Universität Münster

PLDI COPENHAGEN 2024

ACM TOPLAS

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

Ari Rasch

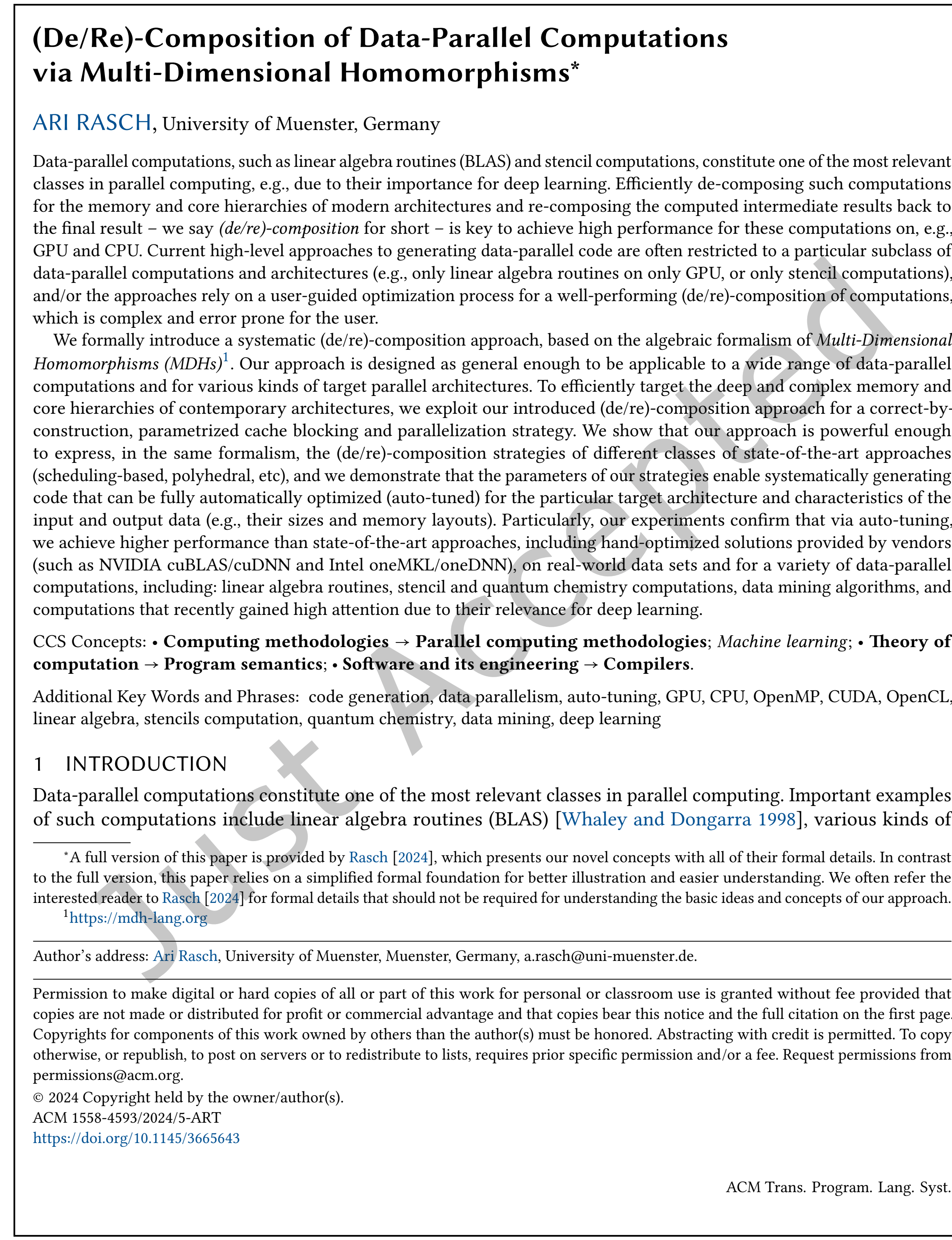
University of Münster, Germany

https://mdh-lang.org

Introductory Remarks

- This talk (briefly!) highlights the main contributions of our paper (~20 slides vs. >70 pages)
- Talk focuses on illustrative examples, rather than formal definitions & details (all provided and thoroughly discussed in the paper)
- There is a (full) arXiv version of the paper that contains all formal details [1]
- Paper is long: >70 pages (>130 pages arXiv)
- Many illustrations and discussions — you can get the basic idea even when skipping the formal details

The paper attempts to make a general, fundamental contribution to the community (see next slide)



(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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s).
ACM 1558-4593/2024/5-ART
https://doi.org/10.1145/3665643

ACM Trans. Program. Lang. Syst.

[1] Rasch, Ari. "Full Version: (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms." *arXiv preprint arXiv:2405.05118* (2024).

Goal of this Work

<u>Questions addressed by this talk:</u>

How can data parallelism be formally defined?

How can data-parallel computations be uniformly expressed via higher-order functions?

How can data-parallel computations be expressed agonistic from of hardware and optimization details (and still capture all information relevant for generating high-performing code)?

How can optimizations for the memory and core hierarchies of state-of-the-art parallel architectures be formally expressed?

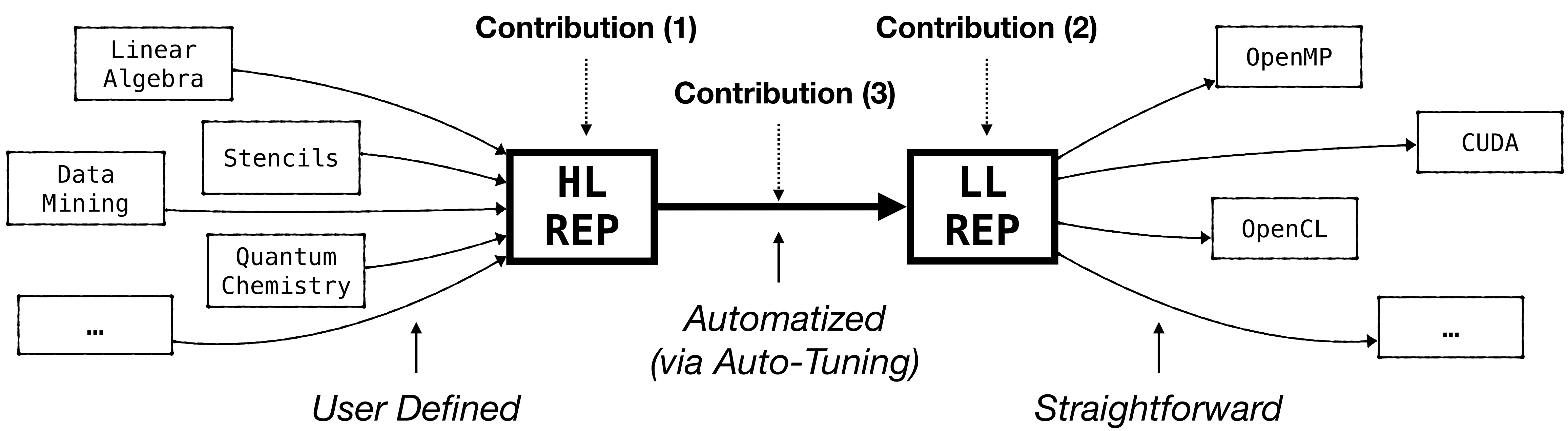
How can such optimizations be generalized to apply to arbitrary data-parallel computations?

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

All questions are answered (fully formally) in the paper!

Goal of this Work

A (formal) framework for expressing & optimizing data-parallel computations:



1. ***Contribution 1 (HL-REP):*** defines *data parallelism* & introduces *higher-order functions* for expressing data-parallel computations, agnostic from hardware and optimization details while still capturing all 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 state-of-the-art parallel architectures & generalizes these optimizations to apply to arbitrary combinations of data-parallel computations and parallel architectures

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

Agenda

1. **<u>Contribution 1:</u>** *High-Level Representation*
2. **<u>Contribution 2:</u>** *Low-Level Representation*
3. **<u>Contribution 3:</u>** *Lowering*: High-Level Representation → Low-Level Representation
4. Experimental Results (Performance & Portability & Productivity)
5. Related Work
6. Conclusion

<u>Contribution 1</u>
3 slides vs 17p.TOPLAS (26p.arXiv)

<u>Contribution 2</u>
1 slide vs 9p.TOPLAS (20p.arXiv)

<u>Contribution 3</u>
1 slide vs 4p.TOPLAS (2p.arXiv)

Experimentale Results
4 slides vs 23p.TOPLAS (23p.arXiv)

Related Work
1 slide vs 11p.TOPLAS (11p.arXiv)

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

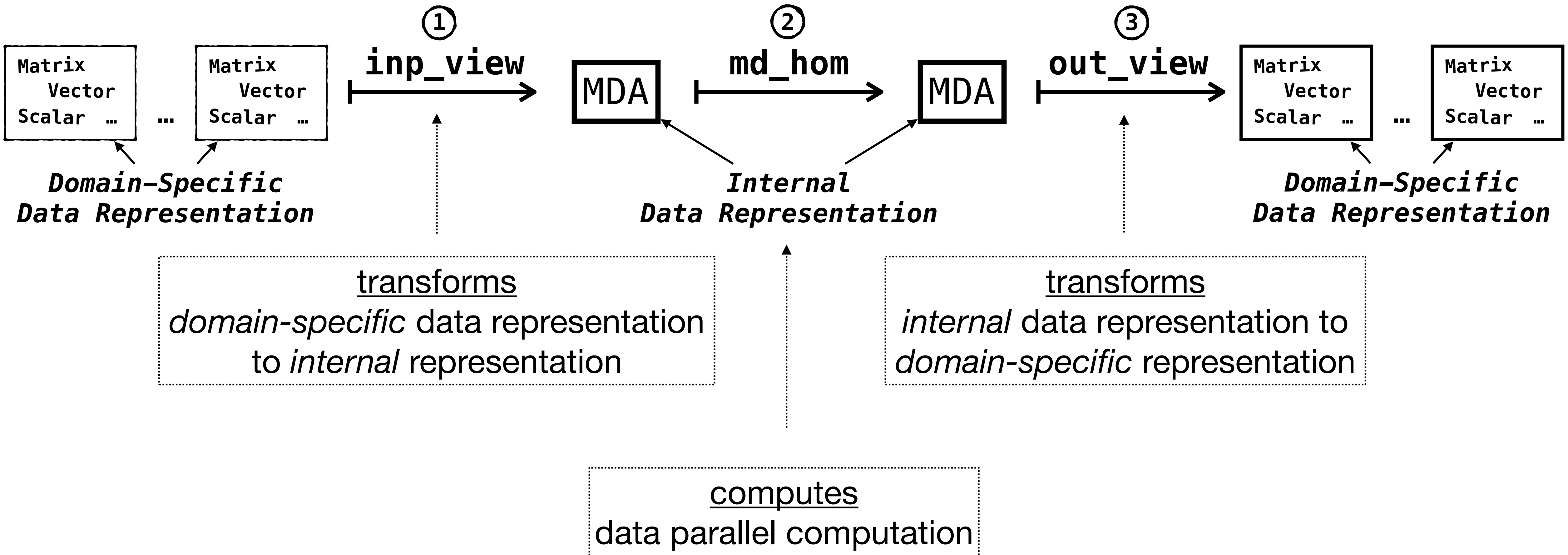
3. **Structured**:
   avoiding compositions and nestings of building blocks as much as possible, thereby further contributing to usability and simplicity of our language

```
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) )
```

***Our High-Level Representation of* MatVec**

High-Level Representation

Overview:



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

High-Level Representation

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

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

1) Linear Algebra Routines

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

Views	inp_view		out_view
	I	F	O
Conv2D	(p,q,r,s) ↦ (p+r,q+s)	(p,q,r,s) ↦ (r,s)	(p,q,r,s) ↦ (p,q)
MCC	(n,p,...) ↦ (n,p+r,q+s,c)	(n,p,...) ↦ (k,r,s,c)	(n,p,...) ↦ (n,p,q,k)
MCC_Capsule	(n,p,...) ↦ (n,p+r,q+s,c,cm,ck)	(n,p,...) ↦ (k,r,s,c,ck,cn)	(n,p,...) ↦ (n,p,q,k,cm,cn)

2) Convolution Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
MBBS	id	$++_{\texttt{prefix-sum}}(+)$	+

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

8) Maximum Bottom Box Sum

md_hom	f	$\circledast_1$	$\circledast_2$
Jacobi1D	J_{1D}	++	/
Jacobi2D	J_{2D}	++	++

Views	inp_view	out_view
	I	O
Jacobi1D	(i) ↦ (i+0) , (i) ↦ (i+1) , (i) ↦ (i+2)	(i) ↦ (i)
Jacobi2D	(i,j) ↦ (i,j+1) , (i,j) ↦ (i+1,j) , ...	(i,j) ↦ (i,j)

3) Jacobi Stencils

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

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

4) Probabilistic Record Linkage

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

Views	inp_view	out_view
	I	O
scan(⊕)	(i) ↦ (i)	(i) ↦ (i)

7) Scan Pattern

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

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

6) Map/Reduce Patterns

md_hom	f	$\circledast_1$	$\circledast_2$
Histo	$f_{\texttt{Histo}}$	++	+

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

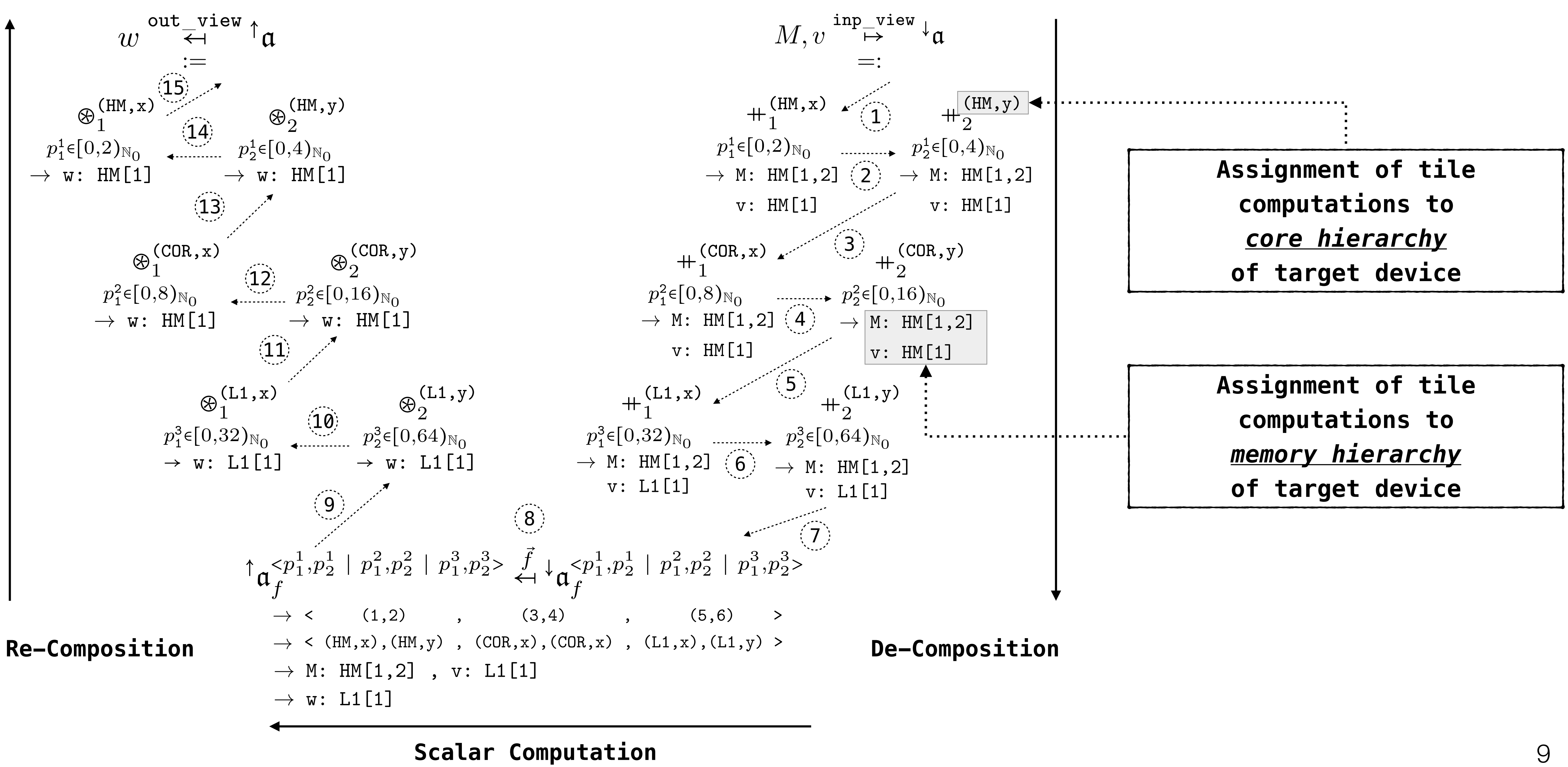
5) Histogram

Our high-level representation is capable of expressing the various kinds of data-parallel computations which often differ in major characteristics

Low-Level Representation

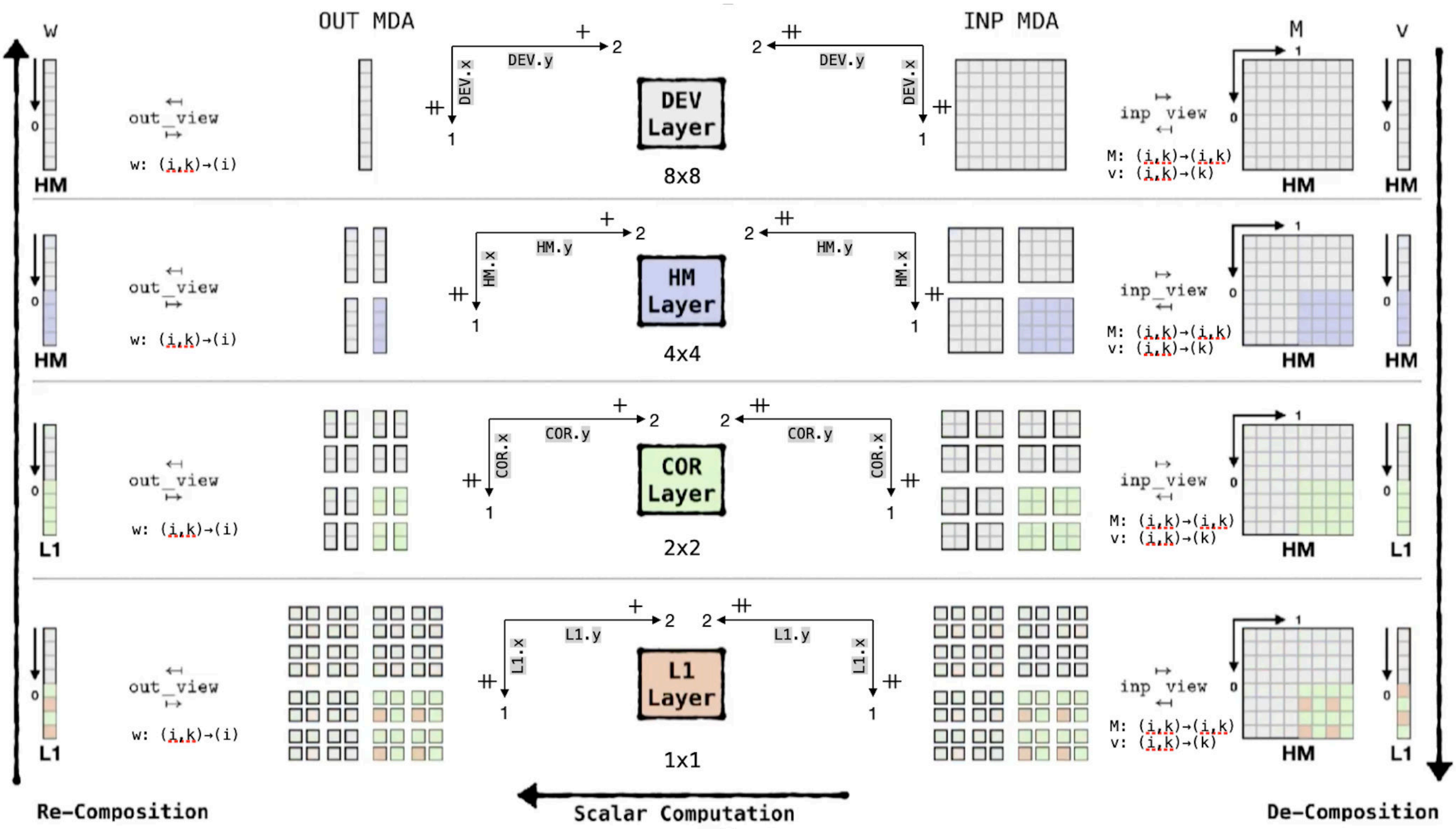
Goals:

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



Low-Level Representation

Excursion: Visualization of MDH Low-Level Programs



Low-Level Representation

Code generation:

```
// 0.1.2. combine operators

  // pre-implemented combine operators

  // inverse concatenation
  ∀d ∈ ℕ:
  cc_inv<<d>><I_1, ..., I_{d-1}, I_{d+1}, ..., I_D ∈ MDA-IDX-SETs, (P, Q) ∈ MDA-IDX-SETs ×̇ MDA-IDX-SETs>(
    T^INP[I_1, ..., I_{d-1}, id(P ⊍ Q), I_{d+1}, ..., I_D] res ) -> ( T^INP[I_1, ..., I_{d-1}, id(P) , I_{d+1}, ..., I_D] lhs ,
                                                              T^INP[I_1, ..., I_{d-1}, id(Q) , I_{d+1}, ..., I_D] rhs )
  {
    int i_1 ∈ I_1
      ⋱
        int i_{d-1} ∈ I_{d-1}
          int i_{d+1} ∈ I_{d+1}
            ⋱
              int i_D ∈ I_D
              {
                int i_d ∈ P
                  res[ i_1,...,i_d,...,i_D] =: lhs[ i_1,...,i_d,...,i_D];
                int i_d ∈ Q
                  res[ i_1,...,i_d,...,i_D] =: rhs[ i_1,...,i_d,...,i_D];
              }
  }
```

Listing 11. Pre-Implemented Combine Operators

```
// 2. scalar phase

int p_ σ_{f-ord}(1,1) ∈^{< ↔_{f-ass}(1,1) >} #PRT ( σ_{f-ord}(1,1) )
  ⋱
    int p_ σ_{f-ord}(L,D) ∈^{< ↔_{f-ass}(L,D) >} #PRT ( σ_{f-ord}(L,D) )
    {
      (
        ll_out_mda<<f>><<
            p_(1,1) ,..., p_(1,D),
                           ...
            p_(L,1) ,..., p_(L,D)>><<b, a>>(
                ⇒^1_MDA ⊛_MDA ( I<<1>><p_(1,1),...,p_(L,1)>(0) ) ,
                ⋮
                ⇒^D_MDA ⊛_MDA ( I<<D>><p_(1,D),...,p_(L,D)>(0) ) )
      )_{b∈[1,B^OB]_ℕ, a∈[1,A^OB_b]_ℕ} := f( ( ll_inp_mda<<f>><< p_(1,1) ,..., p_(1,D),
                                                           ...
                                                           p_(L,1) ,..., p_(L,D)>><<b, a>>(
                                                ⇒^d_MDA #_MDA ( I<<1>><p_(1,1),...,p_(L,1)>(0) ) ,
                                                ⋮
                                                ⇒^d_MDA #_MDA ( I<<D>><p_(1,D),...,p_(L,D)>(0) ) )
                                            )_{b∈[1,B^IB]_ℕ, a∈[1,A^IB_b]_ℕ} )
    }
```

Listing 17. Scalar Phase

```
// 3. re-composition phase

  // 3.1. main

  int p_ σ_{↑-ord}(1,1) ∈^{< ↔_{↑-ass}(1,1) >} #PRT ( σ_{↑-ord}(1,1) )
  {
    int p_ σ_{↑-ord}(1,2) ∈^{< ↔_{↑-ass}(1,2) >} #PRT ( σ_{↑-ord}(1,2) )
    {
      ⋱
        int p_ σ_{↑-ord}(L,D) ∈^{< ↔_{↑-ass}(L,D) >} #PRT ( σ_{↑-ord}(L,D) )
        {
          ll_out_mda<< σ_{↑-ord}(L,D) >> :=_{co< σ_{↑-ord}(L,D) >} out_mda<<f>>;
        }
      ⋰
      ll_out_mda<< σ_{↑-ord}(1,2) >> :=_{co< σ_{↑-ord}(1,2) >} out_mda<< σ_{↑-ord}(1,3) >>;
    }
    ll_out_mda<< σ_{↑-ord}(1,1) >> :=_{co< σ_{↑-ord}(1,1) >} out_mda<< σ_{↑-ord}(1,2) >>;
  }

  // 3.2. finalization
  ll_out_mda<<⊥>> := ll_out_mda<< σ_{↑-ord}(1,1) >>
```

Listing 18. Re-Composition Phase

Our *Code Generation Process* is outlined in [1], based on an imperative-style pseudocode notation *(intended to be described in detail in FW)*

[1] Rasch, Ari. “Full Version: (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms.” arXiv.2405.05118.

Lowering: *High Level → Low-Level*

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

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

Table 1. Tuning parameters of our low-level expressions

exploiting core hierarchy (parallelization) → D2

exploiting memory hierarchy (data movements) → D3, D4

... → S2

... → S3, S4, S5, S6

... → R2

... → R3, R4

Our parameters
unify & generalize & combine
well-proven optimizations
(e.g., data movement & tiling)

We use our ***Auto-Tuning Framework (ATF) [1]*** to *fully automatically* determine optimized values of parameters

[1] Rasch, Schulze, Steuwer, Gorlatch, "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)", TACO'21

Experimental Results

<u>We experimentally evaluate our MDH approach in terms of ***Performance*** & ***Portability*** & ***Productivity***:</u>

<u>Competitors:</u>

1. <u>Scheduling Approach:</u>
   - Apache TVM [2] (GPU & CPU)
2. <u>Polyhedral Compilers:</u>
   - PPCG [3] (GPU)
   - Pluto [4] (CPU)
3. <u>Functional Approach:</u>
   - Lift [5] (GPU & CPU)
4. <u>Domain-Specific Libraries:</u>
   - NVIDIA cuBLAS & cuDNN (GPU)
   - Intel oneMKL & oneDNN (CPU)

[2] Chen et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning", OSDI'18

[3] Verdoolaege et al., "Polyhedral Parallel Code Generation for CUDA", TACO'13

[4] Bondhugula et al., "PLuTo: A Practical and Fully Automatic Polyhedral Program Optimization System", PLDI'08

[5] Steuwer et al., "Generating Performance Portable Code using Rewrite Rules", ICFP'15

<u>Case Studies:</u>

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

Experimental Results

Highlights only

Performance Evaluation: (via runtime comparison)

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	-	-

MDH speedup over

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

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	-	-

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

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

Experimental Results

Highlights only

<u>Productivity Evaluation:</u> (via intuitive argumentation)

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

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

```
for( int i = 0 ; i < M ; ++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)

```
def MatVec(I, K):
    M = te.placeholder((I, K), name='M', dtype='float32')
    v = te.placeholder((K,), name='v', dtype='float32')

    k = te.reduce_axis((0, K), name='k')
    w = te.compute(
        (I,),
        lambda i: te.sum(M[i, k] * v[k], axis=k)
    )
    return [M, v, w]
```

Listing 1. TVM program expressing Matrix-Vector Multiplication (MatVec)

```
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)

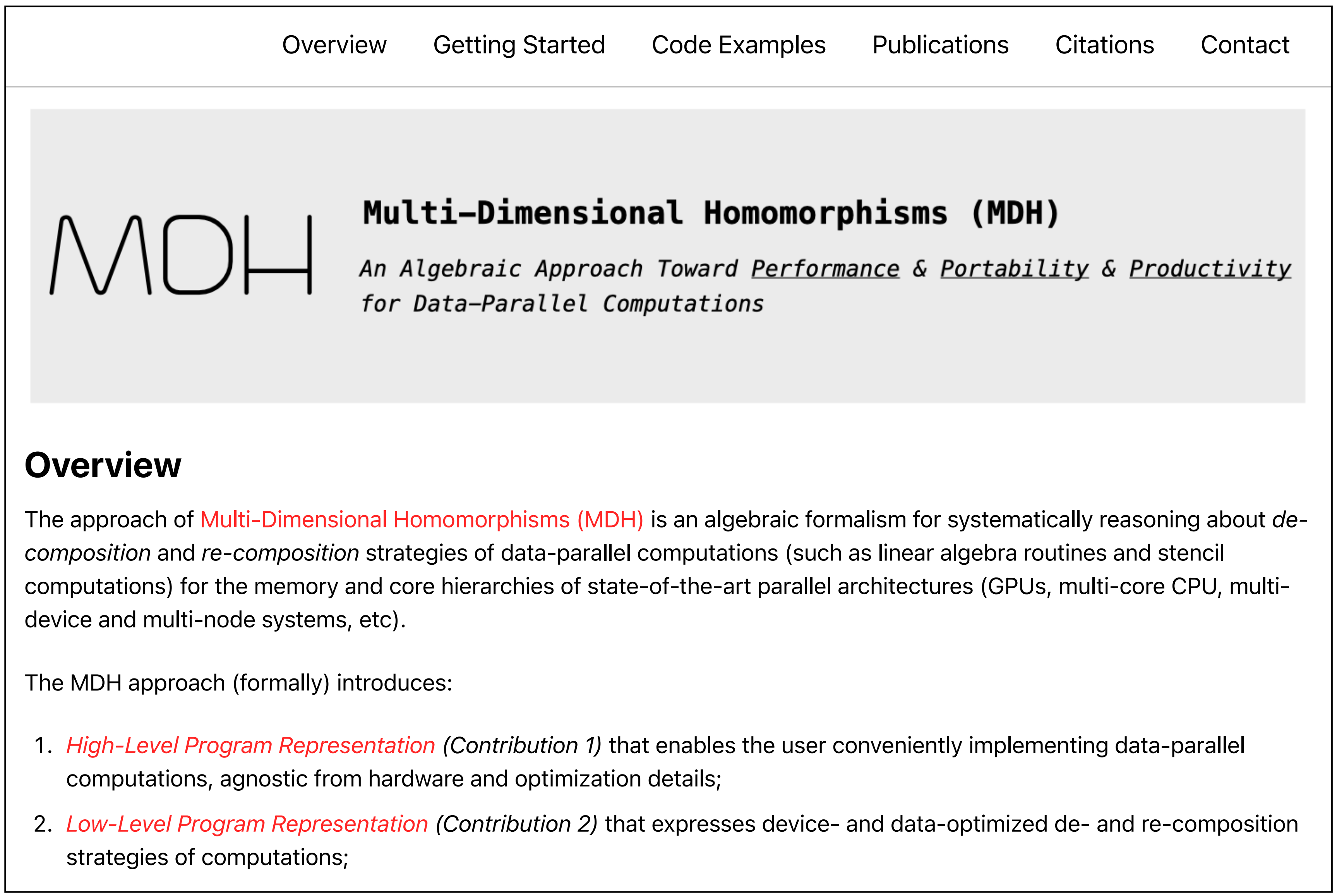
Related Work

<u>MDH often achieves higher *Performance & Portability & Productivity* than state-of-practice approaches:</u>

Class	*Popular Examples*	*Performance*	*Portability*	*Productivity*
	MDH	✔	✔	✔
Scheduling	`TVM, Halide, Fireiron`	✔	often require re-design/extension for new architectures	incorporate user into optimization process
Polyhedral	`TC, PPCG, Pluto`	struggle with reductions (e.g., dot in MatMul)	transformations chosen toward particular architectures and data characteristics	✔
Functional	`Lift`	✔	transformations designed toward particular architectures and data characteristics	often incorporate user into optimization process
Domain-Specific	`cuBLAS, oneMKL`	✔	hand-optimized toward particular architecture and data characteristics	(✔)
Higher-Level	*`Futhark, Dex, ATL, Yang et al.[POPL'21], …`*	*We consider these approaches as greatly combinable with our approach* 👍		

Conclusion

- MDH is a formalism for expressing and optimizing *data-parallel computations.*
- MDH achieves higher *performance* & *portability* & *productivity* than related approaches.


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;

We have a Website

https://mdh-lang.org

Westfälische
Wilhelms-Universität
Münster

Questions?

Ari Rasch
a.rasch@wwu.de

Grateful for any kind of feedback

https://mdh-lang.org

Code Generation

https://atf-tuner.org

Code Optimization

https://hca-project.org

Code Execution