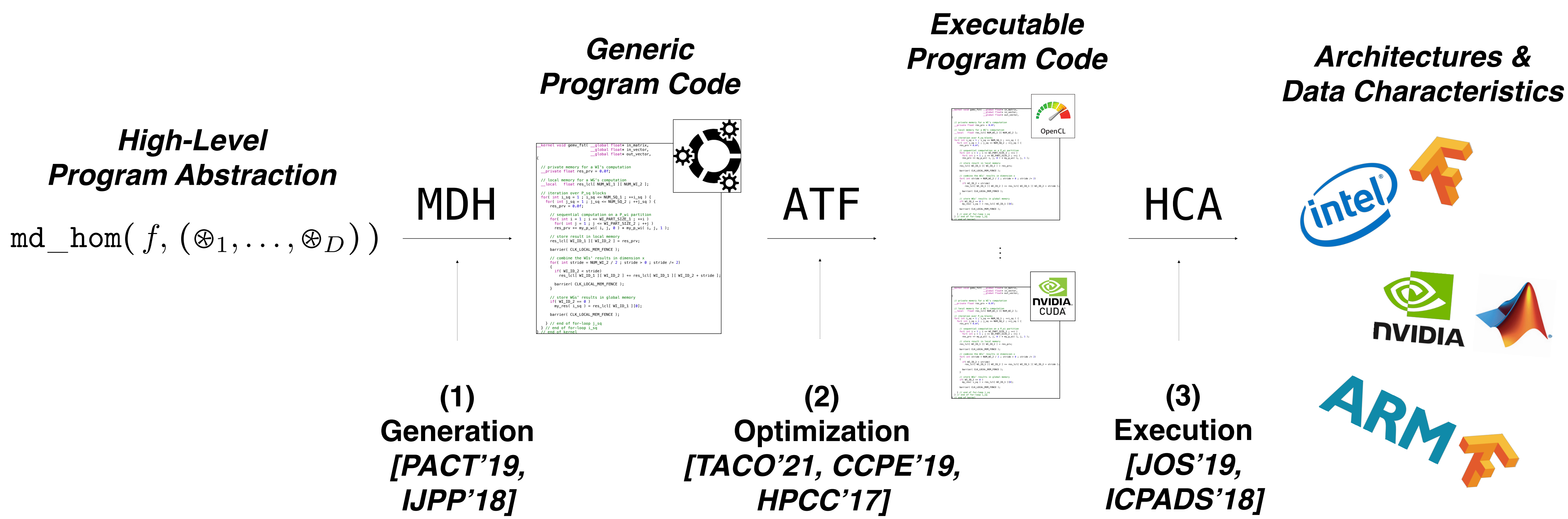
(De/Re)-Compositions Expressed Systematically via MDH-Based Schedules

Ari Rasch, Richard Schulze, Denys Shabalin,
Anne Elster, Sergei Gorlatch, Mary Hall

The MDH(+ATF+HCA) Approaches



Approaches to code *generation* (`MDH`) & *optimization* (`ATF`) & *execution* (`HCA`):

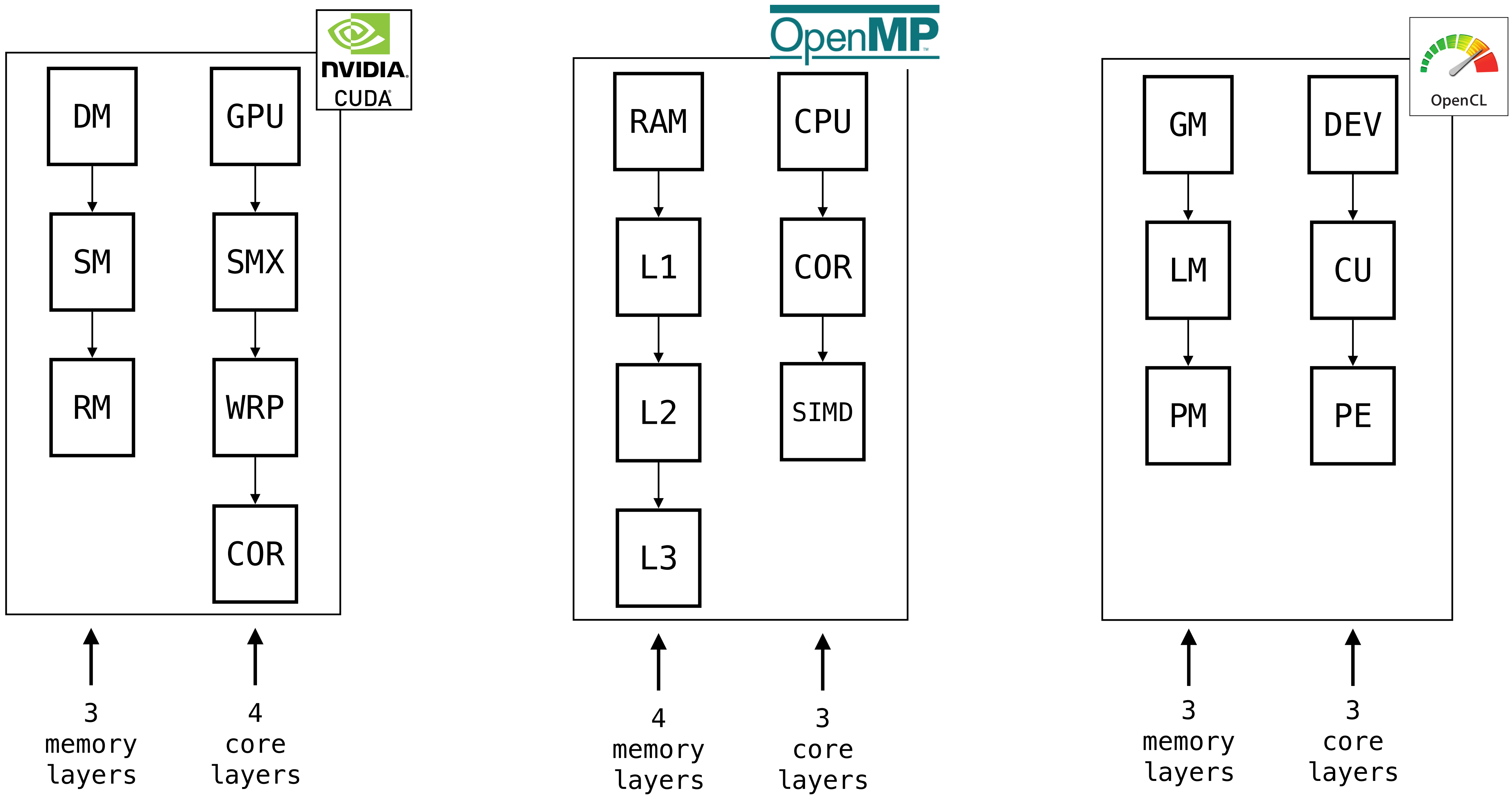
(1) `MDH` (*Multi-Dimensional Homomorphisms*): How to generate automatically optimizable (auto-tunable) code?

(2) `ATF` (*Auto-Tuning Framework*): How to optimize (auto-tune) code?

(3) `HCA` (*Host Code Abstraction*): How to execute code on (distr.) multi-dev. systems?

Observation

<u>State-of-the-art architectures rely on deep *memory* & *core hierarchies*:</u>



Optimizations are required for **both hierarchy kinds**
— ***memory*** and ***core*** —
to achieve the full performance potential of architectures

Observation

- Modern high performance compilers include:
  TVM, Halide, …

- These compilers efficiently target modern architectures, by allowing expert users to explicitly express *code optimizations* in form of so-called *scheduling programs*

- Flaw:

The existing scheduling languages usually rely on a vast set of low-level commands, and the commands have to be combined in complex ways to achieve high performance

We show that this design decision of the existing approaches makes them expressive, but complicates:

1. achieving *high performance*
2. guaranteeing *safety*
3. offering *auto-tuning*
4. *enabling applicability*
5. allowing *visualization*

```
1  # exploiting fast memory resources for "C":
2  matmul_local, = s.cache_write([matmul], "local"
       )
3  matmul_1, matmul_2, matmul_3 = tuple(
       matmul_local.op.axis) + tuple(matmul_local.
       op.reduce_axis)
4  SHR_1, REG_1 = s[matmul_local].split(matmul_1,
       factor=1)
5  # 9 further split commands
6  s[matmul_local].reorder(BLK_1, BLK_2, DEV_1,
       DEV_2, THR_1, THR_2, DEV_3, SHR_3, SHR_1,
       SHR_2, REG_3, REG_1, REG_2)
7
8  # ... (loop unrolling)
9
10 # tiling:
11 matmul_1, matmul_2, matmul_3 = tuple(matmul.op.
       axis) + tuple(matmul.op.reduce_axis)
12 THR_1, SHR_REG_1 = s[matmul].split(matmul_1,
       factor=1)
13 # 5 further split commands
14 s[matmul].reorder(BLK_1, BLK_2, DEV_1, DEV_2,
       THR_1, THR_2, SHR_REG_1, SHR_REG_2)
15 s[matmul_local].compute_at(s[matmul], THR_2)
16
17 # block/thread assignments:
18 BLK_fused = s[matmul].fuse(BLK_1, BLK_2)
19 s[matmul].bind(BLK_fused, te.thread_axis("
       blockIdx.x"))
20 # ... (similar to lines 18 and 19)
21
22 # exploiting fast memory resources for "A":
23 A_shared = s.cache_read(A, "shared", [
       matmul_local])
24 A_shared_ax0, A_shared_ax1 = tuple(A_shared.op.
       axis)
25 A_shared_ax0_ax1_fused = s[A_shared].fuse(
       A_shared_ax0, A_shared_ax1)
26 A_shared_ax0_ax1_fused_o,
       A_shared_ax0_ax1_fused_i = s[A_shared].
       split(A_shared_ax0_ax1_fused, factor=1)
27 s[A_shared].vectorize(A_shared_ax0_ax1_fused_i)
28 # ...
29 s[A_shared].compute_at(s[matmul_local], DEV_3)
30
31 # exploiting fast memory resources for "B":
32 # ... (analogous to lines 23 – 29)
```

Listing 3. TVM+Ansor schedule (shortened for brevity) for Matrix Multiplication as used in ResNet-50 network on NVIDIA Ampere GPU

Contribution of this Work

We introduce a *<u>new scheduling language</u>* for expressing code optimizations in a systematic way

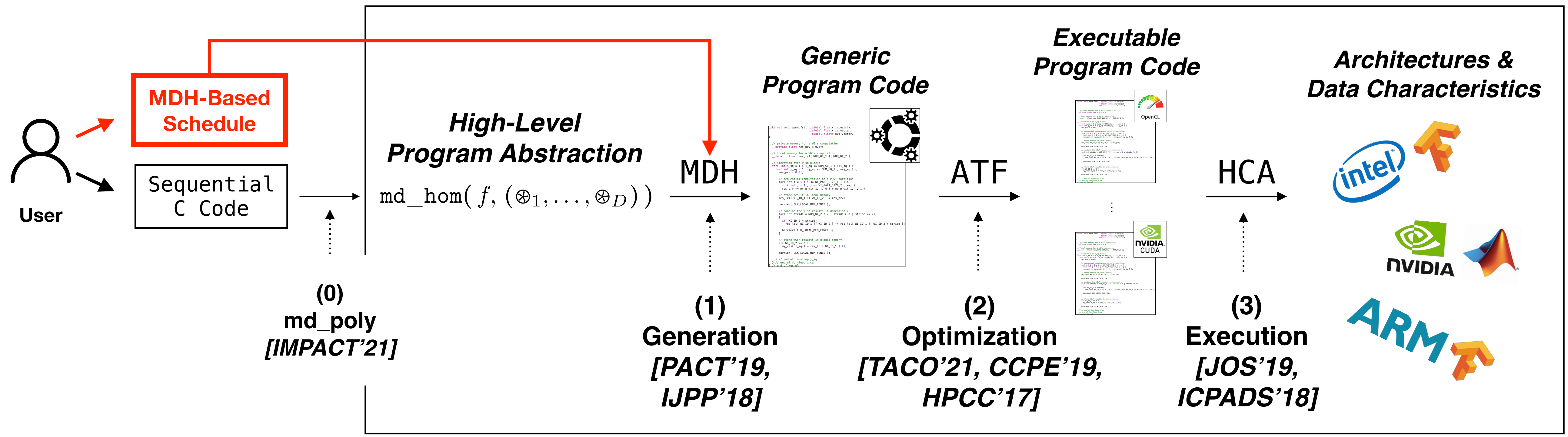
<u>Our systematic language design enables:</u>

1. **<u>Performance:</u>** we can achieve (and often even outperform) the state-of-the-art TVM+Ansor compiler
2. **<u>Safety:</u>** we offer strong error checking, backed by MDH formalism
3. **<u>Auto-Tuning:</u>** any particular optimization decisions can be optionally left for auto-tuning (schedules can be recommended)
4. **<u>Applicability:</u>** our language is used analogously for multiple kinds of programming models (CUDA, OpenMP, OpenCL, …)
5. **<u>Visualization:</u>** our schedules can be visualized and also be generated from visual inputs

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

Overview



<u>In this work:</u>

We extend the existing MDH+ATF+HCA pipeline,

by allowing expert users to explicitly express some/all optimizations

via **MDH-Based Schedules**

<u>Advantages over existing MDH+ATF+HCA:</u>

1. <u>Better Optimization:</u> an auto-tuning system might not always make the same high-quality optimization decisions as an expert user

2. <u>Faster Auto-Tuning:</u> as some (or even all) optimization decisions are made by the expert user and thus are not left to the auto-tuning system

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*

- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*

- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

<u>Initialization (optional):</u>

— the initial iteration space has a size of **16,1000,2048**

— *CUDA's Device Memory* (**DM**) is used for A & B input matrices and C output matrix

— computation is performed by a **GPU**

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*
- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

<u>Block Parallelization:</u>

– iteration space is split into tiles of size **8,20,2048**

– no memory optimizations

– each tile is computed by a *CUDA Block* (**BLK**)

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*
- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

<u>Classical Tiling:</u>

— iteration space is split into tiles of size **4,20,2048**

— no memory optimizations

— no parallelization

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*
- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

<u>Thread Parallelization & Register Memory Utilization:</u>

– iteration space is split into tiles of size **1,1,2048**

– *CUDA Register Memory* (**RM**) is used for computed intermediate results of C output matrix

– each tile is computed by a *CUDA Thread* (**THR**)

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*

- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

<u>Shared Memory Utilization:</u>

– iteration space is split into tiles of size **1,1,256**

– *CUDA Shared Memory* (**SM**) is used for A & B input matrices

– no parallelization

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*
- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

Classical Tiling:

```
– iteration space is split in tiles of size
  1,1,2

– no memory optimizations

– no parallelization
```

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

MDH-Based Schedules

- We allow expert users to express optimizations via *MDH-Based Schedules*
- Our language consists of exactly one primitive which has the following basic structure:

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

- We illustrate our primitive using the example of Matrix Multiplication on `16x2048` & `2048x1000` matrices (taken from ResNet-50):

Classical Tiling:

```
– iteration space is split in tiles of size
  1,1,1

– no memory optimizations

– no parallelization
```

```
// initialization
0: (de/re)-comp( 16,1000,2048 )
               ( A:DM[1,2],B:DM[1,2] ;
                 C:DM[1,2]           )
               ( GPU.y,GPU.x,GPU.z )

// parallelization over CUDA Blocks
1: (de/re)-comp( 8,20,^ )
               ( ^,^ ; ^ )
               ( BLK.y,BLK.x,BLK.z )

// tiling 1
6: (de/re)-comp( 4,^,^ )
               ( ^,^ ; ^ )
               ( FOR.1,FOR.2,FOR.3 )

// parallelization over CUDA Threads &
// utilization of CUDA Register Memory
2: (de/re)-comp( 1,1,^ )
               ( ^,^ ; C:RM[1,2] )
               ( THR.y,THR.x,THR.z )

// utilization of CUDA Shared Memory
3: (de/re)-comp( ^,^,256 )
               ( A:SM[1,2],B:SM[1,2] ; ^ )
               ( FOR.2,FOR.3,FOR.1 )

// tiling 2
4: (de/re)-comp( ^,^,2 )
               ( ^,^ ; ^ )
               ( ^,^,^ )

// tiling 3
5: (de/re)-comp( ^,^,1 )
               ( ^,^ ; ^ )
               ( ^,^,^ )
```

Listing 5. MDH-based schedule for optimizing matrix multiplication on NVIDIA A100 GPU according to the optimization decisions of TVM+Ansor in Listing 3

Language Features

We show that our systematic language design enables:

1. **Performance**
2. **Safety**
3. **Auto-Tuning**
4. **Applicability**
5. **Visualization**

Language Features

1. **Performance:** "*Deep Learning*" case study (TVM's favorable application class!)

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	1.00	1.42	1.00	1.14	1.00	1.00

Deep Learning	NVIDIA Volta 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	1.21	1.00	1.79	1.00	1.11	1.06	1.00	1.00	1.00

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

Deep Learning	Intel Broadwell 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	1.60	1.29	1.53	1.32	1.00	1.27	1.02	2.42	1.92

Speedup (higher is better) of our approach over TVM+Ansor

- TVM [OSDI'18] is a state-of-the-art compiler based on schedules
- We use for TVM the schedules generated by `Ansor` [OSDI'20]
- We report performance for approach using the schedules automatically recommended by our system
- Better performance of our approach is because our language:
  - i) supports *more optimization* (e.g., data layout changes);
  - ii) has *more potential for auto-tuning* (discussed on next slides)

We achieve competitive and often higher performance than `TVM+Ansor`

Language Features

2. **<u>Safety:</u>** We **formally guarantee correctness** of our scheduling programs, by checking the formal constraints defined by the MDH formalism

CUDA

- Tile Size on lower layer <= Tile Size on upper layer
- `BLKs` combine in `{DM}`
- `WRPs` combine in `{SM,DM}`
- `THRs` combine in `{RM,SM,DM}`
- Number of `THRs` limited
- <u>as well as:</u>
  - `BLK/THR.{x,y,z}` can be used only once
  - `(de/re)-comp` order must be permutation
  - …

OpenCL

- Tile Size on lower layer <= Tile Size on upper layer
- `WGs` combine in `{GM}`
- `WIs` combine in `{SM,GM}`
- Number of `WIs` limited
- <u>as well as:</u>
  - `WG/WI.{1,2,3,…}` can be used only once
  - `(de/re)-comp` order must be permutation
  - …

OpenMP

…

In related approaches, e.g., Fireiron [PACT'20], it is possible to implement schedules from which incorrect low-level code is generated, without issued error messages

Language Features

3. **Auto-Tuning:** our language is designed such that optimizations can be left for auto-tuning (via symbol “?”)

```
(de-)comp( 32,32,32 )
        ( A:SM[1,2] , B:SM[1,2] ;
          C:RM[1,2]             )
        ( BLK.x , BLK.y , BLK.z )
```

No tuning

```
(de-)comp( 32,?,? )
        ( A:SM[1,2] , B:SM[1,2] ;
          C:RM[1,2]             )
        ( BLK.x , BLK.y , BLK.z )
```

tile size tuning

```
(de-)comp( 32,?,? )
        ( A:?[?,?] , B:SM[1,2] ;
          C:?[1,2]             )
        ( BLK.x , BLK.y , BLK.z )
```

tile size & memory tuning

```
(de-)comp( 32,?,? )
        ( A:?[?,?] , B:SM[1,2] ;
          C:?[1,2]             )
        ( ?.? , BLK.y , ?.? )
```

tile size & memory & parallelization tuning

- **Our language is designed such that <u>any(!)</u> optimization decision can be left for auto-tuning.**
- **In contrast, the language design of other approaches (such as TVM) support auto-tuning for <u>some optimizations</u> (e.g, choosing tile size values), <u>but not for others</u>**
  (e.g., binding parallelization to inner/outer tiles, using fast memory regions or not, etc).

Language Features

4. **Applicability:** our language is used similarly for different kinds of programming models

```
(de-)comp( /* sub-problem size */ )
         ( /* memory hierarchy assignments */ )
         ( /* core   hierarchy assignments */ )
```

```
IN x … x IN
```

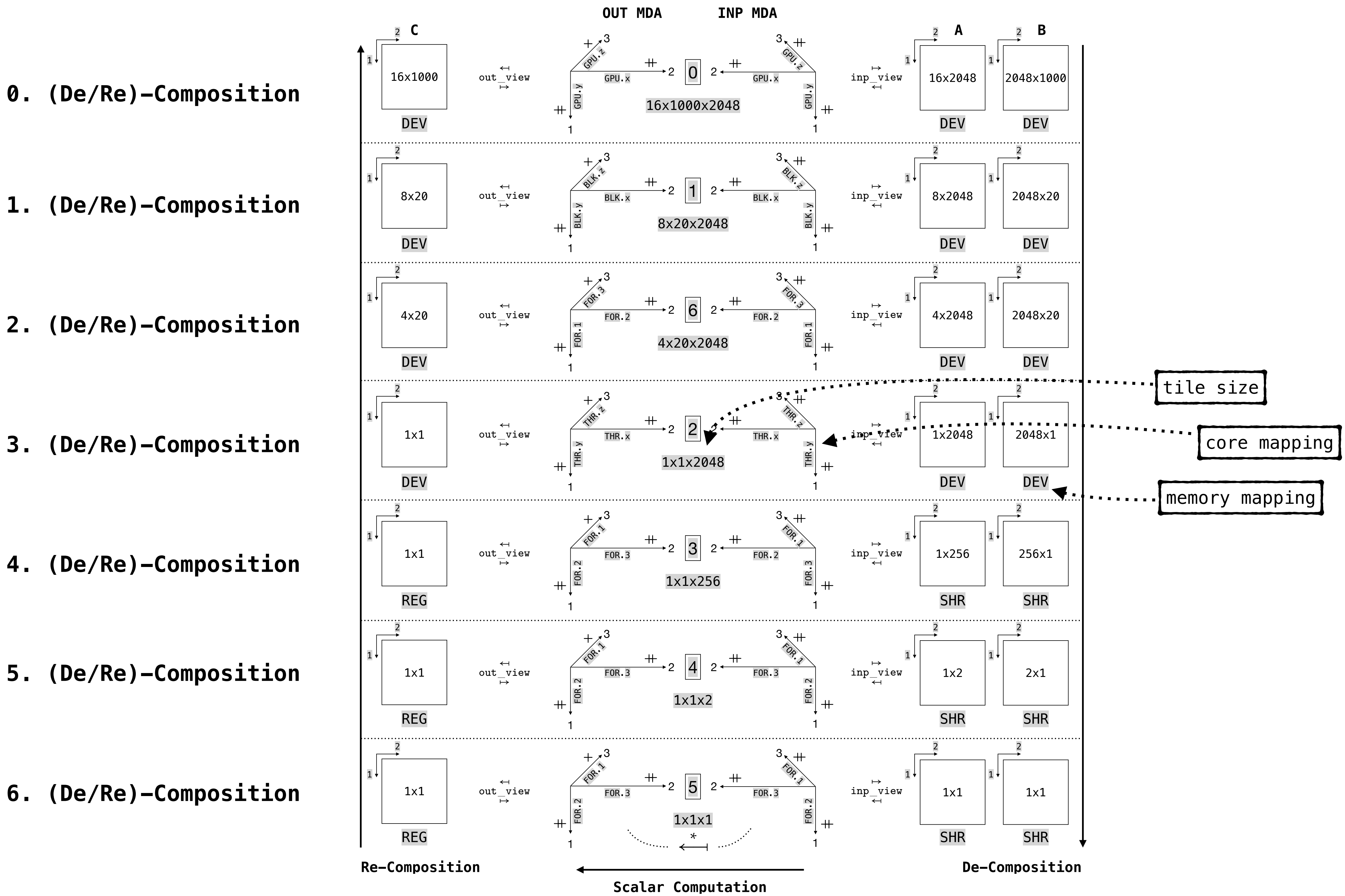
```
CUDA:   {DM,SM,RM}
OpenMP: {MM,L1,L2,L3,…}
OpenCL: {GM,LM,PM}
```

```
CUDA:   {BLK,WRP,THR}.{x,y,z} or FOR.{1,2,3,…}
OpenMP: {THR}.{1,2,3,…}       or FOR.{1,2,3,…}
OpenCL: {WG,WI}.{x,y,z}       or FOR.{1,2,3,…}
```

Our system can be used/extended for C-based programming models targeting arbitrarily deep memory & core hierarchies

Language Features

5. **Visualization:** our schedules can be visualized & also be generated from visual inputs



Related Work

- Popular scheduling approaches include: `TVM [OSDI'18], Halide [PLDI'13], Elevate [ICFP'20], DaCe [SC'19], Tiramisu [CGO'19], CUDA-CHiLL [TACO'13], Fireiron [PACT'20], Distal [PLDI'22],` and `LoopStack [arXiv'22]`

- All these approaches have in common that their scheduling languages rely on fine-grained low-level primitives which are expressive but complex to use, often even for experts

- Our language design allows combining the following advantages over the related work:

  1. **Performance:** competitive to TVM and often higher

  2. **Safety:** backed by MDH formalism

  3. **Auto-Tuning:** any optimization decision can be optionally left for auto-tuning

  4. **Applicability:** our language is used similarly for multiple kinds of programming models

  5. **Visualization:** our schedules can be visualized and also be generated from visual inputs

Conclusion & Future Work

Conclusion:

- We introduce a new scheduling language, based on the formalism of Multi-Dimensional Homomorphisms (MDH)

- The goal of our language design is to express (de/re)-compositions of computations in a systematic, structured way to simplify the complex and error-prone optimization process for performance experts

- Our language design enables: 1) *Performance*, 2) *Safety*, 3) Auto-Tuning, 4) *Applicability*, and 5) *Visualization*

Future Work:

- Computations consisting of multiple loop nests (currently limited to individual nests)

- Targeting domain-specific hardware extensions, e.g., *NVIDIA Tensor Cores*

- Targeting further models, e.g., LLVM to benefit from assembly-level optimizations