Mathematics and Computer Science
Faculty of Mathematics und Natural Sciences
University of Münster, Germany

Toward Performance & Portability & Productivity in Parallel Programming

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

Thesis Defense Presentation

Ari Rasch
University of Münster, Germany
Advisor: Prof. Dr. Sergei Gorlatch

February 13, 2025

Introductory Remark

COMPUTER SCIENCE

TOWARD
PERFORMANCE & PORTABILITY & PRODUCTIVITY
IN PARALLEL PROGRAMMING

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

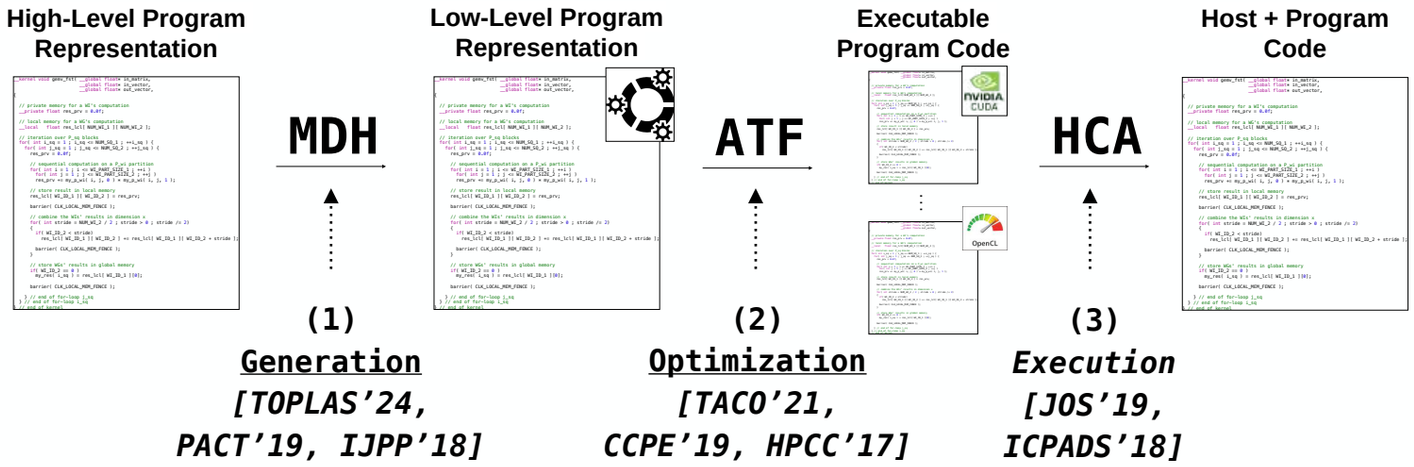
Inaugural Dissertation
for the Award of a Doctoral Degree
Dr. rer. nat.
in the Field of Mathematics and Computer Science
from the Faculty of Mathematics und Natural Sciences
of the University of Münster, Germany

submitted by

ARI RASCH

born in Essen, Germany
– 2024 –

This thesis describes three major (sub-)projects:



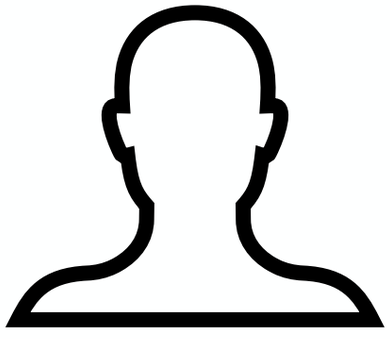
The (sub-)projects complement each other to form a holistic approach to code

Generation & Optimization & Execution

Major Contribution	Medium Contribution	Minor Contribution
Part 1 (15 slides – 187 pages)	*Part 2 (5 slides – 54 pages)*	*Part 3 (4 slides – 21 pages)*

Parallel Programming in Today's World

Parallel programming is hard:



Domain Scientist

Focus of this thesis

Programming Models

NVIDIA CUDA · OpenMP · tvm · OpenCL

Parallel Architectures

Microsoft · intel · NVIDIA · Google · ...

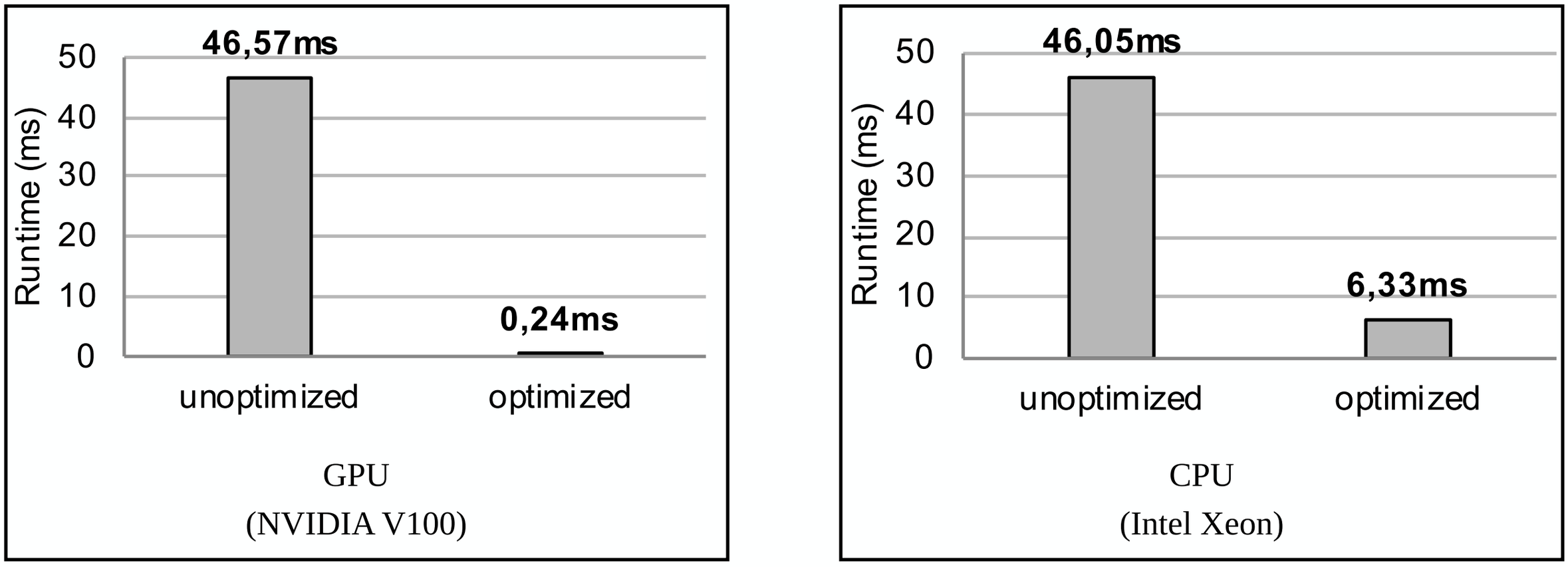
- \+ domain knowledge
- \- but lacks hardware & optimization details

Struggle with *simultaneously* achieving *Performance* & *Portability* & *Productivity*

- \+ potential for high performance
- \- require advanced optimization

Challenges: Performance & Portability & Productivity

The *Performance* challenge:

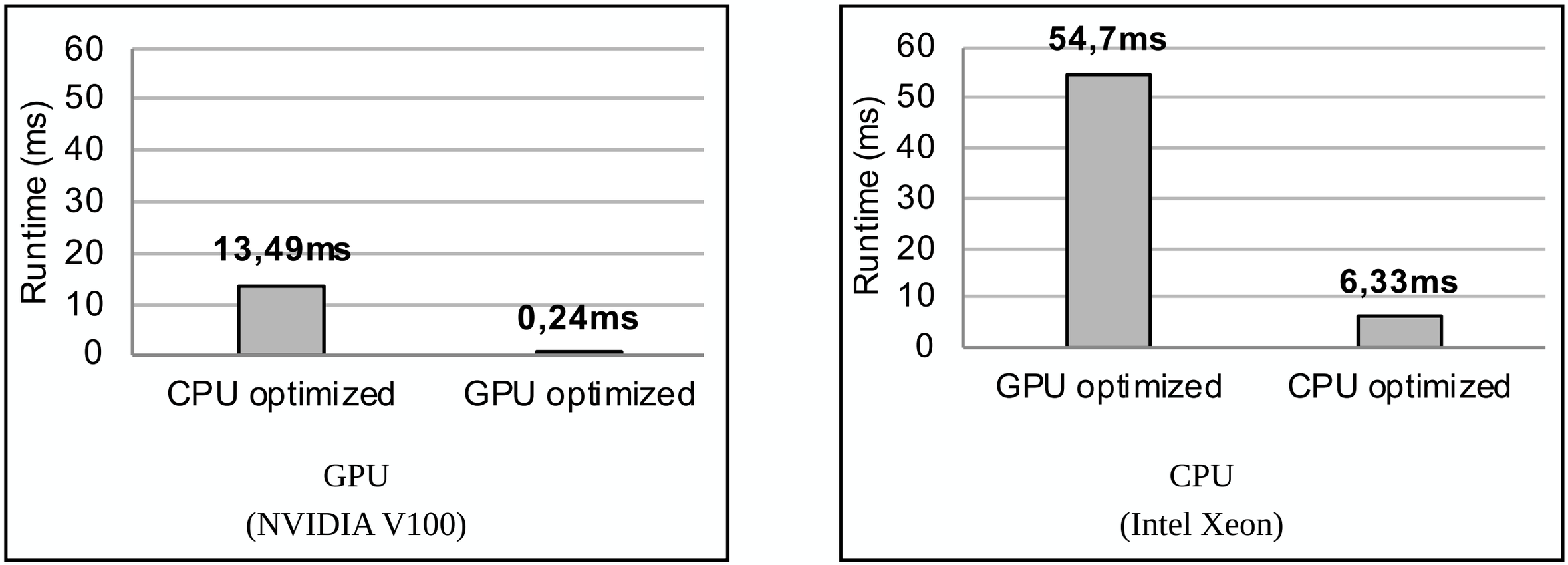


Runtime (lower is better) of unoptimized vs optimized matrix multiplication on GPU (left) and CPU (right).

High *Performance* requires *complex* optimizations

Challenges: Performance & Portability & Productivity

The *Portability* challenge:



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

High *Portability* requires *architecture(/data)-specific* optimizations

Challenges: Performance & Portability & Productivity

The *Productivity* challenge:

```
__kernel void MatMul( __global const float A[M][K] ,
                      __global const float B[K][N] ,
                      __global       float C[M][N] )
{
  int i = get_global_id(0);
  int j = get_global_id(1);

  for( int k=0 ; k<K ; ++k )
    C[i][j] += A[i][k] * B[k][j];
}
```

Naive OpenCL implementation of matrix multiplication

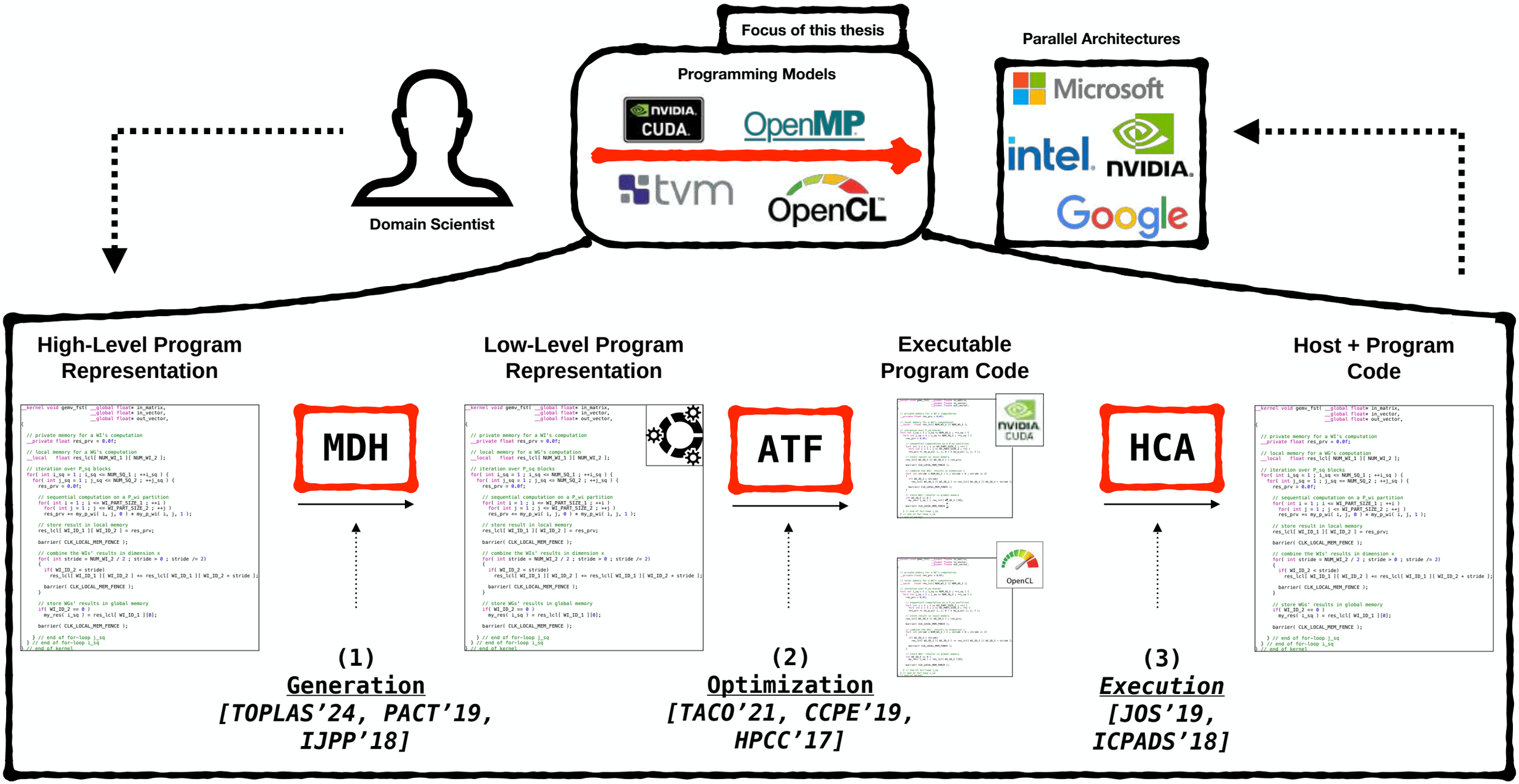
```
__kernel void MatMul( /* ... */ )
{
  const size_t i_wg_l_1 = get_group_id(2);
// ... 5 lines skipped

  __private TYPE_TS res_p[/*...*/][/*...*/];
  {
// ... 7 lines skipped
          for (size_t p_iteration_l_1 = 0; p_iteration_l_1 < (2);
              ++p_iteration_l_1) {
            for (size_t p_iteration_l_2 = 0; p_iteration_l_2 < (1)
                ; ++p_iteration_l_2) {
              size_t p_iteration_r_1 = 0;
              res_p[p_step_l_1][((p_iteration_l_1) * 1 + 0)][(0)][
                  p_step_l_2][((p_iteration_l_2) * 1 + 0)] = f(
                  a[(((l_step_l_1 x* (32 / 1) + (((p_step_l_1 *
                      (2) + (((p_iteration_l_1) * 1 + 0)) / 1) * 1
                   + i_wi_l_1 * 1 + ((((p_iteration_l_1) * 1 +
                   0)) % 1))) / 1) * (64 * 1) + i_wg_l_1 * 1 +
                   ((((p_step_l_1 * (2) + (((p_iteration_l_1)
                  * 1 + 0)) / 1) * 1 + i_wi_l_1 * 1 + ((((
                  p_iteration_l_1) * 1 + 0)) % 1))) % 1))) *
                  1024 + (((l_step_r_1 * (2 / 1) + (((
                  p_step_r_1 * (1) + (((p_iteration_r_1) * 1 +
                   0)) / 1) * 1 + i_wi_r_1 * 1 + ((((
                  p_iteration_r_1) * 1 + 0)) % 1))) / 1) * (2
                  * 1) + i_wg_r_1 * 1 + ((((p_step_r_1 * (1) +
                   (((p_iteration_r_1) * 1 + 0)) / 1) * 1 +
                  i_wi_r_1 * 1 + ((((p_iteration_r_1) * 1 + 0)
                  ) % 1))) % 1)))],
// ... 107 lines skipped
  }
```

Optimized OpenCL implementation of matrix multiplication

High *Productivity* requires *automatic* optimization

Contributions of this Thesis

This thesis introduces a novel, holistic approach to ***Generating*** & ***Optimizing*** & ***Executing*** code:

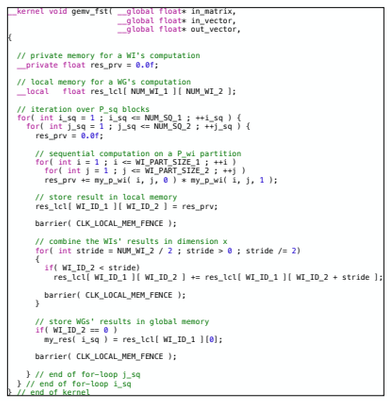


The ultimate goal of **MDH+ATF+HCA** is to simultaneously achieve **Performance** & **Portability** & **Productivity**

Outline

This talk(/thesis) is structured into three main parts:

High-Level Program Representation

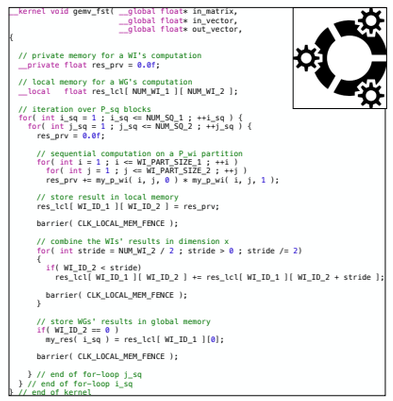


MDH

(1) Generation

[TOPLAS'24, PACT'19, IJPP'18]

Low-Level Program Representation

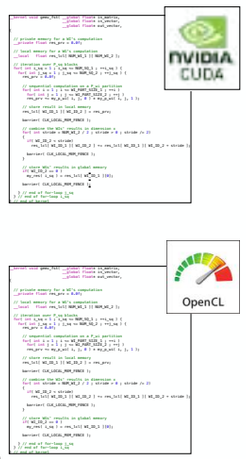


ATF

(2) Optimization

[TACO'21, CCPE'19, HPCC'17]

Executable Program Code

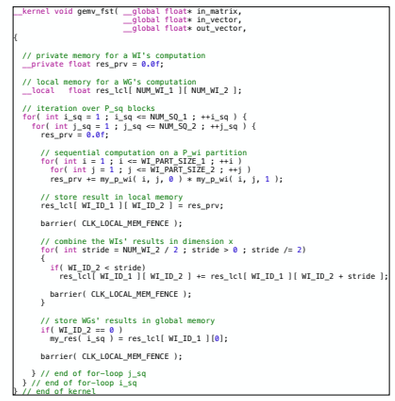


HCA

(3) *Execution*

[JOS'19, ICPADS'18]

Host + Program Code



1. Part: How to generate automatically optimizable (auto-tunable) code?

2. Part: How to optimize (auto-tune) code?

3. Part: How to execute code on (distr.) multi-dev. systems?

→ *Interface Kinds* for MDH+ATF+HCA are outlined at the end of talk

Code Generation via MDH

Overview Getting Started Code Examples Publications Citations Contact

MDH

Multi-Dimensional Homomorphisms (MDH)

An Algebraic Approach Toward Performance & Portability & Productivity for Data-Parallel Computations

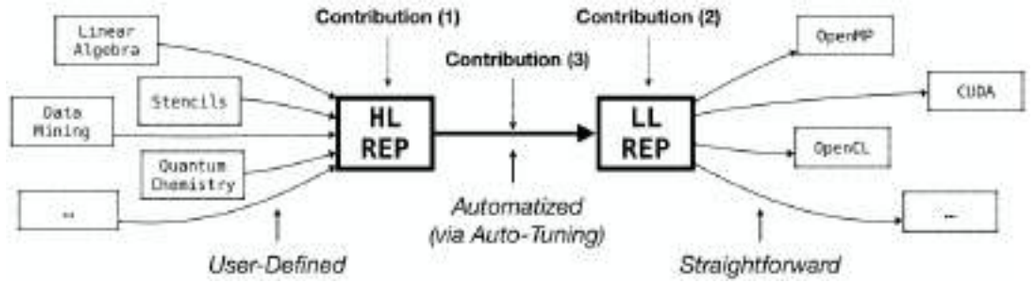
Overview

The approach of Multi-Dimensional Homomorphisms (MDH) is an algebraic formalism for systematically reasoning about *de-composition* and *re-composition* strategies of data-parallel computations (such as linear algebra routines and stencil computations) for the memory and core hierarchies of state-of-the-art parallel architectures (GPUs, multi-core CPU, multi-device and multi-node systems, etc).

The MDH approach (formally) introduces:

1. *High-Level Program Representation (Contribution 1)* that enables the user conveniently implementing data-parallel computations, agnostic from hardware and optimization details;
2. *Low-Level Program Representation (Contribution 2)* that expresses device- and data-optimized de- and re-composition strategies of computations;
3. *Lowering Process (Contribution 3)* that fully automatically lowers a data-parallel computation expressed in its high-level program representation to an optimized instance in its low-level representation, based on concepts from automatic performance optimization (a.k.a. *auto-tuning*), using the Auto-Tuning Framework (ATF).

The MDH's low-level representation is designed such that Code Generation from it (e.g., in OpenMP for CPUs, CUDA for NVIDIA GPUs, or OpenCL for multiple kinds of architectures) becomes straightforward.



Our Experiments report encouraging results on GPUs and CPUs for MDH as compared to state-of-practice approaches, including NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN which are hand-optimized libraries provided by vendors.

https://mdh-lang.org

ACM TOPLAS 2024

(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 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). 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 datasets 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

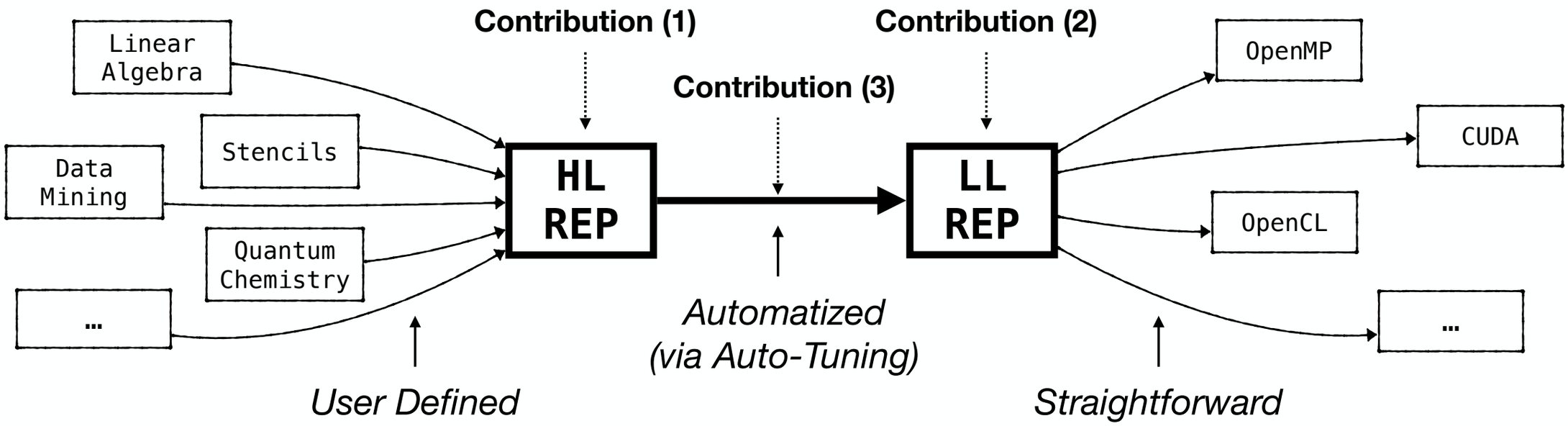
A full version of this article is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this article relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader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

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—project PPP-DL (470527619).

Author's Contact Information: Ari Rasch (Corresponding author), University of Muenster, Muenster, Germany; e-mail: a.rasch@uni-muenster.de.

Goal of **MDH**

MDH is a (formal) framework for *expressing* & *optimizing* data-parallel computations:



1. ***Contribution 1 (HL-REP):*** defines *data parallelism*, based on *common algebraic properties of computations* & introduces *higher-order functions* for expressing these computations, agnostic from hardware and optimization details while still capturing high-level information relevant for generating high-performing code

2. ***Contribution 2 (LL-REP):*** allows *expressing and reasoning about optimizations* for the memory and core hierarchies of contemporary parallel architectures & generalizes these optimizations to apply to arbitrary combinations of data-parallel computations and architectures

3. ***Contribution 3 (→):*** introduces a *structured optimization process* — for arbitrary combinations of data-parallel computations and parallel architectures — that allows *fully automatic* optimization (auto-tuning)

MDH: High-Level Representation

Goals:

1. **Uniform**:
   should be able to express any kind of data-parallel computation, without relying on domain-specific building blocks, extensions, etc.

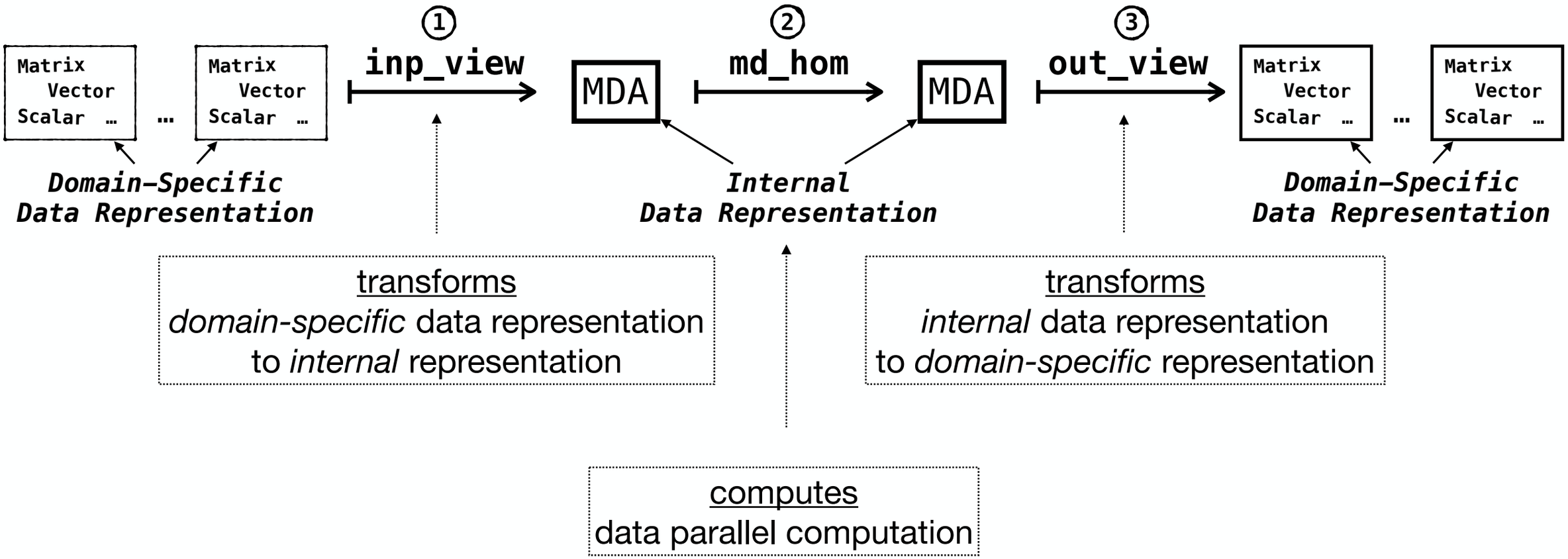
2. **Minimalistic**:
   should rely on less building blocks to keep language small and simple

3. **Structured**:
   avoiding compositions and nestings of building blocks as much as possible, thereby further contributing to the usability and simplicity of our language

```
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) )
```

***MDH High-Level Representation of* MatVec**
(*discussed later*)

MDH: High-Level Representation



Our high-level representation defines data-parallel computations as

Multi-Dimensional Homomorphisms (MDH)

and it expresses data-parallel computations using exactly

three straightforwardly composed higher-order functions only

MDH: High-Level Representation

Example: `MatVec` expressed in MDH

```
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) )
```

MDH

***MDH High-Level Representation of* MatVec**

What is happening here:

- `inp_view` captures the accesses to input data
- `md_hom` expresses the data-parallel computation
- `out_view` captures the accesses to output data

```
void MatVec( T[] M, T[] v, T[] w )
{
  for( int i=0 ; i < I ; ++i )
    for( int k=0 ; k < K ; ++k )
      w[i] += M[i][k] * v[k];
}
```

MatVec in C++

[1]We can generate such MDH expressions also automatically from straightforward (annotated) code in Python or C

MDH: High-Level Representation

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$
Dot	*	+			
MatVec	*	++	+		
MatMul	*	++	++	+	
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)
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	++	$\max_{PRL}$

	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	f_{Histo}	+	++
GenHisto	f	⊕	++

	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	$\circledast_1$	$\circledast_2$
map(f)	f	++	
reduce(⊕)	id	⊕	
reduce(⊕,⊗)	(x) ↦ (x,x)	(⊕,⊗)	

	inp_view	out_view	
Views	I	O_1	O_2
map(f)	(i) ↦ (i)	(i) ↦ (i)	
reduce(⊕)	(i) ↦ (i)	(i) ↦ ()	
reduce(⊕,⊗)	(i) ↦ (i)	(i) ↦ ()	(i) ↦ ()

7) Map/Reduce Patterns

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

	inp_view	out_view
Views	A	Out
scan(⊕)	(i) ↦ (i)	(i) ↦ (i)
MBBS	(i,j) ↦ (i,j)	(i,j) ↦ (i)

8) Prefix Sum Computations

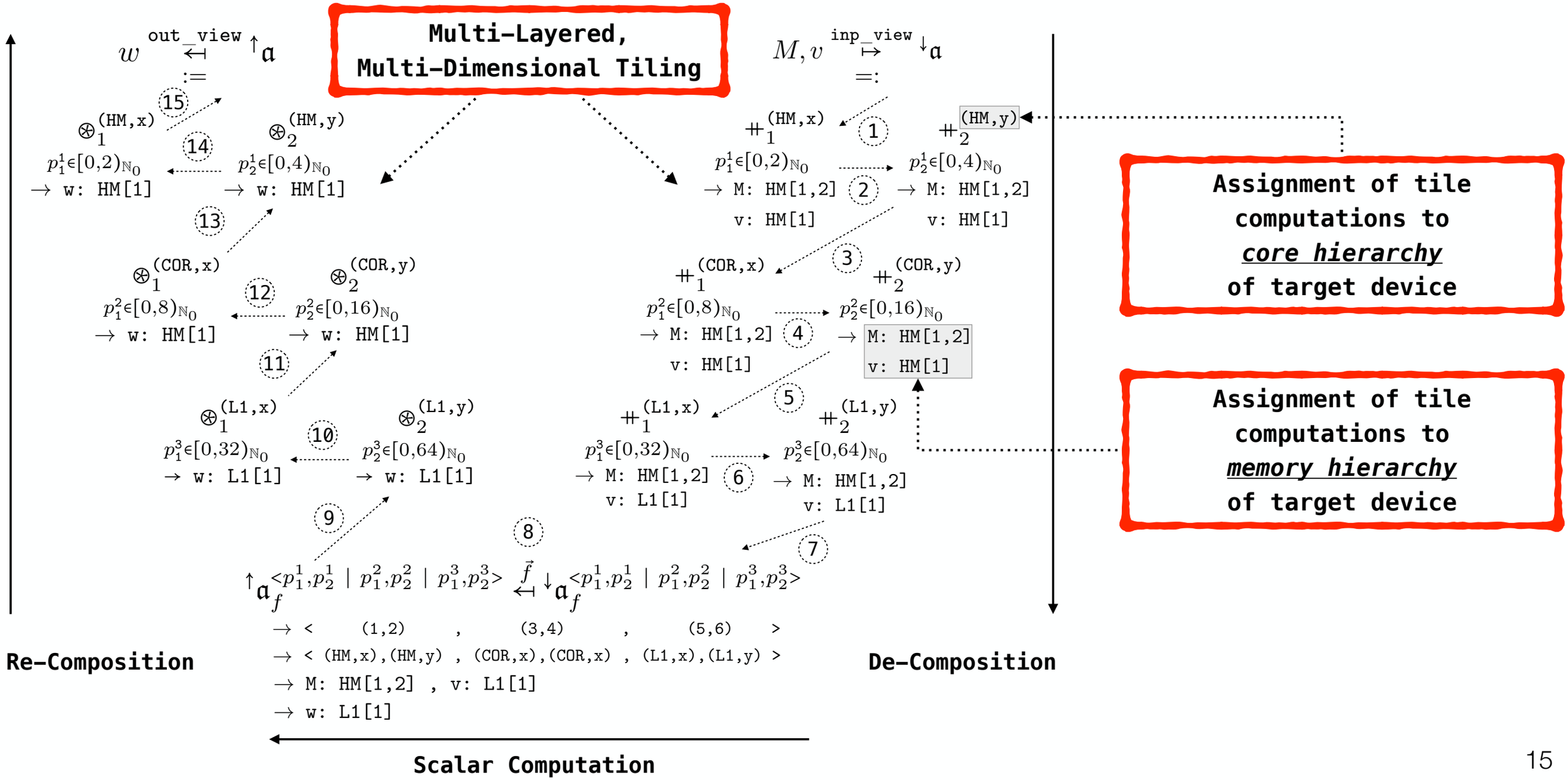
The MDH high-level representation is capable of expressing various kinds of data-parallel computations (with significantly different characteristics)

MDH: Low-Level Representation

Basic Idea

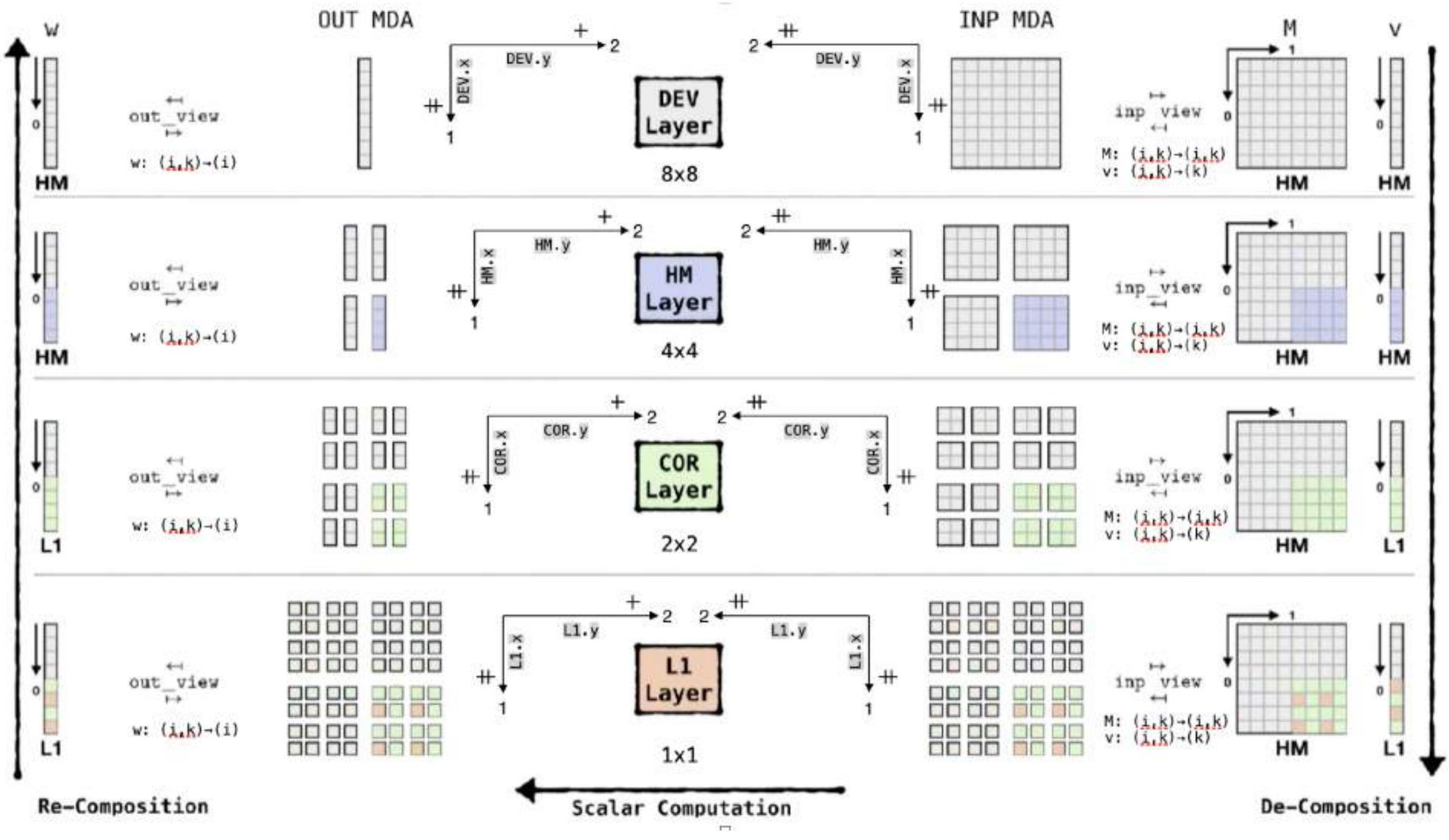
Goals:

1. Expressing a hardware- & data-optimized *de-composition* and *re-composition* of data-parallel computations, based on an *Abstract System Model (ASM)*
2. Being straightforwardly transformable to executable program code (e.g., in OpenMP, CUDA, and OpenCL) — major optimization decisions explicitly expressed in low-level representation



MDH: Low-Level Representation

Excursion: visualizing low-level instances



The (structured) design of our Low-Level Representation allows uniformly visualizing optimizations

MDH: Lowering: *High Level → Low-Level*

Based on (formally defined) performance-critical parameters, for a *structured* optimization process:

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

Table 1. Tuning parameters of our low-level expressions

exploiting core hierarchy (parallelization)

exploiting memory hierarchy (data movements)

...

...

...

...

Our parameters *unify* & *generalize* & *combine* well-proven optimizations (e.g., tiling, data movements, and parallelization)

We use our ***Auto-Tuning Framework (ATF)*** to *automatically* determine optimized values of parameters

MDH: Experimental Results

MDH is experimentally evaluated in terms of ***Performance*** & ***Portability*** & ***Productivity***:

Competitors:

1. Scheduling Approach:
   - Apache TVM [1] (GPU & CPU)
2. Polyhedral Compilers:
   - PPCG [2] (GPU)
   - Pluto [3] (CPU)
3. Functional Approach:
   - Lift [4] (GPU & CPU)
4. Domain-Specific Libraries:
   - NVIDIA cuBLAS & cuDNN (GPU)
   - Intel oneMKL & oneDNN (CPU)

Case Studies:

1. Linear Algebra Routines:
   - Matrix Multiplication (`MatMul`)
   - Matrix-Vector Multiplication (`MatVec`)
2. Stencil Computations:
   - Jacobi Computation (`Jacobi1D`)
   - Gaussian Convolution (`Conv2D`)
3. Quantum Chemistry:
   - Coupled Cluster (`CCSD(T)`)
4. Data Mining:
   - Probabilistic Record Linkage (PRL)
5. Deep Learning:
   - Multi-Channel Convolution (`MCC`)
   - Capsule-Style Convolution (`MCC_Capsule`)

[1] Chen et al., “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”, OSDI’18

[2] Verdoolaege et al., “Polyhedral Parallel Code Generation for CUDA”, TACO’13

[3] Bondhugula et al., “PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System”, PLDI’08

[4] Steuwer et al., “Generating Performance Portable Code using Rewrite Rules”, ICFP’15

MDH: Experimental Results

Performance Evaluation: (via runtime comparison)

Highlights only

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	–	–

NVIDIA

MDH speedup over

- TVM: 0.88x – 2.22x
- PPCG: 2.58x – 13.76x
- *(cuBLAS/cuDNN: 0.91x – 2.67x)*

NVCC vs NVRTC

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	–	–

intel

MDH speedup over

- TVM: 1.05 – 3.01x
- Pluto: 6.29x – 364.43x
- *(oneMKL/oneDNN: 0.39x – 9.01x)*

Case Study "Deep Learning" for which most competitors are highly optimized (most challenging for us!)

Significantly higher speedups for other case studies,
e.g., >170x over TVM on GPU already for straightforward dot products

MDH: Experimental Results

Highlights only

Portability Evaluation: (via Pennycook Metric [6])

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

The other related approaches achieve lowest portability — of “0.00” — only,
because they are designed for particular architectures and/or application classes only

[6] Pennycook et al., “Implications of a Metric for Performance Portability”, FGCS’19

MDH: Experimental Results

Highlights only

Productivity Evaluation: (via intuitive argumentation)

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

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

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

Listing 2. PPCG/Pluto program expressing Matrix-Vector Multiplication (MatVec)[1]

```
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)

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

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

[1] *MDH can also take (annotated) C code as input [IMPACT'20]*

MDH: Summary

- MDH combines three key goals — ***Performance*** & ***Portability*** & ***Productivity*** — as compared to related approaches

- For this, MDH **formally introduces program representations** on both:

  - **high level**, for conveniently expressing — in one uniform formalism — the various kinds of data-parallel computations, agnostic from hardware and optimization details, while still capturing all information relevant for generating high-performance program code

  - **low level**, which allows uniformly reasoning — in the same formalism — about optimized (de/re)-compositions of data-parallel computations for the memory and core hierarchies of contemporary parallel architectures (GPUs, CPUs, etc)

- MDH **lowers** instances in its high-level representation to device- and data-optimized instances in its low-level representation, in a **formally sound** manner, by introducing a generic search space that is based on **performance-critical parameters** & **auto-tuning**

- Our **experiments** confirm that MDH often achieves higher ***Performance*** & ***Portability*** & ***Productivity*** than popular state-of-practice approaches, including hand-optimized libraries provided by vendors

MDH: Summary

That was a very quick, informal dive!

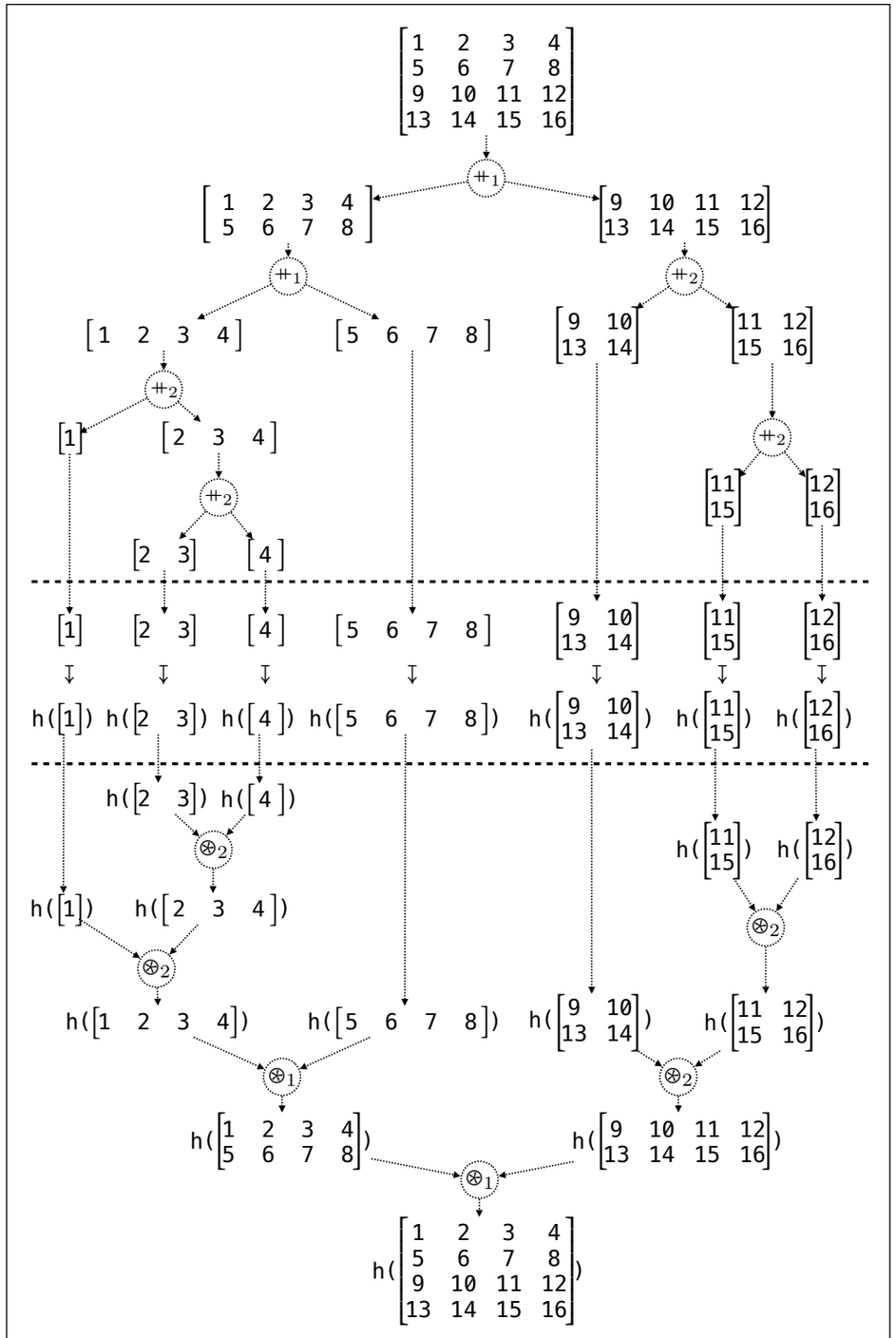


Fig. 10. MDH property recursively applied to a two-dimensional example computation

Definition 3 (Multi-Dimensional Homomorphism). Let $T^{\mathsf{INP}}, T^{\mathsf{OUT}} \in \mathsf{TYPE}$ be two arbitrary scalar types, $D \in \mathbb{N}$ a natural number, and $\Rightarrow^{1}_{\mathsf{MDA}}, \ldots, \Rightarrow^{D}_{\mathsf{MDA}} : \mathsf{MDA\text{-}IDX\text{-}SETs} \to \mathsf{MDA\text{-}IDX\text{-}SETs}$ functions on MDA index sets. Let further $+\!\!+_d := +\!\!+^{<T^{\mathsf{INP}} \mid D \mid d>} \in \mathsf{CO}^{<id \mid T^{\mathsf{INP}} \mid D \mid d>}$ denote concatenation (Definition 1) in dimension $d \in [1, D]_{\mathbb{N}}$ on D-dimensional MDAs that have scalar type T^{INP}.

A function

$$h^{<I_1,\ldots,I_D \in \mathsf{MDA\text{-}IDX\text{-}SETs}>} : T^{\mathsf{INP}}[\, I_1, \ldots, I_D \,] \to T^{\mathsf{OUT}}[\, \Rightarrow^{1}_{\mathsf{MDA}}(I_1), \ldots, \Rightarrow^{D}_{\mathsf{MDA}}(I_D) \,]$$

is a *Multi-Dimensional Homomorphism (MDH)* that has *input scalar type* T^{INP}, *output scalar type* T^{OUT}, *dimensionality* D, and *index set functions* $\Rightarrow^{1}_{\mathsf{MDA}}, \ldots, \Rightarrow^{D}_{\mathsf{MDA}}$, iff for each $d \in [1, D]_{\mathbb{N}}$, there exists a combine operator $\circledast_d \in \mathsf{CO}^{<\Rightarrow^{d}_{\mathsf{MDA}} \mid T^{\mathsf{OUT}} \mid D \mid d>}$ (Definition 2), such that for 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 $\mathsf{MDH}^{<T^{\mathsf{INP}}, T^{\mathsf{OUT}} \mid D \mid (\Rightarrow^{d}_{\mathsf{MDA}})_{d \in [1,D]_{\mathbb{N}}}>}$.

Definition 5 (Buffer). Let $T \in \mathsf{TYPE}$ be an arbitrary scalar type, $D \in \mathbb{N}_0$ a natural number[9], and $N := (N_1, \ldots, N_D) \in \mathbb{N}^D$ a sequence of natural numbers.

A *Buffer (BUF)* $\mathfrak{b}$ that has *dimensionality* D, *size* N, and *scalar type* T is a function with the following signature:

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

Here, $\bot$ denotes the *undefined* value. We refer to $[0, N_1)_{\mathbb{N}_0} \times \ldots \times [0, N_D)_{\mathbb{N}_0} \to T \cup \{\bot\}$ as the *type* of BUF $\mathfrak{b}$, which we also denote as $T^{N_1 \times \ldots \times N_D}$, and we refer to set $\mathsf{BUF\text{-}IDX\text{-}SETs} := \{\, [0, N)_{\mathbb{N}_0} \mid N \in \mathbb{N} \}$ as *BUF index sets*. 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.

Definition 2 (Combine Operator). Let $\mathsf{MDA\text{-}IDX\text{-}SETs} \dot{\times} \mathsf{MDA\text{-}IDX\text{-}SETs} := \{\, (P, Q) \in \mathsf{MDA\text{-}IDX\text{-}SETs} \times \mathsf{MDA\text{-}IDX\text{-}SETs} \mid P \cap Q = \varnothing \,\}$ denote the set of all pairs of MDA index sets that are disjoint. Let further $\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}} : \mathsf{MDA\text{-}IDX\text{-}SETs} \to \mathsf{MDA\text{-}IDX\text{-}SETs}$ be a function on MDA index sets, $T \in \mathsf{TYPE}$ a scalar type, $D \in \mathbb{N}$ an MDA dimensionality, and $d \in [1, D]_{\mathbb{N}}$ an MDA dimension.

We refer to any binary function $\circledast$ of type (parameters in angle brackets are type parameters)

$$\circledast^{<(I_1,\ldots,I_{d-1},I_{d+1},\ldots,I_D) \in \mathsf{MDA\text{-}IDX\text{-}SETs}^{D-1},\ (P,Q) \in \mathsf{MDA\text{-}IDX\text{-}SETs} \dot{\times} \mathsf{MDA\text{-}IDX\text{-}SETs}>} :$$

$$T[\, I_1, \ldots, \underbrace{\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}}(P)}_{\uparrow d}, \ldots, I_D \,] \times T[\, I_1, \ldots, \underbrace{\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}}(Q)}_{\uparrow d}, \ldots, I_D \,] \to T[\, I_1, \ldots, \underbrace{\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}}(P \uplus Q)}_{\uparrow d}, \ldots, I_D \,]$$

as *combine operator* that has *index set function* $\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}}$, *scalar type* T, *dimensionality* D, and *operating dimension* d. We denote combine operator's type concisely as $\mathsf{CO}^{<\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}} \mid T \mid D \mid d>}$.

... (~140 pages)

All concepts are fully formally defined in the thesis

Code Optimization via **ATF**

Overview | Getting Started | Code Examples | Publications | Citations | Contact

Auto-Tuning Framework (ATF)

Efficient Auto-Tuning of Parallel Programs with Constrained Tuning Parameters

Overview

The Auto-Tuning Framework (ATF) is a general-purpose auto-tuning approach: given a program that is implemented as generic in performance-critical program parameters (a.k.a. *tuning parameters*), such as sizes of tiles and numbers of threads, ATF fully automatically determines a hardware- and data-optimized configuration of such parameters.

Key Feature of ATF

A key feature of ATF is its support for *Tuning Parameter Constraints*. Parameter constraints allow auto-tuning programs whose tuning parameters have so-called *interdependencies* among them, e.g., the value of one tuning parameter has to evenly divide the value of another tuning parameter.

ATF's support for parameter constraints is important: modern parallel programs target novel parallel architectures, and such architectures typically have deep memory and core hierarchies thus requiring constraints on tuning parameters, e.g., the value of a tile size tuning parameter on an upper memory layer has to be a multiple of a tile size value on a lower memory layer.

For such parameters, ATF introduces novel concepts for *Generating* & *Storing* & *Exploring* the search spaces of constrained tuning parameters, thereby contributing to a substantially more efficient overall auto-tuning process for such parameters, as confirmed in our *Experiments*.

Generality of ATF

For wide applicability, ATF is designed as generic in:

1. The target program's Programming Language, e.g., *C/C++*, *CUDA*, *OpenMP*, or *OpenCL*. ATF offers *pre-implemented cost functions* for conveniently auto-tuning C/C++ programs, as well as CUDA and OpenCL kernels which require host code for their execution which is automatically generated and executed by ATF's pre-implemented CUDA and OpenCL cost functions. ATF also offers a pre-implemented *generic* cost function that can be used for conveniently auto-tuning programs in any other programming language different from C/C++, CUDA, and OpenCL.

https://atf-tuner.org

ACM TACO 2021

Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)

ARI RASCH and RICHARD SCHULZE, University of Muenster, Germany
MICHEL STEUWER, University of Edinburgh, United Kingdom
SERGEI GORLATCH, University of Muenster, Germany

Auto-tuning is a popular approach to program optimization: it automatically finds good configurations of a program's so-called tuning parameters whose values are crucial for achieving high performance for a particular parallel architecture and characteristics of input/output data. We present three new contributions of the Auto-Tuning Framework (ATF), which enable a key advantage in *general-purpose auto-tuning*: efficiently optimizing programs whose tuning parameters have *interdependencies* among them. We make the following contributions to the three main phases of general-purpose auto-tuning: (1) ATF *generates* the search space of interdependent tuning parameters with high performance by efficiently exploiting parameter constraints; (2) ATF *stores* such search spaces efficiently in memory, based on a novel chain-of-trees search space structure; (3) ATF *explores* these search spaces faster, by employing a multi-dimensional search strategy on its chain-of-trees search space representation. Our experiments demonstrate that, compared to the state-of-the-art, general-purpose auto-tuning frameworks, ATF substantially improves generating, storing, and exploring the search space of interdependent tuning parameters, thereby enabling an efficient overall auto-tuning process for important applications from popular domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

CCS Concepts: • **General and reference → Performance**; • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Parallel programming languages**;

Additional Key Words and Phrases: Auto-tuning, parallel programs, interdependent tuning parameters

ACM Reference format:
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 (January 2021), 26 pages.
https://doi.org/10.1145/3427093

This is a new paper, not an extension of a conference paper.
Authors' addresses: A. Rasch, R. Schulze, and S. Gorlatch, University of Muenster, Muenster, Germany; emails: {a.rasch, r.schulze, gorlatch}@uni-muenster.de; M. Steuwer, University of Edinburgh, Edinburgh, United Kingdom; email: michel.steuwer@glasgow.ac.uk.

1544-3566/2021/01-ART1

Goal of **ATF**

Advantage of *Auto-Tuning Framework (ATF)* over state-of-the-art general-purpose AT approaches:

ATF finds values of performance-critical parameters with

***interdependencies* among them**

via optimized processes to

generating & *storing* & *exploring*

the spaces of interdependent parameters

→ We illustrate ATF by comparing it to MIT's *OpenTuner* [PACT'14] & *CLTune* [MCSoC'15] which is the foundation of many related approaches (e.g., KernelTuner & KTT).

Note: BaCO [ASPLOS'23] & KTT [FGCS'20] recently adopted the ATF techniques to efficiently handle interdependencies among tuning parameters.

ATF: Contributions

The three major contributions of ATF:

1. ATF ***generates*** the search space of interdependent parameters with *high performance*

2. ATF ***stores*** these spaces with *low memory footprint*

3. ATF ***explores*** the spaces *efficiently*

Parameter Constraints

CoT Structure

ATF introduces novel processes to
generating & ***storing*** & ***exploring***
the spaces of interdependent tuning parameters, based on a novel ***constraint design*** and ***search space structure***

ATF: Internals in a Nutshell

The three major contributions of *Auto-Tuning Framework (ATF)*:

Based on Traditional Constraints

```
for ( v1 : r1 )
  ⋱
    for ( vk : rk )
      if( sc(v1,…,vk) )
        add_config( v1,…,vk );
```

```
parallel_for ( G : {G1,…,Gn} )
{
  parallel_for ( v1^G : r1^G )
    if( pc1^G(v1^G) )
      ⋱
        parallel_for ( vtg^G : rtg^G )
          if( pctg^G(vtg^G) )
            for ( vtg+1^G : rtg+1^G )
              if( pc(vtg+1^G) )
                ⋱
                  for( vk^G : rk^G )
                    if( pc(vk^G) )
                      add_config( v1^G,…,vk^G );
}
```

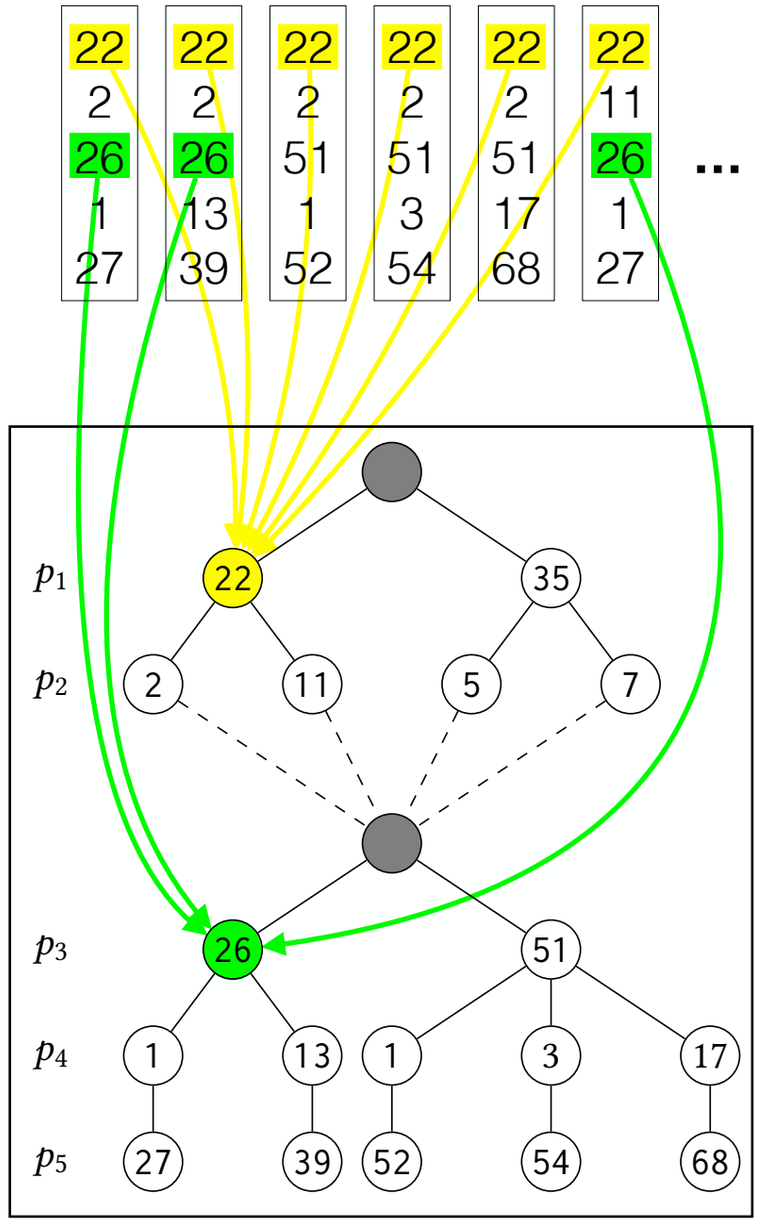
Based on ATF's Parameter Constraints

Fast Space Generation:
- Break loop nest
- Parallelize loop nest
- Skip loops iterations

Memory-Friendly Storing:
- Avoid Redundancies
- interdependent parameters → "tree"
- independent parameters → "chain"

Efficient Exploration:
- CoT maps to Coordinate Space (CS)
- CS efficient structure for search techniques

Straightforward Search Space



ATF's CoT Search Space

ATF: Summary

ATF's efficiently handles tuning parameters with ***interdependencies*** among them:

ATF introduces novel concepts to
Generating & ***Storing*** & ***Exploring***
the search spaces of interdependent parameters, based on its *novel*
constraint design and ***search space representation***

Further ATF features (not presented on slides for brevity):

- ATF's has a DSL-based **user interface** that is **arguably simpler** to use and more expressive than existing auto-tuners (including: OpenTuner & CLTune)
- ATF offers different kinds of **search techniques** and **abort conditions** (extensible)
- ATF offers a **DSL-based** user interface (*offline tuning*), as well as **GPL-based** interfaces (*online* auto-tuning):

pyATF

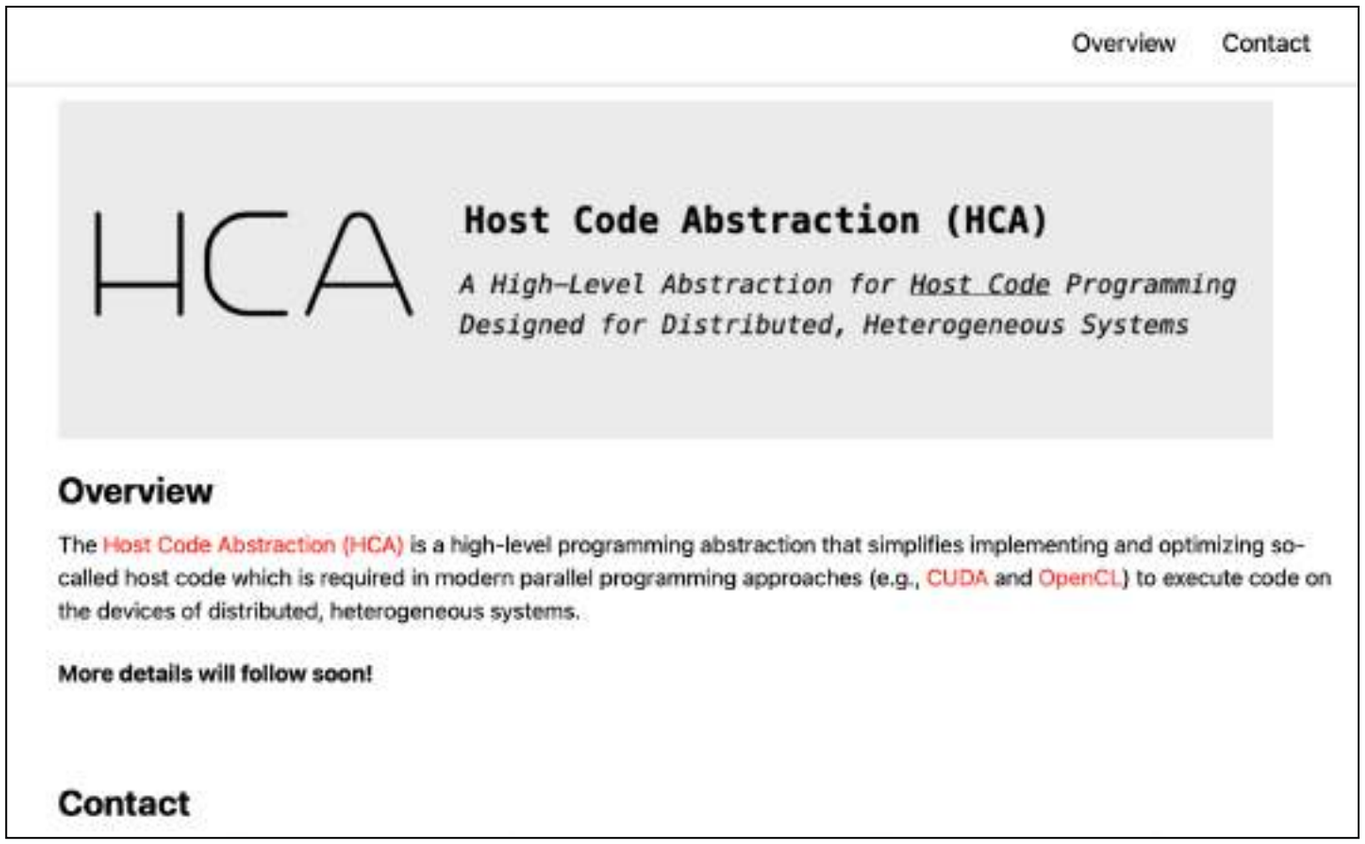
github.com/atf-tuner/pyATF

cppATF

github.com/atf-tuner/cppATF

… (future work)

Code Execution via **HCA**



https://hca-project.org

Just a brief outline of HCA (for brevity)

Journal of Supercomputing 2019

The Journal of Supercomputing (2020) 76:5117–5138
https://doi.org/10.1007/s11227-019-02829-2

Check for updates

dOCAL: high-level distributed programming with OpenCL and CUDA

Ari Rasch[1] · Julian Bigge[1] · Martin Wrodarczyk[1] · Richard Schulze[1] · Sergei Gorlatch[1]

Published online: 30 March 2019

Abstract
In the state-of-the-art parallel programming approaches OpenCL and CUDA, so-called host code is required for program's execution. Efficiently implementing host code is often a cumbersome task, especially when executing OpenCL and CUDA programs on systems with multiple nodes, each comprising different devices, e.g., multi-core CPU and graphics processing units; the programmer is responsible for explicitly managing node's and device's memory, synchronizing computations with data transfers between devices of potentially different nodes and for optimizing data transfers between devices' memories and nodes' main memories, e.g., by using pinned main memory for accelerating data transfers and overlapping the transfers with computations. We develop distributed OpenCL/CUDA abstraction layer (dOCAL)—a novel high-level C++ library that simplifies the development of host code. dOCAL combines major advantages over the state-of-the-art high-level approaches: (1) it simplifies implementing both OpenCL and CUDA host code by providing a simple-to-use, high-level abstraction API; (2) it supports executing arbitrary OpenCL and CUDA programs; (3) it allows conveniently targeting the devices of different nodes by automatically managing node-to-node communications; (4) it simplifies implementing data transfer optimizations by providing different, specially allocated memory regions, e.g., pinned main memory for overlapping data transfers with computations; (5) it optimizes memory management by automatically avoiding unnecessary data transfers; (6) it enables interoperability between OpenCL and CUDA host code for systems with devices from different vendors. Our experiments show that dOCAL significantly simplifies the development of host code for heterogeneous and distributed systems, with a low runtime overhead.

Goal of **HCA**

<u>HCA simplifies implementing host code (e.g., OpenCL & CUDA) for the programmer:</u>

- HCA offers an **easy-to-use high-level API** for host code programming that frees the user from boilerplate low-level commands (e.g., for memory allocations and data transfers)

- HCA allows **conveniently targeting multiple devices** and of potentially **different nodes** (by automatically managing device-to-device communications)
- HCA simplifies implementing **host code optimizations** (e.g., using pinned main memory for overlapping data transfers with computations)

- HCA **optimizes memory management** by automatically avoiding unnecessary data transfers (by internally generating & maintaining a data-dependency graph)
- …

Related approaches usually offer only a subset of these HCA advantages

HCA: Overview

CUDA vs. HCA — for executing parallel reduction CUDA kernel:

```
int main(int argc, char **argv)
{
    // initialization
    int i, j, gpuBase, GPU_N;
    cudaGetDeviceCount(&GPU_N);
    //...

    /* ... prepare input data ... */

    // Allocate device and host memory
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i));
        cudaStreamCreate(&plan[i].stream));
        cudaMalloc((void **)&plan[i].d_Data, plan[i].dataN *
sizeof(float));
        cudaMalloc((void **)&plan[i].d_Sum, ACCUM_N * sizeof(float));
        cudaMallocHost((void **)&plan[i].h_Sum_from_device, ACCUM_N *
sizeof(float));
        cudaMallocHost((void **)&plan[i].h_Data, plan[i].dataN *
sizeof(float));
        for (j = 0; j < plan[i].dataN; j++)
          plan[i].h_Data[j] = (float)rand()/(float)RAND_MAX;
    }

    // Perform data transfers and start device computations
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i);
        cudaMemcpyAsync(plan[i].d_Data, plan[i].h_Data, plan[i].dataN *
sizeof(float), cudaMemcpyHostToDevice, plan[i].stream);
        reduceKernel<<<BLOCK_N, THREAD_N, 0,
plan[i].stream>>>(plan[i].d_Sum, plan[i].d_Data, plan[i].dataN);
        cudaMemcpyAsync(plan[i].h_Sum_from_device, plan[i].d_Sum,
ACCUM_N *sizeof(float), cudaMemcpyDeviceToHost, plan[i].stream);
    }
    // combine GPUs' results
    for (i = 0; i < GPU_N; i++) {
        float sum;
        cudaSetDevice(i);
        cudaStreamSynchronize(plan[i].stream);
        sum = 0;
        for (j = 0; j < ACCUM_N; j++)
          sum += plan[i].h_Sum_from_device[j];
        *(plan[i].h_Sum) = (float)sum;
        cudaFreeHost(plan[i].h_Sum_from_device);
        cudaFree(plan[i].d_Sum);
        cudaFree(plan[i].d_Data);
        cudaStreamDestroy(plan[i].stream);
    }

    /* ... Compare GPU and CPU results ... */
}
```

Excerpt of NVIDIA SDK CUDA host code for executing parallel reduction CUDA kernel

```
#include "hca.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = hca::get_all_devices<CUDA>();

  // 2. declare kernel
  hca::kernel reduction = cuda::source( /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  hca::buffer<float> in ( N     * devices.size() );
  hca::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                  )
       ( dim3( GS ), dim3( BS )                     )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                          );

  auto res = std::accumulate( out.begin(), out.end(), std::plus<float>() );

  std::cout << res << std::endl;
}
```

HCA Code (Equivalent to CUDA code left)

HCA relies on a high user-level of abstraction instead of complex, error-prone low-level functions

HCA: Summary

Goal of HCA:

HCA is a programming library for
simplifying host code programming
by abstracting from low-level details

Further HCA Features:

- enables interoperability between OpenCL and CUDA host code
- supports auto-tuning
- compatible with existing libraries (e.g., NVIDIA cuBLAS/cuDNN)
- simplifies profiling
- …

MDH+ATF+HCA: User Interfaces

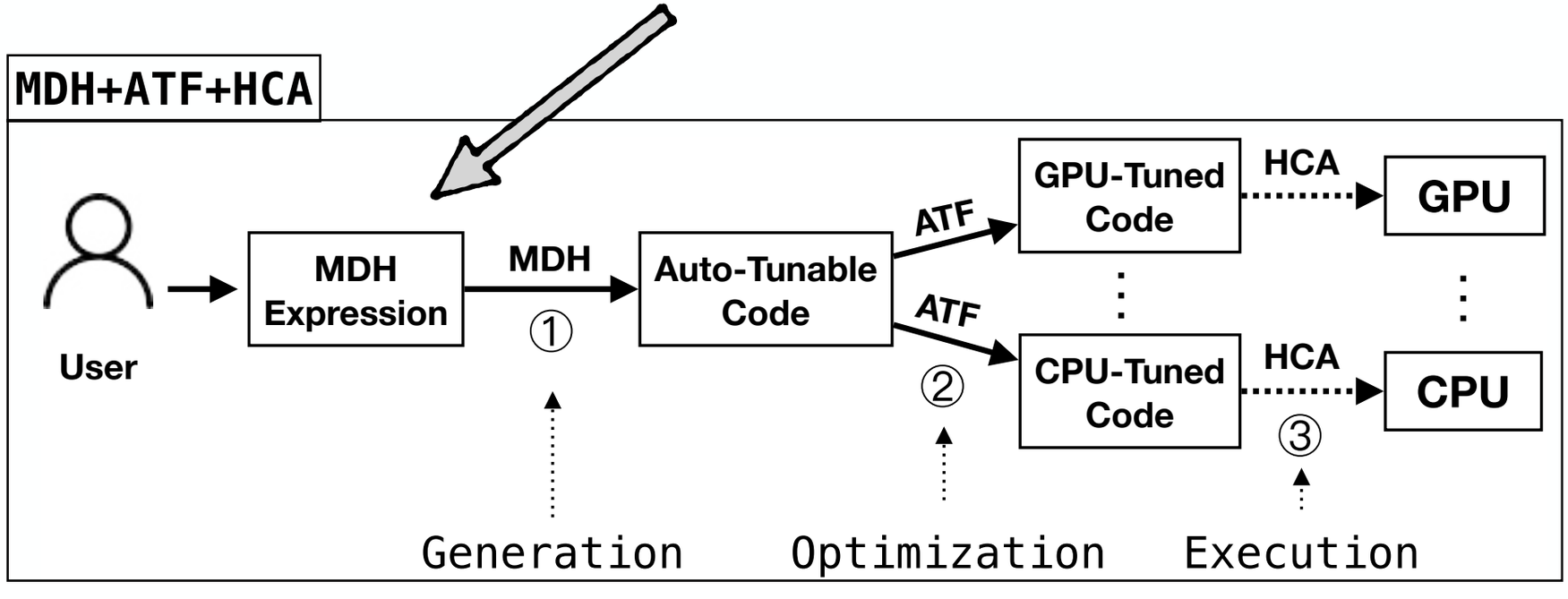
A Domain-Specific Language (DSL):

```
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) )
```

MDH's Formal Representation of MatVec

MDH's DSL Representation of MatVec

```
MatVec<T in TYPE | I,K in IN> := out_view<T>( w:(i,k)->(i) ) o

                                   md_hom<I,K>( *, (++,+) ) o

                                    inp_view<T,T>( M:(i,k)->(i,k),

                                                   v:(i,k)->(k)  )
```



MDH+ATF+HCA: User Interfaces

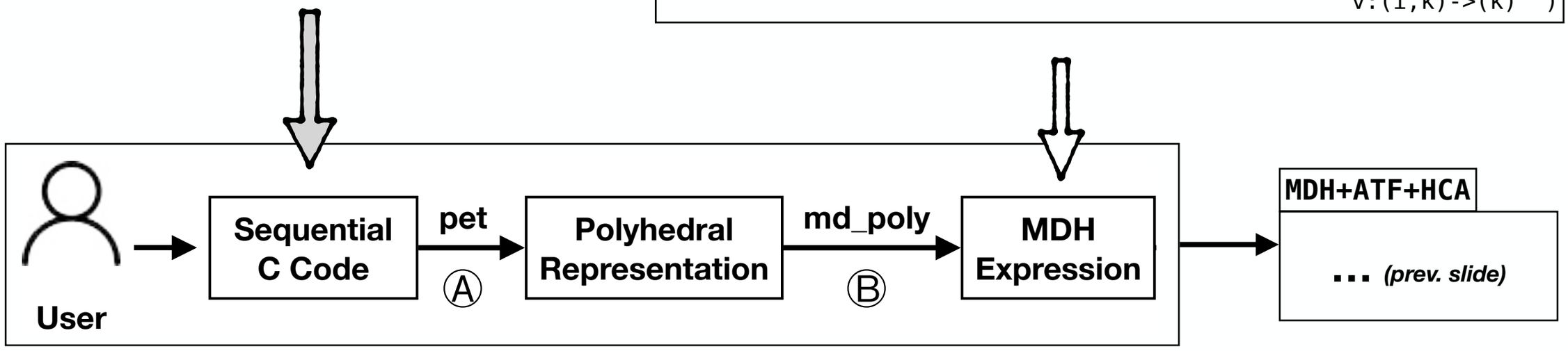
Automatic Parallelization:

C Implementation of MatVec

```
1  for( int i = 0; i < I; ++i )
2    for( int k = 0; k < K; ++k )
3    {
4      w[i] += M[i][k] * v[k];
5    }
```

MDH's DSL Representation of MatVec

```
MatVec<T in TYPE | I,K in IN> := out_view<T>( w:(i,k)->(i) ) o
                                 md_hom<I,K>( *, (++,+) ) o
                                 inp_view<T,T>( M:(i,k)->(i,k),
                                                v:(i,k)->(k)  )
```

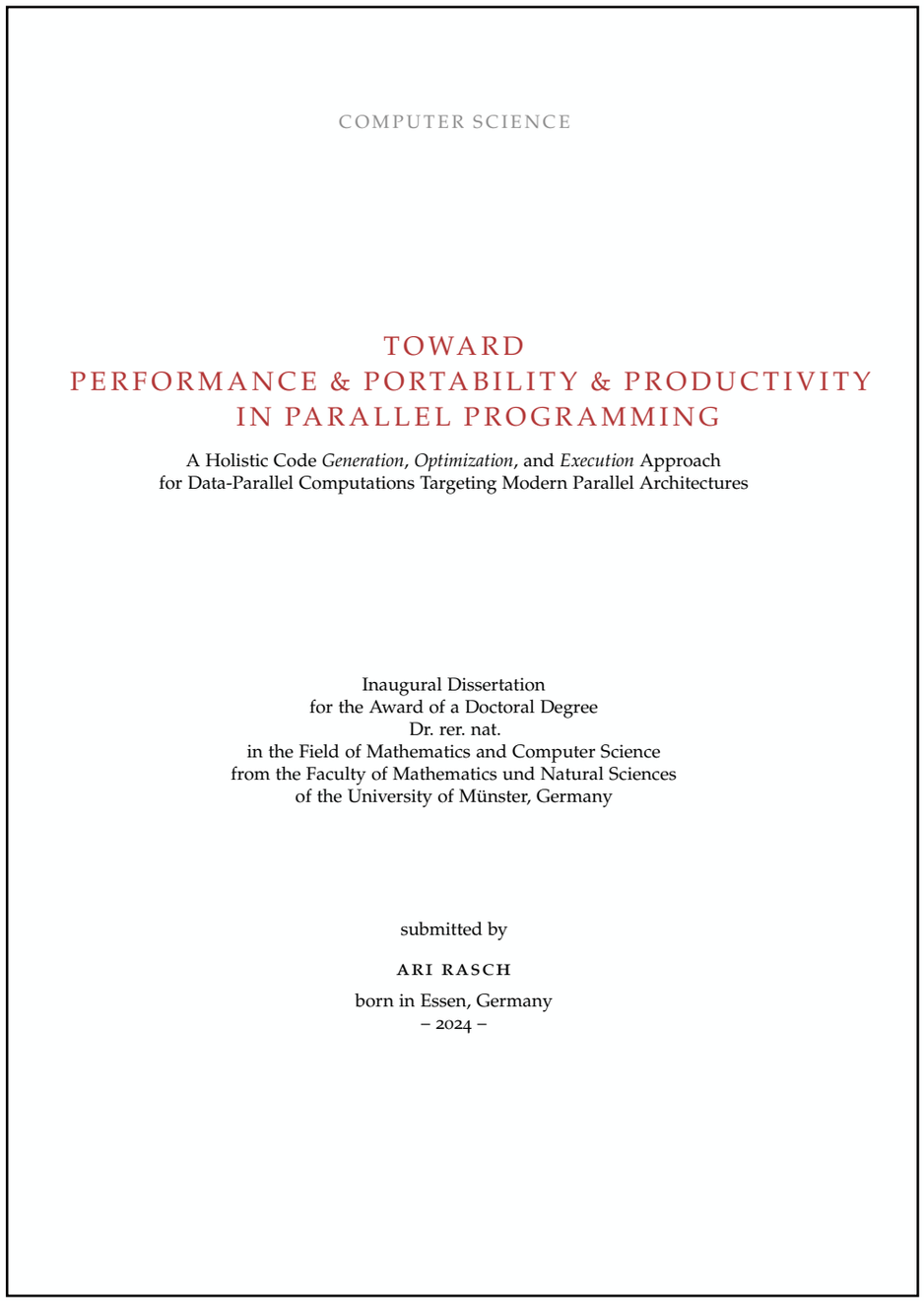


***MDH pragma* enables advanced optimizations:**

```
#pragma mdh ( w[i] : ++ , + )
```

Conclusion

The **MDH+ATF+HCA** approach is a novel approach toward **Performance** & **Portability** & **Productivity** for data-parallel computations targeting modern parallel architectures:



COMPUTER SCIENCE

TOWARD
PERFORMANCE & PORTABILITY & PRODUCTIVITY
IN PARALLEL PROGRAMMING

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

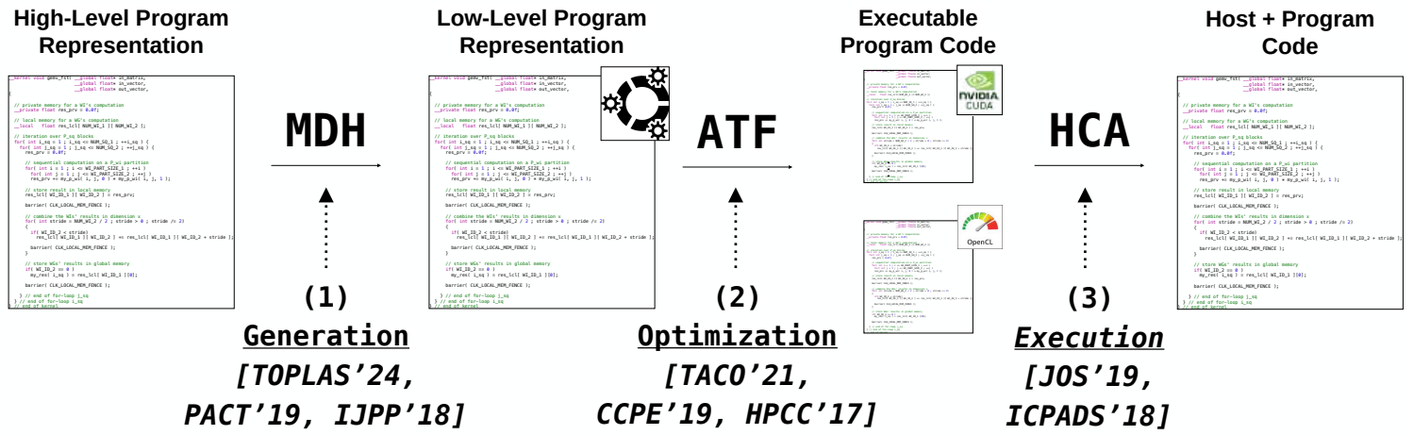
Inaugural Dissertation
for the Award of a Doctoral Degree
Dr. rer. nat.
in the Field of Mathematics and Computer Science
from the Faculty of Mathematics und Natural Sciences
of the University of Münster, Germany

submitted by

ARI RASCH

born in Essen, Germany
– 2024 –

The MDH+ATF+HCA approach:

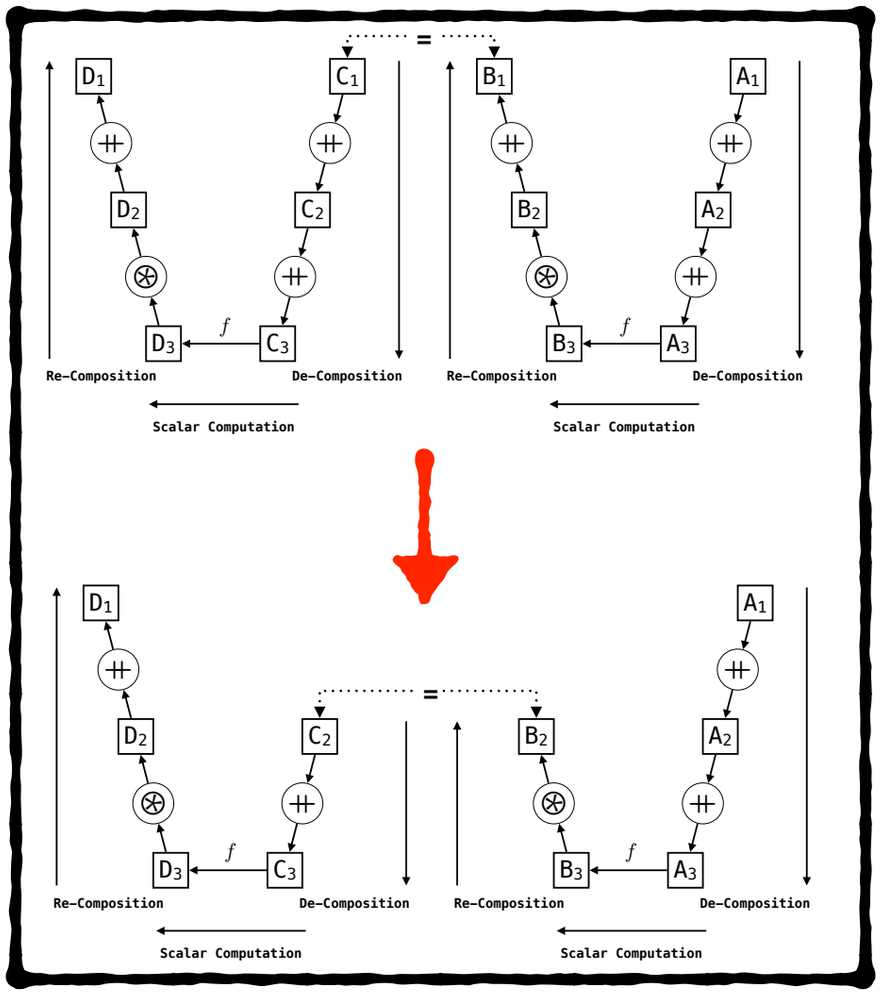


- The three sub-projects — MDH & ATF & HCA — complement each other to a holistic code *Generation* & *Optimization* & *Execution* approach
- Our holistic approach takes as input easy-to-use DSL programs or sequential (annotated) C code[1]
- There are many (promising) future directions for MDH & ATF & HCA (one part of thesis dedicated to FW)

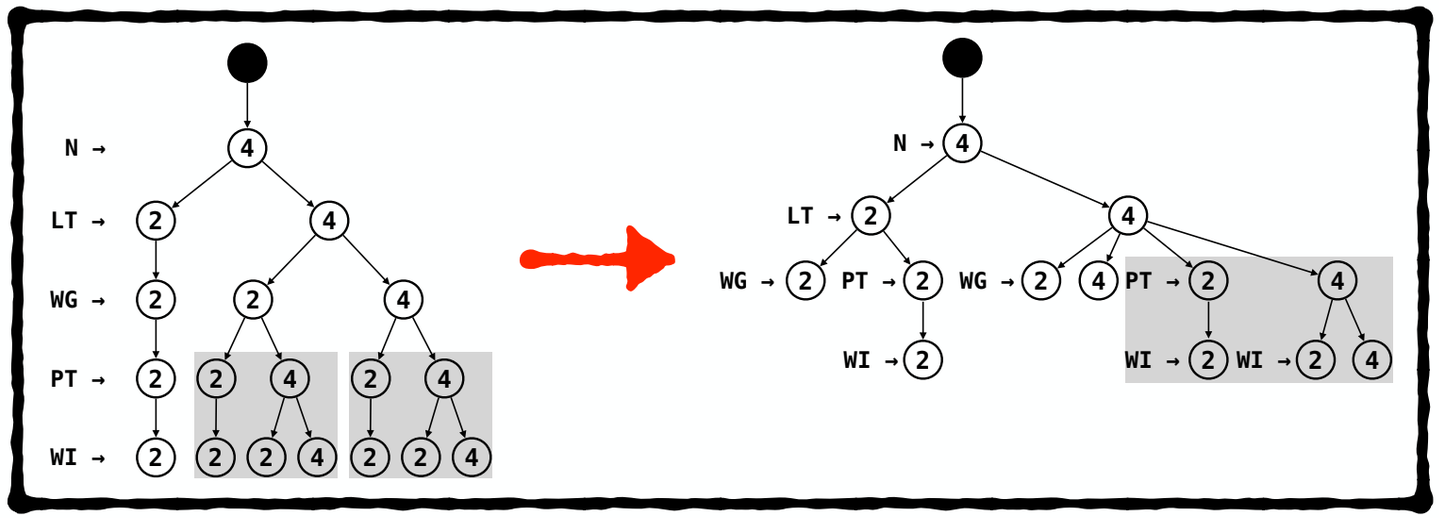
[1] A DSL-equivalent *MLIR* frontend will be introduced soon.

Stay Tuned !

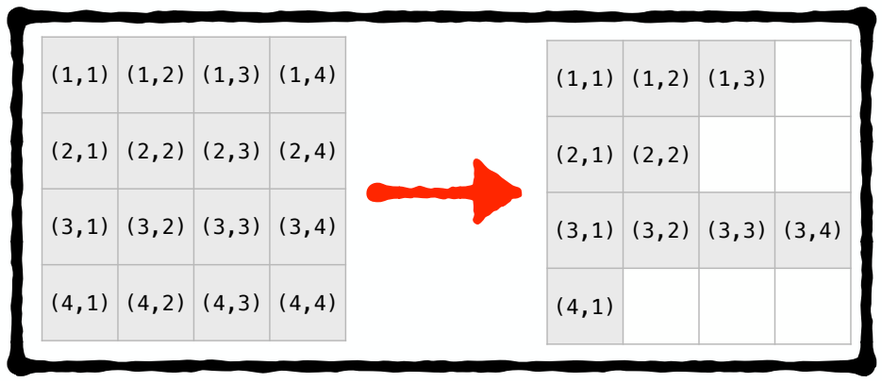
The approaches MDH & ATF & HCA are starting points for many future directions:



Fusion Optimization (MDH)



CoT Optimization (ATF)



Irregular Data Accesses (MDH)

… (many more)

Encouraging WIP results — already awarded & funded:

Funded by DFG Deutsche Forschungsgemeinschaft German Research Foundation

Best Poster Award PUMPS+AI

Lorentz center

Gold Winner: SRC @ CGO

Gold Winner: SRC @ PACT

IHK **Award for:** *Applied Computer Science*

SC Best Poster Finalist

Questions