Expressing Hierarchical Code Optimizations via MDH-Based Schedules

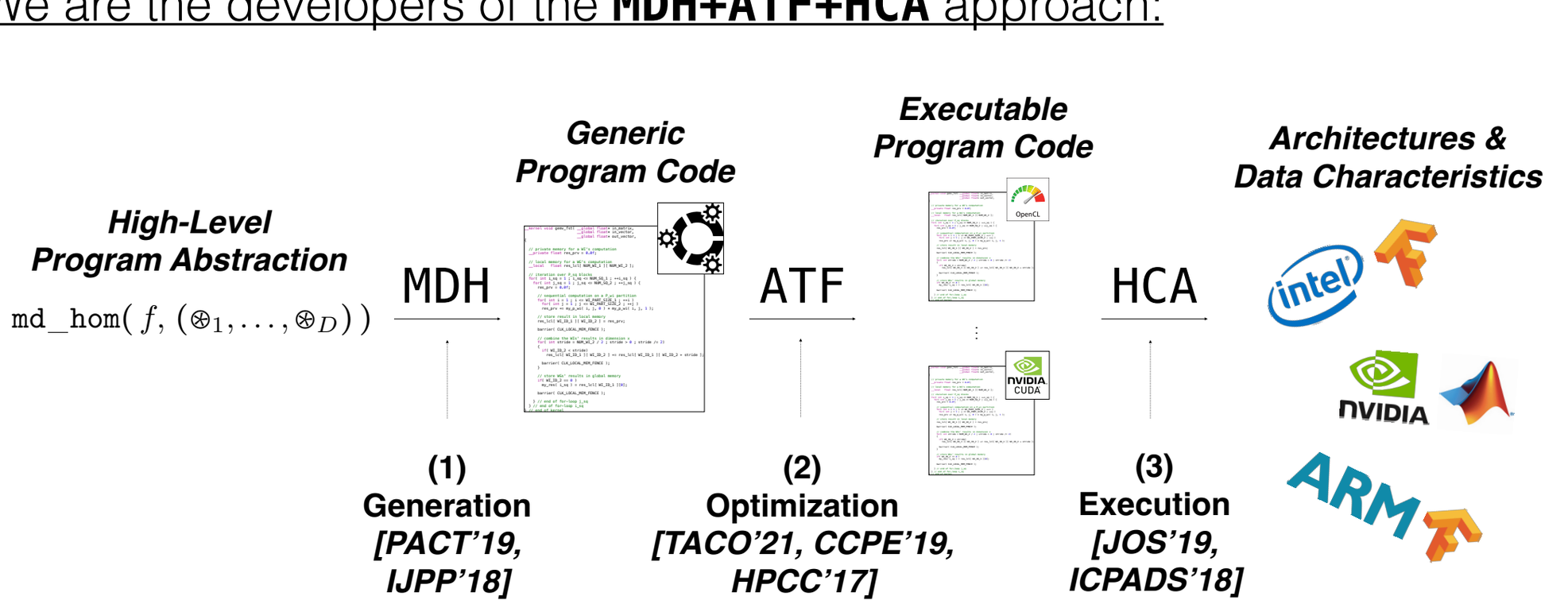
Ari Rasch, Richard Schulze, Sergei Gorlatch

University of Münster, Germany

WIP

Who are we?

We are the developers of the **MDH+ATF+HCA** approach:



Richard Schulze

Ari Rasch

A holistic approach to code *generation* (MDH) & *optimization* (ATF) & *execution* (HCA):

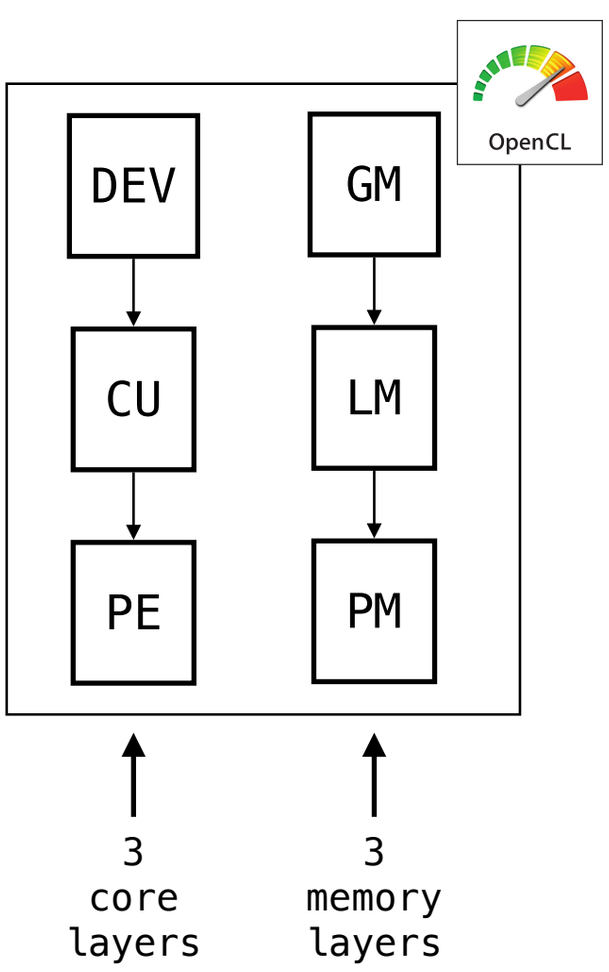
(1) MDH (*Multi-Dimensional Homomorphisms*): How to generate automatically optimizable (auto-tunable) code?

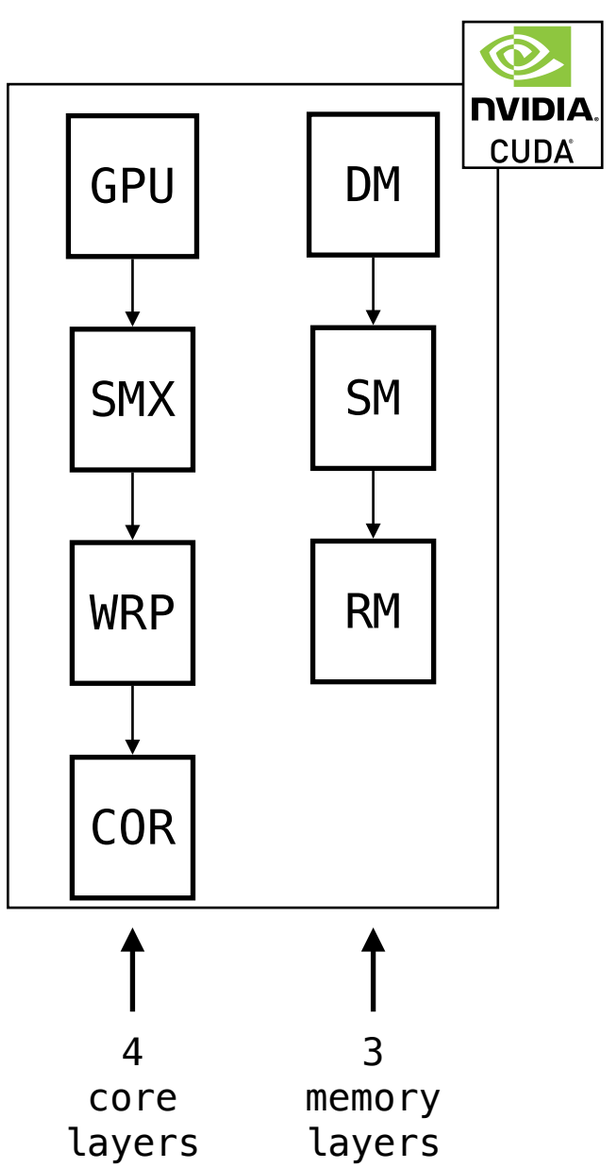
(2) ATF (*Auto-Tuning Framework*): How to optimize (auto-tune) code?

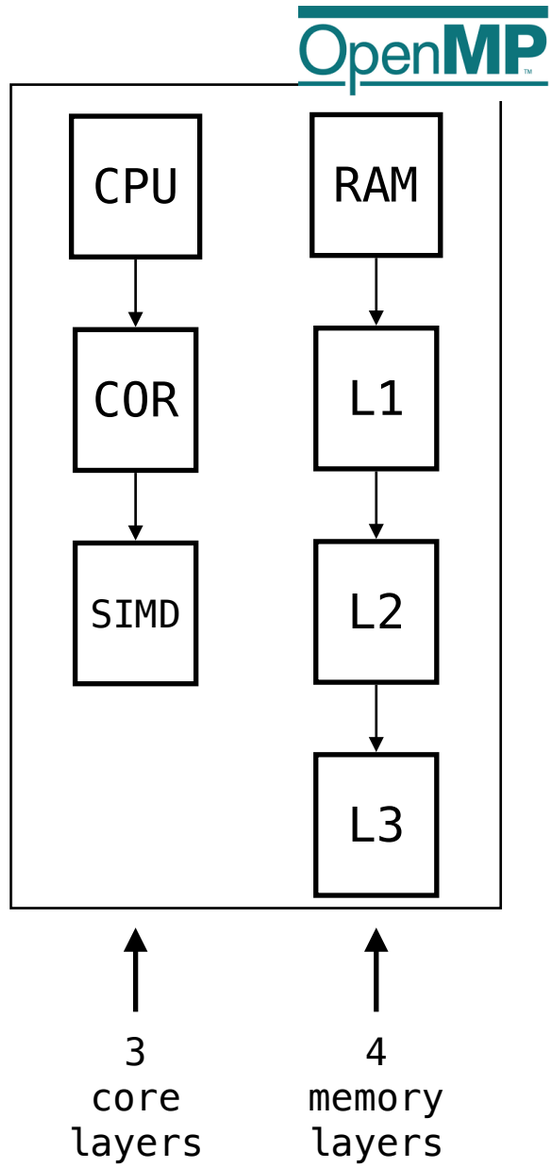
(3) HCA (*Host Code Abstraction*): How to execute code on (distr.) multi-dev. systems?

Observation

State-of-the-art architectures rely on deep *core* & *memory* ***hierarchies***:







Hierarchical *Optimizations* are required
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

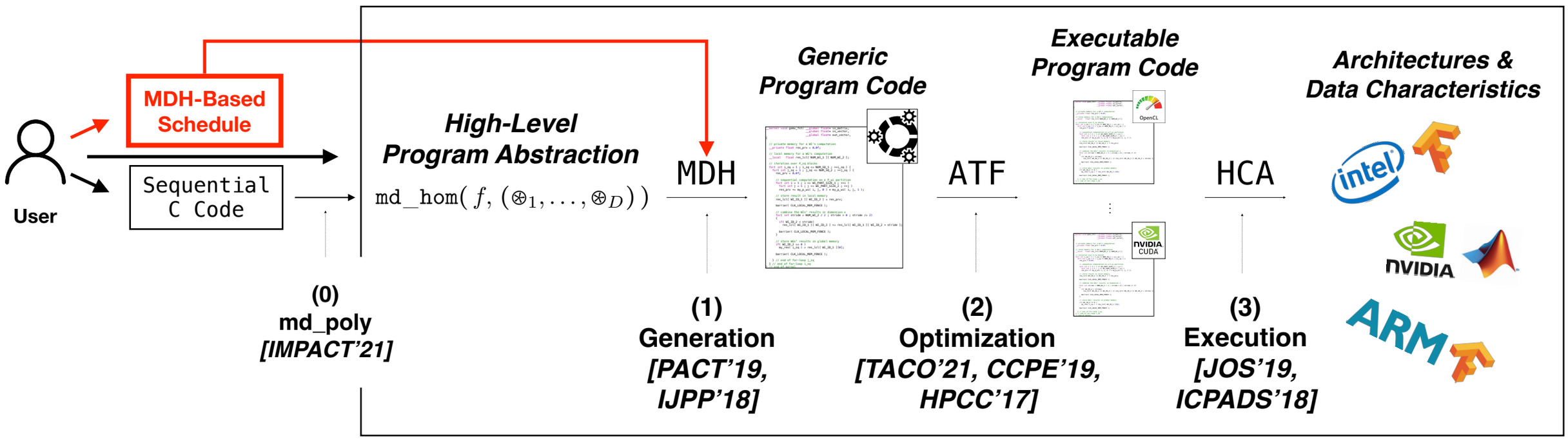
Contribution of this work:

We introduce a new scheduling language for expressing code optimizations in a structured, *hierarchical* way

```
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

Overview



In this work:

We extend the existing MDH pipeline, by allowing expert users to explicitly express some/all optimizations via **MDH-Based Schedules**

Advantages:

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

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

Excursion: The MDH Approach

Example: Matrix Multiplication (MatMul)

```
1  MatMul<Type T | int I,J,K> :=
2        out_view<T>( C:(i,j,k)->(i,j) ) o
3          md_hom<I,J,K>( *, (++,++,+) ) o
4            inp_view<T,T>( A:(i,j,k)->(i,k) ,
5                           B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (MatMul) expressed in the MDH formalism

```
1  #pragma mdh( ++ , ++ , C[i][j]:+ )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

What's happening:

- inp_view: specifies types and accesses to input data
- md_hom: specifies computations
- out_view: specifies types and accesses to output data

Excursion: The MDH Approach

Example: Matrix Multiplication (MatMul)

```
1  MatMul<Type T | int I,J,K> :=
2        out_view<T>( C:(i,j,k)->(i,j) ) o
3          md_hom<I,J,K>( *, (++,++,+) ) o
4            inp_view<T,T>( A:(i,j,k)->(i,k) ,
5                           B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (MatMul) expressed in the MDH formalism

```
1  #pragma mdh( ++ , ++ , C[i][j]:+ )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

What's happening:

- inp_view: specifies types and accesses to input data
- md_hom: specifies computations
- out_view: specifies types and accesses to output data

Excursion: The MDH Approach

Example: Matrix Multiplication (MatMul)

```
1  MatMul<Type T | int I,J,K> :=
2        out_view<T>( C:(i,j,k)->(i,j) ) o
3          md_hom<I,J,K>( *, (++,++,+) ) o
4            inp_view<T,T>( A:(i,j,k)->(i,k) ,
5                           B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (MatMul) expressed in the MDH formalism

```
1  #pragma mdh( ++ , ++ , C[i][j]:+ )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

What's happening:

- inp_view: specifies types and accesses to input data
- md_hom: specifies computations
- out_view: specifies types and accesses to output data

Excursion: The MDH Approach

Example: Matrix Multiplication (MatMul)

```
1  MatMul<Type T | int I,J,K> :=
2        out_view<T>( C:(i,j,k)->(i,j) ) o
3          md_hom<I,J,K>( *, (++,++,+) ) o
4            inp_view<T,T>( A:(i,j,k)->(i,k) ,
5                           B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (MatMul) expressed in the MDH formalism

```
1  #pragma mdh( ++ , ++ , C[i][j]:+ )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

What's happening:

- inp_view: specifies types and accesses to input data
- md_hom: specifies computations
- out_view: specifies types and accesses to output data

Excursion: The MDH Approach

Example: Matrix Multiplication (MatMul)

```
1  MatMul<Type T | int I,J,K> :=
2        out_view<T>( C:(i,j,k)->(i,j) ) o
3          md_hom<I,J,K>( *, (++,++,+) ) o
4            inp_view<T,T>( A:(i,j,k)->(i,k) ,
5                           B:(i,j,k)->(k,j) )
```

Listing 2. Matrix Multiplication (MatMul) expressed in the MDH formalism

```
1  #pragma mdh( ++ , ++ , C[i][j]:+ )
2  for( int i = 0 ; i < I ; ++i )
3    for( int j = 0 ; j < J ; ++j )
4      for( int k = 0 ; k < K ; ++k )
5        C[i][j] += A[i][k] * B[k][j]
```

Listing 1. Matrix multiplication in C (annotated in line 1 with an optional MDH directive enabling advanced optimizations)

We use this representation in this work which is simpler for most users

What's happening:

- inp_view: specifies types and accesses to input data
- md_hom: specifies computations
- out_view: specifies types and accesses to output data

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 discuss our primitive using the example Matrix Multiplication:

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3               ( A:DM[1,2],B:DM[1,2] ;
4                 C:DM[1,2]           )
5               ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9               ( ^,^ ; ^ )
10              ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14              ( ^,^ ; ^ )
15              ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19              ( ^,^ ; ^ )
20              ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24              ( A:SM[1,2],B:SM[1,2] ;
25                C:RM[1,2]           )
26              ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30              ( ^,^ ; ^ )
31              ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35              ( ^,^ ; ^ )
36              ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Initialization (optional):

- the initial iteration space has a size of **16,1000,2048**
- Input/Output matrices **A,B** / **C** are stored in *CUDA's Device Memory* (**DM**)
- computation is performed by a **GPU**

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Block Parallelization:

- iteration space is split in tiles of size **8,20,2048**
- no memory optimizations
- each tile is computed by a *CUDA Block* (**BLK**)

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Classical Tiling:

- iteration space is split in tiles of size **4,20,2048**
- no memory optimizations
- no parallelization

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Thread Parallelization:

- iteration space is split in tiles of size **1,1,2048**
- no memory optimizations
- each tile is computed by a *CUDA Thread* (**THR**)

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Memory Optimization:

- iteration space is split in tiles of size **1,1,256**
- input stored in *CUDA Shared Memory* (SHR); output in *CUDA Register Memory* (**RM**)
- no parallelization

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Classical Tiling:

- iteration space is split in tiles of size **1,1,2**
- no memory optimizations
- no parallelization

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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 discuss our primitive using the example Matrix Multiplication:

Classical Tiling:

- iteration space is split in tiles of size **1,1,1**
- no memory optimizations
- no parallelization

```
1  // initialization
2  0: (de-)comp( 16,1000,2048 )
3                ( A:DM[1,2],B:DM[1,2] ;
4                  C:DM[1,2]           )
5                ( GPU.y,GPU.x,GPU.z )
6
7  // parallelization over CUDA Blocks
8  1: (de-)comp( 8,20,^ )
9                ( ^,^ ; ^ )
10               ( BLK.y,BLK.x,BLK.z )
11
12 // tiling 1
13 6: (de-)comp( 4,^,^ )
14               ( ^,^ ; ^ )
15               ( FOR.1,FOR.2,FOR.3 )
16
17 // parallelization over CUDA Threads
18 2: (de-)comp( 1,1,^ )
19               ( ^,^ ; ^ )
20               ( THR.y,THR.x,THR.z )
21
22 // using CUDA Shared & Register memory
23 3: (de-)comp( ^,^,256 )
24               ( A:SM[1,2],B:SM[1,2] ;
25                 C:RM[1,2]           )
26               ( FOR.2,FOR.3,FOR.1 )
27
28 // tiling 2
29 4: (de-)comp( ^,^,2 )
30               ( ^,^ ; ^ )
31               ( ^,^,^ )
32
33 // tiling 3
34 5: (de-)comp( ^,^,1 )
35               ( ^,^ ; ^ )
36               ( ^,^,^ )
```

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

Further Features:

- Our language can be used analogously also for **other programming models**: OpenMP for CPU, OpenCL for multiple kinds of architectures, …

- We **formally guarantee correctness** of our scheduling programs, by checking the formal constraints defined by the MDH formalism

- Our language is designed such that each optimization decision can alternatively be **auto-tuned** — the user uses symbol "?", rather than a particular optimization value:

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

tile size tuning

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

tile size & parallelization tuning

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

tile size & parallelization & memory region tuning

```
(de-)comp( ?,?,? )
         ( A:?[?,?] , B:?[?,?] ;
           C:?[?,?]            )
         ( ?.? , ?.? , ?.? )
```

tile size & parallelization & memory region+layout tuning

MDH-Based Schedules

Schedule Visualization: our schedules can be visualized & also be generated from visual inputs

0. (De-)Composition

1. (De-)Composition

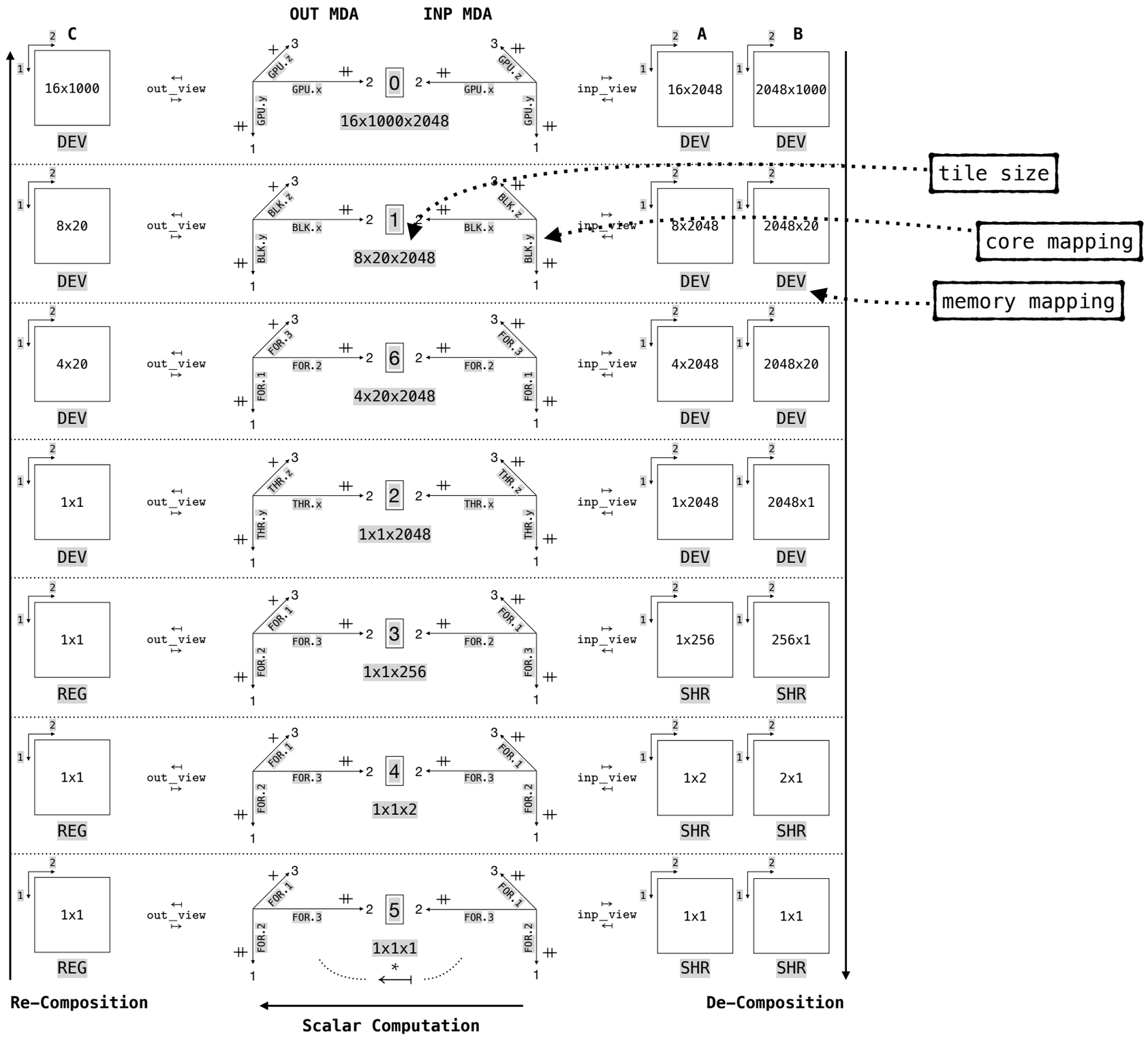
2. (De-)Composition

3. (De-)Composition

4. (De-)Composition

5. (De-)Composition

6. (De-)Composition



Experimental Evaluation

Case Study: "*Deep Learning*" (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
NVIDIA cuDNN	0.92	–	1.85	–	1.22	–	1.94	–	1.81	2.14
NVIDIA cuBLAS	–	1.58	–	2.67	–	0.93	–	1.04	–	–

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
NVIDIA cuDNN	1.21	–	1.29	–	2.80	–	3.50	–	2.32	3.14
NVIDIA cuBLAS	–	1.33	–	1.14	–	1.09	–	1.04	–	–

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
Intel oneDNN	0.39	–	5.07	–	1.22	–	9.01	–	1.05	4.20
Intel oneMKL	–	0.44	–	1.09	–	0.88	–	0.53	–	–
TVM+Ansor (LLVM)	1.20	0.67	0.90	0.26	1.42	0.76	0.66	0.76	0.56	0.36

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
Intel oneDNN	1.30	–	1.81	–	2.94	–	2.85	–	1.83	4.47
Intel oneMKL	–	1.45	–	1.36	–	1.35	–	0.50	–	–
TVM+Ansor (LLVM)	1.45	1.35	1.06	0.72	1.63	0.85	0.98	0.79	1.14	0.52

Figure 3. Speedup (higher is better), of our approach over TVM+Ansor and vendor libraries for time-intensive deep learning computations. Dash symbol indicates unsupported computations. Assembly-optimized approaches (currently beyond the scope of our work) are separated by a straight line and are listed for completeness.

- Speedup (higher is better) of our approach over TVM+Ansor
- Our MDH Schedules are generated automatically by our system, based on auto-tuning
- We achieve the same and often higher performance than TVM+Ansor, for example, because our auto-tuner choses a better parallelization strategy or utilizes fast memory resources more efficiently
- We report speedups/slowdowns also for approaches optimized at the assembly level
- The better performance of assembly approaches is because our approach currently operates at the higher CUDA/ OpenCL abstraction level which offers less optimization opportunities

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

- Moreover, our language is designed such that each particular optimization decisions can alternatively be auto-tuned

- We see the following, further advantages of our approach over the related work:

    - formal correctness guaranteed (via MDH formalism)

    - more expressive: for example, Fireiron works well for data movements but has difficulties with loop-level optimizations, while TVM works well for loop optimizations but not for data movements

    - not restricted to narrow classes of computations and architectures: for example, Fireiron works only for matrix multiplication on NVIDIA GPUs

Conclusion & Future Work

Conclusion:

- We introduce a new scheduling language, based on the approach of Multi-Dimensional Homomorphisms (MDH)
- The goal of our language is to express (de-)compositions in a systematic, structured way to simplify the complex and error-prone optimization process for performance experts
- Correctness of optimizations is formally checked and guaranteed in our approach, backed by the MDH formalism
- We demonstrate that our language can express optimization decisions of the popular TVM compiler for deep learning computations (TVM's favorable application class) and often even outperform TVM on these computations

Future Work:

- Computations consisting of multiple loop nests (currently limited to individual nests)
- Target domain-specific hardware extensions, e.g., *NVIDIA Tensor Cores*
- Target further models, e.g., LLVM to benefit from assembly-level optimizations