Universität Münster

LATHC 2026

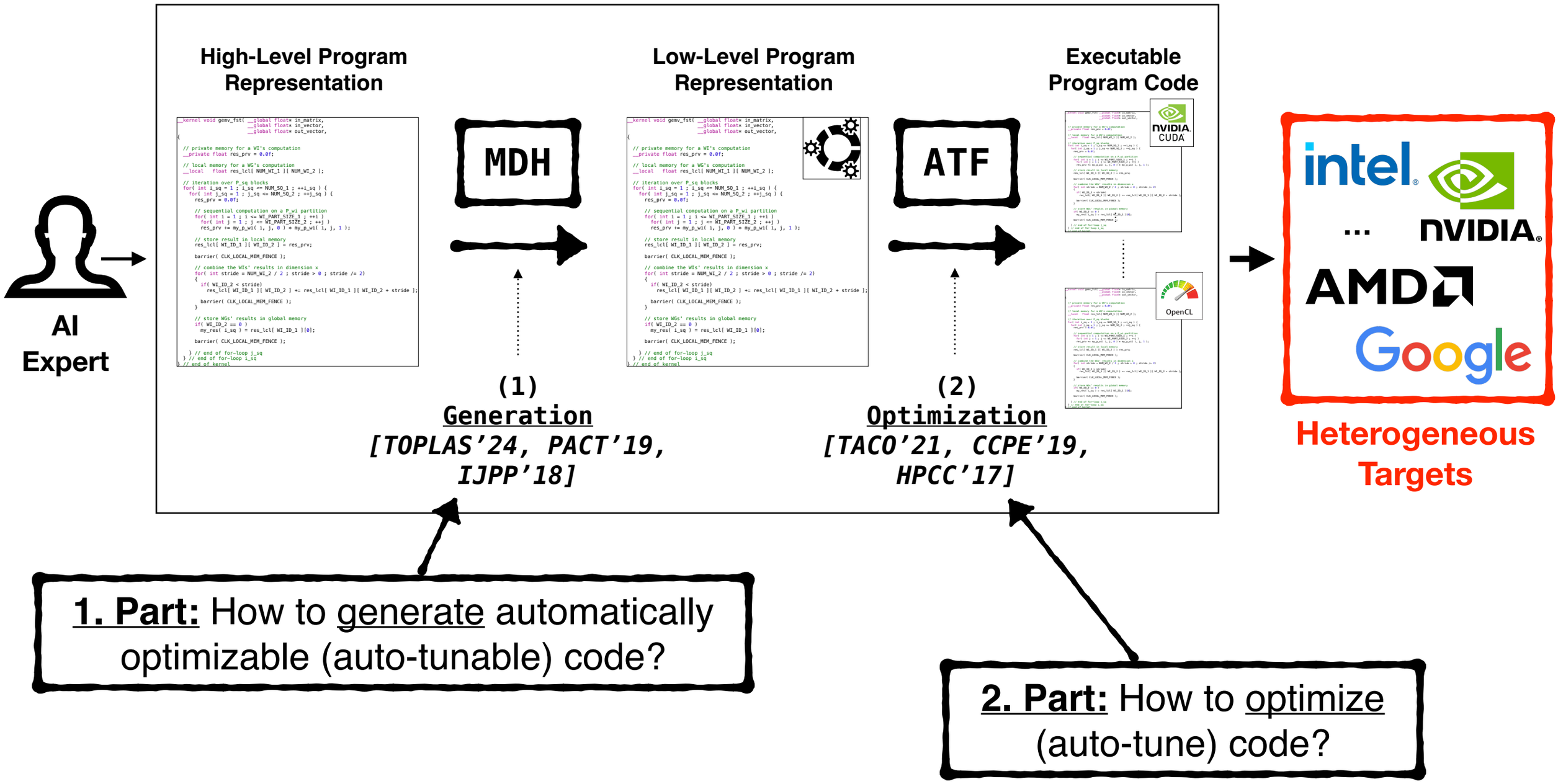
4th Languages, Architectures, and Tools for Heterogeneous Computing

AI Compilation for Heterogeneous Targets via MDH and ATF

Ari Rasch, Richard Schulze
University of Münster, Germany

Goal of MDH+ATF

An approach to ***Generating (MDH)*** & ***Optimizing (ATF)*** code for **AI computations**[1]:



[1] MDH+ATF extend beyond AI workloads and target arbitrary *data-parallel computations*.

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

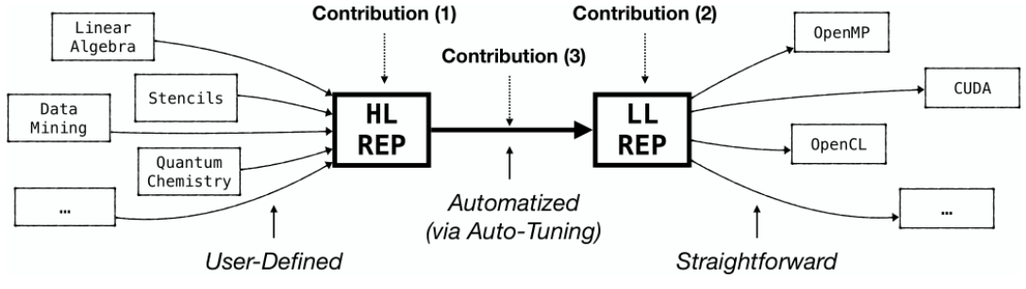
Overview

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

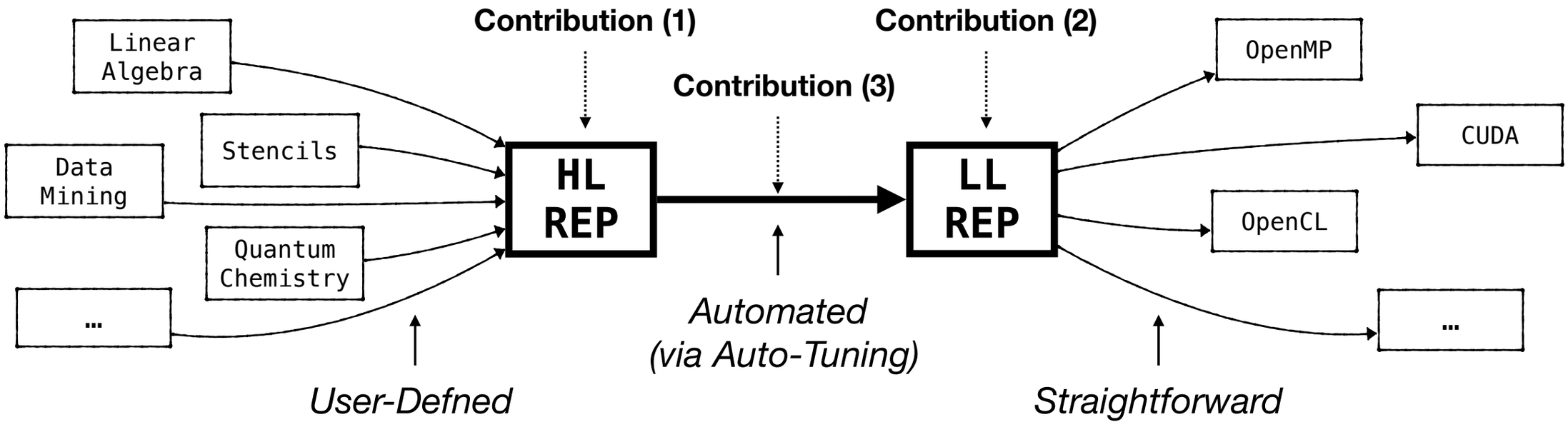
A full version of this article is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this article relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader 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* AI computations:



1. ***Contribution 1 (HL-REP):*** defines *AI computation*—based on *common algebraic properties*—*and* introduces *higher-order functions* for expressing these computations, independent of hardware and optimization details, while capturing high-level semantic information essential for generating high-performance code

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

3. ***Contribution 3 (→):*** introduces a *structured optimization process*—for arbitrary combinations of an AI computations and parallel architectures—that enables *fully automated* optimization (auto-tuning)

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 core algebraic 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, C, …

MDH: High-Level Representation

md_hom	f	$\circledast_1, \ldots, \circledast_D$
MatMul<F,F>	$*$	$+\!\!+, +\!\!+, +$
MatMul<F,T>	$*$	$+\!\!+, +\!\!+, +$
MatMul<T,F>	$*$	$+\!\!+, +\!\!+, +$
MatMul<T,T>	$*$	$+\!\!+, +\!\!+, +$
BatchMatMul<F,F>	$*$	$+\!\!+, \ldots, +\!\!+, +$
⋮	⋮	⋮
BiasAddGrad<NHWC>	id	$+, +, +, +\!\!+$
BiasAddGrad<NCHW>	id	$+, +\!\!+, +, +$
CheckNumerics	$(x) \mapsto (x == \text{NaN})$	$\vee, \ldots, \vee$
Sum<0><F>	id	$+, +\!\!+, +\!\!+, \ldots, +\!\!+$
Sum<0><T>	id	$+, +\!\!+, +\!\!+, \ldots, +\!\!+$
Sum<1><F>	id	$+\!\!+, +, +\!\!+, \ldots, +\!\!+$
Sum<0,1><F>	id	$+, +, +\!\!+, \ldots, +\!\!+$
⋮	⋮	⋮
Prod<0><F>	id	$*, +\!\!+, +\!\!+, \ldots, +\!\!+$
⋮	⋮	⋮
All<0><F>	id	$\&\&, +\!\!+, +\!\!+, \ldots, +\!\!+$
⋮	⋮	⋮

Linear Algebra, Contractions, …
(Computation Specification)

Views	inp_view I_1	inp_view I_2	out_view O
MatMul<F,F>	$(i,j,k) \mapsto (i,k)$	$(i,j,k) \mapsto (k,j)$	$(i,j,k) \mapsto (i,j)$
MatMul<F,T>	$(i,j,k) \mapsto (i,k)$	$(i,j,k) \mapsto (j,k)$	$(i,j,k) \mapsto (i,j)$
MatMul<T,F>	$(i,j,k) \mapsto (k,i)$	$(i,j,k) \mapsto (k,j)$	$(i,j,k) \mapsto (i,j)$
MatMul<T,T>	$(i,j,k) \mapsto (k,i)$	$(i,j,k) \mapsto (j,k)$	$(i,j,k) \mapsto (i,j)$
BatchMatMul<F,F>	$(b_1,\ldots,i,j,k) \mapsto (b_1,\ldots,i,k)$	$(b_1,\ldots,i,j,k) \mapsto (b_1,\ldots,k,j)$	$(b_1,\ldots,i,j,k) \mapsto (b_1,\ldots,i,j)$
⋮	⋮	⋮	⋮
BiasAddGrad<NHWC>	$(n,h,w,c) \mapsto (n,h,w,c)$	/	$(n,h,w,c) \mapsto (n,h,w)$
BiasAddGrad<NCHW>	$(n,c,h,w) \mapsto (n,c,h,w)$	/	$(n,c,h,w) \mapsto (n,h,w)$
CheckNumerics	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto ()$
Sum<0><F>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_2,\ldots,i_D)$
Sum<0><T>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (0,i_2,\ldots,i_D)$
Sum<1><F>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_1,i_3,\ldots,i_D)$
Sum<0,1><F>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_3,\ldots,i_D)$
⋮	⋮	⋮	⋮
Prod<0><F>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_2,\ldots,i_D)$
⋮	⋮	⋮	⋮
All<0><F>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_2,\ldots,i_D)$
⋮	⋮	⋮	⋮

Linear Algebra, Contractions, … *(Data Specification)*

md_hom	f	$\circledast_1, \ldots, \circledast_D$
Fill	id	$+\!\!+, \ldots, +\!\!+$
ExpandDims<0>	id	$+\!\!+, \ldots, +\!\!+$
ExpandDims<1>	id	$+\!\!+, \ldots, +\!\!+$
ExpandDims<0,1>	id	$+\!\!+, \ldots, +\!\!+$
⋮	⋮	⋮
Transpose<σ>	id	$+\!\!+, \ldots, +\!\!+$
Exp	exp	$+\!\!+, \ldots, +\!\!+$
Mul	$*$	$+\!\!+, \ldots, +\!\!+$
BiasAdd<NHWC>	$+$	$+\!\!+, +\!\!+, +\!\!+, +\!\!+$
BiasAdd<NCHW>	$+$	$+\!\!+, +\!\!+, +\!\!+, +\!\!+$
Range	$(s,d,i) \mapsto (s + d * i)$	$+\!\!+$

Point-Wise, Re-Shaping, …
(Computation Specification)

Views	inp_view I_1	inp_view I_2	out_view O
Fill	$(i_1,\ldots,i_D) \mapsto ()$	/	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$
ExpandDims<0>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (0,i_1,i_2,\ldots,i_D)$
ExpandDims<0>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_1,0,i_2,\ldots,i_D)$
ExpandDims<0,1>	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (0,0,i_1,\ldots,i_D)$
⋮	⋮	⋮	⋮
Transpose<σ>	$(i_1,\ldots,i_D) \mapsto (\sigma(i_1),\ldots,\sigma(i_D))$	/	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$
Exp	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	/	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$
Mul	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$ $(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$ $(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_D)$	$(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$ $(i_1,\ldots,i_D) \mapsto (i_1,\ldots,i_D)$
	⋮	⋮	⋮
BiasAdd<NHWC>	$(n,h,w,c) \mapsto (n,h,w,c)$	$(n,h,w,c) \mapsto (c)$	$(n,h,w,c) \mapsto (n,h,w,c)$
BiasAdd<NCHW>	$(n,c,h,w) \mapsto (n,c,h,w)$	$(n,c,h,w) \mapsto (c)$	$(n,c,h,w) \mapsto (n,c,h,w)$
Range	$(i) \mapsto ()$	$(i) \mapsto ()$	$(i) \mapsto (i)$

Point-Wise, Re-Shaping, … *(Data Specification)*

The **MDH high-level representation** is **capable of expressing various kinds of AI computations**

MDH: High-Level Representation

Quick Dive

We offer a **Python interface** for MDH's high-level program representation:

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

in MDH Formalism

MatVec

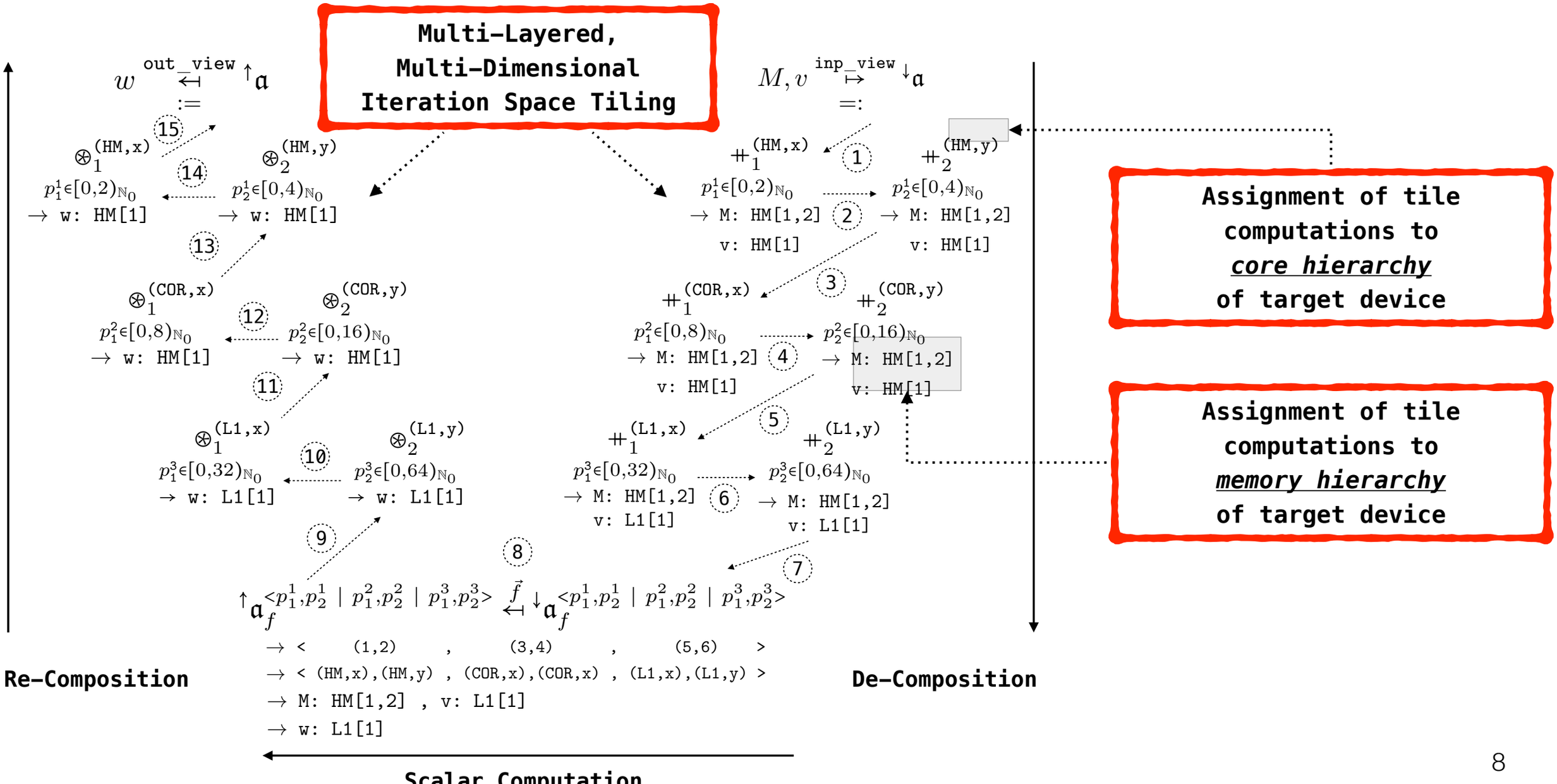
```python
def matvec( T: BasicType, I: int, K: int ):

    @mdh()
    def mdh_matvec():
        return (
            out_view[T]( w = [lambda i,k: (i)] ),
              md_hom[I,K]( mul, ( cc, pw(scalar_plus) ) ),
                inp_view[T,T]( M = [lambda i,k: (i,k)] ,
                               v = [lambda i,k: (k)  ] )
        )
```

The **MDH-Python-Interface** is designed to be
close to MDH's formal representation

MDH: Low-Level Representation

Goals:

1. Expressing a hardware- & data-optimized *de-composition* and *re-composition* of an AI computation, 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: 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)

exploiting core hierarchy (parallelization)

exploiting memory hierarchy (data movements)

...

...

...

...

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

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

[1] We optionally allow (expert) users to incorporate their knowledge into the optimization process via *MDH-Based Schedules* [CC'23]

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**

Auto-Tuning Framework (ATF) provides a key advantage over related approaches:

ATF identifies values of performance-critical parameters with

***interdependencies* among them**

via optimized processes for

generating & *storing* & *exploring*

the spaces of interdependent parameters

ATF auto-tunes programs written in **arbitrary programming languages** (e.g., CUDA and OpenMP).

For this, ATF introduces:

```
tuner.addParameter( “tp_1”, T1 );
tuner.addParameter( “tp_2", T2 );
// ...

tuner.addConstraint(
  [](T1 tp_1, T2 tp_2, … ) -> bool
```

traditional constraints

Defined on:
***search space* (traditional)**
vs. *parameters* (ATF)

```
tuner.addParameter( “tp_1”, R1, [](T1 tp_1) -> bool { /* … */ } );
tuner.addParameter( “tp_2”, R2, [](T2 tp_2) -> bool { /* … */ } );
```

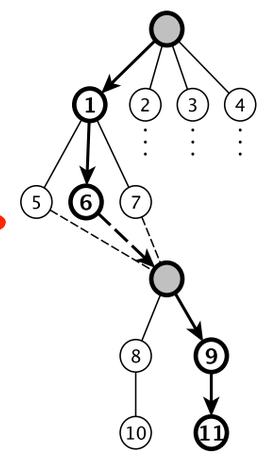
ATF
parameter constraints

```
SP := [ (1,1) | (2,1) | (2,2) | … ]
```

traditional search space

Structure is:
***verbose & 1D* (traditional)**
vs. *compact & nD* (ATF)

ATF
CoT search space



ATF: User Interfaces

In a nutshell

ATF's ***Python-based*** user interface[1]:

ATF Website

Name

Range
(either `interval` or `set`)

Constraint
(may contain tuning parameters)

```python
# Input Size
N = 1000

# Step 1: Generate the Search Space
WPT = TP( 'WPT'                          ,
          Interval( 1,N )                ,
          lambda WPT: N % WPT == 0 )
LS  = TP( 'LS'                                ,
          Interval( 1,N )                     ,
          lambda WPT,LS: (N/WPT) % LS == 0 )

# Step 2: Implement a Cost Function
saxpy_kernel = # … (kernel's code & name)

N = np.int32( N )
a = np.float32( np.random.random() )
x = np.random.rand(N).astype(np.float32)
y = np.random.rand(N).astype(np.float32)

cf = cuda.CostFunction( saxpy_cuda_kernel )      \
          .device_id( 0 )                        \
          .kernel_args( N, a,x,y )               \
          .grid_dim( lambda WPT,LS: N/WPT/LS ) \
          .block_dim( lambda LS: LS )

# Step 3: Explore the Search Space
config = Tuner().tuning_parameters( WPT,LS )     \
                .search_technique( AUCBandit() ) \
                .tune( cf, Evaluations(50) )
```

Arbitrary & pre-implemented *cost functions*

Various pre-implemented
Search Techniques & Abort Conditions

PDF **Schulze, Gorlatch, Rasch, "pyATF: Constraint-Based Auto-Tuning in Python", CC'25**

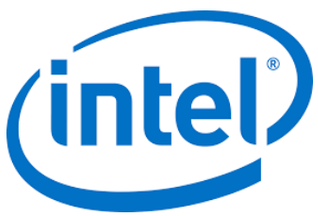
[1] ATF also offers a GPL-based interface for (online-tuning) C++ programs [HPCC'17], as well as a DSL-based interface (offline tuning) [CCPE'18]

Experimental Results

MDH+ATF 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. Domain-Specific Libraries:
   - NVIDIA cuBLAS & cuDNN (GPU)
   - Intel oneMKL & oneDNN (CPU)



[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

Case Studies:

1. Linear Algebra Routines:
   - **Matrix Multiplication (`MatMul`)**
   - Matrix-Vector Multiplication (`MatVec`)
   - Dot Product (`Dot`)
2. Stencil Computations:
   - Jacobi Computation (`Jacobi3D`)
   - 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`)

Experimental Results

Case Study: Matrix Multiplication (MatMul)

Performance:

Linear Algebra	NVIDIA Ampere GPU	
	MatMul	
	10,500,64	1024,1024,1024
TVM+Ansor	1.00	1.00
PPCG	2.20	2.73
PPCG+ATF	1.20	1.87
cuBLAS	1.40	0.92
cuBLASEx	1.20	0.91
cuBLASLt	1.20	0.88

Linear Algebra	Intel Skylake CPU	
	MatMul	
	10,500,64	1024,1024,1024
TVM+Ansor	1.06	1.15
Pluto	3.21	12.25
Pluto+ATF	2.98	4.78
oneMKL	6.27	0.69
oneMKL(JIT)	0.65	-

Portability:

Linear Algebra	Pennycook Metric	
	MatMul	
	10,500,64	1024,1024,1024
MDH+ATF	0.88	0.54
TVM+Ansor	0.83	0.50

Productivity:

```
def matmul(T: BasicType, I: int, J: int, K: int):
    @mdh()
    def matmul__T_I_J_K():
        return (
            out_view[T](C=[lambda i, j, k: (i,j) ),
            md_hom[I, J, K]( f_mul, (cc, cc, pw(add)) ),
            inp_view[T, T](A=[lambda i, j, k: (i,k) ,
                              B=[lambda i, j, k: (k,j) ])
        )
    return matmul__T_I_J_K
```

Higher **Performance** than vendor libraries; Highest **Portability**; **Productive**, by requiring basic algebraic properties only

Experimental Results

Case Study: Multi-Channel Convolution (MCC)

Performance:

Deep Learning	NVIDIA Ampere GPU			
	ResNet-50		VGG-16	
TVM+Ansor	1.00	1.05	0.93	0.88
PPCG	3456.16	–	1661.14	5.77
PPCG+ATF	3.28	13.76	4.26	9.46
cuDNN	0.92	1.85	1.22	1.94

Deep Learning	Intel Skylake CPU			
	ResNet-50		VGG-16	
TVM+Ansor	1.53	1.14	1.97	2.38
Pluto	355.81	364.43	130.80	186.25
Pluto+ATF	13.08	170.69	3.11	53.61
oneDNN	0.39	5.07	1.22	9.01

Portability:

Deep Learning	Pennycook Metric			
	ResNet-50		VGG-16	
MDH+ATF	0.67	0.91	0.98	0.97
TVM+Ansor	0.53	0.89	0.76	0.70

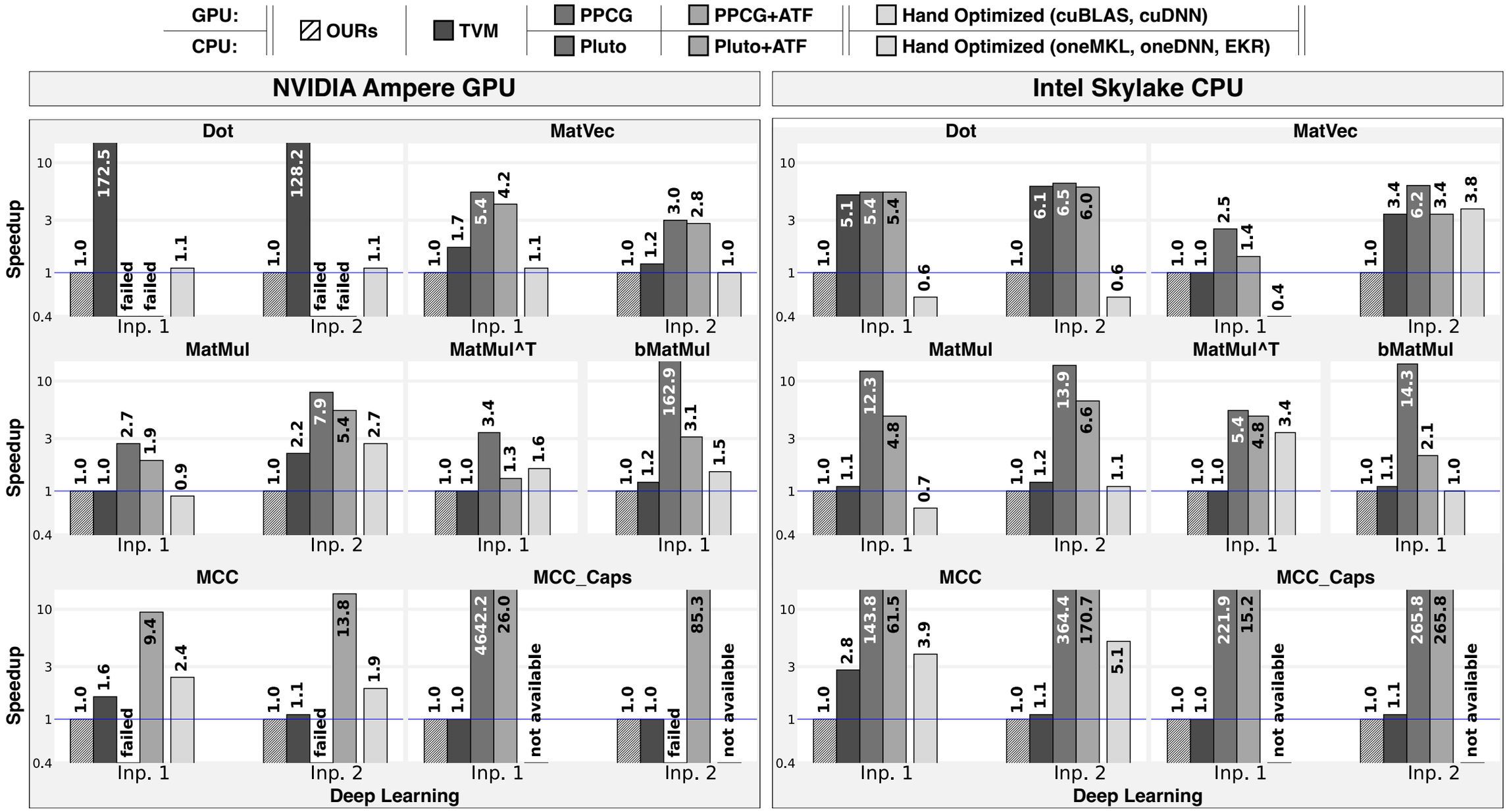
Productivity:

```
def mcc(T: BasicType,
        N: int, P: int, Q: int, K: int, R: int, S: int, C: int):
  @mdh(inp_img=Buffer[T, [N, (2 * P) + R - 1, (2 * Q) + S - 1, C]])
  def mcc__T_N_P_Q_K_R_S_C():
    return (
      out_view[T](...),
      md_hom[N, P, Q, K, R, S, C](...),
      inp_view[T, T](
        img=[lambda n, p, q, k, r, s, c:
                    (n, (2 * p) + r, (2 * q) + s, c)],
        flt=[lambda n, p, q, k, r, s, c:
                    (k, r, s, c)],
      ),
    )

  return mcc__T_N_P_Q_K_R_S_C
```

Higher **Performance** than vendor libraries; Highest **Portability**; **Productive**, by capturing buffer sizes in its type system

Experimental Results

Further Case Studies:



MDH+ATF achieve **similar PPP** advantages for these studies

Experimental Results

Why does MDH+ATF achieves such PPP ?

Performance:

- Exploits **algebraic high-level information** (reductions operators)
- **Rich optimization space** (data layouts, block parallelization, …)

Portability:

- Optimization driven by **common algebraic properties** of computations
- **Auto-tuning–friendly** optimization space

Productivity:

- **User-facing language** designed to be algebraically **uniform & minimalistic & structured**
- **Formally** grounded

MDH+ATF leverages **algebraic properties** of AI computations for **aggressive**, **device-agnostic** optimization and **concise formulation** of computations

Summary

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

- **MDH** formally **introduces program representations** on both:

  - **high level**, for conveniently expressing — in one uniform formalism — the various kinds of AI 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 AI computations for the memory and core hierarchies of contemporary parallel architectures (GPUs, CPUs, etc)

    - **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**

- **ATF** automatically **identifies optimized values of performance-critical program parameters** that may be **constrained**

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

MDH: WIP & Future Directions

Many promising future directions (detailed discussion available *here*):

High-Level Program Transformations (a.k.a. Fusion)

$$\texttt{md\_hom}(g,(++,\ldots,++)) \circ \texttt{md\_hom}(f,(++,\ldots,++)) \rightarrow \texttt{md\_hom}(g \circ f,(++,\ldots,++))$$

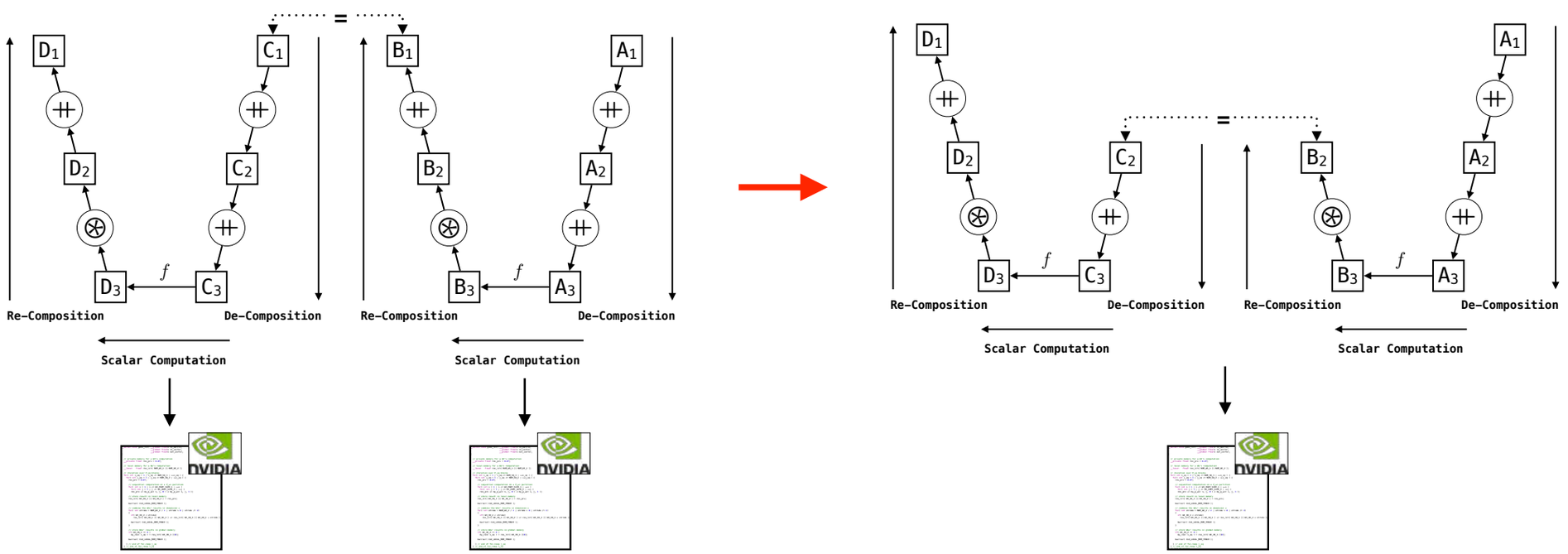
Sparse Computations

...

Domain-Specific HW

...

Low-Level Program Transformations (a.k.a. Fusion)



... (etc)

We consider **MDH** to be a **promising (formal) foundation** for these goals,
e.g., due to its **uniform representation** and its **captured algebraic information**

Universität Münster

Questions?

https://richardschulze.net
r.schulze@uni-muenster.de

Richard Schulze

https://mdh-lang.org

Code Generation

https://atf-tuner.org

Code Optimization

https://arirasch.net
a.rasch@uni-muenster.de

Ari Rasch