis of High-Performance Parallel Programs for a Class of ab Initio Quantum Chemistry Models." In: *Proceedings of the IEEE* 93.2 (2005), pp. 276–292.

[276] Matteo Frigo and Steven G Johnson. "The Design and Implementation of FFTW3." In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[277] M. Puschel et al. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275. DOI: 10.1109/JPROC.2004.840306.

[278] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275.

[279] David A. Wheeler. *SLOCCount*. https://www.dwheeler.com/sloccount/. 2004.

[280] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. "Fast Searches for Effective Optimization Phase Sequences." In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI '04. Washington DC, USA: Association for Computing Machinery, 2004, pp. 171–182. ISBN: 1581138075. DOI: 10.1145/996841.996863. URL: https://doi.org/10.1145/996841.996863.

[281] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis amp; transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[282] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[283] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. "FLAME: Formal Linear Algebra Methods Environment." In: *ACM Trans. Math. Softw.* 27.4 (2001), 422–455. ISSN: 0098-3500. DOI: 10.1145/504210.504213. URL: https://doi.org/10.1145/504210.504213.

[284] Bruce Sagan. *The symmetric group: representations, combinatorial algorithms, and symmetric functions*. Vol. 203. Springer Science & Business Media, 2001.

[285] Maurice V Wilkes. "The memory gap and the future of high performance memories." In: *ACM SIGARCH Computer Architecture News* 29.1 (2001), pp. 2–7.

[286] T Daniel Crawford and Henry F Schaefer. "An introduction to coupled cluster theory for computational chemists." In: *Reviews in computational chemistry* 14 (2000), pp. 33–136.

[287] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. "Constraint satisfaction problems: Algorithms and applications." In: *European Journal of Operational Research* 119.3 (1999), pp. 557–581. ISSN: 0377-2217. DOI: https://doi.org/10.1016/S0377-2217(98)00364-6. URL: https://www.sciencedirect.com/science/article/pii/S0377221798003646.

[288] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-oblivious algorithms." In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 1999, pp. 285–297. DOI: 10.1109/SFFCS.1999.814600.

[289] Sergei Gorlatch. "Extracting and implementing list homomorphisms in parallel program development." In: *Science of Computer Programming* 33.1 (1999), pp. 1–27. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(97)00014-2. URL: https://www.sciencedirect.com/science/article/pii/S0167642397000142.

[290] Stephen Wright and Jorge Nocedal. "Numerical Optimization." In: *Springer Science* 35.67–68 (1999), p. 7.

[291] Hongwei Xi and Frank Pfenning. "Dependent Types in Practical Programming." In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, 214–227. ISBN: 1581130953. DOI: 10.1145/292540.292560. URL: https://doi.org/10.1145/292540.292560.

[292] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. "An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions." In: *J. ACM* 45.6 (Nov. 1998), pp. 891–923. ISSN: 0004-5411. DOI: 10.1145/293347.293348. URL: https://doi.org/10.1145/293347.293348.

[293] M. Frigo and S.G. Johnson. "FFTW: an adaptive software architecture for the FFT." In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*. Vol. 3. 1998, 1381–1384 vol.3. DOI: 10.1109/ICASSP.1998.681704.

[294] Wayne Kelly and William Pugh. *A framework for unifying reordering transformations*. Tech. rep. Technical Report UMIACS-TR-92-126.1, 1998.

[295] Gordon E Moore. "Cramming more components onto integrated circuits." In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[296] R Clinton Whaley and Jack J Dongarra. "Automatically Tuned Linear Algebra Software." In: *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE. 1998, p. 38.

[297] R.C. Whaley and J.J. Dongarra. "Automatically Tuned Linear Algebra Software." In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998, pp. 38–38. DOI: 10.1109/SC.1998.10004.

[298] Krzysztof Ciesielski. *Set theory for the working mathematician*. 39. Cambridge University Press, 1997.

[299] S. Gorlatch and C. Lengauer. "(De) composition rules for parallel scan and reduction." In: *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No.97TB100228)*. 1997, pp. 23–32. DOI: 10.1109/MPPM.1997.715958.

[300] Cristina Hristea, Daniel Lenoski, and John Keen. "Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks." In: *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*. SC '97. San Jose, CA: Association for Computing Machinery, 1997, 1–12. ISBN: 0897919858. DOI: 10.1145/509593.509638. URL: https://doi.org/10.1145/509593.509638.

[301] Per Stenström and Jonas Skeppstedt. "A performance tuning approach for shared-memory multiprocessors." In: *Euro-Par'97 Parallel Processing*. Ed. by Christian Lengauer, Martin Griebl, and Sergei Gorlatch. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 72–83. ISBN: 978-3-540-69549-3.

[302] Walid Taha and Tim Sheard. "Multi-Stage Programming with Explicit Annotations." In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '97. Amsterdam, The Netherlands: Association for Computing Machinery, 1997, 203–217. ISBN: 0897919173. DOI: 10.1145/258993.259019. URL: https://doi.org/10.1145/258993.259019.

[303] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. "Improving Data Locality with Loop Transformations." In: *ACM Trans. Program. Lang. Syst.* 18.4 (1996), 424–453. ISSN: 0164-0925. DOI: 10.1145/233561.233564. URL: https://doi.org/10.1145/233561.233564.

[304] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. "Cost models for future software life cycle processes: COCOMO 2.0." In: *Annals of Software Engineering* 1.1 (1995), pp. 57–94. DOI: 10.1007/BF02249046. URL: https://doi.org/10.1007/BF02249046.

[305] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. "Cost models for future software life cycle processes: COCOMO 2.0." In: *Annals of Software Engineering* 1.1 (1995), pp. 57–94. DOI: 10.1007/BF02249046. URL: https://doi.org/10.1007/BF02249046.

[306] MURRAY I. COLE. "PARALLEL PROGRAMMING WITH LIST HOMOMORPHISMS." In: *Parallel Processing Letters* 05.02 (1995), pp. 191–203. DOI: 10.1142/S0129626495000175. eprint: https://doi.org/10.1142/S0129626495000175. URL: https://doi.org/10.1142/S0129626495000175.

[307] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing." In: *ACM Comput. Surv.* 26.4 (1994), 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: https://doi.org/10.1145/197405.197406.

[308] Xavier Redon and Paul Feautrier. "Detection of recurrences in sequential programs with loops." In: *International Conference on Parallel Architectures and Languages Europe*. Springer. 1993, pp. 132–145.

[309] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time." In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. DOI: 10.1007/BF01407835. URL: https://doi.org/10.1007/BF01407835.

[310] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimizations of Blocked Algorithms." In: *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, 63–74. ISBN: 0897913809. DOI: 10.1145/106972.106981. URL: https://doi.org/10.1145/106972.106981.

[311] M.E. Wolf and M.S. Lam. "A loop transformation theory and an algorithm to maximize parallelism." In: *IEEE Transactions on Parallel and Distributed Systems* 2.4 (1991), pp. 452–471. DOI: 10.1109/71.97902.

[312] Guy E. Blelloch. *Prefix Sums and Their Applications*. Tech. rep. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[313] Richard S. Bird. "Lectures on Constructive Functional Programming." In: *Constructive Methods in Computing Science*. Ed. by Manfred Broy. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 151–217. ISBN: 978-3-642-74884-4.

[314] P. P. Chang and W.-W. Hwu. "Inline Function Expansion for Compiling C Programs." In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*. PLDI '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 246–257. ISBN: 089791306X. DOI:

10.1145/73141.74840. URL: https://doi.org/10.1145/73141.74840.

[315] E Oran Brigham. *The Fast Fourier Transform and Its Applications*. Prentice-Hall, Inc., 1988.

[316] Richard S Bird. "An introduction to the theory of lists." In: *Logic of Programming and Calculi of Discrete Design: International Summer School directed by FL Bauer, M. Broy, EW Dijkstra, CAR Hoare*. Springer, 1987, pp. 5–42.

[317] Michael Wolfe. "Loops skewing: The wavefront method revisited." In: *International Journal of Parallel Programming* 15.4 (1986), pp. 279–293. DOI: 10.1007/BF01407876. URL: https://doi.org/10.1007/BF01407876.

[318] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing." In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. eprint: https://www.science.org/doi/pdf/10.1126/science.220.4598.671. URL: https://www.science.org/doi/abs/10.1126/science.220.4598.671.

[319] Gordon D Plotkin. *A Structural Approach to Operational Semantics*. Aarhus university, 1981.

[320] Haskell B. Curry. "Some Philosophical Aspects of Combinatory Logic." In: *The Kleene Symposium*. Ed. by Jon Barwise, H. Jerome Keisler, and Kenneth Kunen. Vol. 101. Studies in Logic and the Foundations of Mathematics. Elsevier, 1980, pp. 85–101. DOI: https://doi.org/10.1016/S0049-237X(08)71254-0. URL: https://www.sciencedirect.com/science/article/pii/S0049237X08712540.

[321] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage." In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[322] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[323] Maurice H Halstead. *Elements of software science*. Elsevier computer science library: operational programming systems series. 1977.

[324] T.J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[325] N. Ahmed, T. Natarajan, and K.R. Rao. "Discrete Cosine Transform." In: *IEEE Transactions on Computers* C-23.1 (1974), pp. 90–93. DOI: 10.1109/T-C.1974.223784.

[326] Dana Scott. *Outline of a Mathematical Theory of Computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[327] J. von Neumann. "Eine Axiomatisierung der Mengenlehre." In: 1925.154 (1925), pp. 219–240. DOI: doi:10.1515/crll.1925.154.219. URL: https://doi.org/10.1515/crll.1925.154.219.

[328] *A "Hands-on" Introduction to OpenMP*. https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf.

[329] *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[330] *ACM SIGPLAN Int. Conf. on Compiler Construction (CC)*.

[331] *ACM/IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*.

[332] *Boost.Asio*. https://www.boost.org/doc/libs/1_75_0/doc/html/boost_asio.html.

[333] *Compilers for Machine Learning (C4ML)*.

[334] *European Forum for Experts in Computer Architecture, Programming Models, Compilers and Operating Systems for Embedded and General-Purpose Systems (HiPEAC)*.

[335] *IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*.

[336] *Int. Workshop on Polyhedral Compilation Techniques (IMPACT)*.

[337] *Intel Math Kernel Library for Deep Learning Networks*. URL: https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation.

[338] *Intel Math Kernel Library*. URL: https://software.intel.com/en-us/mkl.

[339] *Intel oneMKL GEMM*. https://oneapi-src.github.io/oneMKL/domains/blas/gemm.html#onemkl-blas-gemm.

[340] *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[341] *KTT - Kernel Tuning Toolkit*. https://github.com/HiPerCoRe/KTT.

[342] *Khronos OpenCL*. https://www.khronos.org/opencl/.

[343] *Khronos SYCL*. https://www.khronos.org/sycl/.

[344] *MPI Forum*. https://www.mpi-forum.org.

[345] *NVIDIA Ampere GPU Architecture*. https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html.

[346] *NVIDIA CUDA Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[347] *NVIDIA CUDA Samples – Simple MPI*. https://github.com/NVIDIA/cuda-samples/blob/master/Samples/simpleMPI/simpleMPI.cpp.

[348] *NVIDIA CUDA Samples – Simple Multi GPU*. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/simpleMultiGPU.

[349] *NVIDIA CUDA Toolkit*. https://developer.nvidia.com/cuda-toolkit.

[350] *NVIDIA CUDA*. https://developer.nvidia.com/cuda-toolkit.

[351] *NVIDIA CUTLASS*. https://github.com/NVIDIA/cutlass.

[352] *NVIDIA cuBLAS GEMM*. https://docs.nvidia.com/cuda/cublas/index.html#cublas-lt-t-gt-gemm.

[353] NVIDIA. *OpenCL Samples*. https://github.com/sschaetz/nvidia-opencl-examples/.

[354] *OpenACC: Directives for GPUs*. https://developer.nvidia.com/blog/openacc-directives-gpus/.

[355] *OpenACC*. https://www.openacc.org.

[356] *OpenCL SGEMM Tuning*. https://cnugteren.github.io/tutorial/pages/page3.html.

[357] *OpenMP*. https://www.openmp.org.

[358] *Performance, Portability, and Productivity in HPC (P3HPC)*.

[359] *Principles and Practice of Parallel Programming (PPoPP)*.

[360] *PyTorch*. https://pytorch.org.

[361] *RDNA Architecture*. https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[362] Standard C++ Foundation Foundation Members. *ISO C++*. https://isocpp.org.

[363] *Tensor Flow*. https://www.tensorflow.org.

[364] *Top500 List*. https://www.top500.org.

[365] *Workshop on Languages and Compilers for Parallel Computing (LCPC)*.

[366] *Workshop on Performance Portable Programming Models for Accelerators (P3MA)*.

[367] *Workshop on Programming and Performance Visualization Tools (ProTools)*. https://protools22.github.io.

CURRICULUM VITAE

PERSONAL INFORMATION

	Ari Rasch
	born on 31.12.1985 in Essen, Germany
	Nationality: german
	Name of Father: Dr. Hemin Rasch
	Name of Mother: Dr. Nishtaman Rasch

SCHOOL EDUCATION

1992 – 1996	Primary School, Essen, Germany
1996 – 2005	High School, Gronau Westf., Germany
	Abitur on 26.06.2005

STUDIES

2005 – 2013	Diploma degree in *Computer Science* with a minor in *Mathematics* (equivalent to combined *MSc* and *BSc*), University of Münster, Germany
	Examination Day: 20.09.2013

ACTIVITIES

2011 – 2013	Student Assistant, University of Münster, Germany
since 2014	Research Associate, University of Münster, Germany

START OF DISSERTATION

since 6/2016	Computer Science Department, University of Münster, Germany
	Supervisor: Prof. Dr. Sergei Gorlatch

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/