Universität Münster

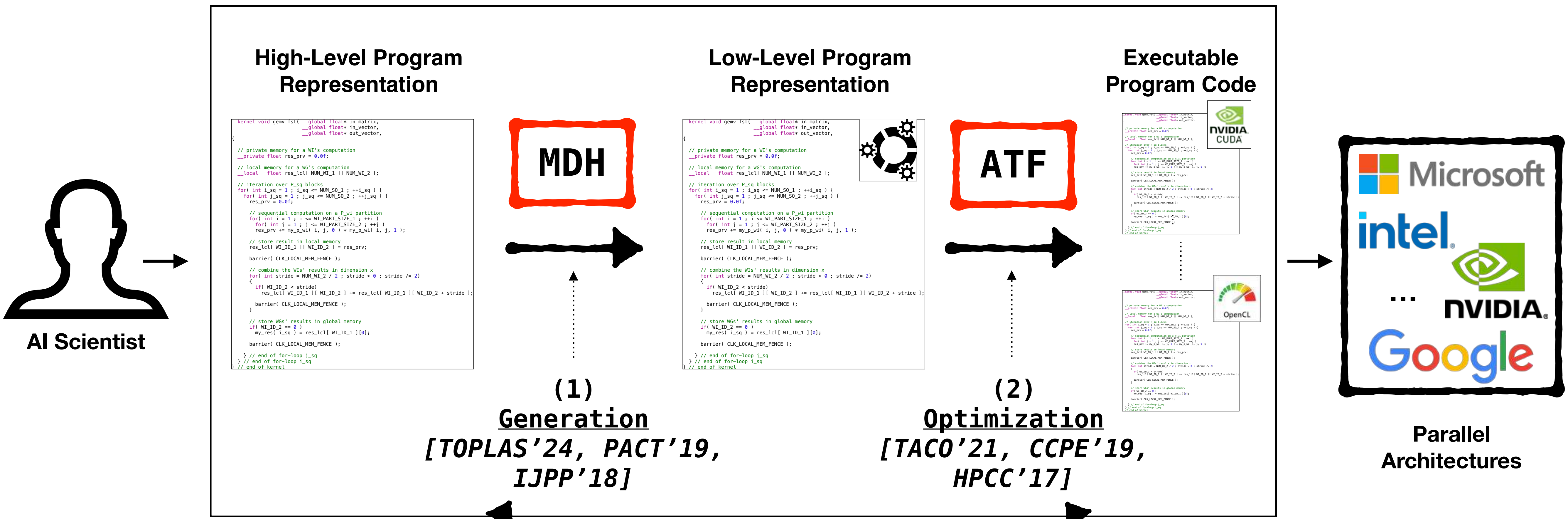
Compilers for Machine Learning

MDH+ATF: Code *Generation* & *Optimization* for Deep-Learning Computations

Ari Rasch, Richard Schulze
University of Münster, Germany

Goal of MDH+ATF

An approach to ***Generating (MDH)*** & ***Optimizing (ATF)*** code for **DL computations**:

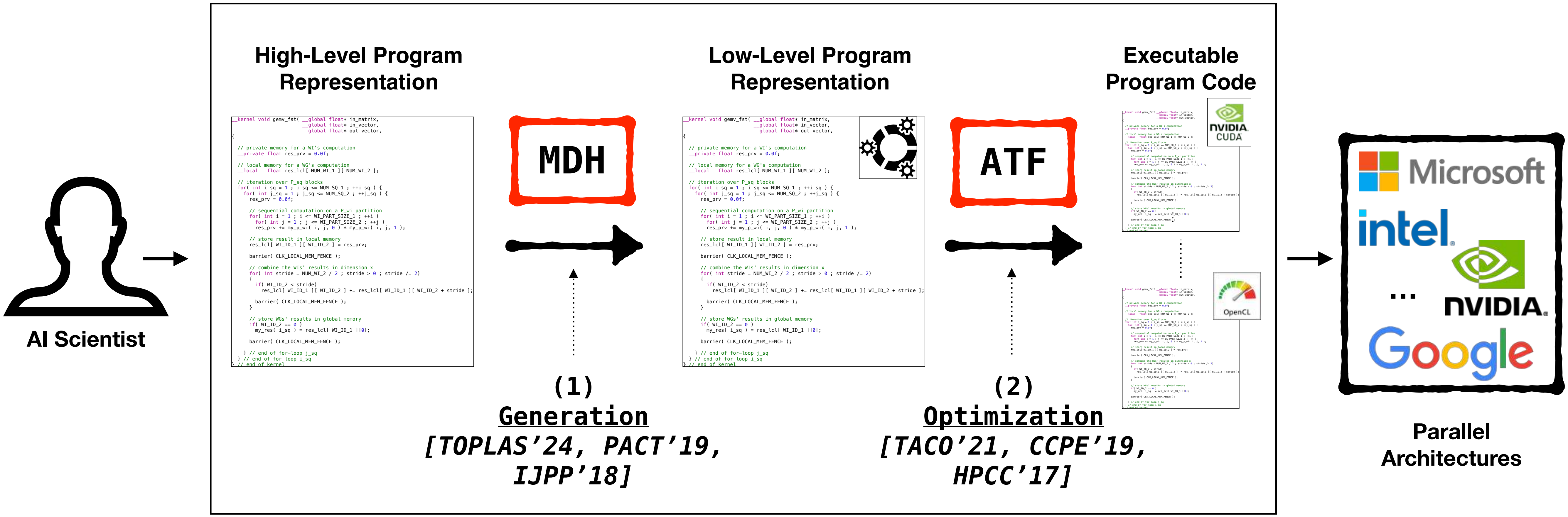


1. Part: How to generate automatically optimizable (auto-tunable) DL code?

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

Goal of MDH+ATF

An approach to ***Generating (MDH)*** & ***Optimizing (ATF)*** code for **DL computations:**



The ultimate goal of **MDH+ATF** is to simultaneously achieve
Performance & **Portability** & **Productivity**
for **DL computations**[1]

[1] MDH not limited to DL: targets arbitrary kinds of 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

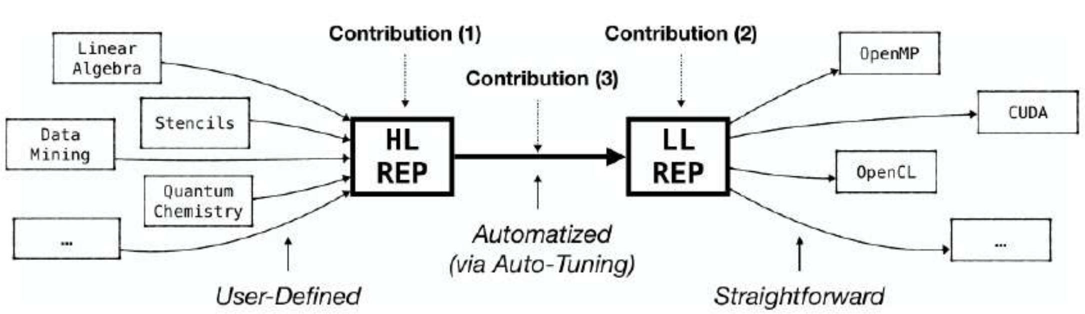
Overview

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

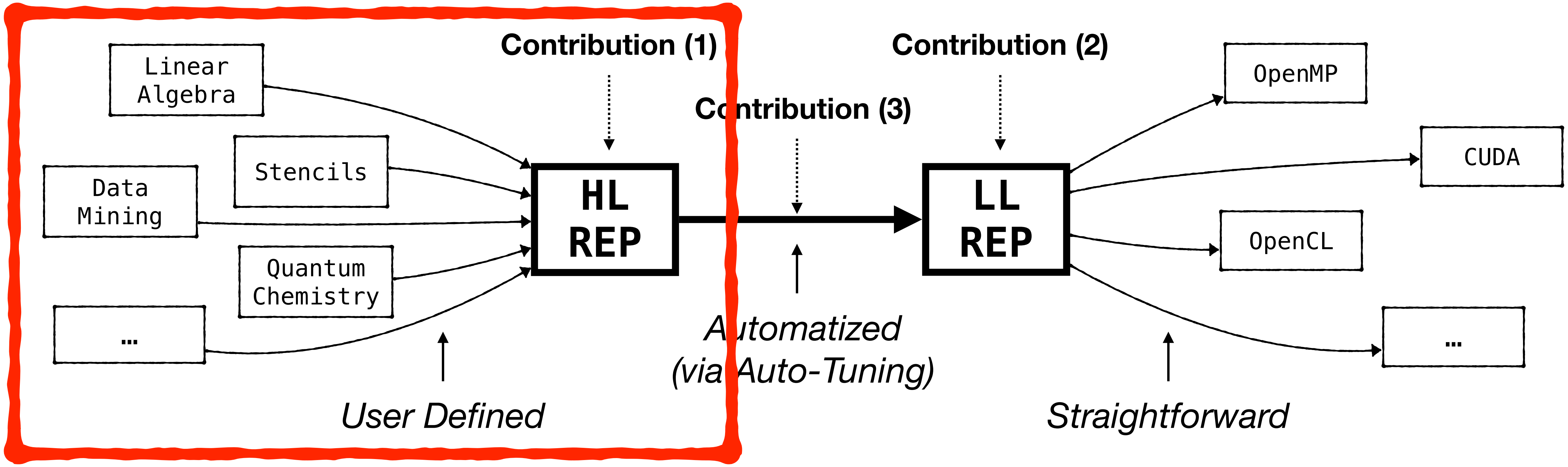
A full version of this article is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this article relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader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

Focus Today

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.

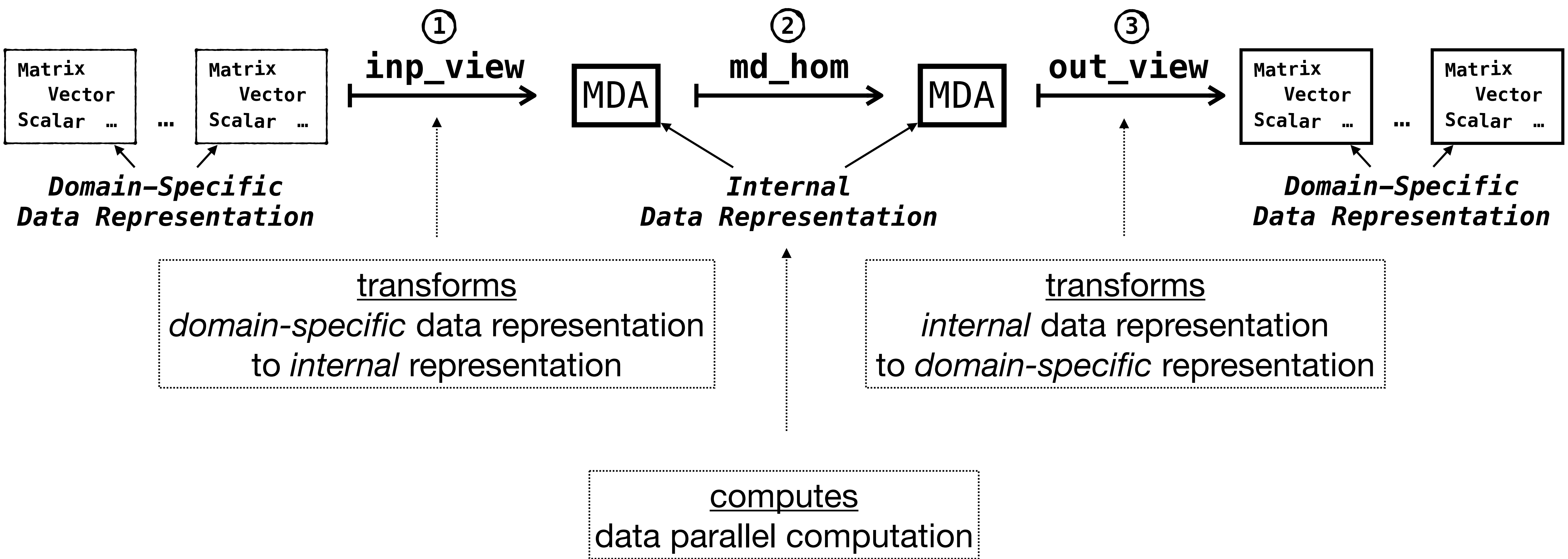
2. **Minimalistic**:
   should rely on less building blocks to keep language small and simple

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 (formally) defines DL computations[1], and it expresses these computations using exactly three straightforwardly composed higher-order functions only

[1] MDH not limited to DL: targets arbitrary kinds of data-parallel computations

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 High-Level Representation of MatVec

```
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++

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

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

<u>MDH:</u> 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	$(x) \mapsto (x == \texttt{NaN})$	$\vee, \ldots, \vee$
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, Reductions, … (<u>Non-Endomorphic Operators</u>)
— Computation Specification —

Views	inp_view		out_view
	I_1	I_2	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, Reductions, … (<u>Non-Endomorphic Operators</u>)
— 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, … (<u>Endomorphic Operators</u>)
— Computation Specification —

Views	inp_view		out_view
	I_1	I_2	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, … (<u>Endomorphic Operators</u>)
— Data Specification —

Important <u>DL computations</u> can be (naturally & uniformly) expressed in MDH

MDH: High-Level Representation

Differences to existing approaches:

In a nutshell

```
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

MatVec in MDH

MDH **explicitly captures** the **operations for combining intermediate results** (e.g., add)

MDH **separates** the **scalar operation** (e.g., `mul`) from the **operations for combining intermediate results** (e.g., add)

The *additional semantic information* & *their explicit separation* allows MDH to achieve higher *"Performance"* & *"Expressivity"* [1]

```
#map1 = affine_map<(d0, d1) -> (d0, d1)>
#map2 = affine_map<(d0, d1) -> (d1)    >
#map3 = affine_map<(d0, d1) -> (d0)    >
module {
  func.func @main() {
    %M = memref.alloc() : memref<128x64xf32>
    %v = memref.alloc() : memref<64xf32>
    %w = memref.alloc() : memref<128xf32>
    linalg.generic
    {
      indexing_maps = [#map1, #map2, #map3],
      iterator_types = ["parallel", "reduction"]
    } ins(%M,%v:memref<128x64xf32>,memref<64xf32>)
      outs(%w:memref<128xf32>) {
    ^bb0(%in_1: f32, %in_2: f32, %out: f32):
      %0 = arith.mulf %in_1, %in_2 : f32
      %1 = arith.addf %out, %0 : f32
      linalg.yield %1 : f32
    }
    return
  }
```

MatVec in linalg

[1] "Linalg vs MDH: A Comparison of two MLIR Dialects", EuroLLVM'24

Note: MDH also describes code generation from its high-level representation

MDH: High-Level Representation

Differences to existing approaches:

In a nutshell

```
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) )
```

MatVec in MDH

MDH allows **multiple combine operators** and has **(formally defined) semantics** for them

```
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 ]
```

MatVec in TVM

MDH allows using **arbitrary combine operators**, whereas TVM is limited to: `min`, `max`, `sum`

tqchen Feb '20

This is a restriction in the current tensor expression language, because reduction is quite complicated to be processed in nested form.

There are ongoing effort to enhance low-level IR passes to enable more powerful tensorization, which hopefully will resolve the issue you raised.

MDH's <u>stronger support for combine operators</u> contributes to high *"Expressivity"* (e.g., for potentially upcoming DL 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) ) ∘
                                md_hom<I,K>( *, (++,+) ) ∘
                                    inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

MDH

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 very close to MDH's formal representation

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) ) ∘
                              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():
        @scalar_function(
            out_scalar_type = [T  ],
            inp_scalar_type = [T,T]
        )
        def mul( out,inp ):
            out['w'] = inp['M'] * inp['v']

        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)   ] )
        )
```

We allow custom scalar functions

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) ) ∘
                          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():

        def cc(T: BasicType, D: int, d: int):
            @combine_operator(
                index_set_function   = IndexSetFunction.id,
                scalar_type          = T,
                dimensionality       = D,
                operating_dimension = d
            )
            def cc__T_D_d(res: MDA, lhs: MDA, rhs: MDA):
                for i[1, ..., d - 1] in I[1, ..., d - 1]:
                    for i[d + 1, ..., D] in I[d + 1, ..., D]:
                        for i[d] in P:
                            res[i[1, ..., d, ..., D]] = lhs[i[1, ..., d, ..., D]]
                        for i[d] in Q:
                            res[i[1, ..., d, ..., D]] = rhs[i[1, ..., d, ..., D]]

            return cc__T_D_d

        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)   ] )
        )
```

We allow custom combine operators

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) ) ∘
                            md_hom<I,K>( *, (⧺,+) ) ∘
                              inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

MDH

in MDH Formalism

MatVec

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

    @mdh( out( w = Buffer[T]                   ) ,
          inp( M = Buffer[T], v = Buffer[T] ) ,
          combine_ops( cc, pw(scalar-plus) )
         )
    def mdh_matvec(w, M, v):
        for i in range(I):
            for k in range(K):
                w[i] = M[i, k] * v[k]
```

MDH also takes as input
straightforward sequential Python code
(instead of DSL programs)

MDH: High-Level Representation

Quick Dive

We offer an **MLIR interface** for MDH's high-level program representation:

```
out_view<f32>( w:(i,k)↦(i) )
   md_hom<128,64>( *, (++,+) )
      inp_view<f32,f32>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

MDH

in MDH Formalism

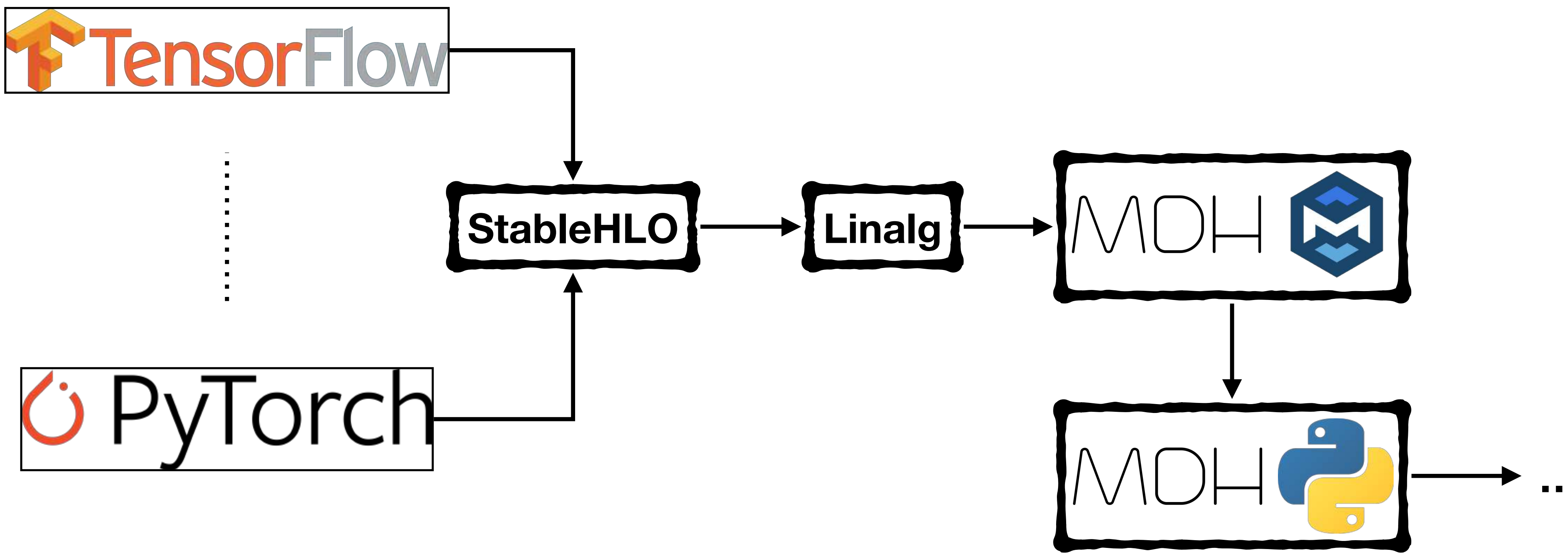
MatVec

```
func.func @main()
{
  %M = memref.alloc() : memref<128x64xf32>
  %v = memref.alloc() : memref<64xf32>

  %w = mdh.compute "mdh_matvec"
  {
    inp_view =
    [
      [ affine_map<( i,k ) -> ( i,k )> ],
      [ affine_map<( i,k ) -> ( k )   > ]
    ],

    md_hom =
    {
      scalar_func = @mul,
      combine_ops = [ "cc", ["pw",@add] ]
    },

    out_view =
    [
      [ affine_map<( i,k ) -> ( i )> ]
    ]
  }
  {
    inp_types = [ f32, f32 ],
    mda_size  = [ 128,64 ],
    out_types = [ f32 ]
  }( %M,%v ):( memref<128x64xf32>,memref<64xf32> )
            -> memref<128xf32>

  return
}
```

MLIR interface for MDH

MDH: High-Level Representation

Quick Dive



Implemented by
Jens & Lars Hunloh

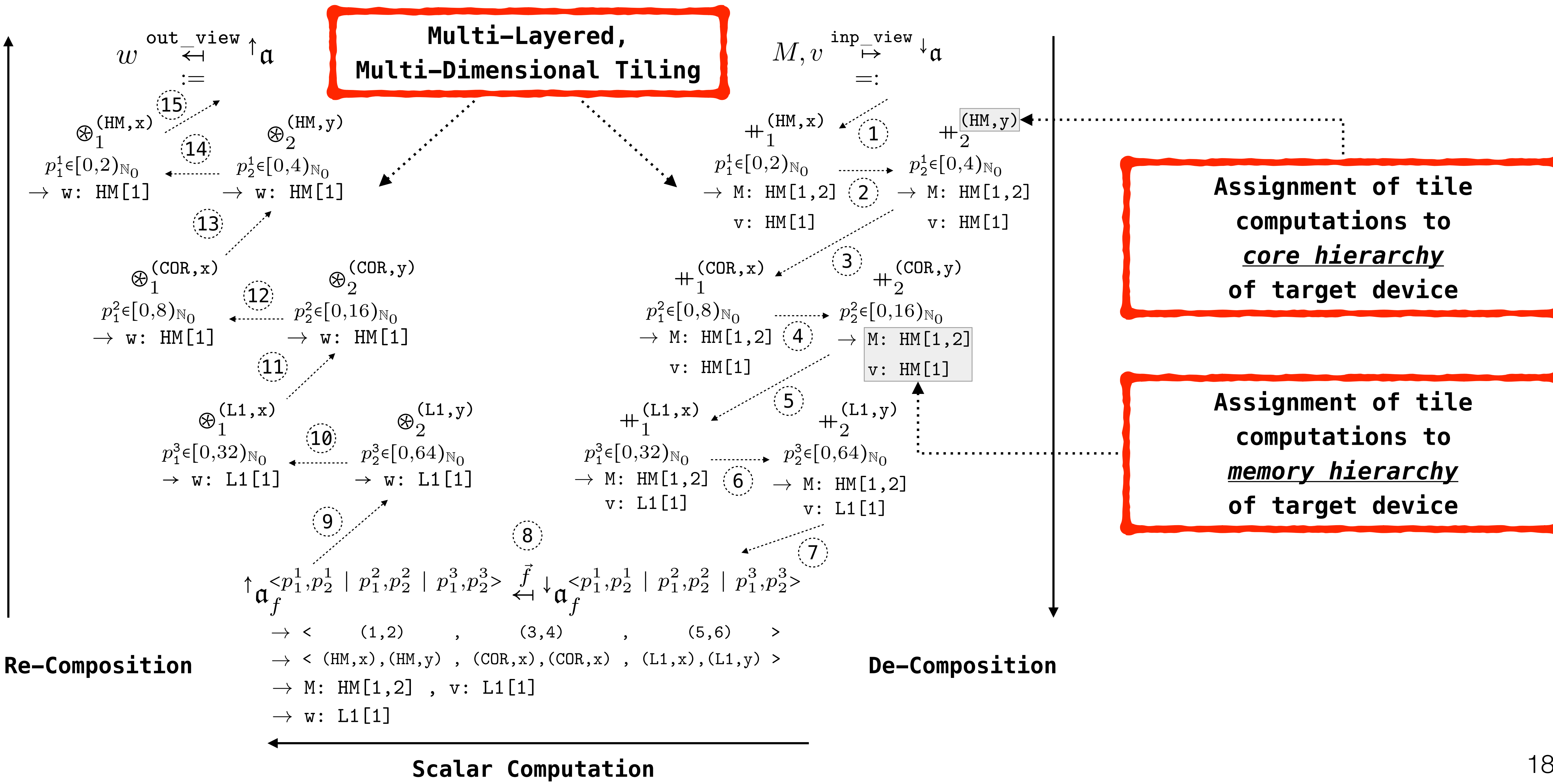
Our MLIR interface allows
easy integration of MDH into DL Frameworks

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: 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 → $\mathbb{N}$	number of parts
D1	$\sigma_{\downarrow\text{-ord}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ MDH-LVL	de-composition order
D2	$\leftrightarrow_{\downarrow\text{-ass}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ ASM-LVL	ASM assignment (de-composition)
D3	$\downarrow\text{-mem}^{\text{<ib>}}$	MDH-LVL → MR	memory regions of input BUFs (ib)
D4	$\sigma^{\text{<ib>}}_{\downarrow\text{-mem}}$	MDH-LVL → $[1,\ldots,D^{\text{IB}}_{\text{ib}}]_S$	memory layouts of input BUFs (ib)
S1	$\sigma_{f\text{-ord}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ MDH-LVL	scalar function order
S2	$\leftrightarrow_{f\text{-ass}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ ASM-LVL	ASM assignment (scalar function)
S3	$f^{\downarrow}\text{-mem}^{\text{<ib>}}$	MR	memory region of input BUF (ib)
S4	$\sigma^{\text{<ib>}}_{f^{\downarrow}\text{-mem}}$	$[1,\ldots,D^{\text{IB}}_{\text{ib}}]_S$	memory layout of input BUF (ib)
S5	$f^{\uparrow}\text{-mem}^{\text{<ob>}}$	MR	memory region of output BUF (ob)
S6	$\sigma^{\text{<ob>}}_{f^{\uparrow}\text{-mem}}$	$[1,\ldots,D^{\text{OB}}_{\text{ob}}]_S$	memory layout of output BUF (ob)
R1	$\sigma_{\uparrow\text{-ord}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ MDH-LVL	re-composition order
R2	$\leftrightarrow_{\uparrow\text{-ass}}$	MDH-LVL $\hookrightarrow\!\!\!\!\rightarrow$ ASM-LVL	ASM assignment (re-composition)
R3	$\uparrow\text{-mem}^{\text{<ob>}}$	MDH-LVL → MR	memory regions of output BUFs (ob)
R4	$\sigma^{\text{<ob>}}_{\uparrow\text{-mem}}$	MDH-LVL → $[1,\ldots,D^{\text{OB}}_{\text{ob}}]_S$	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
(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]

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

NVCC vs NVRTC

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	–	–

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)

```
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)[1]

```
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)

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

MDH: WIP & Future Directions

Many promising future directions:

High-Level Program Transformations

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

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

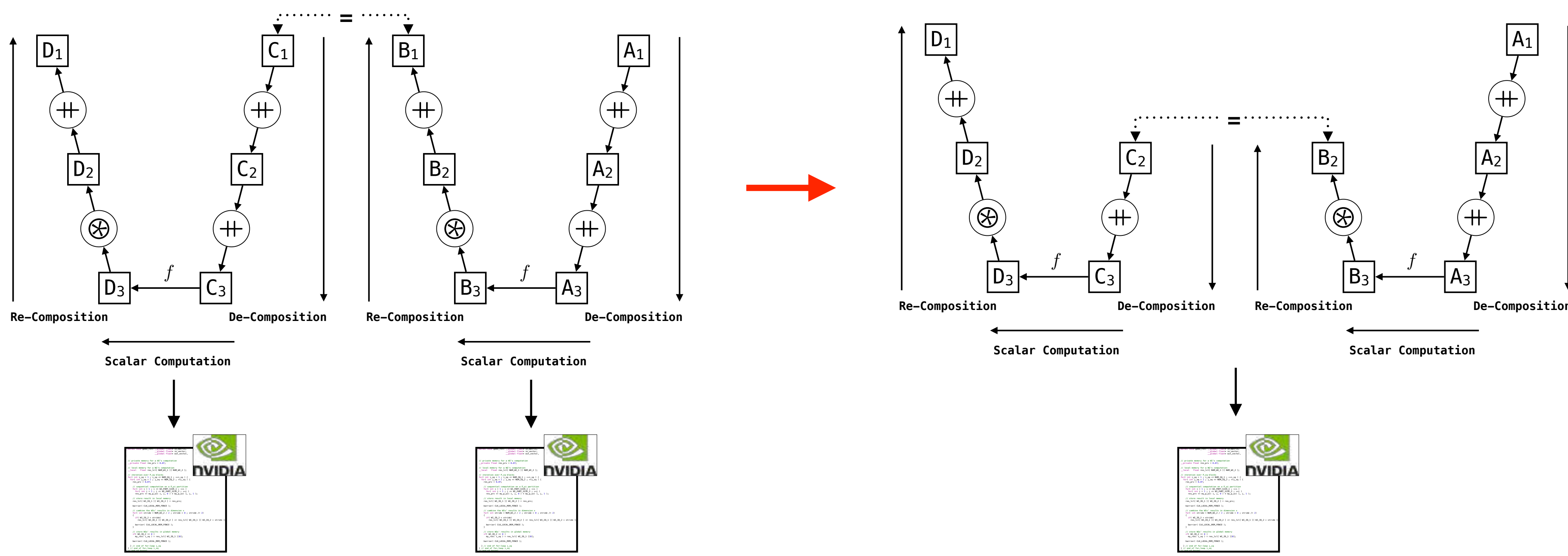
Sparse Computations

...

Domain-Specific HW

...

Fusion Optimizations



... (etc)

We expect MDH to be a promising (formal) foundation for these goals, e.g., due to the semantic/algebraic information captured in its representation

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

- **Many promising future directions, particularly for DL!**

MDH: Summary

That was a very quick, informal dive!

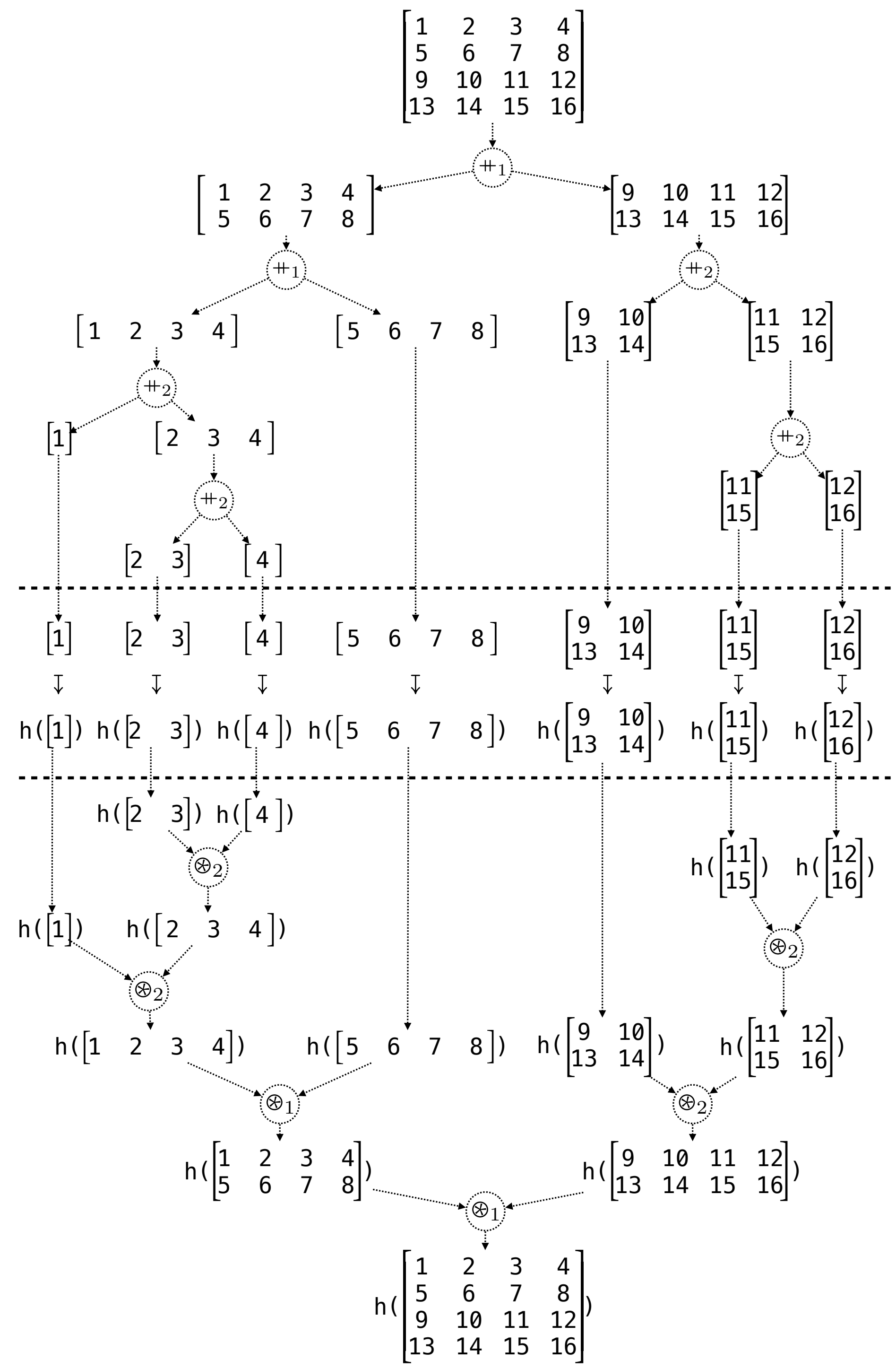


Fig. 10. MDH property recursively applied to a two-dimensional example computation

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 $\overset{1}{\Rightarrow}{}^{\mathsf{MDA}}_{\mathsf{MDA}}, \ldots, \overset{D}{\Rightarrow}{}^{\mathsf{MDA}}_{\mathsf{MDA}} : \mathsf{MDA\text{-}IDX\text{-}SETs} \to \mathsf{MDA\text{-}IDX\text{-}SETs}$ 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

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

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

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)

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

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

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 $\mathsf{CO}^{<\Rightarrow^{\mathsf{MDA}}_{\mathsf{MDA}} \mid T \mid D \mid d>}$.

... (>130 pages)

All concepts are fully formally defined in the MDH paper (arXiv version)

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

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 { /* … */ } );
```

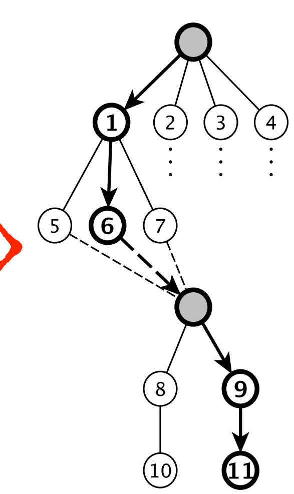
ATF
parameter constraints **[1]**

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

traditional search space

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

ATF
CoT search space **[1]**



[1] “Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)”, *TACO’21*

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)*

Arbitrary & pre-implemented *cost functions*

Various pre-implemented *Search Techniques & Abort Conditions*

```
# 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_code = # ...

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 = opencl.CostFunction( saxpy_code )
           .platform_id( 0 )                    \
           .device_id( 0 )                      \
           .kernel_args( N, a,x,y )             \
           .glb_size( lambda WPT,LS: N/WPT )    \
           .lcl_size( lambda     LS: LS    )

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

ATF's ***Python-based*** user interface was **presented at CC: Sat, 1. March, 10:30am**

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, as well as a DSL-based interface (offline tuning)

ATF: Summary

ATF 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 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)

Universität
Münster

We have a poster at C4ML !

Questions?

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

Ari
Rasch

https://mdh-lang.org

Code
Generation

https://atf-tuner.org

Code
Optimization

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

Richard
Schulze