(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms[*]

ARI RASCH, University of Muenster, Germany

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 the 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.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; *Machine learning*; • **Theory of computation → Program semantics**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: code generation, data parallelism, auto-tuning, GPU, CPU, OpenMP, CUDA, OpenCL, linear algebra, stencils computation, quantum chemistry, data mining, deep learning

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) [Whaley and Dongarra 1998], various kinds of

[*] A full version of this paper is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this paper relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader to Rasch [2024] for formal details that should not be required for understanding the basic ideas and concepts of our approach.

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

Author's address: Ari Rasch, University of Muenster, Muenster, Germany, a.rasch@uni-muenster.de.

ACM 1558-4593/2024/5-ART
https://doi.org/10.1145/3665643

$$\begin{pmatrix} M_{1,1} & \dots & M_{1,K} \\ \vdots & \ddots & \vdots \\ M_{I,1} & \dots & M_{I,K} \end{pmatrix}, \begin{pmatrix} v_1 \\ \vdots \\ v_K \end{pmatrix} \overset{\texttt{MatVec}}{\mapsto} \overset{\circledast_2}{\overrightarrow{\left. \begin{matrix} f(M_{1,1},v_1) & \dots & f(M_{1,K},v_K) \\ \vdots & \ddots & \vdots \\ f(M_{I,1},v_1) & \dots & f(M_{I,K},v_K) \end{matrix} \right\downarrow_{\circledast_1}}} = \begin{pmatrix} M_{1,1}*v_1 + \dots + M_{1,K}*v_K \\ \vdots \\ M_{I,1}*v_1 + \dots + M_{I,K}*v_K \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_I \end{pmatrix}$$

Fig. 1. Data parallelism illustrated using the example *Matrix-Vector Multiplication* (MatVec)

stencil computations (e.g., Jacobi method and convolutions) [Hagedorn et al. 2018], quantum chemistry computations [Kim et al. 2019], and data mining algorithms [Rasch et al. 2019b]. 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*.

Figures 1 and 2 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 $\circledast_2(\,(x_1,\dots,x_n)\,,\,(y_1,\dots,y_n)\,) := (x_1+y_1,\dots,x_n+y_n)$ (point-wise 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 paper.

Achieving high performance for data-parallel computations is considered important in both academia and industry, but has proven to 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 3 and discussed thoroughly in this paper) 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 [Sun et al. 2019] – 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 [OpenMP 2022] for CPU, CUDA [NVIDIA 2022g] for GPU, and OpenCL [Khronos 2022b] for multiple

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \overset{\texttt{Jacobi1D}}{\mapsto} \left. \begin{matrix} f(v_1,v_2,v_3) \\ f(v_2,v_3,v_4) \\ \vdots \end{matrix} \right\downarrow_{\circledast_1} = \begin{pmatrix} c*(v_1+v_2+v_3) \\ c*(v_2+v_3+v_4) \\ \vdots \end{pmatrix} = \begin{pmatrix} w_1 \\ \vdots \\ w_{N-2} \end{pmatrix}$$

Fig. 2. Data parallelism illustrated using the example *Jacobi 1D* (Jacobi1D)

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 [Ragan-Kelley et al. 2013], Apache TVM [Chen et al. 2018a], Fireiron [Hagedorn et al. 2020a], and LoopStack [Wasti et al. 2022] 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.

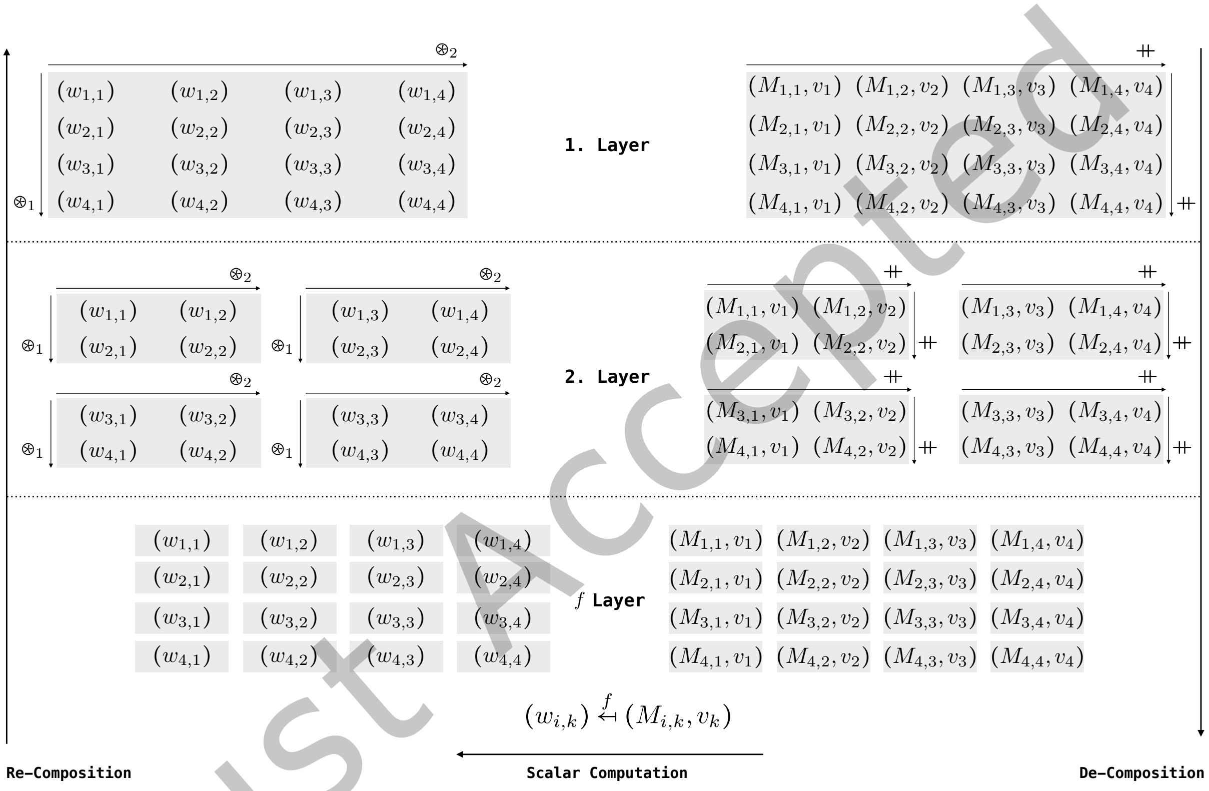


Fig. 3. Example (de/re)-composition of `MatVec` (Figure 1) on a 4×4 input matrix M and a 4-sized vector v: i) the *de-composition phase* (right part of the figure) partitions the concatenated input data into parts (a.k.a. *tiles* in programming), where ++ denotes the concatenation operator; ii) to each 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 paper, 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 paper. Phases are arranged from right to left, inspired by the application order of function composition, as we also discuss later.

In contrast, *polyhedral approaches*, such as Pluto [Bondhugula et al. 2008b], PPCG [Verdoolaege et al. 2013], and Facebook's TC [Vasilache et al. 2019], 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 [Steuwer et al. 2015], are productive for functional programmers (e.g., with experience in *Haskell* [Haskell.org 2022] 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 [Rasch et al. 2019a]. 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 [Remmelg et al. 2016] and stencil computations [Hagedorn et al. 2018]), which further hinders the productivity of the user.

In this paper, 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 the algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)* [Rasch and Gorlatch 2016][2]. 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[3]. 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 [Balaprakash et al. 2018]) 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 [Barham and Isard 2019]. 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).

Summarized, we make the following three major contributions (illustrated in Figure 4):

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

[2] We thoroughly compare to the existing MDH work in Section 6.6.

[3] We consider as main users of our approach compiler engineers and library designers. Rasch et al. [2020b] show that our approach can also take straightforward, sequential code as input, which makes our approach attractive also to end users.

(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.

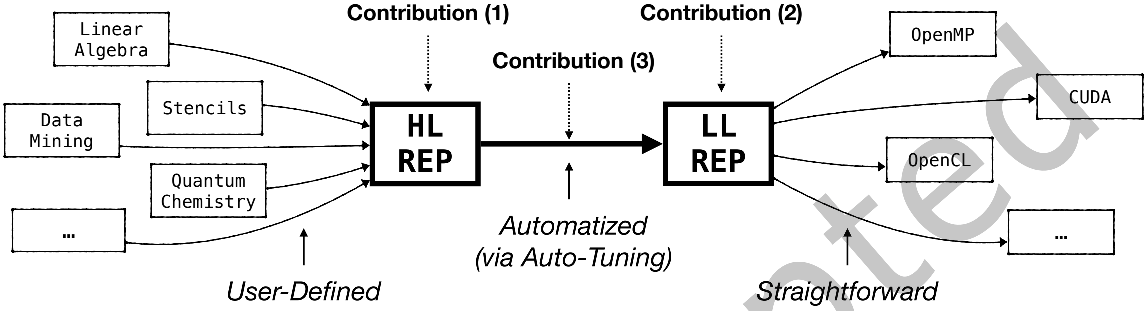


Fig. 4. Overall structure of our approach (contributions highlighted in bold)

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 the paper is structured as follows. We introduce our high-level functional representation (Contribution 1) in Section 2, and we show how this representation is used for expressing various kinds of popular data-parallel computations. In Section 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 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 5, discuss related work in Section 6, conclude in Section 7, and we present our ideas for future work in Section 8.

We provide a full version of this paper [Rasch 2024] that contains details for the interested reader that should not be required for understanding the basic concepts introduced in this paper. In particular, our full version contains formal details – for all the following definition, examples, and theorems in Sections 2-4 – whereas the formalism in this paper is simplified for better illustration and easier understanding of our basic ideas and concepts.

2 HIGH-LEVEL REPRESENTATION FOR DATA-PARALLEL COMPUTATIONS

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 from hardware and optimization details, and thus in a user-productive manner, while still capturing all information relevant for generating high-performance program code. The building blocks of our abstraction are based on the algebraic formalism of *Multi-Dimensional Homomorphisms (MDHs)* which is an approach toward formalizing data parallelism (we compare in detail to the existing work on MDHs in Section 6.6).

Figure 5 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* [Gorlatch and Cole 2011] 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.

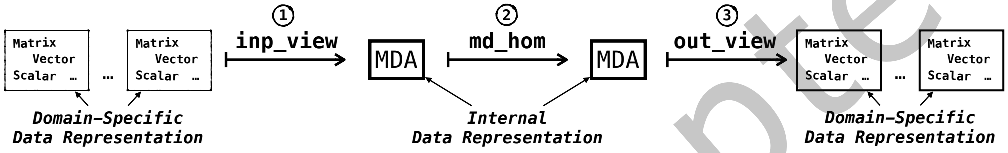


Fig. 5. High-level representation (overview)

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 6 shows how our high-level representation is used for expressing the example of matrix-vector multiplication MatVec[4](Figure 1). Computation MatVec takes as input a matrix $M \in T^{I\times K}$ and vector $v \in T^K$ of arbitrary scalar type[5] T and sizes $I \times K$ (matrix) and K (vector), for arbitrary but fixed positive natural numbers $I, K \in \mathbb{N}$[6]. 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

[4] The expression in Figure 6 can also be extracted from straightforward, annotated sequential code [Rasch et al. 2020b,c].

[5] 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.

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

(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) )
```

Fig. 6. High-level expression for Matrix-Vector Multiplication (MatVec)[7]

2.2 Function md_hom

Higher-order function md_hom is introduced by Rasch and Gorlatch [2016] to express *Multi-Dimensional Homomorphisms (MDHs)* – a 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 1 (Multi-Dimensional Array). A *Multi-Dimensional Array (MDA)* $\mathfrak{a}$ that has *dimensionality* $D \in \mathbb{N}$, *size* $N \in \mathbb{N}^D$, *index sets* $I_1, \dots, I_D \subset \mathbb{N}_0$, and *scalar type* $T \in \mathsf{TYPE}$ is a function with the following signature:

$$\mathfrak{a} : I_1 \times \ldots \times I_D \to T$$

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

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

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

Figure 7 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 7 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 3.

[7] Our technical implementation takes as input a representation that is equivalent to Figure 6, expressed via straightforward program code (see Rasch [2024], Section A.4)

[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.).

$$\mathfrak{a}^{(1,1)} = \underbrace{\begin{bmatrix} \underbrace{1}_{\mathfrak{a}[0,0]} & \underbrace{2}_{\mathfrak{a}[0,1]} & \underbrace{3}_{\mathfrak{a}[0,2]} & \underbrace{4}_{\mathfrak{a}[0,3]} \end{bmatrix}}_{\in T[\ I_1^{(1,1)}:=\{0\}\ ,\ I_2^{(1,1)}:=\{0,1,2,3\}\ ]} \qquad \mathfrak{a}^{(1,2)} = \underbrace{\begin{bmatrix} \underbrace{5}_{\mathfrak{a}[1,0]} & \underbrace{6}_{\mathfrak{a}[1,1]} & \underbrace{7}_{\mathfrak{a}[1,2]} & \underbrace{8}_{\mathfrak{a}[1,3]} \end{bmatrix}}_{\in T[\ I_1^{(1,2)}:=\{1\}\ ,\ I_2^{(1,2)}:=\{0,1,2,3\}\ ]}$$

$$\mathfrak{a} = \underbrace{\begin{bmatrix} \underbrace{1}_{\mathfrak{a}[0,0]} & \underbrace{2}_{\mathfrak{a}[0,1]} & \underbrace{3}_{\mathfrak{a}[0,2]} & \underbrace{4}_{\mathfrak{a}[0,3]} \\ \underbrace{5}_{\mathfrak{a}[1,0]} & \underbrace{6}_{\mathfrak{a}[1,1]} & \underbrace{7}_{\mathfrak{a}[1,2]} & \underbrace{8}_{\mathfrak{a}[1,3]} \end{bmatrix}}_{\in T[\ I_1:=\{0,1\}\ ,\ I_2:=\{0,1,2,3\}\ ]}$$

$$\mathfrak{a}^{(2,1)} = \underbrace{\begin{bmatrix} \underbrace{1}_{\mathfrak{a}[0,0]} & \underbrace{2}_{\mathfrak{a}[0,1]} \\ \underbrace{5}_{\mathfrak{a}[1,0]} & \underbrace{6}_{\mathfrak{a}[1,1]} \end{bmatrix}}_{\in T[\ I_1^{(2,1)}:=\{0,1\}\ ,\ I_2^{(2,1)}:=\{0,1\}\ ]} \qquad \mathfrak{a}^{(2,2)} = \underbrace{\begin{bmatrix} \underbrace{3}_{\mathfrak{a}[0,2]} \\ \underbrace{7}_{\mathfrak{a}[1,2]} \end{bmatrix}}_{\in T[\ I_1^{(2,2)}:=\{0,1\}\ ,\ I_2^{(2,2)}:=\{2\}\ ]} \qquad \mathfrak{a}^{(2,3)} = \underbrace{\begin{bmatrix} \underbrace{4}_{\mathfrak{a}[0,3]} \\ \underbrace{8}_{\mathfrak{a}[1,3]} \end{bmatrix}}_{\in T[\ I_1^{(2,3)}:=\{0,1\}\ ,\ I_2^{(2,3)}:=\{3\}\ ]}$$

Fig. 7. MDA examples

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 1 (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 afterwards using the example operators *concatenation* and *point-wise combination*.

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

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

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

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

and that is computed as:

$$+\!\!+_d(\,\mathfrak{a}_1,\mathfrak{a}_2\,)[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] := \begin{cases} \mathfrak{a}_1[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] & ,\ i_d \in P \\ \mathfrak{a}_2[\,i_1,\ldots,\ i_d\ ,\ldots,i_D\,] & ,\ i_d \in Q \end{cases}$$

The function is well defined when P and Q are disjoint. We usually use an infix notation for $+\!\!+_d$, i.e., we write $\mathfrak{a}_1 +\!\!+_d \mathfrak{a}_2$ instead of $+\!\!+_d(\mathfrak{a}_1,\mathfrak{a}_2)$, 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 $\oplus : T \times T \to T$ (e.g. addition), as function $\overrightarrow{\bullet}_d$ of type

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

that is computed as:

$$\vec{\bullet}_d(\oplus)(\mathfrak{a}_1,\mathfrak{a}_2)[i_1,\ldots,\underset{\overset{\uparrow}{d}}{0},\ldots,i_D] := \mathfrak{a}_1[i_1,\ldots,\underset{\overset{\uparrow}{d}}{0},\ldots,i_D]\oplus\mathfrak{a}_2[i_1,\ldots,\underset{\overset{\uparrow}{d}}{0},\ldots,i_D]$$

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 classical approaches, e.g., *list homomorphisms* [Bird 1989; COLE 1995; Gorlatch 1999], a major characteristic of MDH functions is that they allow (de/re)-composing computations in multiple dimensions (e.g., in Figure 1, 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 in this paper 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 8 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 $\mathfrak{a}_1$ and $\mathfrak{a}_2$, 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 8, 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 $\circledast_2$.

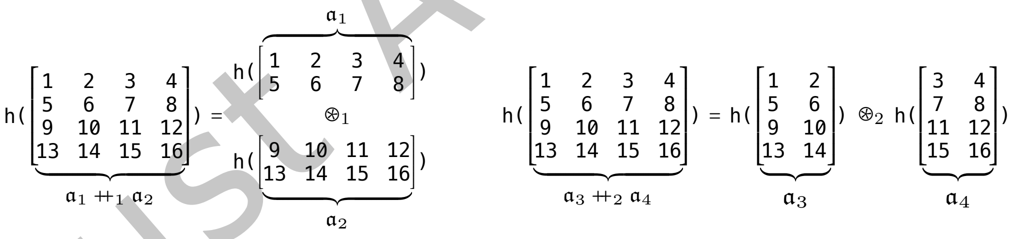


Fig. 8. MDH property illustrated on a two-dimensional example computation

Figure 9 shows an artificial example in which we apply the MDH property (illustrated in Figure 8) recursively. We refer in Figure 9 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

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

is a *Multi-Dimensional Homomorphism (MDH)* that has *input scalar type* $T^{\mathsf{INP}} \in \mathsf{TYPE}$, *output scalar type* $T^{\mathsf{OUT}} \in \mathsf{TYPE}$, 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

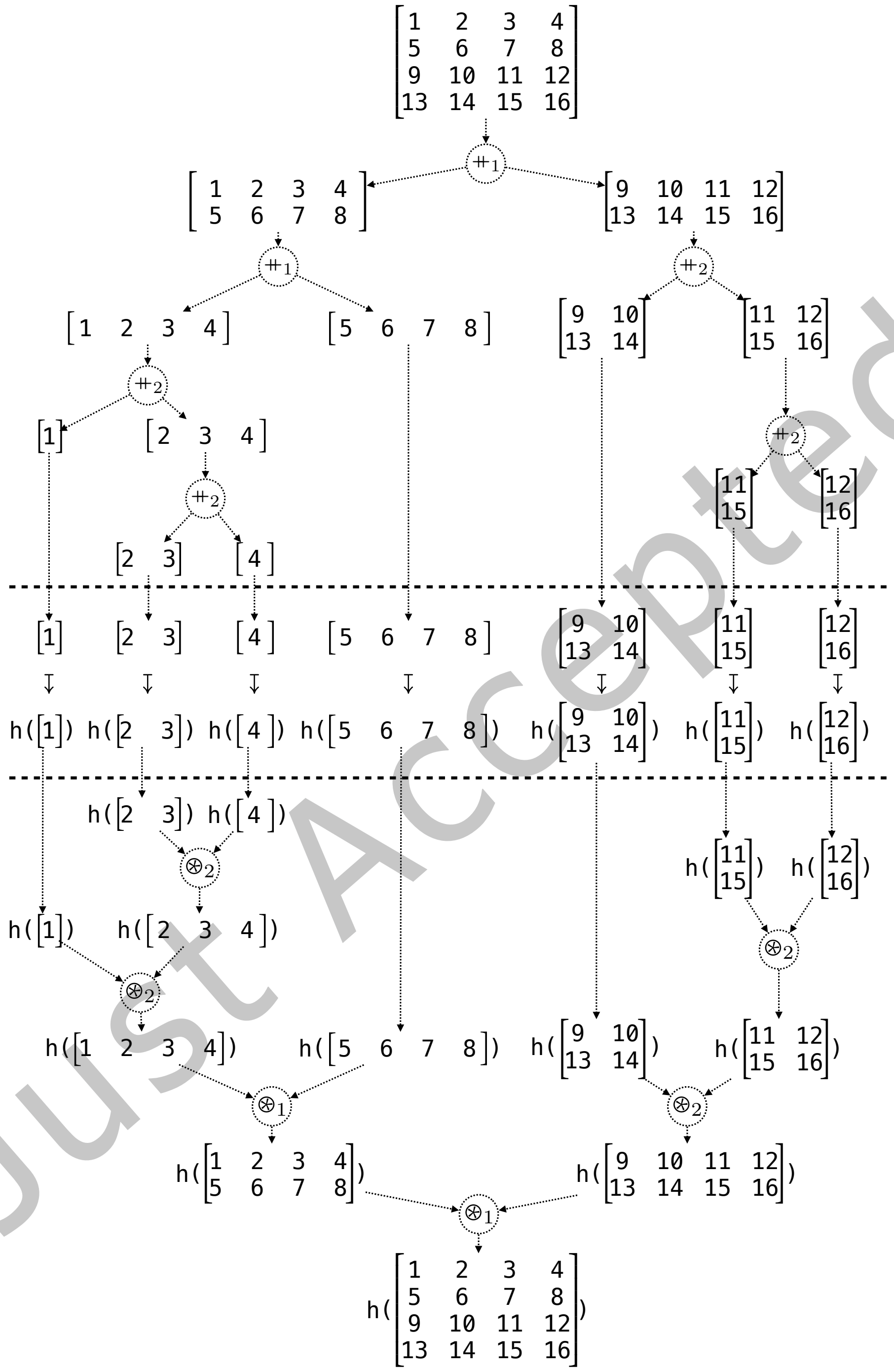


Fig. 9. MDH property recursively applied to a two-dimensional example computation

any concatenated input MDA $\mathfrak{a}_1 +\!\!+_d\, \mathfrak{a}_2$ in dimension d, the *homomorphic property* is satisfied:

$$h(\ \mathfrak{a}_1 +\!\!+_d\, \mathfrak{a}_2\ )\ =\ h(\mathfrak{a}_1)\ \circledast_d\ h(\mathfrak{a}_2)$$

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 $\mathfrak{a}_1$ and $\mathfrak{a}_2$ and combining the intermediate results afterwards by using its combine operator $\circledast_d$, as also informally discussed above.

Example 3 (Function Mapping)**.** A simple example MDH is *function mapping* [González-Vélez and Leyton 2010], computed by higher-order function $\texttt{map}(f)(\mathfrak{a})$, which applies a user-defined scalar function $f : T^{\texttt{INP}} \rightarrow T^{\texttt{OUT}}$ to each element within a D-dimensional MDA $\mathfrak{a}$. Function $\texttt{map}(f)$ is an MDH whose combine operators are concatenation $+\!\!+$ in all of its D dimensions (Example 1).

Example 4 (Reduction)**.** A further MDH function is *reduction* [González-Vélez and Leyton 2010], implemented as higher-order function $\texttt{red}(\oplus)(\mathfrak{a})$, which combines all elements within a D-dimensional MDA $\mathfrak{a}$ using a user-defined binary function $\oplus : T \times T \rightarrow T$. Reduction's combine operators are point-wise combination $\overrightarrow{\bullet}(\oplus)$ 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 2.4, based on higher-order functions `md_hom`, `inp_view`, and `out_view` (Figure 5) 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:

$$\texttt{h(}\begin{bmatrix}1&2&3&4\\5&6&7&8\\9&10&11&12\\13&14&15&16\end{bmatrix}\texttt{)}\ =\ \begin{array}{cccc|l}\multicolumn{4}{c}{\xrightarrow{\circledast_2}} & \\ \texttt{h(}\boxed{1}\texttt{)} & \texttt{h(}\boxed{2}\texttt{)} & \texttt{h(}\boxed{3}\texttt{)} & \texttt{h(}\boxed{4}\texttt{)} & \\ \texttt{h(}\boxed{5}\texttt{)} & \texttt{h(}\boxed{6}\texttt{)} & \texttt{h(}\boxed{7}\texttt{)} & \texttt{h(}\boxed{8}\texttt{)} & \\ \texttt{h(}\boxed{9}\texttt{)} & \texttt{h(}\boxed{10}\texttt{)} & \texttt{h(}\boxed{11}\texttt{)} & \texttt{h(}\boxed{12}\texttt{)} & \\ \texttt{h(}\boxed{13}\texttt{)} & \texttt{h(}\boxed{14}\texttt{)} & \texttt{h(}\boxed{15}\texttt{)} & \texttt{h(}\boxed{16}\texttt{)} & \big\downarrow \circledast_1\end{array}\ =\ \begin{array}{cccc|l}\multicolumn{4}{c}{\xrightarrow{\circledast_2}} & \\ \texttt{f(1)} & \texttt{f(2)} & \texttt{f(3)} & \texttt{f(4)} & \\ \texttt{f(5)} & \texttt{f(6)} & \texttt{f(7)} & \texttt{f(8)} & \\ \texttt{f(9)} & \texttt{f(10)} & \texttt{f(11)} & \texttt{f(12)} & \\ \texttt{f(13)} & \texttt{f(14)} & \texttt{f(15)} & \texttt{f(16)} & \big\downarrow \circledast_1\end{array}$$

Here, f is the function on scalar values that behaves the same as h when restricted to singleton MDAs: $f(\mathfrak{a}[i_1,\ldots,i_D]) := h(\mathfrak{a})$, for any MDA $\mathfrak{a} \in T[\{i_1\},\ldots,\{i_D\}]$ 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 4 (Higher-Order Function `md_hom`)**.** The higher-order function `md_hom` is of type

$$\texttt{md\_hom} : \underbrace{\texttt{SF}}_{f}\ \times\ \underbrace{(\ \texttt{CO} \times \ldots \times \texttt{CO}\ )}_{\circledast_1\,,\ldots,\,\circledast_D}\ \rightarrow_p\ \underbrace{\texttt{MDH}}_{\texttt{md\_hom(}\ f\ ,\ (\circledast_1,\ldots,\circledast_D)\ \texttt{)}}$$

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 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

$$\texttt{md\_hom}(\ f\ ,\ (\circledast_1, \ldots, \circledast_D)\ )(\ \mathfrak{a}\ ) \ := \underset{i_1 \in I_1}{\circledast_1} \ldots \underset{i_D \in I_D}{\circledast_D} \ f(\ \mathfrak{a}[i_1, \ldots, i_D]\ )$$

The combine operators' underset notation denotes straightforward iteration[9]. 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:

$$\begin{aligned}&\texttt{md\_hom}(\ f\ ,\ (\circledast_1, \ldots, \circledast_D)\ )(\ \mathfrak{a}_1 +\!\!+_d\ \mathfrak{a}_2\ ) \ = \\ &\qquad\qquad \texttt{md\_hom}(\ f\ ,\ (\circledast_1, \ldots, \circledast_D)\ )(\ \mathfrak{a}_1\ ) \ \circledast_d \ \texttt{md\_hom}(\ f\ ,\ (\circledast_1, \ldots, \circledast_D)\ )(\ \mathfrak{a}_2\ )\end{aligned}$$

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 $T \in \texttt{TYPE}$. 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 by Rasch [2024], Section B.5. □

MDH functions are defined (Definition 3) such that they uniformly operate on MDAs (Figure 5). 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.

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 2.3.1 and 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 5); 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 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. Finally, we discuss in Section 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

[9] 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 by Rasch [2024], Definition 4).

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* $D \in \mathbb{N}_0$[10], *size* $N := \{N_1, \ldots, N_D\} \in \mathbb{N}^D$, and *scalar type* $T \in$ TYPE is a function with the following signature:

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

Here, we use $\perp$ to denote the *undefined value*. We refer to $[0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\perp\}$ 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 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.

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 1). 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 2).

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 $\mathfrak{iv}$ of type:

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

We denote the type of $\mathfrak{iv}$ 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 $\mathfrak{idx}_{b,a} : \{0, 1, \ldots\} \to \{0, 1, \ldots\}$, $b \in [1, B]_{\mathbb{N}}$, $a \in [1, A_b]_{\mathbb{N}}$, 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.

[10] We use the case $D = 0$ to represent scalar values (details provided Rasch [2024], Section B.7).

In the case of `MatVec` (Figure 1), 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: $\mathfrak{idx}_{1,1}(i,k) := (i,k)$ (used to access the matrix) and $\mathfrak{idx}_{2,1}(i,k) := (k)$ (used for the vector).

In contrast, for `Jacobi1D` (Figure 2), 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: $\mathfrak{idx}_{1,1}(i) := (i+0)$, $\mathfrak{idx}_{1,2}(i) := (i+1)$, and $\mathfrak{idx}_{1,3}(i) := (i+2)$.

Figures 10 and 11 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 i_1 and i_2 for index variables; for `Jacobi1D`, we use identifier v instead of $\mathfrak{b}_1$, and i instead of i_1.

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

$$\texttt{inp\_view} : \underbrace{\Big(\underbrace{\overset{B}{\underset{b=1}{\times}}}_{\text{Buffer}} \underbrace{\overset{A_b}{\underset{a=1}{\times}}}_{\text{Access}} \underbrace{\texttt{IDX-FCT}}_{\text{Index Function: } \mathfrak{idx}_{b,a}} \Big)}_{\text{Index Functions: } \mathfrak{idx}_{1,1},\dots,\mathfrak{idx}_{B,A_B}} \to \underbrace{\texttt{IV}}_{\text{Input View: } \mathfrak{iv}}$$

and it is defined as:

$$\underbrace{(\mathfrak{idx}_{b,a})_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}}_{\text{Index Functions}} \mapsto \underbrace{\underbrace{(\mathfrak{b}_1,\dots,\mathfrak{b}_B)}_{\text{BUFs}} \overset{\mathfrak{iv}}{\mapsto} \underbrace{\mathfrak{a}}_{\text{MDA}}}_{\text{Input View}}$$

for

$$\mathfrak{a}[i_1,\dots,i_D] := (\,\mathfrak{a}_{b,a}[i_1,\dots,i_D]\,)_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}}$$

and

$$\mathfrak{a}_{b,a}[i_1,\dots,i_D] \;:=\; \mathfrak{b}_b[\;\mathfrak{idx}_{b,a}(i_1,\dots,i_D)\;]$$

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 10 and 11.

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 [Hall 2020], 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 Section 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. [2021]:

$$\texttt{inp\_view}(\;\texttt{ID}_1 : \mathfrak{idx}_{1,1},\dots,\mathfrak{idx}_{1,A_1}\;\;,\dots,\;\;\texttt{ID}_B : \mathfrak{idx}_{B,1},\dots,\mathfrak{idx}_{B,A_B}\;)$$

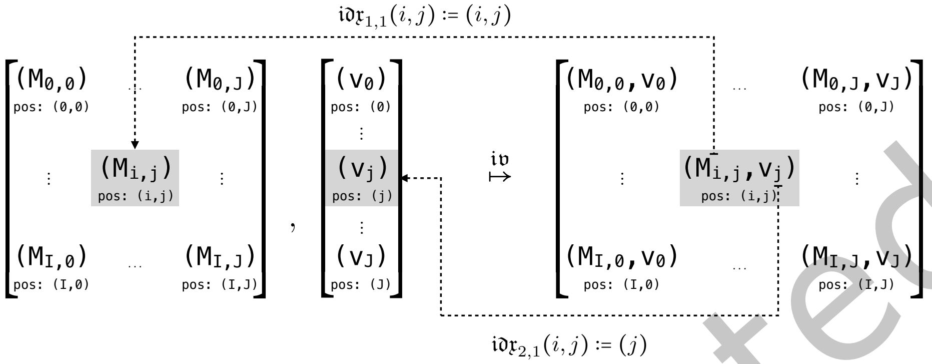


Fig. 10. Input view illustrated using the example MatVec

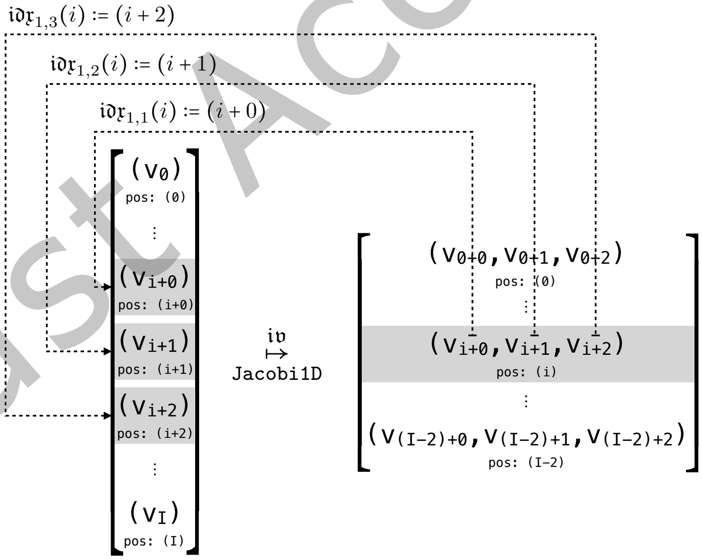

$\mathfrak{idx}_{1,3}(i) := (i+2)$
$\mathfrak{idx}_{1,2}(i) := (i+1)$
$\mathfrak{idx}_{1,1}(i) := (i+0)$
(v0) pos: (0)
(vi+0) pos: (i+0)
(vi+1) pos: (i+1)
(vi+2) pos: (i+2)
(vI) pos: (I)
$\overset{\mathfrak{iv}}{\mapsto}$
Jacobi1D
(v0+0,v0+1,v0+2) pos: (0)
(vi+0,vi+1,vi+2) pos: (i)
(v(I-2)+0,v(I-2)+1,v(I-2)+2) pos: (I-2)

Fig. 11. Input view illustrated using the example Jacobi1D

Here, $\texttt{ID}_1, \ldots, \texttt{ID}_\texttt{B}$ 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:

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

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

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).

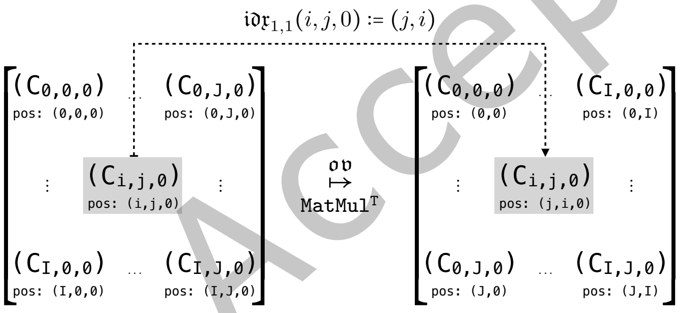


Fig. 12. Output view illustrated using the example *transposed Matrix Multiplication*

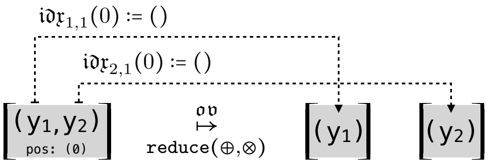


Fig. 13. Output view illustrated using the example *Double Reduction*

Figures 12 and 13 illustrate output views informally using the examples *transposed Matrix Multiplication* and *Double Reduction*.

In the case of transposed matrix multiplication (Figure 12), 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 13), 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 $(0) \mapsto ()$[11] 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:

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

We denote the type of $\mathfrak{ov}$ as `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:

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

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

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

$$\texttt{out\_view} : \underbrace{\underbrace{\mathop{\times}_{b=1}^{B}}_{\text{Buffer}} \underbrace{\mathop{\times}_{a=1}^{A_b}}_{\text{Access}} \underbrace{\texttt{IDX-FCT}}_{\text{Index Function: } \mathfrak{idx}_{b,a}}}_{\text{Index Functions: } \mathfrak{idx}_{1,1}, \ldots, \mathfrak{idx}_{B,A_B}} \to \underbrace{\texttt{OV}}_{\text{Output View: } \mathfrak{ov}}$$

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:

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

[11] The empty braces denote accessing a scalar value (details provided by Rasch [2024], Section B.7).

for

$$\mathfrak{b}_b[\ \mathfrak{idx}_{b,a}(i_1,\dots,i_D)\ ] \ := \ \mathfrak{a}_{b,a}[i_1,\dots,i_D]$$

and

$$(\ \mathfrak{a}_{b,a}[i_1,\dots,i_D]\ )_{b\in[1,B]_{\mathbb{N}},a\in[1,A_b]_{\mathbb{N}}} := \mathfrak{a}[i_1,\dots,i_D]$$

i.e., $\mathfrak{a}_{b,a}[i_1,\dots,i_D]$ is the element at point $i_1,\dots,i_D$ within MDA $\mathfrak{a}$ that belongs to the a-th access of the b-th BUF. We set $\mathfrak{b}_b[\ j_1,\dots,j_{D_b}\ ] := \perp$ (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 $\to_p$ in Definition 8), because for non-injective index functions, it must hold $\mathfrak{idx}_{b,a}(i_1,\dots,i_D) = \mathfrak{idx}_{b,a'}(i'_1,\dots,i'_D) \Rightarrow \mathfrak{a}_{b,a}[i_1,\dots,i_D] = \mathfrak{a}_{b,a'}[i'_1,\dots,i'_D]$ 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:

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

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

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

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 5. In our implementation presented later, we aim to access data uniformly in the 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 10).

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 5), but we use the view functions in our implementation to implicitly forward the MDA accesses to accesses to the physically stored BUF representation.

Lemma 2. Let

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

and

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

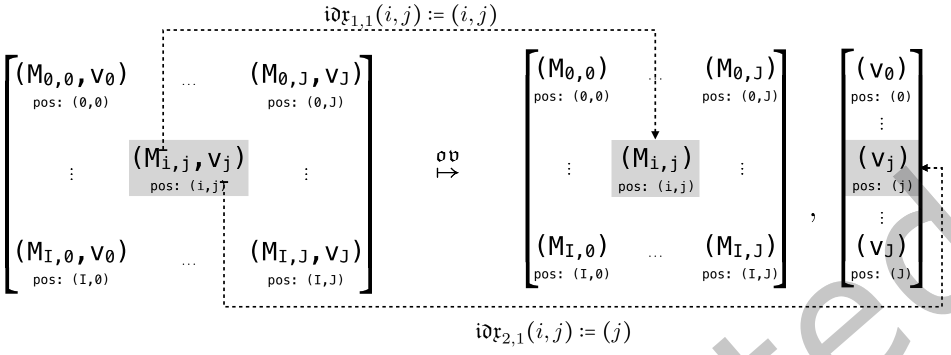
be two arbitrary instances of functions `inp_view` and `out_view` (in Notations 2 and 3), both using the same index functions $\mathfrak{idx}_{1,1},\dots,\mathfrak{idx}_{B,A_B}$.

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

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

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 10):



2.4 Examples

Figure 14 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 6 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) MatMul^T (*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 MatMul^T, the basic computation part of MatMul^T and MatMul are the same, which is exactly reflected in our formalisms: both MatMul^T and MatMul are expressed using exactly the same md_hom instances. The differences between MatMul^T and MatMul lies only in the data accesses – transposed accesses in the case of MatMul^T 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 MatMul^T, 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 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 [Podlozhnyuk 2007]; 2) MCC

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

	inp_view		out_view
Views	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)
$\texttt{MatMul}^{T}$	(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	$*$	$+\!\!+$	$+\!\!+$	$+\!\!+$	$+\!\!+$	$+$	$+$	$+$	$+\!\!+$	$+\!\!+$	$+$

	inp_view		out_view
Views	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)	$*$	$+\!\!+$	...	$+\!\!+$	$+$

	inp_view		out_view
Views	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}	$+\!\!+$	$+\!\!+$	$+\!\!+$

	inp_view	out_view
Views	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	$+\!\!+$	$\texttt{max}_{\texttt{PRL}}$

	inp_view		out_view
Views	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	$\texttt{f}_{\texttt{Histo}}$	$+$	$+\!\!+$
GenHisto	f	$\oplus$	$+\!\!+$

	inp_view		out_view
Views	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)	($\oplus,\otimes$)

	inp_view	out_view	
Views	I	$\texttt{O}_1$	$\texttt{O}_2$
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	$+\!\!+_{\texttt{prefix-sum}}(\oplus)$	
MBBS	id	$+\!\!+_{\texttt{prefix-sum}}(+)$	$+$

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

8) Prefix Sum Computations

Fig. 14. Data-parallel computations expressed in our high-level representation

expresses a so-called *Multi-Channel Convolution* [Dumoulin and Visin 2018] – a generalization of Conv2D that is heavily used in the area of deep learning; 3) MCC_Capsule is a recent generalization of MCC [Hinton et al. 2018] which attracted high attention due to its relevance for advanced deep learning neural networks [Barham and Isard 2019].

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 [Hagedorn et al. 2018]. 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 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 [Li et al. 2016].

Subfigure 3. We show how quantum chemistry computation *Coupled Cluster* (CCSD(T)) [Kim et al. 2019] 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 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 $\mathtt{J}_{\mathtt{nD}}$ [Cecilia et al. 2012], and *Probabilistic Record Linkage (PRL)* [Christen 2012], 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 $\mathtt{max}_{\mathtt{PRL}}$ (point-wise combination via the PRL-specific binary operator $\mathtt{max}_{\mathtt{PRL}}$) [Rasch et al. 2019b]. Histograms, in their generalized version [Henriksen et al. 2020] (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 $f = f_{\mathtt{Histo}}$, where $f_{\mathtt{Histo}}(e,b) = 1$ iff $e = b$ and $f_{\mathtt{Histo}}(e,b) = 0$ otherwise.

Subfigure 7. We show how typical map and reduce patterns [González-Vélez and Leyton 2010] are implemented in our high-level representation. Examples map(f) and reduce($\oplus$) (discussed in Examples 3 and 4) are simple and thus straightforwardly expressed in our representation. In contrast, example reduce($\oplus,\otimes$) is more complex and shows how reduce($\oplus$) is extended to combine the input vector simultaneously twice – once combining vector elements via operator $\oplus$ and once using operator $\otimes$. The outcome of reduce($\oplus,\otimes$) are two scalars – one representing the result of combination via $\oplus$ and the other of combination via $\otimes$ – which we map via the output view to output elements $\mathtt{O}_1$ (result of $\oplus$) and $\mathtt{O}_2$ (result of $\otimes$), correspondingly; this is also illustrated in Figure 13.

Subfigure 8. We present *prefix-sum computations* [Blelloch 1990] 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($\oplus$) uses as combine operator $+\!\!+_{\mathtt{prefix\text{-}sum}}(\oplus)$ which computes prefix-sum [Gorlatch and Lengauer 1997] (formally defined by Rasch [2024], Section B.9) according to binary operator ⊕, and `MBBS` (Maximum Bottom Box Sum) [Farzan and Nicolet 2019] uses a particular instance of prefix-sum for $\oplus = +$ (addition).

3 LOW-LEVEL REPRESENTATION FOR DATA-PARALLEL COMPUTATIONS

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 3). 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 hierarchy of the target architecture.

In the following, after briefly discussing an introductory example in Section 3.1, we introduce in Section 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 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 4), according to the methodologies presented later in Section 4 and auto-tuning [Rasch et al. 2021].

3.1 Introductory Example

Figure 15 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 $T \in \texttt{TYPE}$; 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) *de-composition* (steps 1-7, in the right part of Figure 15), 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 15) partitions input MDA ${}^{\downarrow}\mathfrak{a}$ (in the top right of Figure 15) to the structure ${}^{\downarrow}\mathfrak{a}^{<\dots>}_{f}$ (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 $p^1_1 \in [0,2)_{\mathbb{N}_0}$, $p^1_2 \in [0,4)_{\mathbb{N}_0}$, $p^2_1 \in [0,8)_{\mathbb{N}_0}$, $p^2_2 \in [0,16)_{\mathbb{N}_0}$, $p^3_1 \in [0,32)_{\mathbb{N}_0}$, $p^3_2 \in [0,64)_{\mathbb{N}_0}$ refers to an MDA that represents an individual part of MDA ${}^{\downarrow}\mathfrak{a}$ (a.k.a. *tile* in programming – informally illustrated in Figure 7). The partitions are arranged on multiple layers (indicated by the p's superscripts) and in multiple dimensions (indicated by subscripts) – as illustrated in Figure 16 – 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 15, 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

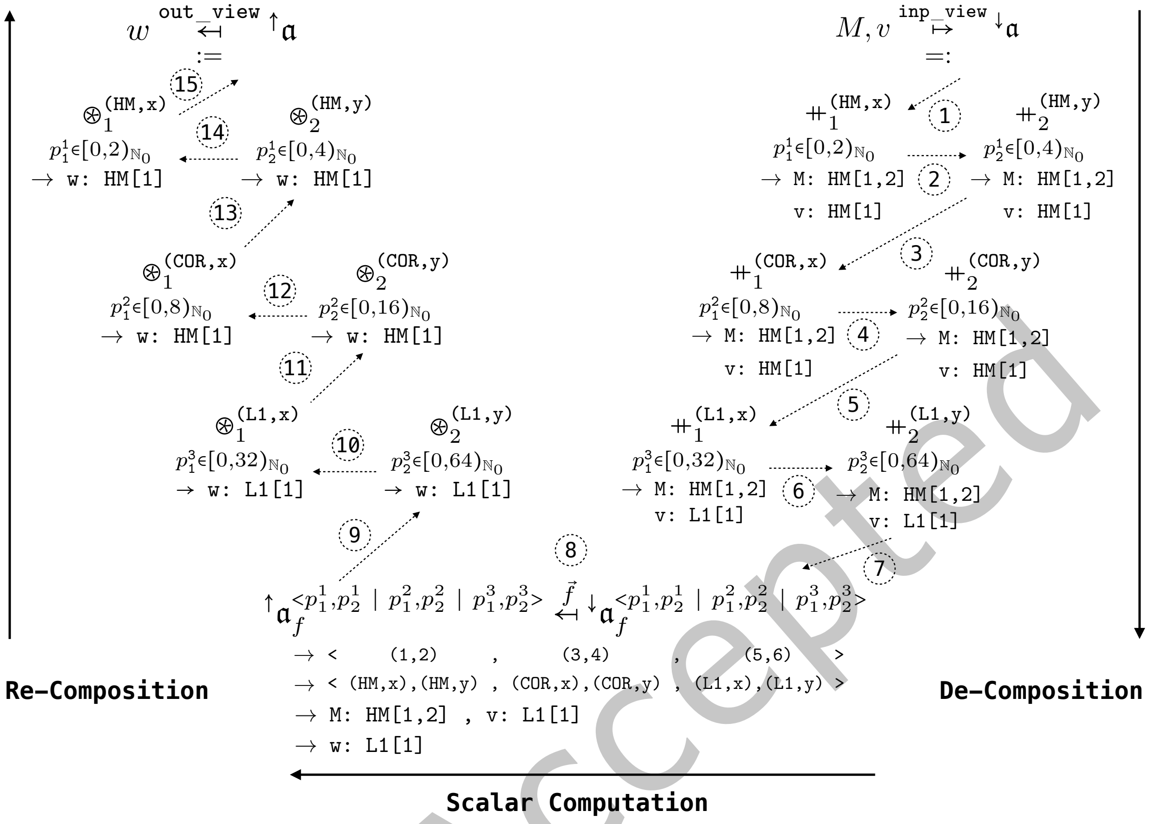


Fig. 15. Low-level expression for straightforwardly computing Matrix-Vector Multiplication (MatVec) on a simple, artificial architecture with two memory layers (HM and L1) and one core layer (COR). Dotted lines indicate data flow.

dimensions of the target computation (in the case of MatVec, the i-dimension and also the k-dimension – see Figure 1), 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 15). For example, we partition in Figure 15 MDA $^{\downarrow}\mathfrak{a}$ first via the inverse of $+\!\!+_1^{(\mathtt{HM,x})}$ in dimension 1 (indicated by the subscript 1 of $+\!\!+_1^{(\mathtt{HM,x})}$; the superscript $(\mathtt{HM,x})$ 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. Afterwards, each of the obtained parts is further partitioned, in the second dimension, via $+\!\!+_2^{(\mathtt{HM,y})}$ 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[12], 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 16, the individual elements are denoted

[12] Our future work (outlined in Section 8) aims to additionally allow coarser-grained partitioning schemas, e.g., to target domain-specific hardware extensions (such as *NVIDIA Tensor Cores* [NVIDIA 2017] which compute 4 × 4 matrices immediately in hardware, rather than 1 × 1 matrices as obtained in the case of a full partitioning).

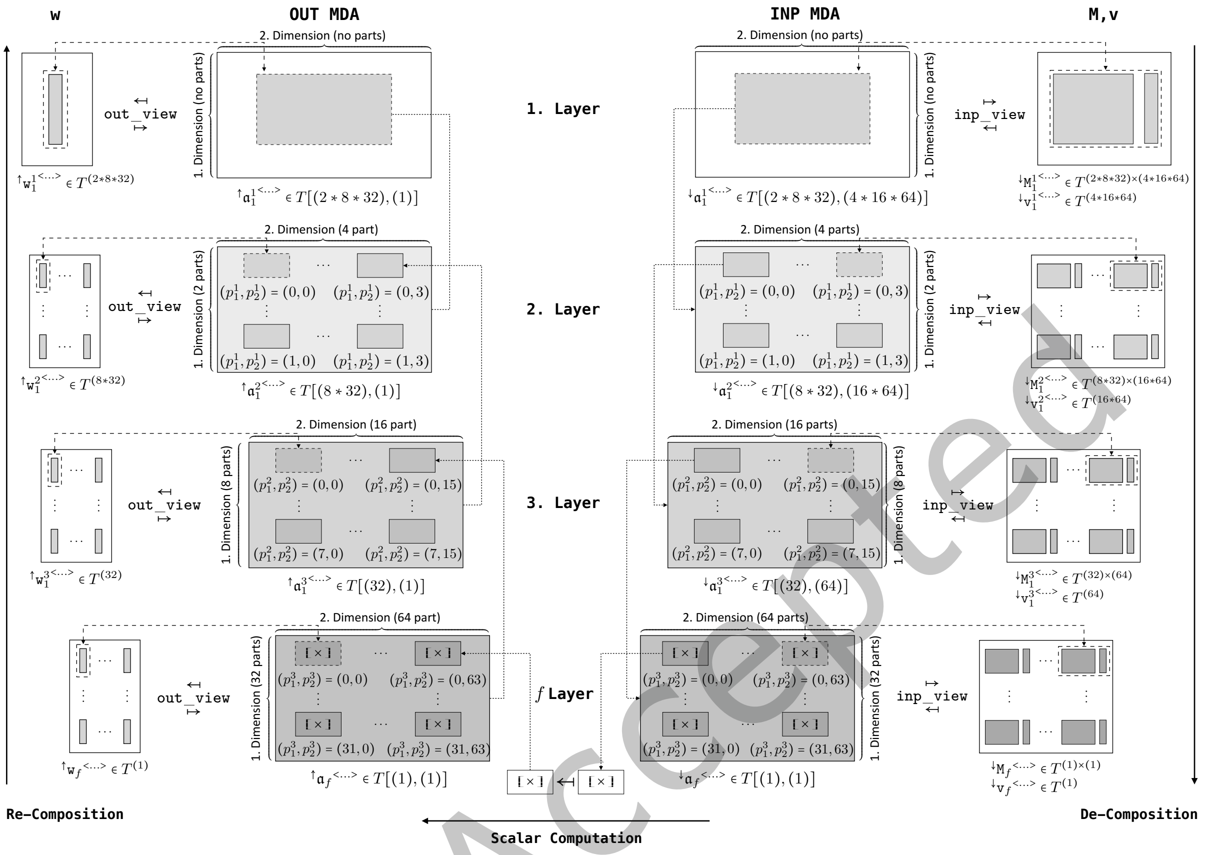

w
OUT MDA
INP MDA
M,v
1. Layer
2. Layer
3. Layer
f Layer
out_view
inp_view
2. Dimension (no parts)
1. Dimension (no parts)
2. Dimension (4 part)
2. Dimension (4 parts)
1. Dimension (2 parts)
2. Dimension (16 part)
2. Dimension (16 parts)
1. Dimension (8 parts)
2. Dimension (64 part)
2. Dimension (64 parts)
1. Dimension (32 parts)
$(p_1^1,p_2^1)=(0,0)$
$(p_1^1,p_2^1)=(0,3)$
$(p_1^1,p_2^1)=(1,0)$
$(p_1^1,p_2^1)=(1,3)$
$(p_1^2,p_2^2)=(0,0)$
$(p_1^2,p_2^2)=(0,15)$
$(p_1^2,p_2^2)=(7,0)$
$(p_1^2,p_2^2)=(7,15)$
$(p_1^3,p_2^3)=(0,0)$
$(p_1^3,p_2^3)=(0,63)$
$(p_1^3,p_2^3)=(31,0)$
$(p_1^3,p_2^3)=(31,63)$
$[\times]$
$^{\uparrow}\mathtt{w}_1^{1<\ldots>}\in T^{(2*8*32)}$
$^{\uparrow}\mathfrak{a}_1^{1<\ldots>}\in T[(2*8*32),(1)]$
$^{\downarrow}\mathfrak{a}_1^{1<\ldots>}\in T[(2*8*32),(4*16*64)]$
$^{\downarrow}\mathtt{M}_1^{1<\ldots>}\in T^{(2*8*32)\times(4*16*64)}$
$^{\downarrow}\mathtt{v}_1^{1<\ldots>}\in T^{(4*16*64)}$
$^{\uparrow}\mathtt{w}_1^{2<\ldots>}\in T^{(8*32)}$
$^{\uparrow}\mathfrak{a}_1^{2<\ldots>}\in T[(8*32),(1)]$
$^{\downarrow}\mathfrak{a}_1^{2<\ldots>}\in T[(8*32),(16*64)]$
$^{\downarrow}\mathtt{M}_1^{2<\ldots>}\in T^{(8*32)\times(16*64)}$
$^{\downarrow}\mathtt{v}_1^{2<\ldots>}\in T^{(16*64)}$
$^{\uparrow}\mathtt{w}_1^{3<\ldots>}\in T^{(32)}$
$^{\uparrow}\mathfrak{a}_1^{3<\ldots>}\in T[(32),(1)]$
$^{\downarrow}\mathfrak{a}_1^{3<\ldots>}\in T[(32),(64)]$
$^{\downarrow}\mathtt{M}_1^{3<\ldots>}\in T^{(32)\times(64)}$
$^{\downarrow}\mathtt{v}_1^{3<\ldots>}\in T^{(64)}$
$^{\uparrow}\mathtt{w}_f^{<\ldots>}\in T^{(1)}$
$^{\uparrow}\mathfrak{a}_f^{<\ldots>}\in T[(1),(1)]$
$^{\downarrow}\mathfrak{a}_f^{<\ldots>}\in T[(1),(1)]$
$^{\downarrow}\mathtt{M}_f^{<\ldots>}\in T^{(1)\times(1)}$
$^{\downarrow}\mathtt{v}_f^{<\ldots>}\in T^{(1)}$
Re-Composition
De-Composition
Scalar Computation

Fig. 16. Illustration of multi-layered, multi-dimensional MDA partitioning using the example MDA from Figure 15. In this example, we use three layers and two dimensions, according to Figure 15.

via symbol ×, 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 2.2).

The superscript of combine operators, e.g., $(\mathtt{COR},\mathtt{x})$ of operator $+\!\!+_1^{(\mathtt{COR},\mathtt{x})}$, 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 $(\mathtt{COR},\mathtt{x})$ indicates that parts processed by operator $+\!\!+_1^{(\mathtt{COR},\mathtt{x})}$ 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 [NVIDIA 2022g]), as we also discuss in the next subsection. In contrast, tag $(\mathtt{HM},\mathtt{x})$ refers to a memory layer (host memory `HM`) and thus, operator $+\!\!+_1^{(\mathtt{HM},\mathtt{x})}$ 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 afterwards in dimension 2) are chosen arbitrarily for this example. These choices are critical for performance and should be optimized[13] 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 15), we apply MDH's scalar function f to the individual MDA elements

$$^{\downarrow}\mathfrak{a}_f^{<p_1^1,p_2^1 \,|\, p_1^2,p_2^2 \,|\, p_1^3,p_2^3>}$$

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

$$^{\uparrow}\mathfrak{a}_f^{<p_1^1,p_2^1 \,|\, p_1^2,p_2^3 \,|\, p_1^3,p_2^3>}$$

In the figure, $\vec{f}$ is the slight adaption of function f that operates on a singleton MDA, rather than a scalar (see Footnote 9).

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 15) 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 15) combines the computed parts $^{\uparrow}\mathfrak{a}_f^{<p_1^1,p_2^1 \,|\, p_1^2,p_2^2 \,|\, p_1^3,p_2^3>}$ (bottom left in the figure) to the final result $^{\uparrow}\mathfrak{a}$ (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; afterwards, 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 [NVIDIA 2022f]. 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 by Rasch [2024], Section C.3, for the interested reader).

[13] We currently rely on auto-tuning [Rasch et al. 2021] for choosing optimized values of performance-critical parameters, as we discuss in Section 5.

Excursion: Code Generation[14]

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 15. Details about our code generation process are provided by Rasch [2024], Section E, and will be presented and illustrated in detail in our future work.

We implement combine operators as sequential or parallel loops. For example, the operator $+\!\!+_1^{(\mathtt{HM},\mathtt{x})}$ is assigned to memory layer HM and thus implemented as a sequential loop (loop range indicated by $[0,2)_{\mathbb{N}_0}$), and operator $+\!\!+_1^{(\mathtt{COR},\mathtt{x})}$ 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 [OpenMP 2022], or variable `threadIdx.x` in CUDA [NVIDIA 2022g]). 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 15. 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 [NVIDIA 2022g], 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 15 are beyond the scope of this work and outlined by Rasch [2024], Section F, 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 [MDH Project 2024]. Our compiler takes as input a high-level MDH expression (as in Figure 6), 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 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. Afterwards, in Section 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.

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

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

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

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

Our ASM representation is capable of modeling architectures with arbitrarily deep memory and core hierarchies[15]: 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 15 is represented as an ASM instance as follows (bar symbols denote set cardinality):

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

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

$$\begin{array}{lcccccccc}
\mathtt{ASM}_{\mathtt{OpenMP}} & := & (& |\{\mathtt{MM}, \mathtt{L2}, \mathtt{L1}\}| & , & |\{\mathtt{COR}\}| &) & = & (3, 1) \\
\mathtt{ASM}_{\mathtt{OpenMP+L3}} & := & (& |\{\mathtt{MM}, \mathtt{L3}, \mathtt{L2}, \mathtt{L1}\}| & , & |\{\mathtt{COR}\}| &) & = & (4, 1) \\
\mathtt{ASM}_{\mathtt{OpenMP+L3+SIMD}} & := & (& |\{\mathtt{MM}, \mathtt{L3}, \mathtt{L2}, \mathtt{L1}\}| & , & |\{\mathtt{COR}, \mathtt{SIMD}\}| &) & = & (4, 2) \\
\mathtt{ASM}_{\mathtt{CUDA}} & := & (& |\{\mathtt{DM}, \mathtt{SM}, \mathtt{RM}\}| & , & |\{\mathtt{SMX}, \mathtt{CC}\}| &) & = & (3, 2) \\
\mathtt{ASM}_{\mathtt{CUDA+WRP}} & := & (& |\{\mathtt{DM}, \mathtt{SM}, \mathtt{RM}\}| & , & |\{\mathtt{SMX}, \mathtt{WRP}, \mathtt{CC}\}| &) & = & (3, 3) \\
\mathtt{ASM}_{\mathtt{OpenCL}} & := & (& |\{\mathtt{GM}, \mathtt{LM}, \mathtt{PM}\}| & , & |\{\mathtt{CU}, \mathtt{PE}\}| &) & = & (3, 2)
\end{array}$$

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* [Klemm et al. 2012]), 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 [NVIDIA 2022g], but often exploited by programmers – via special intrinsics (e.g., *shuffle* and *tensor core intrinsics* [NVIDIA 2017, 2018]) – 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 [StreamHPC 2016], 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

[15] 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.

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 [NVIDIA 2022f] – 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:

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

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:

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

It adds to $\mathtt{ASM}_{\mathtt{Multi\text{-}GPU}}$ 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 [Chen et al. 2010].

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 16); 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 $\mathfrak{a}_{ll}$ of type:

$$\mathfrak{a}_{ll}^{<\overbrace{(p^1_1,\ldots,p^1_D)\in P^1_1\times\ldots\times P^1_D}^{\text{Partitioning: Layer 1}} \;|\; \ldots \;|\; \overbrace{(p^L_1,\ldots,p^L_D)\in P^L_1\times\ldots\times P^L_D}^{\text{Partitioning: Layer } L}>} :$$

$$I_1^{<p^1_1,\ldots,p^1_D \,|\, \ldots \,|\, p^L_1,\ldots,p^L_D>} \times \ldots \times I_D^{<p^1_1,\ldots,p^1_D \,|\, \ldots \,|\, p^L_1,\ldots,p^L_D>} \to T$$

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 [NVIDIA 2022f], 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 $\mathfrak{b}_{ll}$ of type ($\lhook\joinrel\twoheadrightarrow$ denotes bijection):

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

$$[0,N_1^{<p_1^1,\dots,p_D^1 \mid \dots \mid p_1^L,\dots,p_D^L>})_{\mathbb{N}_0} \times \dots \times [0,N_D^{<p_1^1,\dots,p_D^1 \mid \dots \mid p_1^L,\dots,p_D^L>})_{\mathbb{N}_0} \to T$$

We refer to $\texttt{MEM}$ as low-level BUF's *memory region* and to σ as its *memory layout*, and we refer to the function

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

$$[0,N_{\sigma(1)}^{<p_1^1,\dots,p_D^1 \mid \dots \mid p_1^L,\dots,p_D^L>})_{\mathbb{N}_0} \times \dots \times [0,N_{\sigma(D)}^{<p_1^1,\dots,p_D^1 \mid \dots \mid p_1^L,\dots,p_D^L>})_{\mathbb{N}_0} \to T$$

that is defined as

$$\mathfrak{b}_{ll}^{\texttt{trans}<\texttt{MEM}\mid\sigma><p_1^1,\dots,p_D^1\mid\dots\mid p_1^L,\dots,p_D^L>}(i_{\sigma(1)},\dots,i_{\sigma(D)}) \ := \ \mathfrak{b}_{ll}^{<\texttt{MEM}\mid\sigma><p_1^1,\dots,p_D^1\mid\dots\mid p_1^L,\dots,p_D^L>}(i_1,\dots,i_D)$$

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* [Phothilimthana et al. 2019]). 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* [NVIDIA 2022f] 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)[16]:

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

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 15, 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.

[16] 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 LOWERING: FROM HIGH LEVEL TO LOW LEVEL

We have designed our formalism such that an expression in our high-level representation (as in Figure 6) can be *systematically lowered* to an expression in our low-level representation (as in Figure 15). 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 by Rasch [2024], Section 4, for the interested reader. We chose optimized values of tuning parameters fully automatically via auto-tuning [Rasch et al. 2021]; Section 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 by Rasch [2024], Section 4), but the expressions will be translated to differently optimized code variants.[17]

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 [NVIDIA 2022f]. We discuss algorithmic constraints in the following, together with our tuning parameters; model constraints are discussed by Rasch [2024], Section C.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 15 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.

Definition 15 (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`):

$$\texttt{MDH-LVL} := \{ \, (l_{\texttt{MDH}}, d_{\texttt{MDH}}) \mid l_{\texttt{MDH}} \in [1, L]_{\mathbb{N}}, d_{\texttt{MDH}} \in [1, D]_{\mathbb{N}} \}$$ [18]

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 15.

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

[17] Rasch [2024] (Section 3.5) 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 [Chen et al. 2018a] and polyhedral compilers PPCG [Verdoolaege et al. 2013] and Pluto [Bondhugula et al. 2008b].

[18] The same as for identifier `ASM-LVL` (Definition 13), we refrain from annotating identifier `MDH-LVL` with values L and D. Note that `MDH-LVL` and `ASM-LVL` both refer to the same set of pairs, but we use identifier `MDH-LVL` when referring to MDH levels and identifier `ASM-LVL` when referring to ASM levels, correspondingly, for better clarity.

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

Table 1. Tuning parameters of our low-level expressions

example, in Figure 15, we use $\texttt{\#PRT}(1,1) := 2$ which causes combine operators $+\!\!+_1^{(\texttt{HM},\texttt{x})}$ and $\circledast_1^{(\texttt{HM},\texttt{x})}$ 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 $\texttt{\#PRT}(1,2) := 4$ to let operators $+\!\!+_2^{(\texttt{HM},\texttt{y})}$ and $\circledast_2^{(\texttt{HM},\texttt{x})}$ 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 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* [Lam et al. 1991]) 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 15, 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 [Bacon et al. 1994], 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 $\lhook\joinrel\twoheadrightarrow$ in Table 1), determining *when* data is 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* [McKinley et al. 1996]). 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 15 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 by Rasch [2024], Section F).

Parameters `D2,S2,R2`*:* These parameters (symbol $\lhook\joinrel\twoheadrightarrow$ 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 is processed (e.g., threads or `for`-loops), similar to the concept of `bind` in scheduling languages [Apache TVM Documentation 2022a]. Consequently, the parameters determine which parts should be computed sequentially in our generated code and which parts in parallel. For example, in Figure 15, 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 de-composition 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 15, 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 CORs 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 [Phothilimthana et al. 2019]. 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 15 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 is read from or written to, respectively. In the table, we use $\texttt{ib} \in \mathbb{N}$ 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 $\texttt{ob} \in \mathbb{N}$ 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 $\texttt{MR} := [1,\texttt{NUM\_MEM\_LYRs}]_{\mathbb{N}}$ 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 [Bondhugula 2020; Hristea et al. 1997; Mei et al. 2014; Salvador Rohwedder et al. 2023], particularly due to the performance gap between processors' cores and their memory systems [Oliveira et al. 2021; Wilkes 2001].

Parameters D4,R4 *and* S4,S6*:* These parameters set the memory layouts of BUFs, thereby determining *how* data is accessed in memory; for brevity in Table 1, we denote the set of all BUF permutations $[1,D]_{\mathbb{N}} \hookrightarrow\!\!\!\!\twoheadrightarrow [1,D]_{\mathbb{N}}$ (Definition 12) as $[1,\ldots,D]_{\mathcal{S}}$ (symbol $\mathcal{S}$ is taken from the notation of *symmetric groups* [Sagan 2001]). In the case of our matrix-vector multiplication example in Figure 15, we use a standard memory layout for all matrices, which we express via the parameters by setting them to the identity function, e.g., $\sigma^{\texttt{<M>}}_{\downarrow\texttt{-mem}}(1,1) := id$ (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.

5 EXPERIMENTAL RESULTS

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

(1) *scheduling approach*: TVM [Chen et al. 2018a] which generates GPU and CPU code from programs expressed in TVM's own high-level program representation;

(2) *polyhedral compilers*: PPCG [Verdoolaege et al. 2013] for GPUs[19] and Pluto [Bondhugula et al. 2008b] for CPUs, which automatically generate executable program code in CUDA (PPCG) or OpenMP (Pluto) from straightforward, unoptimized C programs;

[19] We cannot compare to polyhedral compiler TC [Vasilache et al. 2019] 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 [Facebook Research 2022]. Rasch et al. [2019a] show that our approach – already in its proof-of-concept version – achieves higher performance than TC for popular computations on real-world data sets.

(3) *functional approach*: Lift [Steuwer et al. 2015] which generates OpenCL code from a Lift-specific, functional program representation;

(4) *domain-specific libraries*: NVIDIA cuBLAS [NVIDIA 2022b] and NVIDIA cuDNN [NVIDIA 2022e], as well as Intel oneMKL [Intel 2022c] and Intel oneDNN [Intel 2022b], 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 [NVIDIA 2022c], and the most recent `cublasLtMatmul` which is part of the `cuBLASLt` extension [NVIDIA 2022d]; 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* [Intel 2019] which is specifically designed and optimized for small problem sizes. We also compare to library *EKR* [Hentschel et al. 2008] which computes data mining example `PRL` (Figure 14) 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* [Pennycook et al. 2019] which mathematically defines portability[20] 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 on platform i; value $\Phi(a,p,H)$ is 0, if any platform in H is unsupported by a running p.“ [Pennycook et al. 2019]. 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 $a^{\text{best}}_{p,i}$ 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 6). Classical code metrics, such as *Lines of Code (LOC)*, COCOMO [Boehm et al. 1995],

[20] 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.

McCabe's cyclomatic complexity [McCabe 1976], and Halstead development effort [Halstead 1977] 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 5.1-5.4.

Application Case Studies

We use for experiments in this section popular example computations from Figure 14 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` [He et al. 2015], `VGG-16` [Simonyan and Zisserman 2014], and `MobileNet` [Howard et al. 2017], according to their TensorFlow implementations [TensorFlow 2022a,b,c]. 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 classical `Conv2D` variant of convolution (Figure 14) – the deep learning variants are considered as significantly more challenging to optimize than their classical variants [Barham and Isard 2019].

We use for experiments this subset of computations from Figure 14 to make experimenting challenging for us: the computations differ in major characteristics (as discussed in Section 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) 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 [Springer and Bientinesi 2016] 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* [Krizhevsky et al. 2012] data set (the particular characteristics are listed by Rasch [2024], Section D.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), in order 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* [NVIDIA 2018] 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* [NVIDIA 2017], in all experiments, which compute the multiplication of small matrices with high performance [Feng et al. 2023], 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` [NVIDIA 2022h] 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` [Apache 2022] 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` [Michael Kruse 2022] 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 the *Auto-Tuning Framework (ATF)* [Rasch et al. 2021] for choosing optimized tile size values (as we discuss in the next subsection);
- `Pluto` [Uday Bondhugula 2022] 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` [NVIDIA 2022b] from `CUDA Toolkit 11.4`, using the NVIDIA-recommended compiler flags `-fast -O3 -DNDEBUG`;
- `NVIDIA cuDNN` [NVIDIA 2022e] from `CUDA Toolkit 11.4`, using the NVIDIA-recommended compiler flags `-fast -O3 -DNDEBUG`;
- `Intel oneMKL` [Intel 2022c] compiled with `Intel's icpx` compiler version `2022.0.0`, using flags `-DMKL_ILP64 -qmkl=parallel -L${MKLROOT}/lib/intel64 -liomp5 -lpthread -lm -ldl`, as recommended for oneMKL by Intel's *Link Line Advisor* tool [Intel 2022a], as well as standard flags `-O3 -NDEBUG`;
- `Intel oneDNN` [Intel 2022b] also compiled with `Intel's icpx` compiler version `2022.0.0`, using flags `-I${DNNLROOT}/include -L${DNNLROOT}/lib -ldnnl`, according to oneDNN's documentation, as well as standard flags `-O3 -NDEBUG`;

- EKR [Hentschel et al. 2008] 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 [GNU/Linux 2022]. In the case of C++ libraries Intel oneMKL and Intel oneDNN, we use the C++ chrono library [C++ reference 2022] 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*[21] 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 [2015].

Auto-Tuning

The auto-tuning process of our approach relies on the generic *Auto Tuning Framework (ATF)* [Rasch et al. 2021]. The ATF framework has proven to be efficient for exploring large search spaces of constrained tuning parameters (as our space introduced in Section 4). We use ATF, out of the box, exactly as described by Rasch et al. [2021]: 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 cost of our generated OpenCL and CUDA codes (in this paper, 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* [Ansel et al. 2014]). 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* [Zheng et al. 2020a] 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

[21] *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 the high-level library *dOCAL* [Rasch et al. 2020a, 2018] for host code programming in this work.

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 Section 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 $(l, d) \in$ 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 5.1-5.4. We aim to substantially improve our naive cost model in future work, based on *operational semantics* for our low-level representation, in order to improve the auto-tuning quality and to reduce (or even avoid) tuning time.

Code Generation

We provide an open source *MDH compiler* [MDH Project 2024] for generating executable program code from expressions in our high-level representation (as illustrated in Figure 4). Our compiler takes as input the high-level representation of the target computation (Figure 14), 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 paper and the ATF auto-tuning framework [Rasch et al. 2021].

In our future work, we aim to integrate our code generation approach into the *MLIR* compiler framework [Lattner et al. 2021], building on work-in-progress results [Google SIG MLIR Open Design Meeting 2020], thereby making our work better accessible to the community. We consider approaches such as *AnyDSL* [Leißa et al. 2018] and *BuildIt* [Brahmakshatriya and Amarasinghe 2021] as further, interesting frameworks in which our compiler could be implemented.

5.1 Scheduling Approaches

Performance. Figures 17-22 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 21, we achieve a speedup $> 2\times$ 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 14), its 3rd-dimension (a.k.a. k-dimension). The difficulties of

TVM with reduction computations becomes particularly obvious when computing dot products (Dot) on GPUs (Figure 17): the Dot's main computation part is a reduction computation (via point-wise addition, see Figure 14), thus requiring reduction-focussed optimization, in particular when targeting the highly-parallel architecture of GPUs: in the case of Dot (Figure 17), 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 [Apache TVM Community 2022a]. Furthermore, while TVM's Ansor rigidly parallelizes outer dimensions [Zheng et al. 2020a], 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 18), 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 the Jacobi's index functions used for memory accesses (Figure 14) are injective. For the MatMul example of ResNet-50's training phase (Figure 21), 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 [Apache TVM Community 2022c]. For MatVec on NVIDIA Ampere GPU (Figure 17), we achieve a speedup over TVM of 1.22× for the small input size, by exploiting a so-called *swizzle pattern* [Phothilimthana et al. 2019]: 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* [NVIDIA 2022f]). 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× over TVM on Intel Skylake CPU for MCC (Figure 22) 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 14) 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 [Apache TVM Documentation 2022b] (Figure 18), rather than from neural networks (as in Figures 21 and 22). 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 17-22. 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 21), 2) MCC (capsule variant) from VGG-16's training phase on NVIDIA Ampere GPU (Figure 21), and 3) MCC (capsule variant) from ResNet-50's training phase on Intel Skylake CPU (Figure 22).

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 Section 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 [Apache TVM Community 2022d].

Portability. Figure 23 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 GEMM^{T}, we always achieve higher portability than TVM. The higher portability of TVM for GEMM^{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 1 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 1. TVM program expressing Matrix-Vector Multiplication (MatVec)

While we consider the MatVec implementations of TVM (Listing 1) and our approach (Figure 6) 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 14. 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 [Apache TVM Community 2020, 2022b]. 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 described by Rasch [2024], Definition 8, 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 1, 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

5.2 Polyhedral Compilers

Performance. Figures 17-22 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 17-22 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 21 and 22). 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× on NVIDIA Ampere GPU and of > 60× 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 Figure 14), 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 17), 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 6.2. In Section 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 20), 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 19): 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 23, 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 24-28 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 2 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 [Vasilache et al. 2019]), because both compilers fully automatically generate optimized parallel code from unoptimized, sequential program code.

Rasch et al. [2020b,c] 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 6). The sequential input program is then transformed via polyhedral tool *pet* [Verdoolaege and Grosser 2012] to its polyhedral representation which is then automatically transformed to our high-level program representation, according to the methodology presented by Rasch et al. [2020b,c].

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

Listing 2. PPCG/Pluto program expressing Matrix-Vector Multiplication (MatVec)

5.3 Functional Approaches

Our previous work [Rasch et al. 2019a] already shows that while functional approaches provide a solid formal foundation for computations, they typically suffer from performance and portability issues. For this, our previous work compares our approach (in its original, proof-of-concept implementation [Rasch et al. 2019a]) to the state-of-the-art Lift [Steuwer et al. 2015] 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 6.3.

Performance/Portability. Already experimentally evaluated in previous work [Rasch et al. 2019a] and discussed in general terms in Section 6.3.

Productivity. Listing 3 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. Afterwards, 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.

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

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

Listing 3. Lift program expressing Matrix-Vector Multiplication (MatVec)

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 [Remmelg et al. 2016] and primitives slide and pad together with optimization *overlapped tiling* for expressing stencil computations [Hagedorn et al. 2018]. 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 5) to express various kinds of data-parallel computations (Figure 14): 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 5), 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[22] and beyond: for example, we are currently not aware of any Lift program being able to express the prefix-sum examples in Figure 14. 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 5.2, our approach can take as input also straightforward, sequential program code, which further contributes to the productivity of our approach.

5.4 Domain-Specific Approaches

Performance. Figures 17-22 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 17-22 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[23] 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 [Goto and Geijn 2008; Lai and Seznec 2013]. The cuBLASEx extension

[22] This work is focussed on dense computations. Lift supports sparse computations [Pizzuti et al. 2020] which we consider as future work for our approach (as also outlined in Section 8). We consider Lift's approach, based on their so-called *position dependent arrays*, as a great inspiration for our future goal.

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

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	–

Fig. 17. Speedup (higher is better) of our approach for linear algebra routines on GPUs and CPUs 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

Fig. 18. Speedup (higher is better) of our approach for stencil computations on GPUs and CPUs 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	–	–	–	–	–	–	–	–

Fig. 19. Speedup (higher is better) of our approach for quantum chemistry computations Coupled Cluster (CCSD(T)) on GPUs and CPUs 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.

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 [NVIDIA 2017]. 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

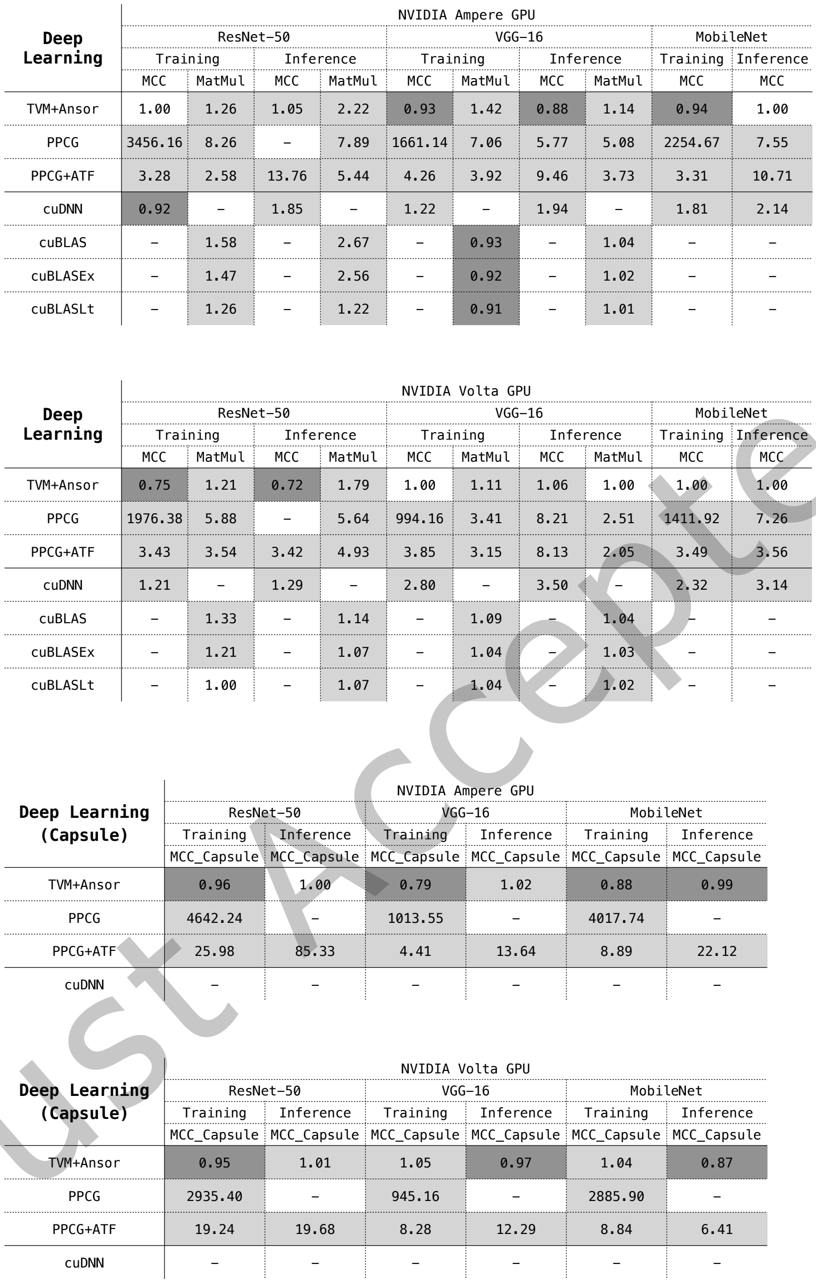
Data Mining	NVIDIA Ampere GPU					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
TVM+Ansor	–	–	–	–	–	–
PPCG	1.49	1.05	1.12	1.22	1.37	1.56
PPCG+ATF	1.40	1.22	1.50	1.63	1.83	2.12

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

Fig. 20. Speedup (higher is better) of our approach for data mining algorithm Probabilistic Record Linkage (PRL) on GPUs and CPUs 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	–	–	–	–	–	–

Fig. 21. 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 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.

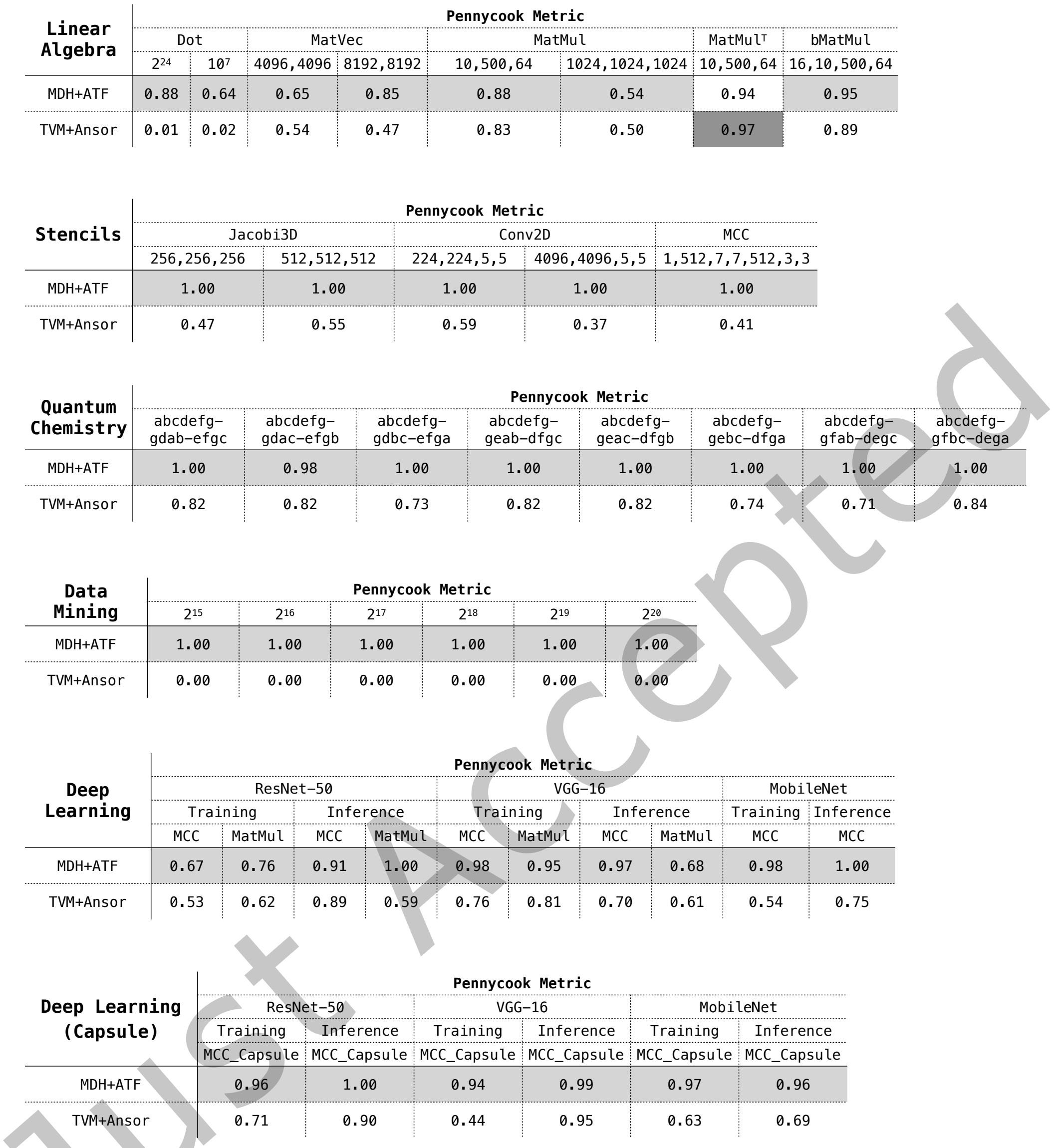
Deep Learning	Intel Skylake CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.05	1.14	1.20	1.97	1.14	2.38	1.27	3.01	1.40
Pluto	355.81	49.57	364.43	13.93	130.80	93.21	186.25	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	–	–	–	–	–	–

Fig. 22. 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 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

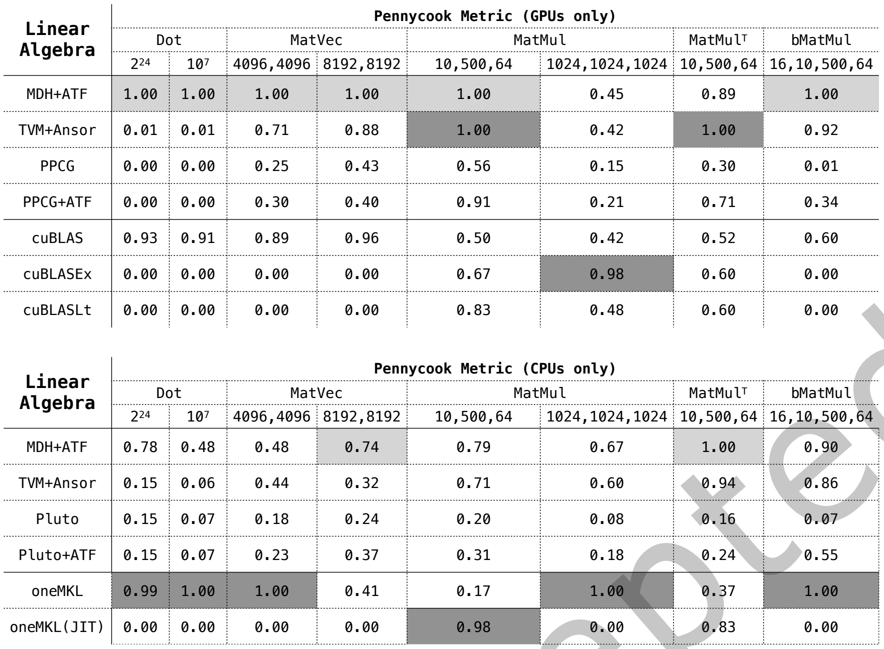
Data Mining	Pennycook Metric					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH+ATF	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.00	0.00	0.00	0.00	0.00	0.00

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

Fig. 23. Portability (higher is better), according to Pennycook metric, of our MDH-based approach and TVM over GPUs and CPUs for case studies. 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.

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



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

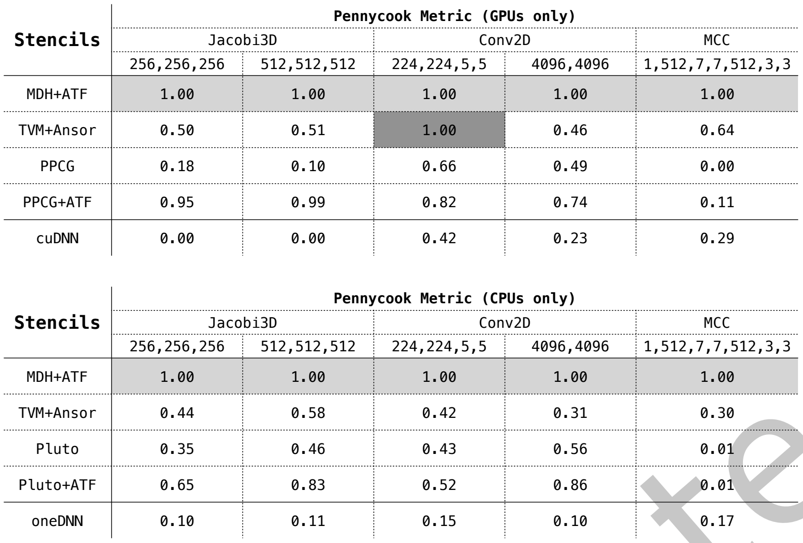
Fig. 24. Portability (higher is better), according to Pennycook metric, for linear algebra routines computed on only GPUs or CPUs, respectively. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

float type and thus meets the accuracy expected from the computation, we achieve a speedup of 1.11× over cuBLASEx.[24]

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 [Tillet and Cox 2017]. 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 18), because the libraries re-use optimizations for these computations originally intended for linear algebra routines [Li et al. 2016], 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 20), 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.

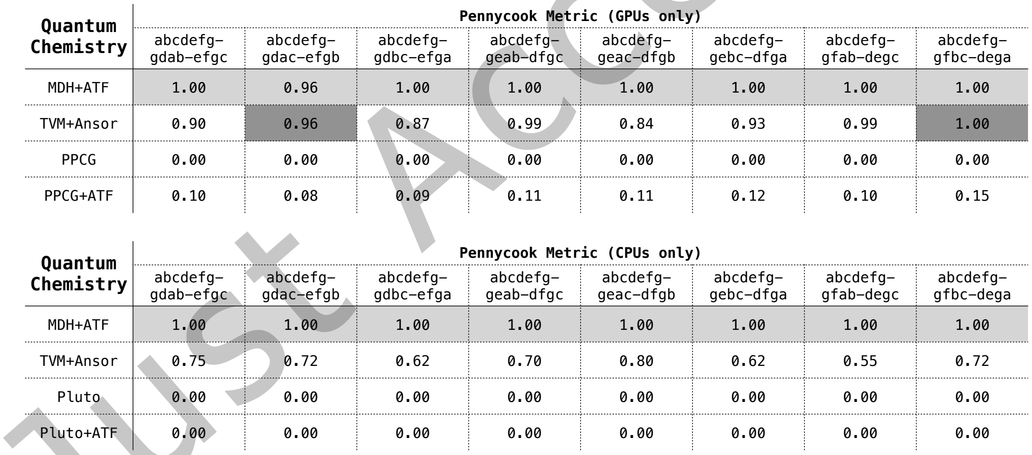
[24] For the interested reader, Rasch [2024] (Section D.2) reports the runtime of cuBLASEx for all its algorithm variants, including reports for the accuracy achieved by the different variants.



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

Fig. 25. Portability (higher is better), according to Pennycook metric, for stencil computations computed on only GPUs or CPUs, respectively. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

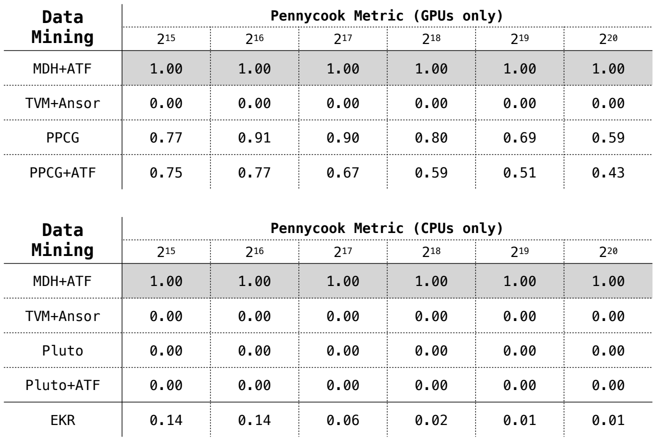


Quantum Chemistry	Pennycook Metric (GPUs only)							
	abcdefg–gdab–efgc	abcdefg–gdac–efgb	abcdefg–gdbc–efga	abcdefg–geab–dfgc	abcdefg–geac–dfgb	abcdefg–gebc–dfga	abcdefg–gfab–degc	abcdefg–gfbc–dega
MDH+ATF	1.00	0.96	1.00	1.00	1.00	1.00	1.00	1.00
TVM+Ansor	0.90	0.96	0.87	0.99	0.84	0.93	0.99	1.00
PPCG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

Fig. 26. Portability (higher is better), according to Pennycook metric, for quantum chemistry computation Coupled Cluster (CCSD(T)) computed on only GPUs or CPUs, respectively. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

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 23 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



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

Fig. 27. Portability (higher is better), according to Pennycook metric, for data mining algorithm Probabilistic Record Linkage (PRL) computed on only GPUs or CPUs, respectively. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

unsupported studies also a portability of only 0 in Figures 24-28 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 Section 8).

Productivity. Listing 4 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 4.

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 only, 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 17-22 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.

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

Fig. 28. Portability (higher is better), according to Pennycook metric, for deep learning computations computed on only GPUs or CPUs, respectively. The restriction simplifies for frameworks with limited architectural support (such as polyhedral compilers and vendor libraries) the portability comparisons against our approach.

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

Listing 4. cuBLAS program expressing Matrix-Vector Multiplication (MatVec)

6 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 6.1-6.3 our approach to each of these three classes – in terms of *performance*, *portability*, and *productivity*. In contrast to Section 5, which has compared our approach against these classes experimentally, this section is focussed on discussions in a more general, non-experimental context. Afterwards, we outline domain-specific approaches in Section 6.4, which are specifically designed and optimized toward their target application domains. In Section 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. Finally, we discuss in Section 6.6 the differences between our approach introduced in this paper and the already existing work on MDHs.

6.1 Scheduling Approaches

Popular examples of scheduling approaches include *UTF* [Kelly and Pugh 1998], *URUK* [Girbal et al. 2006], *CHill* [Chen et al. 2008; Khan et al. 2013], *Halide* [Ragan-Kelley et al. 2013], *Clay* [Bagnères et al. 2016], *TVM* [Chen et al. 2018a], *TeML* [Susungi et al. 2020], *Tiramisu* [Baghdadi et al. 2019], *DaCe* [Ben-Nun et al. 2019], *Fireiron* [Hagedorn et al. 2020a], *Elevate* [Hagedorn et al. 2020b], *DISTAL* [Yadav et al. 2022], and *LoopStack* [Wasti et al. 2022]. While scheduling approaches usually achieve high performance, they often have difficulties with achieving portability and productivity, as we discuss in the following.[25]

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 5 show that compared to the state-of-the-art scheduling approach TVM (using its recent Ansor optimizer [Zheng et al. 2020a] 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 5 discusses that our better performance is due to the design and structure of our general optimization space (Table 1) which can be efficiently explored, fully automatically, using state-of-the-art auto tuning techniques [Rasch et al. 2021]. 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 [Zheng et al. 2020a], including Halide, pyTorch [Paszke et al. 2019], and the recent FlexTensor optimizer [Zheng et al. 2020b].

Recent approach TensorIR [Feng et al. 2023] 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* [NVIDIA 2017] extensions of NVIDIA GPUs, which compute in

[25] Rasch et al. [2023] 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.

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 paper, 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 14). 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 15) 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 [Hennessy and Patterson 2019].

In contrast, we introduce in this paper 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 3) as target for our high-level programs (Section 2), and we describe formally how our high-level programs are automatically lowered to our low-level representation (Section 4), based on the architecture-agnostic optimization space in Table 1. Rasch [2024] (Section E) 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 [Chen et al. 2018b], 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 5.1) as well as scheduling language [Apache TVM Community 2022e]. Safety in parallel programming is an ongoing major demand, in particular from industry [Khronos 2022a].

Auto schedulers, such as Halide's optimization engine [Mullapudi et al. 2016] and TVM's recent Ansor [Zheng et al. 2020a], 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× already for straightforward dot products (Figure 17). 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 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 15). In contrast, approaches such as introduced by Clément and Cohen [2022] 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 [Apache TVM Community 2020, 2022b], e.g., as required for expressing the *Maximum Bottom Box Sum* example in Figure 14. Also, TVM has difficulties with user-defined combine operators [Apache TVM Community 2022d] and thus crashes for example *Probabilistic Record Linkage* in Figure 14. In contrast to TVM, we introduce a formal methodology about of how to manage different kinds of arbitrary, user-defined combine operators (Section 3), which is considered challenging [Apache TVM Community 2020].

6.2 Polyhedral Approaches

Polyhedral approaches, as introduced by Feautrier [1992], as well as *Pluto* [Bondhugula et al. 2008b], *Polly* [Grosser et al. 2012], *PPCG* [Verdoolaege et al. 2013], *Polyhedral Tensor Schedulers* [Meister et al. 2019], *TC* [Vasilache et al. 2019], and *AKG* [Bastoul et al. 2022] rely on a formal, geometrically-inspired 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 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* [Wolf and Lam 1991]), 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* [Bondhugula et al. 2008a]), 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 [Consolaro et al. 2024].

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 2 of this paper: 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, these semantic information are 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 14), 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 5. Similarly to our approach, 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 14, e.g., PRL which is popular in data mining. Moreover, TC outsources the computation of its combine operators to the NVIDIA CUB library [NVIDIA 2022a]; 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 [Doerfert et al. 2015; Reddy et al. 2016]. However, these approaches are limited to a subset of computations, e.g., by not supporting user-defined scalar types [Doerfert et al. 2015] (as required for our PRL example in Figure 14), or by being limited to GPUs only [Reddy et al. 2016]. 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 [Google SIG MLIR Open Design Meeting 2020].

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 [Hennessy and Patterson 2019] with potentially more complex memory and core hierarchies than GPUs and CPUs. Moreover, while we introduce an explicit low-level representation (Section 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 explicit 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 by Rasch [2024], Section E, and planned to be discussed thoroughly in future work). In contrast, code generation from the polyhedral model has proven challenging [Bastoul et al. 2022; Grosser et al. 2015; Vasilache et al. 2022].

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 2; thus, our approach can be considered as less productive than many polyhedral compilers. However, Rasch et al. [2020b,c] 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` [Verdoolaege and Grosser 2012] 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 6.5), and mapping the optimized and parallelizable code eventually to parallel architectures based on the concepts and methodologies introduced in this paper.

6.3 Functional Approaches

Functional approaches map data-parallel computations that are expressed via small, formally defined building blocks (a.k.a. patterns [Gorlatch and Cole 2011], 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 [Chakravarty et al. 2011], Obsidian [Svensson et al. 2011], so-called *skeleton libraries* [Aldinucci et al. 2017; Enmyren and Kessler 2010; Ernstsson et al. 2018; Steuwer et al. 2011], and the modern Lift approach [Steuwer et al. 2015] (recently also known as RISE [Steuwer et al. 2022]).

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 [Steuwer et al. 2017] 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 [Hagedorn et al. 2020b] 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 5), without involving the user for optimization decisions. In particular, our previous work already showed that our approach – even in its original, proof-of-concept implementation [Rasch et al. 2019a] – can significantly outperform Lift on GPU and CPU [Rasch et al. 2019a]. 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* [NVIDIA 2018]) 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 architectures* [Hennessy and Patterson 2019], 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 [Steuwer et al. 2017]; 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 15) 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 6 and 15), 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 5.3); 2) relying on complex function nestings and compositions for expressing computations, rather than avoiding nesting and relying on a fixed composition structure of functions, as in our approach (Figure 5); 3) requiring new patterns for targeting new classes of data-parallel computations (such as patterns `slide` and `pad` for stencils [Hagedorn et al. 2018]), which have to be non-trivially integrated into Lift's type and optimization system (often via extensions of the systems [Hagedorn et al. 2018; Remmelg et al. 2016]), instead of relying on a fixed set of expressive patterns (Figure 6) and generalized optimizations (Table 1) that work for various kinds of data-parallel computations (Figure 14); 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 4). 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, because the existing patterns need to be combined with the new ones via composition and nesting (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 14).

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) [Remmelg et al. 2016], as per design in Lift, multi-dimensional data is 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 5). Rasch et al. [2020b,c] 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 [Steuwer et al. 2015] required notable extension for targeting, e.g., matrix multiplications (so-called *macro-rules* had to be added to Lift [Remmelg et al. 2016]) and stencil computations (primitives `slide` and `pad` were added, and Lift's tiling optimization had to be extended toward *overlapped tiling* [Hagedorn et al. 2018]). 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.

6.4 Domain-Specific Approaches

Many approaches focus on code generation and optimization for particular domains. A popular domain-specific approach is *ATLAS* [Whaley and Dongarra 1998] for linear algebra routines on CPUs[26]. Similar to ATLAS, approach *FFTW* [Frigo and Johnson 1998] targets *Fast Fourier Transform (FFT)*, and *SPIRAL* [Puschel et al. 2005] works for *Digital Signal Processing (DSP)*.

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 [NVIDIA 2022b] 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 [Intel 2022c] computes with high performance linear algebra routines on Intel CPUs, and libraries NVIDIA cuDNN [NVIDIA 2022e] and Intel oneDNN [Intel 2022b] 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 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.

[26] Previous work [Rasch et al. 2021] shows that MDH (already in its original, proof-of-concept implementation) achieves higher performance than ATLAS.

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 [Tillet and Cox 2017]. Thereby, our approach often outperforms domain-specific approaches (as confirmed in Section 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 5.4), which is cumbersome for the user.

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, like *loop skewing* [Wolf and Lam 1991], 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 et al. [2020b,c]. Similarly, we consider the approaches introduced by Farzan and Nicolet [2019]; Frigo et al. [1999]; Gunnels et al. [2001]; Yang et al. [2021], 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 [Henriksen et al. 2017], Dex [Paszke et al. 2021], and ATL [Liu et al. 2022] are further approaches focussed on high-level program transformations, like advanced *flattening* mechanisms [Henriksen et al. 2019], 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.

6.6 Existing Work on MDH

Our work is inspired by the algebraic formalism of Multi-Dimensional Homomorphisms (MDHs) which is introduced in the work-in-progress paper [Rasch and Gorlatch 2016]. The MDH approach, as presented in the previous work, relies on a semi-formal foundation and focuses on code generation for the OpenCL programming model only [Rasch et al. 2019a]. This work makes major contributions over the existing work on MDHs and its OpenCL code generation approach.

We introduce a full formalization of MDH's high-level program representation. In our new formalism, we rely on expressive typing: for example, we encode MDHs' data sizes into our type system, e.g., by introducing *index sets* for MDAs (Definition 1), and we respect and maintain these sets thoroughly during MDH computations. Our expressive typing significantly contributes to correct and simplified code generation, as all relevant type and data size information are contained in our formal, low-level program representation (Figure 15) from which we eventually generate executable program code (Section 3). In contrast, the existing MDH work considers multi-dimensional arrays (MDAs) of arbitrary sizes and dimensionalities to be all of the same, straightforward type, which has greatly simplified the design of the proof-of-concept MDH formalism introduced by Rasch and Gorlatch [2016] (in particular, the definition and usage of combine operators), but at the cost of significantly harder and error-prone code generation: all the missing, type-relevant information need to be elaborated by the implementer of the code generator in the existing MDH work, e.g., allocation sizes of fast memory resources used for caching input data or for storing computed intermediate results. Furthermore, while the original MDH work [Rasch and Gorlatch 2016] is focused on introducing higher-order function `md_hom` only, this work particularly also introduces higher order functions `inp_view` and `out_view` (Section 2.3) which express input and output views in a formally structured and concise manner, and which are central building blocks in our new approach for expressing computations (Figure 14). Also, by introducing and exploiting the index set concept for MDAs, we have improved the definition of the concatenation operator $+\!\!+$ (Example 1) toward commutativity, which is required for important optimizations. e.g., loop permutations (expressed via Parameters `D1`, `S1`, `R1` in Table 1).

A further substantial improvement is the introduction of our low-level representation (Section 3). It relies on a novel combination of tuning parameters (Table 1) that enhance, generalize, and extend the existing, proof-of-concept MDH parameters which capture a subset of OpenCL-orientated features only [Rasch et al. 2019a]. Moreover, while the existing MDH work introduces formally only parameters for flexibly choosing numbers of threads [Rasch and Gorlatch 2016] (which corresponds to a very limited variant of our tuning parameter `0` in Table 1, because our parameter `0` also choses numbers of memory tiles and is not restricted to OpenCL), the other OpenCL parameters are introduced and discussed by Rasch et al. [2019a] only informally, from a technical perspective. With our novel parameter set, we are able to target various kinds of programming models (e.g., also CUDA, as in Section 5) and also to express important optimizations that are beyond the existing work on MDH, e.g., optimizing the memory access pattern of computations: for example, we achieve speedups > 2× over existing MDH for the deep learning computations discussed in Section 5. Our new tuning parameters are expressive enough to represent state-of-the-art, data-parallel implementations, e.g., as generated by scheduling and polyhedral approaches (shown by Rasch [2024], Figures 20-23), and our experiments in Section 5 confirm that auto-tuning our parameters enables performance beyond the state of the art, including hand-optimized solutions provided by vendors, which is not possible when using the existing MDH approach. The expressivity of our parameters particularly also enables comparing significantly differently optimized implementations, based on the values of formally specified tuning parameters, which we consider as promising for structured performance analysis in future work. Moreover, our new low-level representation targets architectures that may have arbitrarily deep memory and core hierarchies, by having optimized our representation toward an *Abstract System Model* (Definition 10). In contrast, the existing MDH work is focused on OpenCL-compatible architectures only.

Our experimental evaluation extends the previous MDH experiments by comparing also to the popular state-of-practice approach TVM which is attracting increasing attention from both academia [Apache Software Foundation 2021] and industry [OctoML 2022]. Also, we compare to the popular polyhedral compilers PPCG and Pluto, as well as the currently newest versions of hand-optimized, high-performance libraries provided by vendors. Furthermore, we have included a real-world case study in our experiments, considering the most time-intensive computations within the three popular deep learning neural networks `ResNet-50`, `VGG-16`, and `MobileNet`; the study also includes Capsule-style convolution computations, which are considered challenging to optimize [Barham and

Isard 2019]. Moreover, Table 14 analyzes MDH's expressivity using new examples: it shows that MDH – based on the new contributions of this work (e.g., view functions) – is capable of expressing computations bMatMuL, MCC_Capsule, Histo, scan, and MBBS, which have not been expressed via MDH in previous work. Our experiments confirm that we achieve high performance for bMatMuL and MCC_Capsule on GPUs and CPUs, and our future work aims to thoroughly analyze our approach for computations Histo, scan, and MBBS in terms of performance, portability, and productivity.

7 CONCLUSION

We introduce a formal (de/re)-composition approach for data-parallel computations targeting state-of-the-art parallel architectures. 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 from 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.

8 FUTURE WORK

We consider this work as a promising starting point for future directions. A major future goal is to extend our approach toward expressing and optimizing simultaneously multiple data-parallel computations (e.g., matrix multiplication followed by convolution), rather than optimizing computations individually and thus independently from each other (e.g., only matrix multiplication or only convolution). Such extension enables optimizations, such as *kernel fusion*, which is important for the overall application performance and considered challenging [Fukuhara and Takimoto 2022; Li et al. 2022; Wahib and Maruyama 2014]. We see this work as a promising foundation for our future goal, because it enables expressing and reasoning about different computations in the same formal framework. Targeting computations on sparse input/output data formats, inspired by Ben-Nun et al. [2017]; Hall [2020]; Kjolstad et al. [2017]; Pizzuti et al. [2020], is a further major goal, which requires extending our approach toward irregularly-shaped input and output data, similarly as done by Pizzuti et al. [2020]. Regarding our optimization process, we aim to introduce an analytical cost model for computations expressed in our formalism – based on operational semantics – thereby accelerating (or even avoiding) the auto-tuning overhead, similarly as done by Li et al. [2021]; Muller and Hoffmann [2021]. Moreover, we aim to incorporate methods from machine learning into our optimization process [Leather et al. 2014; Merouani et al. 2024], instead of relying on empirical auto-tuning methods only. To make our work better accessible for the community, we aim to implement our approach into *MLIR* [Lattner et al. 2021] which offers a reusable compiler infrastructure. The contributions of this work give a precise, formal recipe of how to implement our introduced methods into approaches such as MLIR. Moreover, relying on the MLIR framework will contribute to a structured code generation process in assembly-level programming models, such as LLVM [Lattner and Adve 2004] and NVIDIA PTX [NVIDIA 2022i]. We consider targeting assembly languages as important for our future work: assembly code offers further, low-level optimization opportunities [Goto and Geijn 2008; Lai and Seznec 2013], thereby enabling our approach to potentially achieve higher performance than reported in Section 5 for our generated CUDA and OpenCL code. Also, we aim to extend our approach toward distributed multi-device systems that are heterogeneous, inspired by dynamic load balancing approaches [Chen et al. 2010] and advanced data distributions techniques [Yadav et al. 2022]. Targeting domain-specific hardware extensions, such as *NVIDIA Tensor Cores* [NVIDIA 2017] is also an important goal for our future work, as such extensions allow significantly accelerating computations for the target of the extensions (e.g., deep learning [Markidis et al. 2018]). Finally, we aim to support more target backends (additionally to OpenMP, CUDA, and OpenCL), e.g., AMD's *HIP* [AMD 2024] which is efficient for programming AMD GPUs. Similarly, we consider *Triton* [Tillet et al. 2019], *AMOS* [Zheng et al. 2022], and *Graphene* [Hagedorn et al. 2023] as further, promising backends for our approach.

ACKNOWLEDGMENTS

This work was funded by the *Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)* – project *PPP-DL (470527619)*. I would like to thank Richard Schulze for conducting the extensive set of experiments and the reviewers for their thorough reading of the paper and their comments and remarks that helped us to improve this work.

REFERENCES

Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing* (2017).

AMD. 2024. HIP: C++ Heterogeneous-Compute Interface for Portability. https://github.com/ROCm/HIP.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

Apache. 2022. TVM: Open Deep Learning Compiler Stack. https://github.com/apache/tvm.

Apache Software Foundation. 2021. TVM and Open Source ML Acceleration Conference. https://www.tvmcon.org.

Apache TVM Community. 2020. Non top-level reductions in compute statements. https://discuss.tvm.apache.org/t/non-top-level-reductions-in-compute-statements/5693.

Apache TVM Community. 2022a. Bind reduce axis to blocks. https://discuss.tvm.apache.org/t/bind-reduce-axis-to-blocks/2907.

Apache TVM Community. 2022b. Expressing nested reduce operations. https://discuss.tvm.apache.org/t/expressing-nested-reduce-operations/8784.

Apache TVM Community. 2022c. Implementing Array Packing via cache_read. https://discuss.tvm.apache.org/t/implementing-array-packing-via-cache-read/13360.

Apache TVM Community. 2022d. Invalid comm_reducer. https://discuss.tvm.apache.org/t/invalid-comm-reducer/12788.

Apache TVM Community. 2022e. Undetected parallelization issue. https://discuss.tvm.apache.org/t/undetected-parallelization-issue/13224.

Apache TVM Documentation. 2022a. Bind `ivar` to thread index `thread_ivar`. https://tvm.apache.org/docs/reference/api/python/te.html?highlight=bind#tvm.te.Stage.bind.

Apache TVM Documentation. 2022b. Tuning High Performance Convolution on NVIDIA GPUs. https://tvm.apache.org/docs/how_to/tune_with_autotvm/tune_conv2d_cuda.html.

David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (dec 1994), 345–420. https://doi.org/10.1145/197405.197406

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. Association for Computing Machinery, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083. https://doi.org/10.1109/JPROC.2018.2841200

Paul Barham and Michael Isard. 2019. Machine Learning Systems Are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441

Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. 2022. Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 313–324. https://doi.org/10.1109/CGO53902.2022.9741260

Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. 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)*.

Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. *SIGPLAN Not.* 52, 8 (Jan. 2017), 235–248. https://doi.org/10.1145/3155284.3018756

Richard S. Bird. 1989. Lectures on Constructive Functional Programming. In *Constructive Methods in Computing Science*, Manfred Broy (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–217.

Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. 1995. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering* 1, 1 (1995), 57–94. https://doi.org/10.1007/BF02249046

Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. arXiv:cs.PF/2003.00532

Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008a. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, Laurie Hendren (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–146.

Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. 2008b. 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.

Ajay Brahmakshatriya and Saman Amarasinghe. 2021. 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)*. 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

C++ reference. 2022. Date and time utilities. https://en.cppreference.com/w/cpp/chrono.

José María Cecilia, José Manuel García, and Manuel Ujaldón. 2012. CUDA 2D Stencil Computations for the Jacobi Method. In *Applied Parallel and Scientific Computing*, Kristján Jónasson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–183.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Technical Report 08-897, U. of Southern California.

Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. 2010. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470413

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018a. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018b. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/8b5700012be65c9da25f49408d959ca0-Paper.pdf

Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.

Basile Clément and Albert Cohen. 2022. End-to-End Translation Validation for the Halide Language. In *OOPSLA 2022 - Conference on Object-Oriented Programming Systems, Languages, and Applications (Proceedings of the ACM on Programming Languages (PACMPL))*, Vol. 6. Auckland, New Zealand. https://doi.org/10.1145/3527328

MURRAY I. COLE. 1995. PARALLEL PROGRAMMING WITH LIST HOMOMORPHISMS. *Parallel Processing Letters* 05, 02 (1995), 191–203. https://doi.org/10.1142/S0129626495000175 arXiv:https://doi.org/10.1142/S0129626495000175

Gianpietro Consolaro, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Nassim Tchoulak, Adilla Susungi, Artur Cesar Araujo Alves, Renwei Zhang, Denis Barthou, Corinne Ancourt, and Cedric Bastoul. 2024. PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 28–40. https://doi.org/10.1109/CGO57630.2024.10444791

Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions. *CoRR* abs/1505.07716 (2015). arXiv:1505.07716 http://arxiv.org/abs/1505.07716

Vincent Dumoulin and Francesco Visin. 2018. A guide to convolution arithmetic for deep learning. arXiv:stat.ML/1603.07285

Johan Enmyren and Christoph W. Kessler. 2010. 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)*. Association for Computing Machinery, New York, NY, USA, 5–14. https://doi.org/10.1145/1863482.1863487

August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46, 1 (2018), 62–80. https://doi.org/10.1007/s10766-017-0490-5

Facebook Research. 2022. Tensor Comprehensions. https://github.com/facebookresearch/TensorComprehensions.

Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-Conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 610–624. https://doi.org/10.1145/3314221.3314612

Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (1992), 313–347. https://doi.org/10.1007/BF01407835

Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. 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)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3575693.3576933

M. Frigo and S.G. Johnson. 1998. 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. 1381–1384 vol.3. https://doi.org/10.1109/ICASSP.1998.681704

M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. 285–297. https://doi.org/10.1109/SFFCS.1999.814600

Junji Fukuhara and Munehiro Takimoto. 2022. 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)*. Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/3519941.3535078

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

GNU/Linux. 2022. clock_gettime(3) – Linux man page. https://linux.die.net/man/3/clock_gettime.

Horacio González-Vélez and Mario Leyton. 2010. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience* 40, 12 (2010), 1135–1160. https://doi.org/10.1002/spe.1026 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1026

Google SIG MLIR Open Design Meeting. 2020. *Using MLIR for Multi-Dimensional Homomorphisms*. https://www.youtube.com/watch?v=RQR_9tHscMI

Sergei Gorlatch. 1999. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming* 33, 1 (1999), 1–27. https://doi.org/10.1016/S0167-6423(97)00014-2

Sergei Gorlatch and Murray Cole. 2011. Parallel skeletons. In *Encyclopedia of parallel computing*. Springer-Verlag GmbH, 1417–1422.

S. Gorlatch and C. Lengauer. 1997. (De) composition rules for parallel scan and reduction. In *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No.97TB100228)*. 23–32. https://doi.org/10.1109/MPPM.1997.715958

Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1142/S0129626412500107 arXiv:https://doi.org/10.1142/S0129626412500107

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (jul 2015), 50 pages. https://doi.org/10.1145/2743016

John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27, 4 (dec 2001), 422–455. https://doi.org/10.1145/504210.504213

Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020a. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '20)*. Association for Computing Machinery, New York, NY, USA, 71–82. https://doi.org/10.1145/3410463.3414632

Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3582016.3582018

Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020b. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408974

Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

Mary Hall. 2020. Research Challenges in Compiler Technology for Sparse Tensors. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. viii–viii. https://doi.org/10.1109/IA351965.2020.00006

Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

Haskell.org. 2022. Haskell: An advanced, purely functional programming language. https://www.haskell.org.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (jan 2019), 48–60. https://doi.org/10.1145/3282307

Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00101

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. 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)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 53–67. https://doi.org/10.1145/3293883.3295707

K Hentschel et al. 2008. Das Krebsregister-Manual der Gesellschaft der epidemiologischen Krebsregister in Deutschland e.V. Zuckschwerdt Verlag.

Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. 2018. Matrix capsules with EM routing. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HJWLfGWRb

Torsten Hoefler and Roberto Belli. 2015. 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)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

Cristina Hristea, Daniel Lenoski, and John Keen. 1997. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (SC '97)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/509593.509638

Intel. 2019. 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.

Intel. 2022a. oneAPI Math Kernel Library Link Line Advisor. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html.

Intel. 2022b. oneDNN. https://oneapi-src.github.io/oneDNN/group_dnnl_api.html.

Intel. 2022c. 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.

Wayne Kelly and William Pugh. 1998. *A framework for unifying reordering transformations*. Technical Report. Technical Report UMIACS-TR-92-126.1.

Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. *ACM Trans. Archit. Code Optim.* 9, 4, Article 31 (jan 2013), 25 pages. https://doi.org/10.1145/2400682.2400690

Khronos. 2022a. 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.

Khronos. 2022b. OpenCL: Open Standard For Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/.

Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2019. A Code Generator for High-Performance Tensor Contractions on GPUs. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 85–95. https://doi.org/10.1109/CGO.2019.8661182

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (oct 2017), 29 pages. https://doi.org/10.1145/3133901

Michael Klemm, Alejandro Duran, Xinmin Tian, Hideki Saito, Diego Caballero, and Xavier Martorell. 2012. 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)*. Springer-Verlag, Berlin, Heidelberg, 59–72. https://doi.org/10.1007/978-3-642-30961-8_5

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Junjie Lai and André Seznec. 2013. 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)*. 1–10. https://doi.org/10.1109/CGO.2013.6494986

Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. 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)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/106972.106981

C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

Hugh Leather, Edwin Bonilla, and Michael O'boyle. 2014. Automatic Feature Generation for Machine Learning–Based Optimising Compilation. *ACM Trans. Archit. Code Optim.* 11, 1, Article 14 (Feb. 2014), 32 pages. https://doi.org/10.1145/2536688

Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: a partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 119 (oct 2018), 30 pages. https://doi.org/10.1145/3276489

Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 14–27. https://doi.org/10.1109/CGO53902.2022.9741270

Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. 2021. Analytical Characterization and Design Space Exploration for Optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 928–942. https://doi.org/10.1145/3445814.3446759

Xiaqing Li, Guangyan Zhang, H. Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance Analysis of GPU-Based Convolutional Neural Networks. In *2016 45th International Conference on Parallel Processing (ICPP)*. 67–76. https://doi.org/10.1109/ICPP.2016.15

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.* 18, 4 (jul 1996), 424–453. https://doi.org/10.1145/233561.233564

MDH Project. 2024. Multi-Dimensional Homomorphisms (MDH): An Algebraic Approach Toward Performance & Portability & Productivity for Data-Parallel Computations. https://mdh-lang.org.

Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. 2014. Benchmarking the Memory Hierarchy of Modern GPUs. In *Network and Parallel Computing*, Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–156.

Benoît Meister, Eric Papenhausen Akai Kaeru, and Benoît Pradelle Silexica. 2019. Polyhedral Tensor Schedulers. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 504–512. https://doi.org/10.1109/HPCS48598.2019.9188233

Massinissa Merouani, Khaled Afif Boudaoud, Iheb Nassim Aouadj, Nassim Tchoulak, Islem Kara Bernou, Hamza Benyamina, Fatima Benbouzid-Si Tayeb, Karima Benatchba, Hugh Leather, and Riyadh Baghdadi. 2024. LOOPer: A Learned Automatic Code Optimizer For Polyhedral Compilers. arXiv:cs.PL/2403.11522

Michael Kruse. 2022. Polyhedral Parallel Code Generation. https://github.com/Meinersbur/ppcg, commit = 8a74e46, date = 19.11.2020.

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. *Proc. ACM Program. Lang.* 5, POPL, Article 25 (Jan. 2021), 31 pages. https://doi.org/10.1145/3434306

NVIDIA. 2017. *Programming Tensor Cores in CUDA 9*. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

NVIDIA. 2018. *Warp-level Primitives*. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/

NVIDIA. 2022a. CUB. https://docs.nvidia.com/cuda/cub/.

NVIDIA. 2022b. cuBLAS. https://developer.nvidia.com/cublas.

NVIDIA. 2022c. cuBLAS – BLAS-like Extension. https://docs.nvidia.com/cuda/cublas/index.html#blas-like-extension

NVIDIA. 2022d. cuBLAS – Using the cuBLASLt API. https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublaslt-api

NVIDIA. 2022e. CUDA Deep Neural Network library. https://developer.nvidia.com/cudnn

NVIDIA. 2022f. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

NVIDIA. 2022g. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/.

NVIDIA. 2022h. NVRTC. https://docs.nvidia.com/cuda/nvrtc.

NVIDIA. 2022i. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution.

OctoML. 2022. Accelerated Machine Learning Deployment. https://octoml.ai.

Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. 2021. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. *IEEE Access* 9 (2021), 134457–134502. https://doi.org/10.1109/ACCESS.2021.3110993

OpenMP. 2022. The OpenMP API Specification for Parallel Programming. https://www.openmp.org.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point: Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. *Proc. ACM Program. Lang.* 5, ICFP, Article 88 (aug 2021), 29 pages. https://doi.org/10.1145/3473593

S.J. Pennycook, J.D. Sewall, and V.W. Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958. https://doi.org/10.1016/j.future.2017.08.007

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/3377555.3377896

Victor Podlozhnyuk. 2007. Image Convolution with CUDA. *NVIDIA Corporation White Paper* (2007).

M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* 93, 2 (2005), 232–275. https://doi.org/10.1109/JPROC.2004.840306

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

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

Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2020a. dOCAL: high-level distributed programming with OpenCL and CUDA. *The Journal of Supercomputing* 76, 7 (2020), 5117–5138. https://doi.org/10.1007/s11227-019-02829-2

Ari Rasch and Sergei Gorlatch. 2016. Multi-Dimensional Homomorphisms and Their Implementation in OpenCL. In *International Workshop on High-Level Parallel Programming and Applications (HLPP)*. 101–119.

Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019a. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 354–369. https://doi.org/10.1109/PACT.2019.00035

Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2020b. `md_poly`: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms. In *Proceedings of the International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. 1–4.

Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2020c. `md_poly`: A Performance-Portable Polyhedral Compiler based on Multi-Dimensional Homomorphisms. In *ACM SRC Grand Finals Candidates, 2019 - 2020*. 1–5.

Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019b. High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. Association for Computing Machinery, New York, NY, USA, 526–533. https://doi.org/10.1145/3297280.3297330

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

Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1, Article 1 (Jan. 2021), 26 pages. https://doi.org/10.1145/3427093

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

Chandan Reddy, Michael Kruse, and Albert Cohen. 2016. 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)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/2967938.2967950

Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. 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)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2884045.2884046

Bruce Sagan. 2001. *The symmetric group: representations, combinatorial algorithms, and symmetric functions*. Vol. 203. Springer Science & Business Media.

Caio Salvador Rohwedder, Nathan Henderson, João P. L. De Carvalho, Yufei Chen, and José Nelson Amaral. 2023. 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)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3579990.3580024

Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. https://doi.org/10.48550/ARXIV.1409.1556

Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR* (2016). arXiv:quant-ph/1607.00145 http://arxiv.org/abs/1607.00145

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. 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)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1176–1182. https://doi.org/10.1109/IPDPS.2011.269

Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. *CoRR* abs/2201.03611 (2022). arXiv:2201.03611 https://arxiv.org/abs/2201.03611

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 74–85. https://doi.org/10.1109/CGO.2017.7863730

StreamHPC. 2016. Comparing Syntax for CUDA, OpenCL and HiP. https://streamhpc.com/blog/2016-04-05/comparing-syntax-cuda-opencl-hip/.

Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David R. Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data. *CoRR* abs/1911.11313 (2019). arXiv:1911.11313 http://arxiv.org/abs/1911.11313

Adilla Susungi, Norman A. Rink, Albert Cohen, Jeronimo Castrillon, and Claude Tadonki. 2020. Meta-programming for cross-domain tensor optimizations. *SIGPLAN Not.* 53, 9 (apr 2020), 79–92. https://doi.org/10.1145/3393934.3278131

Joel Svensson, Mary Sheeran, and Koen Claessen. 2011. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–173.

TensorFlow. 2022a. MobileNet v1 models for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/mobilenet.py.

TensorFlow. 2022b. ResNet models for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/resnet.py.

TensorFlow. 2022c. VGG16 model for Keras. https://github.com/keras-team/keras/blob/master/keras/applications/vgg16.py.

Philippe Tillet and David Cox. 2017. 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)*. Association for Computing Machinery, New York, NY, USA, Article 43, 12 pages. https://doi.org/10.1145/3126908.3126939

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

Uday Bondhugula. 2022. Pluto: An automatic polyhedral parallelizer and locality optimizer. https://github.com/bondhugula/pluto, commit = 12e075a, date = 31.10.2021.

Nicolas Vasilache, Oleksandr Zinenko, Aart J. C. Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, Stella Laurenzo, and Albert Cohen. 2022. Composable and Modular Code Generation in MLIR: A Structured and Retargetable Approach to Tensor Compiler Construction. arXiv:cs.PL/2202.03293

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, Vol. 141.

Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 191–202. https://doi.org/10.1109/SC.2014.21

Bram Wasti, José Pablo Cambronero, Benoit Steiner, Hugh Leather, and Aleksandar Zlateski. 2022. LoopStack: a Lightweight Tensor Algebra Compiler Stack. https://doi.org/10.48550/ARXIV.2205.00618

R.C. Whaley and J.J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 38–38. https://doi.org/10.1109/SC.1998.10004

Maurice V Wilkes. 2001. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News* 29, 1 (2001), 2–7.

M.E. Wolf and M.S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (1991), 452–471. https://doi.org/10.1109/71.97902

Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: The Distributed Tensor Algebra Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3519939.3523437

Cambridge Yang, Eric Atkinson, and Michael Carbin. 2021. Simplifying Dependent Reductions in the Polyhedral Model. *Proc. ACM Program. Lang.* 5, POPL, Article 20 (Jan. 2021), 33 pages. https://doi.org/10.1145/3434301

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020a. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 874–887. https://doi.org/10.1145/3470496.3527440

Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020b. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508