Westfälische Wilhelms-Universität Münster

Using MLIR for Multi-Dimensional Homomorphisms

Ari Rasch, Richard Schulze, Sergei Gorlatch

University of Münster, Germany

Special Thanks to Alex Z.

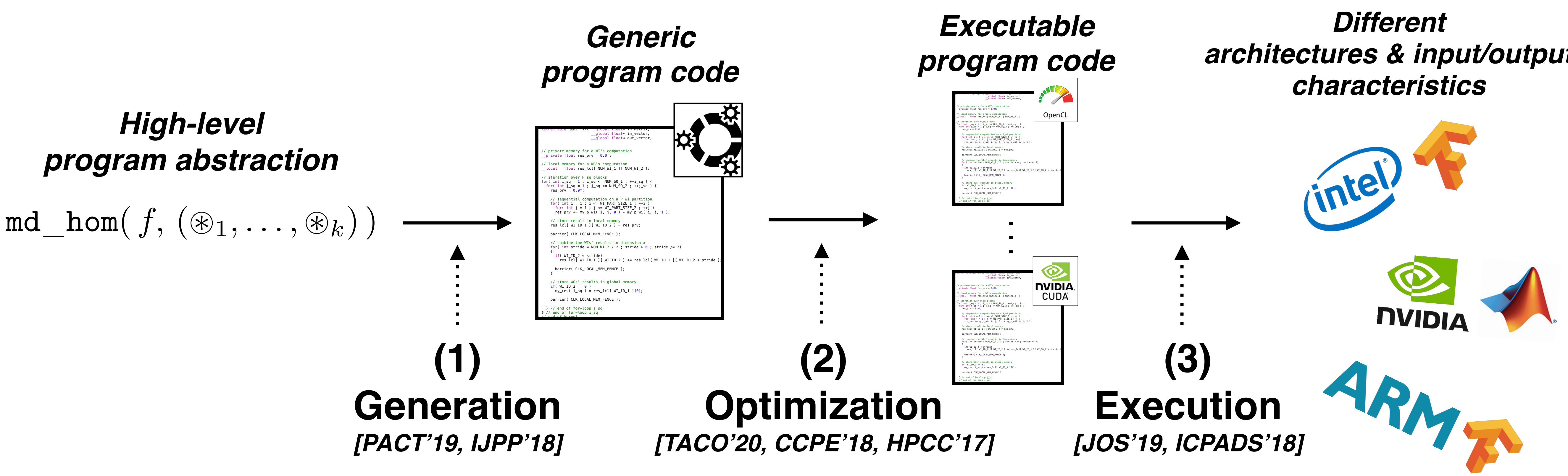
⚠ Please Notice ⚠

<u>This talk will be on a quite high level:</u>

- **we have no technical contribution so far;**
- **we have a vision/idea about how our MDH approach could look like in MLIR;**
- **we want to first discuss and asses with you guys how useful such an integration could potentially be from your point of view.**

Our Background

We are the developers of the MDH approach:



- **Multi-Dimensional Homomorphisms (MDHs)** are **formally defined** to **cover data-parallel computations**: linear algebra routines (BLAS), stencils computations, …
- We enable **conveniently** implementing MDHs by providing a **high-level DSL** for them.
- We provide a **DSL compiler** that automatically **generates auto-tunable low-level code** (OpenCL, CUDA, OpenMP, …) for MDHs;
- Our generated code is **fully automatically optimizable** (auto-tunable) — for any particular combination of a **target architecture** and/or **input/output characteristics** — by being generated as targeted to an **abstract machine model** and as **parametrized in all these abstract model's performance-critical parameters.**

Experimental Results

Stencils

CPU	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
Lift [2]	4.90	5.96	1.94	2.49
MKL-DNN	6.99	14.31	N/A	N/A

GPU	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
Lift [2]	2.33	1.09	1.14	1.02
cuDNN	3.78	19.11	N/A	N/A

[2] Hagedorn et. al, "High Performance Stencil Code Generation with LIFT.", **CGO'18 (Best Paper Award)**.

Data Mining

CPU	Probabilistic Record Linkage					
	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
EKR [5]	1.87	2.06	4.98	13.86	28.34	39.36

[5] Forchhammer et al. "Duplicate Detection on GPUs.", **HFSL'13**.

Our MDH approach achieves often better performance than well-performing competitors [1]

[1] Rasch, Schulze, Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.", PACT'19

Tensor Contractions

GPU	Tensor Contractions								
	RW 1	RW 2	RW 3	RW 4	RW 5	RW 6	RW 7	RW 8	RW 9
COGENT [3]	1.26	1.16	2.12	1.24	1.18	1.36	1.48	1.44	1.85
F-TC [4]	1.19	2.00	1.43	2.89	1.35	1.54	1.25	2.02	1.49

[3] Kim et. al. "A Code Generator for High-Performance Tensor Contractions on GPUs.", **CGO'19**.

[4] Vasilache et al. "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically.", ***TACO'19***.

Linear Algebra

CPU	GEMM		GEMV	
	RW	PC	RW	PC
Lift [6]	fails	3.04	1.51	1.99
MKL	4.22	0.74	1.05	0.87

GPU	GEMM		GEMV	
	RW	PC	RW	PC
Lift [6]	4.33	1.17	3.52	2.98
cuBLAS	2.91	0.83	1.03	1.00

[6] Steuwer et. al, "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation", **CGO'17**.

Experimental Results

Our better results are because:

Lift
Relies on transformation rules which require hand pruning (infinitely-large) optimization space for exploration.

EKR Java
Not auto-tunable for input size & inefficient memory usage.

We rely on a large optimization space [1] — designed & optimized toward an arbitrary: MDH, architecture, and input/output characteristics — which we explore fully automatically via advanced auto-tuning mechanisms [2] .

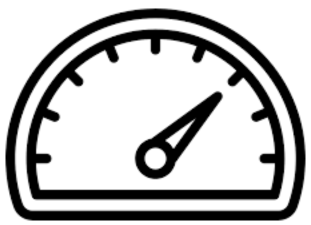
Intel MKL/MKL-DNN & NVIDIA cuBLAS/cuDNN
Optimized toward only average high performance over different input/output characteristics.

Tensor Comprehensions & COGENT
Rely on smaller optimizations spaces and/or no parallelization in summation dimensions.

[1] Rasch, Schulze, Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.", PACT'19
[2] Rasch, Schulze, Steuwer, Gorlatch. "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework ATF", TACO'20

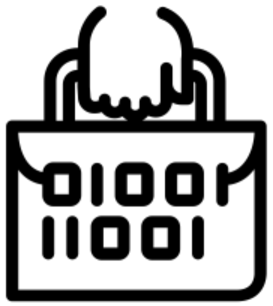
Motivation — MDH in MLIR

The MDH approach aims at combining important advantages over related approaches:



Performance

competitive to best available solutions



Portability

functional and performance — over architectures and input/output characteristics

Productivity

easy to use & expressive

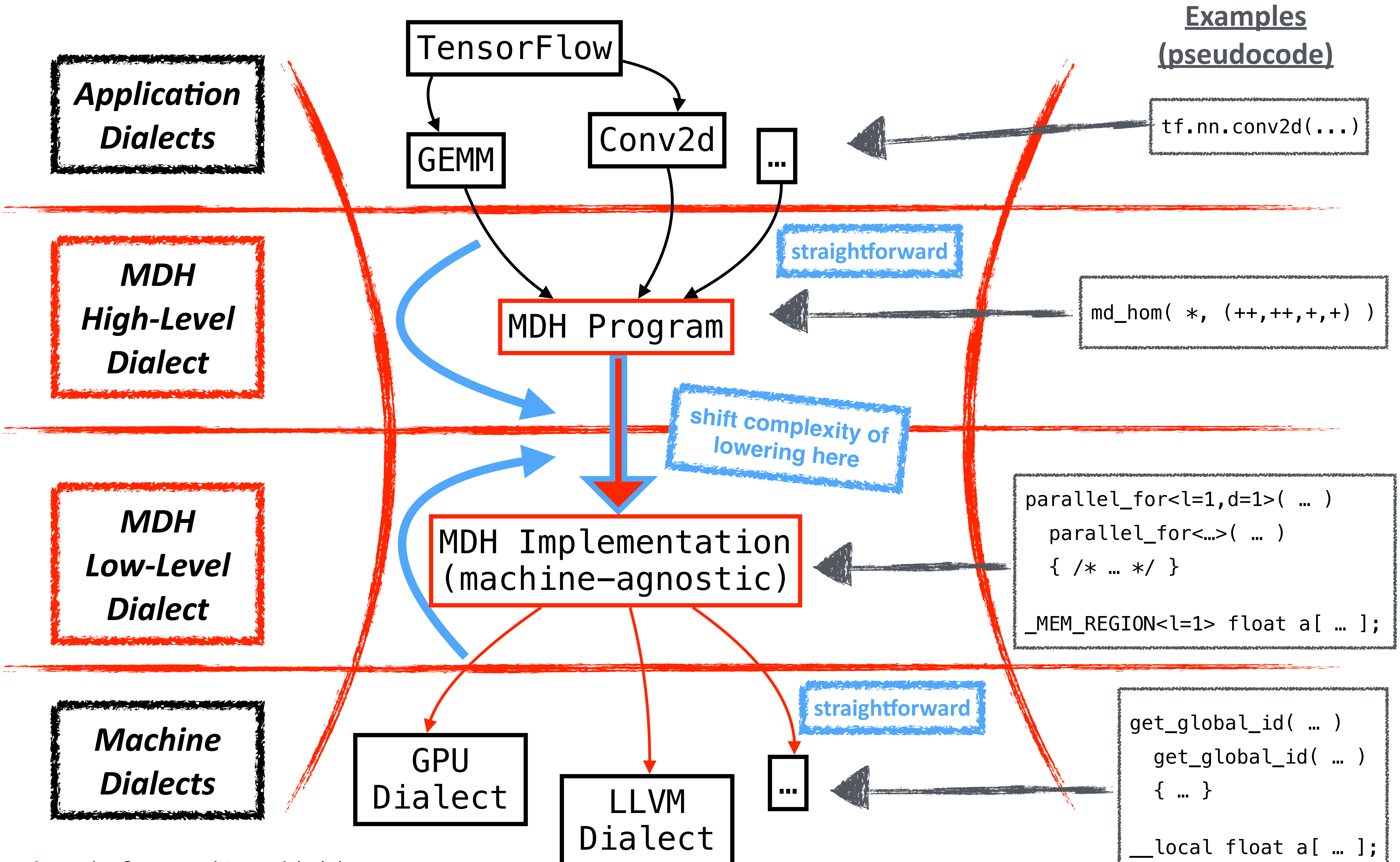
However, our current implementation has weaknesses:

- Prototype implementation — technically inconvenient to use in praxis (e.g., for TensorFlow);
- Systematic code generation for particular models, but not over models;
- Implementation hard to maintain & extend → makes collaboration complicated.

→ Let's make it better in a new MLIR implementation!

MDH in MLIR — Our Vision

Overview:



red: implemented/provided by us
black: implemented/provided by user & community

MDH in MLIR — Our Vision

Dialects:

Level	Requirements	Example (pseudocode)
Applications	- Given by the user (TensorFlow, etc).	`tf.nn.conv2d(...)`
MDH High-Level Dialect	- Agnostic from hardware & optimization details. - Expressive enough to represent various kinds of data-parallel computations. - Should capture — in a structured manner — all high-level information relevant for generating efficient low-level code.	`md_hom( *, (++,++,+,+) )`

MDH Low-Level Dialect	- Optimizations expressible (parallelization, tiling, memory, etc).
	- Uniform for different machine dialects.
		`parallel_for<l=1,d=1>( … )`
		`  parallel_for<…>( … )`
		`  { /* … */ }`
		
		`_MEM_REGION<l=1> float a[ … ];`

Machine Dialects	- Provided by MLIR community (GPU, LLVM, etc).	`get_global_id( … )`
		`  get_global_id( … )`
		`  { … }`
		
		`__local float a[ … ];`

MDH in MLIR — Our Vision

<u>Lowering:</u>

Level	Requirements	Example (pseudocode)

Applications

```
tf.nn.conv2d(...)
```

shall be **<u>very easy</u>** for the programmer due to MDH's high-level abstraction

MDH High-Level Dialect

```
md_hom( *, (++,++,+,+) )
```

all optimizations performed at this step
→ step shall be **<u>uniform</u>** (via auto-tuning) for any target combination of:
1) MDH, 2) device, 3) input/output characteristics

MDH Low-Level Dialect

```
parallel_for<l=1,d=1>( … )
  parallel_for<…>( … )
  { /* … */ }

_MEM_REGION<l=1> float a[ … ];
```

shall be **<u>straightforward</u>** so that new/further machine dialects can be easily targeted with our approach

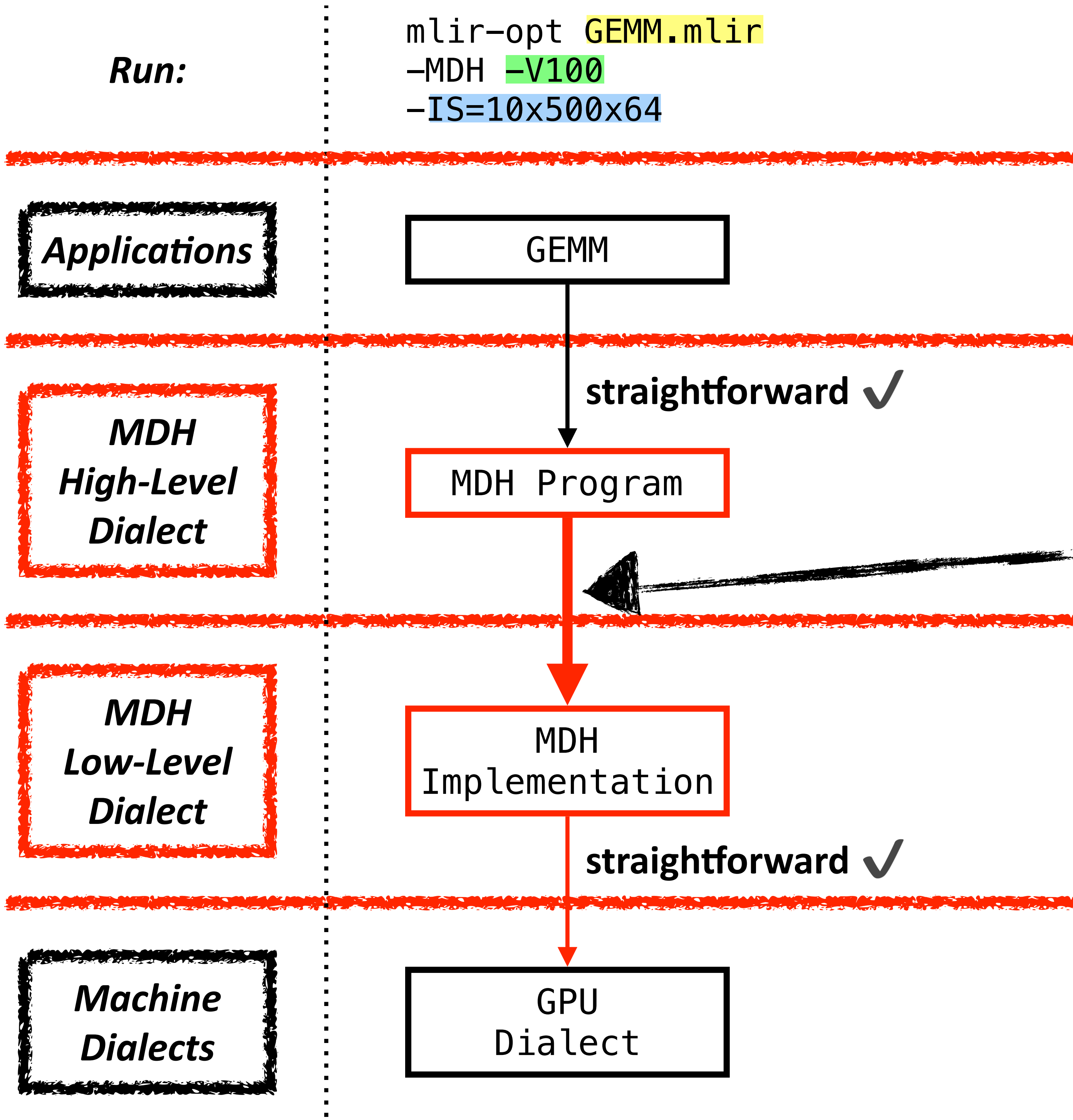
Machine Dialects

```
get_global_id( … )
  get_global_id( … )
  { … }

__local float a[ … ];
```

MDH in MLIR — Our Vision

Workflow:



MDH	DEV	IS	Config
GEMM	V100	10x500x64	*NT=10, TS=5, …*
GEMM	V100	*	*NT=16, TS=8, …*
GEMM	*	*	*NT=128, TS=4, …*
…	…	…	…

Generated & Extended via Auto-Tuning

(→ alternatively: analytical cost model, ML, …)

Agenda

1. MDH – Domain-Specific Language & Examples
2. MDH in MLIR – The MDH High-Level Dialect

High-Level Dialect

3. MDH Code Generation & Optimization Approach
4. MDH in MLIR – The MDH Low-Level Dialect

Low-Level Dialect
(rather briefly)

5. Conclusion

MDH — Domain-Specific Language

<u>The MDH representation (DSL) relies on three higher-order functions (a.k.a. *patterns*):</u>

1. `in_view` → prepares (domain-specific) *input data*
2. `md_hom` → uniformly specifies *computations*
3. `out_view` → prepares (domain-specific) *output data*

<u>Example:</u> MatMul → *`MatMul = out_view( … ) o md_hom( … ) o in_view( … )`*

```
1. in_view( A,B )( i,j,k )( A[i,k], B[k,j] )
```

input matrices

indices for input matrices

data accesses

```
2. md_hom( *, (++,++,+) )
```

multiply elements in `A` and `B`

concatenate in dimensions `i` & `j`

sum in dimension `k`

```
3. out_view( C )( i,j )( C[i,j] )
```

output matrix

indices for output matrix

data accesses

→ Close to MLIR's Linalg/Affine Dialects — we compare soon!

MDH — Examples

Popular computations as MDHs:

Linear Algebra [1]

```
GEMM = md_hom( *, (++, ++, +) ) o in_view( A,B )( i,j,k )( A[i,k], B[k,j] )
GEMV = md_hom( *, (++,     +) ) o in_view( A,B )( i,  k )( A[i,k], B[k]   )
DOT  = md_hom( *, (        +) ) o in_view( A,B )(     k )( A[k]  , B[k]   )
```

Stencil Computations [1]

Access neighboring elements within their input buffer

```
Conv_2D   = md_hom( *     , (++,++,+,+) ) o in_view(…)
Jacobi_3D = md_hom( J_func, (++,++,++ ) ) o in_view(…)
```

Data Mining [2]

```
PRL = md_hom( weight, (++, ⊗max) ) o in_view(…)
```

Has user-defined combine operator that operates on user-defined data type

Machine Learning [1]

Often very high dimensional (e.g., 7 dims)

```
TC = md_hom( *, (++,…,++ , +,…,+) ) o in_view(…)
```

Further examples: MLP, SVM, ECC, …, Mandelbrot, Parallel Reduction, …

[1] Rasch, Schulze, Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.", PACT'19
[2] Rasch, Schulze, et al. "High-Performance Probabilistic Record Linkage via Multi-Dimensional Homomorphisms", SAC'19

MDH — Examples

“Machine Learning Systems are Stuck in a Rut” [HotOS’19]:

$$\forall n, x, y, c_o : V^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} P^{n,c_i}_{sx+k_x, sy+k_y} \cdot W^{c_i,c_o}_{k_x,k_y}$$

MatMul

Capsule Networks

```
conv2d-CapNT( … ) =

 in_view( P,W )( n,x,y,c0 , kx,ky,ci, i,j,k )( P[ n,ci , s*x+kx,
                                          s*y+ky, i,k ], W[ci,c0 , kx,ky, k,j] ) o

                               md_hom( *, (++,++,++,++ , +,+,+, ++,++,+ ) ) o


out_view( V )( n,x,y,c0, i,j )( V[ n,c0,x,y, i,j ] )
```

Existing MLIR Dialects vs. MDH

Questions:

Linalg vs. MDH	**Affine vs. MDH**
Is *Linalg stronger than MDH*? (**MDH ≤ Linalg**)	Is *Affine* stronger than *MDH*? (**Affine ≤ MDH**)
Is *MDH* stronger than *Linalg*? (**Linalg ≤ MDH**)	Is MDH stronger than *Affine*? (**MDH ≤ Affine**)

("stronger" in terms of "information content" → definition on next slide)

Existing MLIR Dialects vs. MDH

For two representation R_1 and R_2, we say representation R_2 is:

- <u>*stronger than*</u> *R_1* ($R_1 \leq R_2$) iff there exist transformations $\rightarrow_1 : R_1 \rightarrow R_2$ and $\rightarrow_2 : R_2 \rightarrow R_1$ such that for all $r_1 \in R_1$, it holds: $r_1 \rightarrow_1 r_2 \rightarrow_2 r_1' \Rightarrow r_1 = r_1'$;

- <u>strictly *stronger than*</u> *R_1* ($R_1 < R_2$) iff: $R_1 \leq R_2$ and $R_2 \nleq R_1$.

<u>Example:</u> `C++ < OpenMP`

1. <u>`C++ ≤ OpenMP:`</u> `C++` $\underset{\textit{identity}}{\rightarrow_1}$ `OpenMP` $\underset{\textit{removes pragmas}}{\rightarrow_2}$ `C++`

2. <u>`OpenMP ≰ C++:`</u> `OpenMP` $\underset{\textit{removes pragmas}}{\rightarrow_1}$ `C++` $\underset{\textit{pragmas unrecoverable} \triangle}{\boxed{\not\rightarrow_2}}$ `OpenMP`

Is Linalg Stronger than MDH (MDH ≤ Linalg) ?

No, Linalg is **not stronger** than MDH (*please correct us if we are wrong*):

MDH $\rightarrow_1$ Linalg $\nrightarrow_2$ MDH

Linalg does not capture all information required to recover the original MDH program ⚠

Example: MatMul

MDH $\rightarrow_1$ Linalg:

```
1. in_view( A,B )( m,n,k )( A[m,k], B[k,n] )
2. md_hom( *, (++,++,+) )
3. out_view( C )( m,n )( C[ m,n ] )
```

MatMul in MDH

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  iterator_types = ["parallel",
"parallel", "reduction"]
}

linalg.generic #matmul_trait
  ins(%A, %B : …) outs(%C : …){
  ^bb0(…) :
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    linalg.yield %e : f32
}
```

MatMul in Linalg

Is Linalg Stronger than MDH (MDH ≤ Linalg) ?

<u>**No**, Linalg is **not stronger** than MDH (*please correct us if we are wrong*):</u>

$\text{MDH} \rightarrow_1 \text{Linalg} \boxed{\not\leftrightarrow_2} \text{MDH}$

Linalg does not capture all information required to recover the MDH program ⚠

Example:
MatMul

Linalg $\boxed{\not\leftrightarrow_2}$ MDH:

```
#matmul_trait = {
 iterator_types = ["parallel",
"parallel", "reduction"]
}

linalg.generic #matmul_trait
  ins(%A, %B : …) outs(%C : …){
  ^bb0(…) :
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    linalg.yield %e : f32
}
```

MatMul in Linalg

```
1. in_view( A,B )( m,n,k )( A[m,k], B[k,n] )

2. md_hom( *, (++,++,+) )

3. out_view( C )( m,n )( C[ m,n ] )
```

MatMul in MDH

cannot recover combine operator ⚠

```
1. in_view( A,B )( m,n,k )( A[m,k], B[k,n] )

2. md_hom( (a,b,c)↦c+a*b, (++,++,?) )

3. out_view( C )( m,n )( C[ m,n ] )
```

MatMul in MDH (generated from Linalg)

Is Linalg Stronger than MDH (MDH ≤ Linalg) ?

Question: why does Linalg not explicitly capture combine operators?

```
#matmul_accesses = [
  (m, n, k) -> (m, k),
  (m, n, k) -> (k, n),
  (m, n, k) -> (m, n)
]
#matmul_trait = {
  doc = "C(m, n) += A(m, k) * B(k, n)",
  indexing_maps = #matmul_accesses,
  library_call = "linalg_matmul",
  iterator_types = ["parallel", "parallel", "reduction"]
}

linalg.generic #matmul_trait
  ins(%A, %B : memref<?x?xf32, stride_specification>,
               memref<?x?xf32, stride_specification>)
  outs(%C : memref<?x?xf32, stride_specification>)
  {other-optional-attributes} {
  ^bb0(%a: f32, %b: f32, %c: f32) :
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    linalg.yield %e : f32
}
```

MatMul in Linalg

```
1. in_view( A, B )( m,n,k )( A[m,k], B[k,n] )
2. md_hom( *, (++,++,+) )
3. out_view( C )( m,n )( C[ m,n ] )
```

MatMul in MDH

When generated from Linalg: have to use "?" (a.k.a. *"unknown combine operator"*) instead of "+" [1]:

md_hom((a,b,c)↦c+a*b, (++,++,?))

[1] Rasch, Schulze, Gorlatch. md_poly: A Performance-Portable Polyhedral Compiler Based on Multi-Dimensional Homomorphisms. IMPACT'20 (WIP)

Why not here?

(better: parallelization, expressiveness, …)

(→ questions summarized at the end of talk)

Is Linalg Stronger than MDH (MDH ≤ Linalg) ?

"Modular Divide-and-Conquer Parallelization of Nested Loops " [PLDI'19]:

```
#parallel for reduce ⊕
for( … ) {

  #parallel for reduce +vec
  for( … ) {
    Vinner +vec= /* … */;
  }

  Vouter ⊕= Vinner;
}
```

Maximum Bottom Box Sum (MBBS)

MBBS = md_hom(id, ($\oplus, +_{\texttt{vec}}$))

$$(a_1,\ldots,a_n) \quad \oplus \quad (b_1,\ldots,b_n) \;:=\; (\; a_1,\ldots,a_n,a_n+b_1,\ldots,a_n+b_n \;)$$

$$(a_1,\ldots,a_m) \quad + \quad (b_1,\ldots,b_m) \;:=\; (\; a_1+b_1,\ldots,a_m+b_m \;)$$

MBBS – MDH Implementation

→ can MBBS be efficiently implemented in Linalg?

Is Linalg Stronger than MDH (MDH ≤ Linalg) ?

"Modular Divide-and-Conquer Parallelization of Nested Loops" [PLDI'19]:

Linalg seems suboptimal for MBBS

```
#parallel for reduce ⊕
for( … ) {

  #parallel for reduce +vec
  for( … ) {
    v_inner +vec= /* … */;
  }

  v_outer ⊕= v_inner;
}
```

Maximum Bottom Box Sum (MBBS)

```
#accesses = [ … ]

#trait = {
 …
 iterator_types = ["reduction"]
}

linalg.generic #trait
  ins( … ) outs( … ) { … } {

   #parallel for reduce +vec
   for( … ) {
     v_inner +vec= /* … */;
   }
}
```

MBBS – MLIR Linalg

Inner for loop of MBBS inherently sequential/ opaque in Linalg?

$$\text{MBBS} = \text{md\_hom}(\ \text{id},\ (\ \oplus, +_{\text{vec}}\ )\ )$$

MBBS – MDH Implementation

Is MDH Stronger than Linalg (Linalg ≤ MDH) ?

Yes, MDH is **stronger** than Linalg (*please correct us if we are wrong*):

Example:
MatMul

```
#matmul_trait = {
 iterator_types = ["parallel",
“parallel", "reduction"]
}

linalg.generic #matmul_trait
  ins(%A, %B : …) outs(%C : …){
  ^bb0(…) :
    %d = mulf %a, %b: f32
    %e = addf %c, %d: f32
    linalg.yield %e : f32
}
```

MatMul in Linalg

```
1. in_view( A,B )( m,n,k )( A[m,k], B[k,n] )

2. md_hom( (a,b,c)↦c+a*b, (++,++,?) )

3. out_view( C )( m,n )( C[ m,n ] )
```

MatMul in MDH (generated from Linalg)

(examples from slides 17/18)

MDH vs. Affine

MDH and Affine seem **equivalent** (*please correct us if we are wrong*):

Equivalent in Representation

```
Conv_2D = … o md_hom( *, (++,++,+,+) ) o …
```

MDH implementation

```
func @conv_2d(%D : memref<100x100xf32>, %K : memref<3x3xf32>) -> (memref<98x98xf32>) {
  %O = alloc memref<98x98xf32>
  affine.parallel (%x, %y) = (0, 0) to (98, 98) {
    %0 = affine.parallel (%kx, %ky) = (0, 0) to (2, 2) reduce ("addf") {
      %1 = affine.load %D[%x + %kx, %y + %ky] : memref<100x100xf32>
      %2 = affine.load %K[%kx, %ky] : memref<3x3xf32>
      %3 = mulf %1, %2 : f32
      affine.yield %3 : f32
    }
    affine.store %0, O[%x, %y] : memref<98x98xf32>
  }
  return %O
}
```

Explicit in Representation

Conv2D

MLIR Affine Dialect

In contrast to Linalg, Affine seems to explicitly capture combine operators.

Existing MLIR Dialects vs. MDH

Summary:

Linalg vs. MDH		**Affine vs. MDH**	
Is *Linalg* *stronger than* *MDH*? (**MDH ≤ Linalg**)	✗	Is *Affine* stronger than *MDH*? (**Affine ≤ MDH**)	✔
Is *MDH* stronger than *Linalg*? (**Linalg ≤ MDH**)	✔	Is MDH stronger than *Affine*? (**MDH ≤ Affine**)	✔

please treat with caution!

MDH in MLIR — The MDH High-Level Dialect

<u>Example:</u> square all elements in a tensor and sum up results

```
func @main() {
  %tnsr = constant dense <[1.000000e+00, 2.000000e+00, 3.141500e+00]>
    : tensor<3xf64>
  %result = "mdh.hom"(%tnsr) {func = @pow2, op = @"+"}
    : (tensor<3xf64>) -> f64
  return
}

func @pow2(%arg0: f64) -> f64 {
  %square = mulf %arg0, %arg0 : f64
  return %square : f64
}

func @"+"(%arg0: f64, %arg1: f64) -> f64 {
  %product = addf %arg0, %arg1 : f64
  return %product : f64
}
```

MDH High-Level Dialect

- Implemented within a student project (thanks to Benedikt Rips & Jan Speer!)
- First steps toward a high-level dialect in MLIR for MDHs
- <u>Currently many(!) restrictions:</u> one combine operator, no input/output views, …

Summary: MDH High-Level Dialect

Dialects:

Level	Requirements	Example (pseudocode)
Applications	…	…
MDH High-Level Dialect	- Agnostic from hardware & optimization details. ✓ - Expressive enough to represent various kinds of data-parallel computations. ✓ - Should capture — in a structured manner — all high-level information relevant for generating efficient low-level code. ✓	`md_hom( *, (++,++,+,+) )`
MDH Low-Level Dialect	…	…
Machine Dialects	…	…

Agenda

1. MDH – Domain-Specific Language & Examples
2. MDH in MLIR – The MDH High-Level Dialect
3. MDH Code Generation & Optimization Approach
4. MDH in MLIR – The MDH Low-Level Dialect
5. Conclusion

High-Level Dialect ✓

Low-Level Dialect *(rather briefly)*

Reminder: MDH Low-Level Dialect

Dialects:

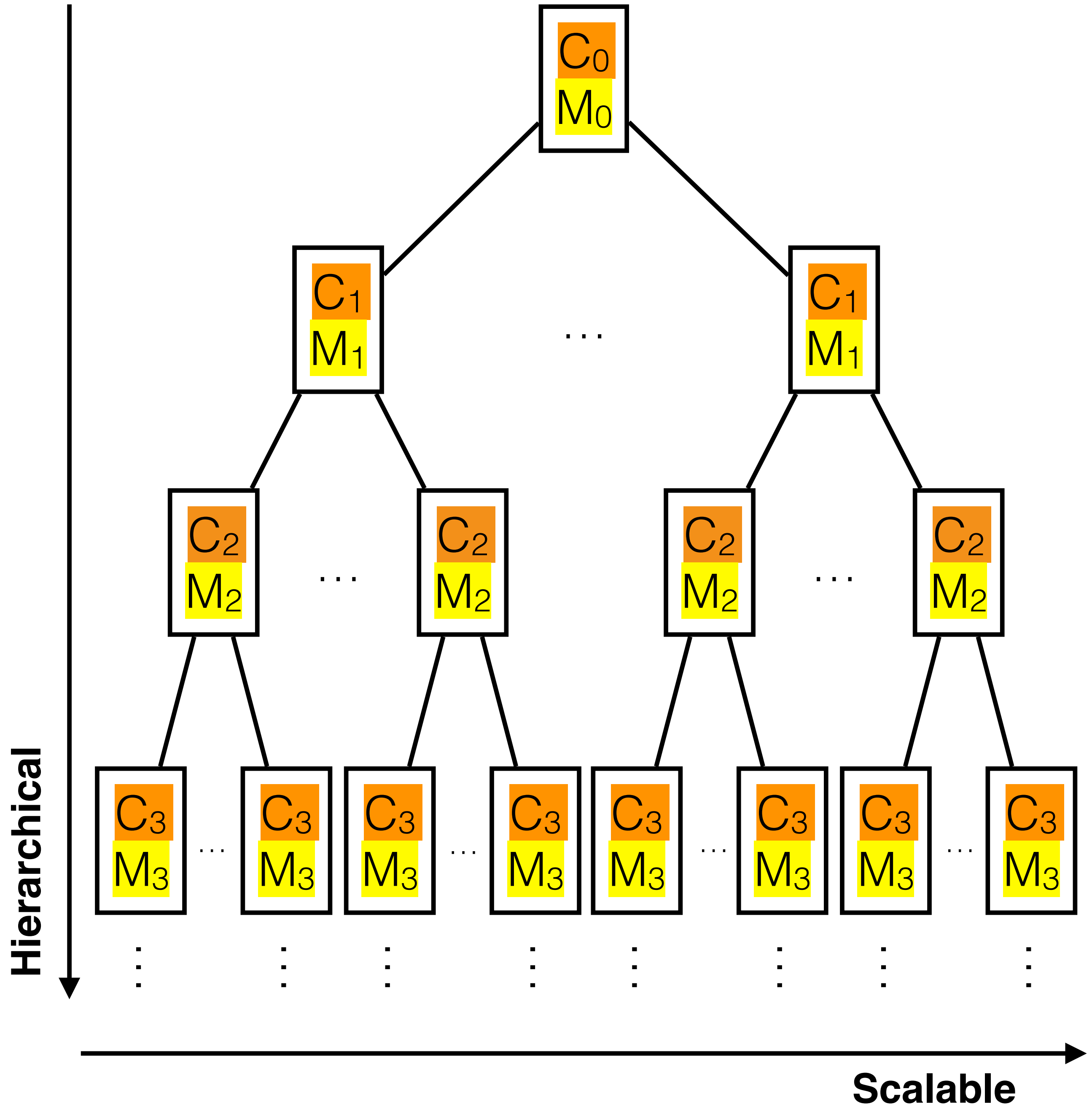
Level	Requirements	Example (pseudocode)
Applications	…	…
MDH High-Level Dialect	…	…

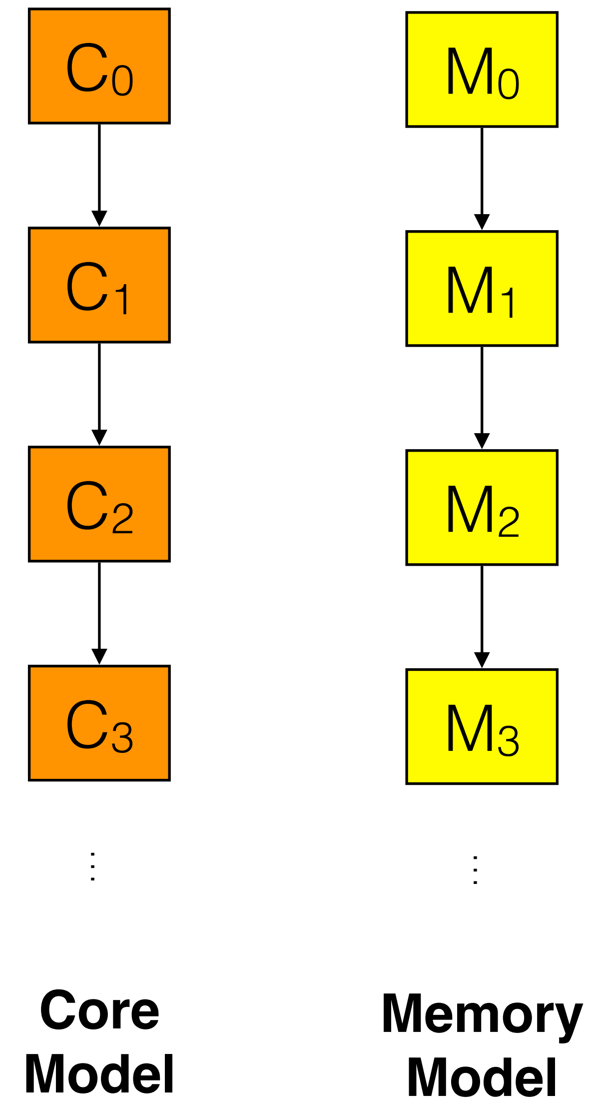
MDH Low-Level Dialect	- Optimizations expressible (parallelization, tiling, memory, etc).
	- Uniform for different machine dialect.
		`parallel_for<l=1,d=1>( … )`
		`  parallel_for<…>( … )`
		`  { /* … */ }`
		
		`_MEM_REGION<l=1> float a[ … ];`

Machine Dialects	…	…

MDH — Target Machine Model

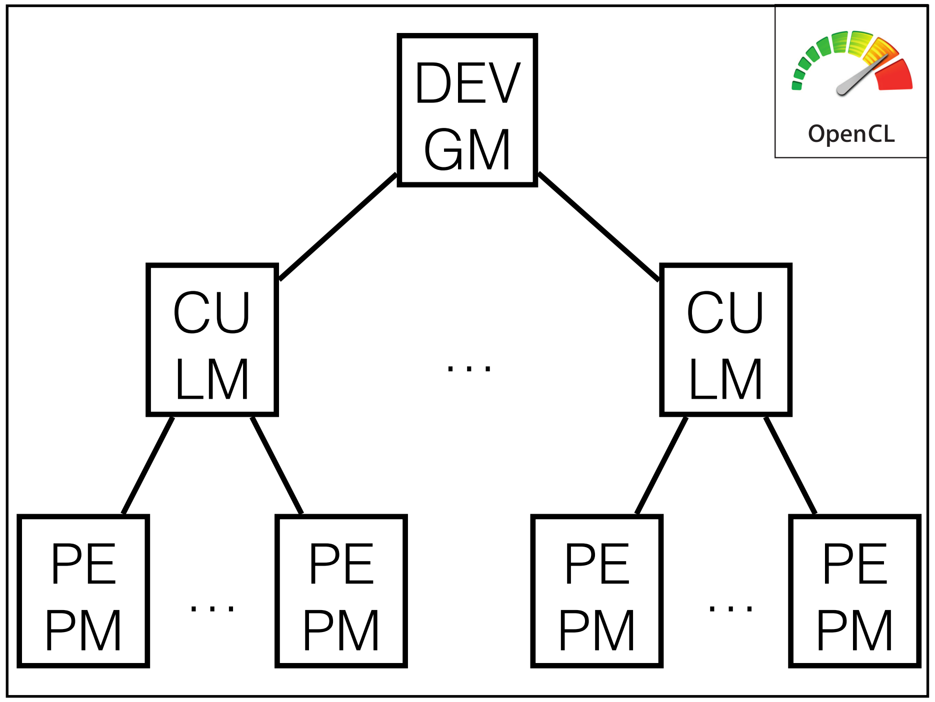
We use a uniform *Abstract Machine Model (AMM)* for MDHs:

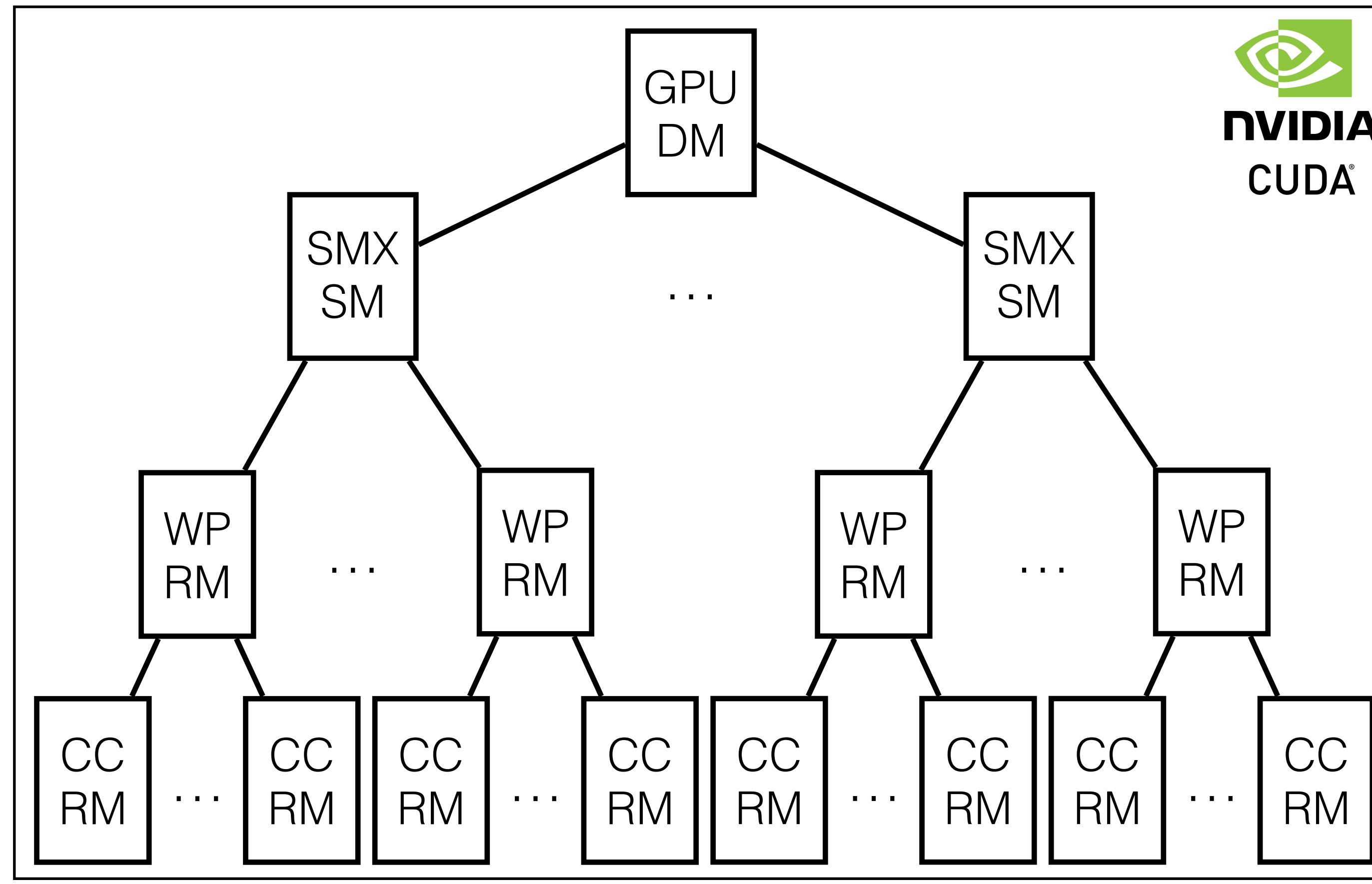


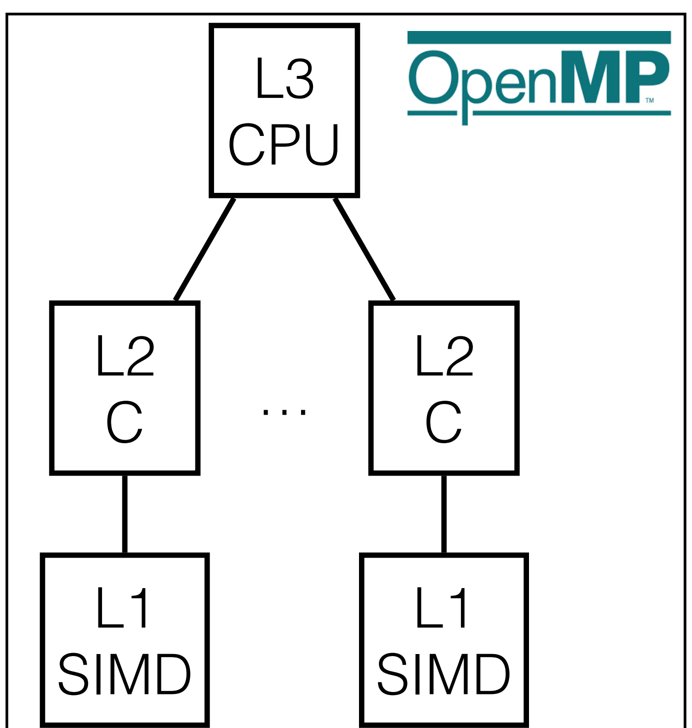


MDH — Target Machine Model

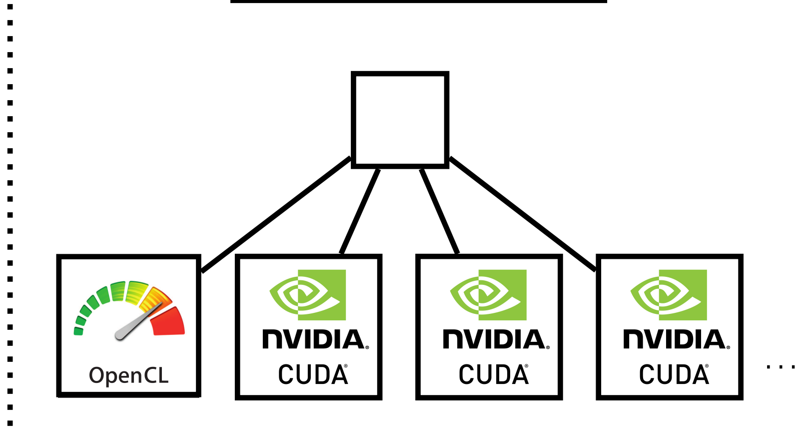
Examples/Instances of our abstract *machine model*:



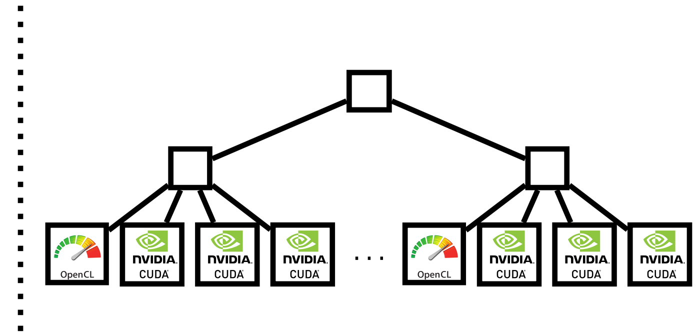




Multi-Device



Multi-Node



. . .

MDH Implementation

<u>We rely on a uniform approach for generating auto-tunable low-level code for MDHs [1]:</u>

No.	Name	Range	Description
1	$\texttt{NUM\_THREADS}^{<l,d>}$	$\{1,\ldots,N_d\}$	number of threads
2	$\texttt{TILE\_SIZE}^{<l,d>}$	$\{1,\ldots,N_d\}$	sizes of tiles
3	$\sigma_{\texttt{mdh-co}}$	$S_{L\times D}$	computation order
4	$\sigma^{<l>}_{\texttt{threads}}$	S_D	thread arrangement
5	$\texttt{MEM\_INP}^{<l,d,inp>}$	$\{1,\ldots,L\}$	memory regions for input
6	$\sigma^{<l,inp>}_{\texttt{inp-buff-do}}$	S_D	input buffer dimension order
7	$\texttt{MEM\_OUT}^{<l,d,out>}$	$\{1,\ldots,L\}$	memory regions for output
8	$\sigma^{<l,out>}_{\texttt{out-buff-do}}$	S_D	output buffer dimension order

Auto-Tunable Parameters

All parameters are chosen as <u>optimized</u> for an <u>arbitrary</u>:

- **MDH**
- **abstract machine model**
- **input/output characteristics**

[1] Rasch, Schulze, Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.", PACT'19

MDH in MLIR — The MDH Low-Level Dialect

Example: Matrix Multiplication — for 3-layered machine model (e.g. OpenCL)

```
// 1. Core/Memory Layer
 parallel_for( p<1,0> = 1,…,32 )
   parallel_for( p<1,1> = 1,…,32 )
     parallel_for( p<1,2> = 1,…,32 )

 for<1>( t<1,0> = 1,…,8 )
   for<1>( t<1,1> = 1,…,8 )
     for<1>( t<1,2> = 1,…,8 )
       copy: A<0> -> A<1>
       copy: B<0> -> B<1>

       __MEM_REGION_0 RES_1[ … ][ … ];
```

```
// 2. Core/Memory Layer
 parallel_for( p<2,0> = 1,…,32 )
   parallel_for( p<2,1> = 1,…,32 )
     parallel_for( p<2,2> = 1,…,32 )

 for<1>( t<2,0> = 1,…,8 )
   for<1>( t<2,1> = 1,…,8 )
     for<1>( t<2,2> = 1,…,8 )
       copy: A<1> -> A<2>
       copy: B<1> -> B<2>

       __MEM_REGION_1 RES_2[ … ][ … ];
```

```
/* ... */
```

pseudocode

Low-Level Dialect

```
MatMul = … o md_hom( *, (++, ++, +) ) o …
```

High-Level Dialect

1. Core/Memory Layer

2. Core/Memory Layer

3. Core/Memory Layer

MDH in MLIR — The MDH Low-Level Dialect

<u>Lowering:</u> *MDH High-Level Dialect → MDH Low-Level Dialect*

```
// 1. Core/Memory Layer
  parallel_for( p<1,0> = 1,…,32 )
    parallel_for( p<1,1> = 1,…,32 )
      parallel_for( p<1,2> = 1,…,32 )

  for<1>( t<1,0> = 1,…,8 )
    for<1>( t<1,1> = 1,…,8 )
      for<1>( t<1,2> = 1,…,8 )
        copy: A<0> -> A<1>
        copy: B<0> -> B<1>

        __MEM_REGION_0 RES_1[ … ][ … ];

// 2. Core/Memory Layer
  parallel_for( p<2,0> = 1,…,32 )
    parallel_for( p<2,1> = 1,…,32 )
      parallel_for( p<2,2> = 1,…,32 )

  for<1>( t<2,0> = 1,…,8 )
    for<1>( t<2,1> = 1,…,8 )
      for<1>( t<2,2> = 1,…,8 )
        copy: A<1> -> A<2>
        copy: B<1> -> B<2>

        __MEM_REGION_1 RES_2[ … ][ … ];

/* … */
```

pseudocode

Low-Level Dialect

```
MatMul = … o md_hom( *, (++, ++, +) ) o …
```

High-Level Dialect

1. Dimension

2. Dimension

3. Dimension

Described in [1]

[1] Rasch, Schulze, Gorlatch. "Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.", PACT'19

MDH in MLIR — The MDH Low-Level Dialect

Lowering: *MDH Low-Level Dialect* → MLIR *Machine Dialects*

```
// 1. Core/Memory Layer
parallel_for( p<1,0> = 1,…,32 )
  parallel_for( p<1,1> = 1,…,32 )
    parallel_for( p<1,2> = 1,…,32 )

for<1>( t<1,0> = 1,…,8 )
  for<1>( t<1,1> = 1,…,8 )
    for<1>( t<1,2> = 1,…,8 )
      copy: A<0> -> A<1>
      copy: B<0> -> B<1>

      __MEM_REGION_0 RES_1[ … ][ … ];
```

pseudocode

```
// 1. Core/Memory Layer
p<1,0> = get_group_id( … );
p<1,1> = get_group_id( … );
p<1,2> = get_group_id( … );

for<1>( t<1,0> = 1,…,8 )
  for<1>( t<1,1> = 1,…,8 )
    for<1>( t<1,2> = 1,…,8 )
      copy_A_glb_to_lcl_(…);
      copy_B_glb_to_lcl_(…);

      __global RES_1[ … ][ … ];
```

pseudocode

Low-Level Dialect

Machine Dialect

straightforward (per design)

Conclusion

1. The **MDH approach** aims at combining the goals of **performance**, **portability**, and **productivity** for **data-parallel computations** targeting **multi- and many-core architectures**;

2. The **MDH approach** often achieves **competitive/higher performance** than well-performing competitors (MKL, cuBLAS, etc);

3. **MLIR** enables **using MDH** in a **structured manner** for **different applications** (e.g., TensorFlow) and **systematically** generating code for **different programming models** (OpenCL, CUDA, OpenMP, etc);

Our Questions:

1. Does Linalg explicitly capture combine operators? If not — why?

2. What is the difference between Linalg and Affine regarding the level of abstraction from your point of view?

Your Questions?