Westfälische Wilhelms-Universität Münster

Multi-Dimensional Homomorphisms and Their Implementation in OpenCL

An Algebraic Approach to Performance, Portability, and Productivity for Data-Parallel Applications on Multi- and Many-Core Architectures

Ari Rasch, Richard Schulze, and Sergei Gorlatch

University of Münster, Germany

Motivation

Observation:

Applications

Linear Algebra (BLAS)

GEMM GEMV

DOT

… 152 routines!

Tensor Contractions

…

Stencils

…

X

Architectures

X

Input Sizes

Machine Learning

…

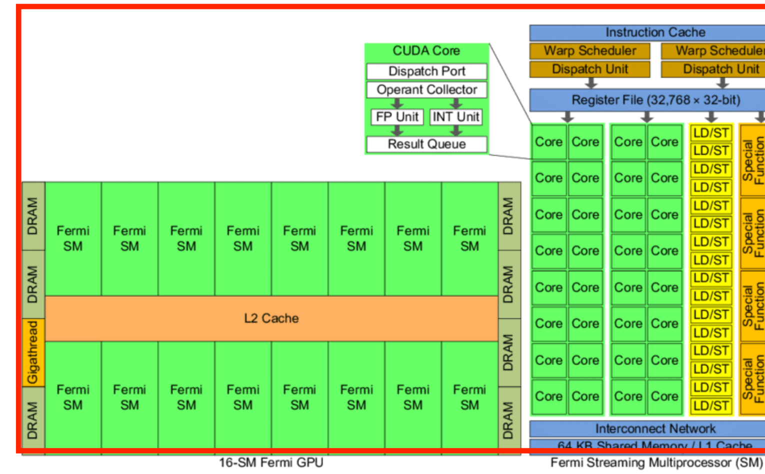
Numerical Computations

…

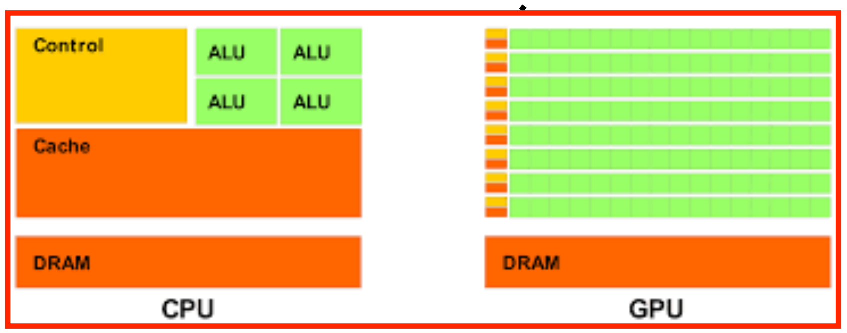
Combinatorial Explosion

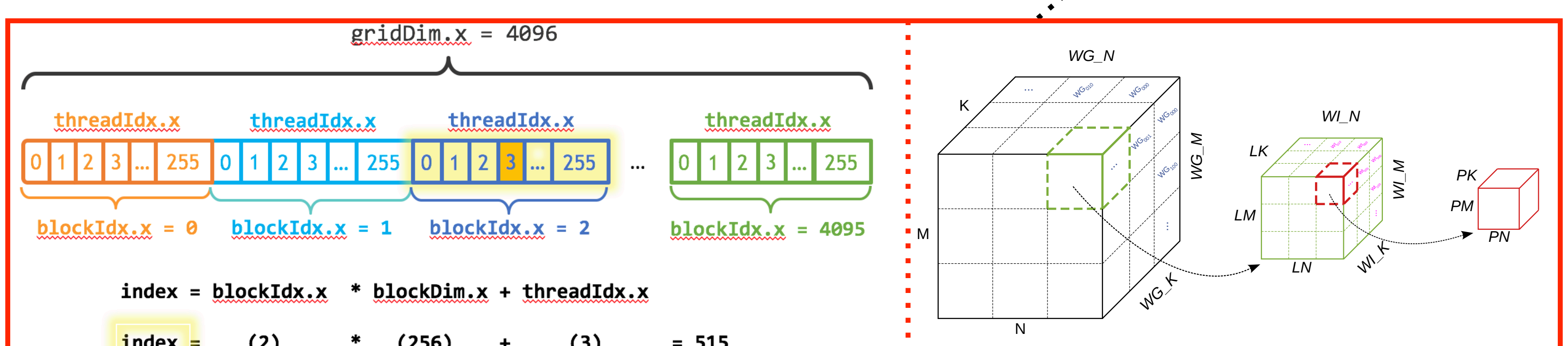
Motivation

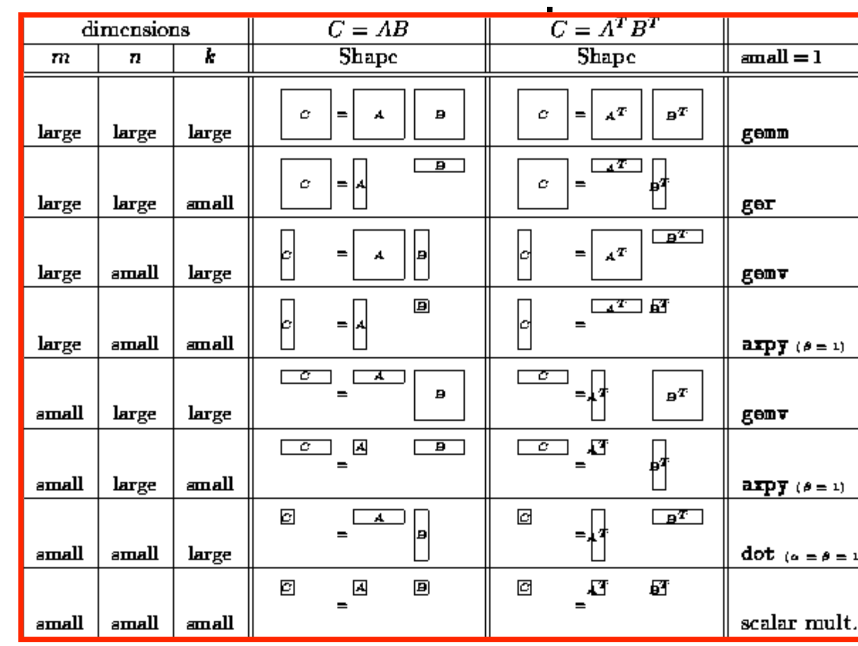
In a perfect world:



Write **one piece of code** for our application that provides **high performance**, is **performance-portable over different architectures** and **input sizes** and that is **easy to implement**.







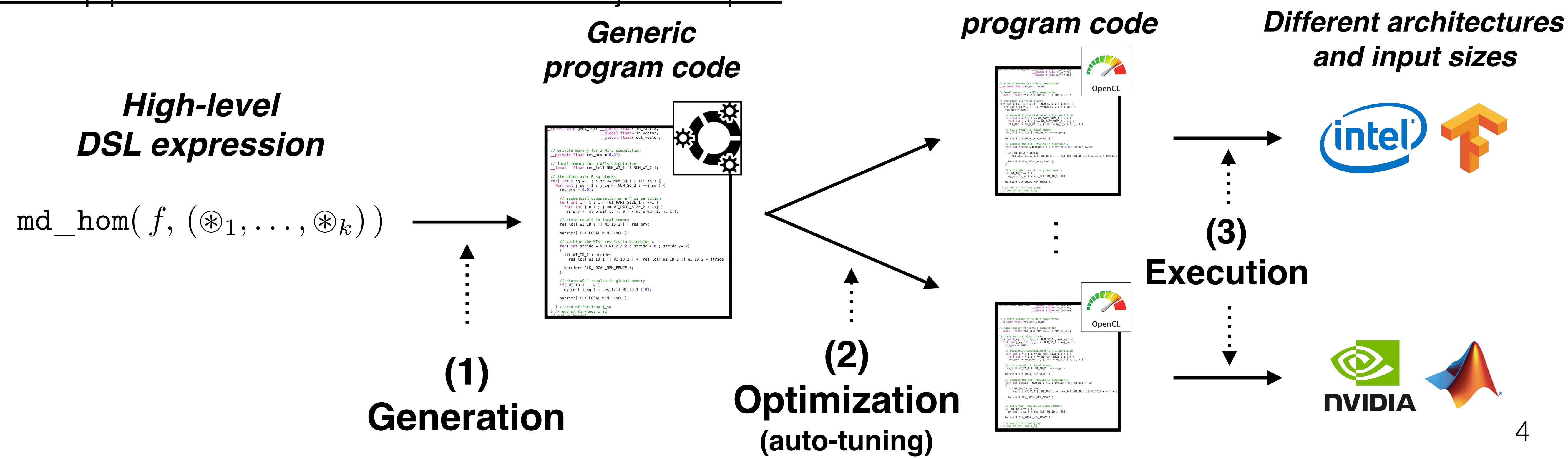
dimensions			$C = AB$	$C = A^T B^T$	
m	n	k	Shape	Shape	small = 1
large	large	large			gemm
large	large	small			ger
large	small	large			gemv
large	small	small			axpy ($\beta = 1$)
small	large	large			gemv
small	large	small			axpy ($\beta = 1$)
small	small	large			dot ($\alpha = \beta = 1$)
small	small	small			scalar mult.

Our Approach

We provide an approach to address all these challenges for our class of *Multi-Dimensional Homomorphisms*:

- **Multi-Dimensional Homomorphisms (MDHs)** are **formally-defined** class of functions that **cover important data-parallel applications**, e.g.: linear algebra (BLAS), stencils computations, …
- We **enable conveniently expressing MDHs** by providing a **high-level DSL** for them.
- We provide a **DSL compiler** to **generate OpenCL code** so that we can target various parallel architectures (e.g., Intel CPU, NVIDIA GPU, …).
- Our OpenCL code is **fully automatically optimizable** (auto-tunable) — for each combination of an **MDH, target architecture**, and **input size** — by being generated as targeted to the **OpenCL's abstract device models** (and not to a particular architecture, e.g., NVIDIA GPU) and as **parametrized in the models' performance-critical parameters.**

Our approach consists of three major steps:



Agenda

We discuss the three major steps of our approach:

1. **Generation**
2. **Optimization**
3. **Execution**

Afterwards:

4. **Experimental Results**
5. **Current/Future Work**

Multi-Dimensional Homomorphisms

Our class of targeted applications is formally specified as:

Definition: [*Multi-Dimensional Homomorphisms [1]*]

Let T and T' be two arbitrary types. A function $h : T[\,N_1\,]\ldots[\,N_d\,] \to T'$ on d-dimensional arrays is called a *Multi-Dimensional Homomorphism (MDH)* iff there exist *combine operators* $\circledast_1, \ldots, \circledast_d : T' \times T' \to T'$, such that for each $k \in [1, d]$ and arbitrary, concatenated input MDA $a +\!\!+_k\, b$:

$$h(\; a +\!\!+_k\, b \;) \;=\; h(a) \;\circledast_k\; h(b)$$

Definition: [`md_hom`]

We write

$$\texttt{md\_hom}(\; f, \, (\circledast_1, \ldots, \circledast_d) \;)$$

for the unique d-dimensional homomorphism with combine operators $\circledast_1, \ldots, \circledast_d$ and action f on singleton arrays.

[1] Rasch, Ari et al. "Multi-Dimensional Homomorphisms and Their Implementation in OpenCL." *International Journal of Parallel Programming*

MDH Examples

Important functions are MDHs — we can express them conveniently using our `md_hom` pattern:

Linear Algebra

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

Whats happening?

1. prepare the domain-specific input for `md_hom` as multi-dimensional array using function `view`
   - here: fuse matrices A and B to 3-dimensional array of pairs — array indices are mapped to the elements of A and B to multiply: `i,j,k ↦ (A[i,k],B[k,j])`
2. apply multiplication (denoted as `*`) to each pair
3. combine results in dimension `k` by addition (`+`)
4. combine results in dimension `i` and `j` by concatenation (`++`)

MDH Examples

Important functions are MDHs — we can express them easily using our `md_hom` pattern:

Linear Algebra

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

- `GEMV` and `DOT` are expressed similarly as `GEMM`
- both are special cases of `GEMM`:
  - `GEMV`: `B` is `Kx1` matrix (i.e., no dimension `j`)
  - `DOT`: `A` is `1xK` matrix, B is `Kx1` matrix (i.e., no dimensions `i` and `j`)
- Note: `GEMV` and `DOT` are not required in our approach — we can use `GEMM` for them (provides the same high performance).

MDH Examples

Important functions are MDHs — we can express them easily using our `md_hom` pattern:

Linear Algebra

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

Stencil Computations

```
Gaussian_2D = md_hom( G_func, (++,++)     ) o view(…)
Jacobi_3D   = md_hom( J_func, (++,++, ++) ) o view(…)
```

Data Mining

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

Machine Learning

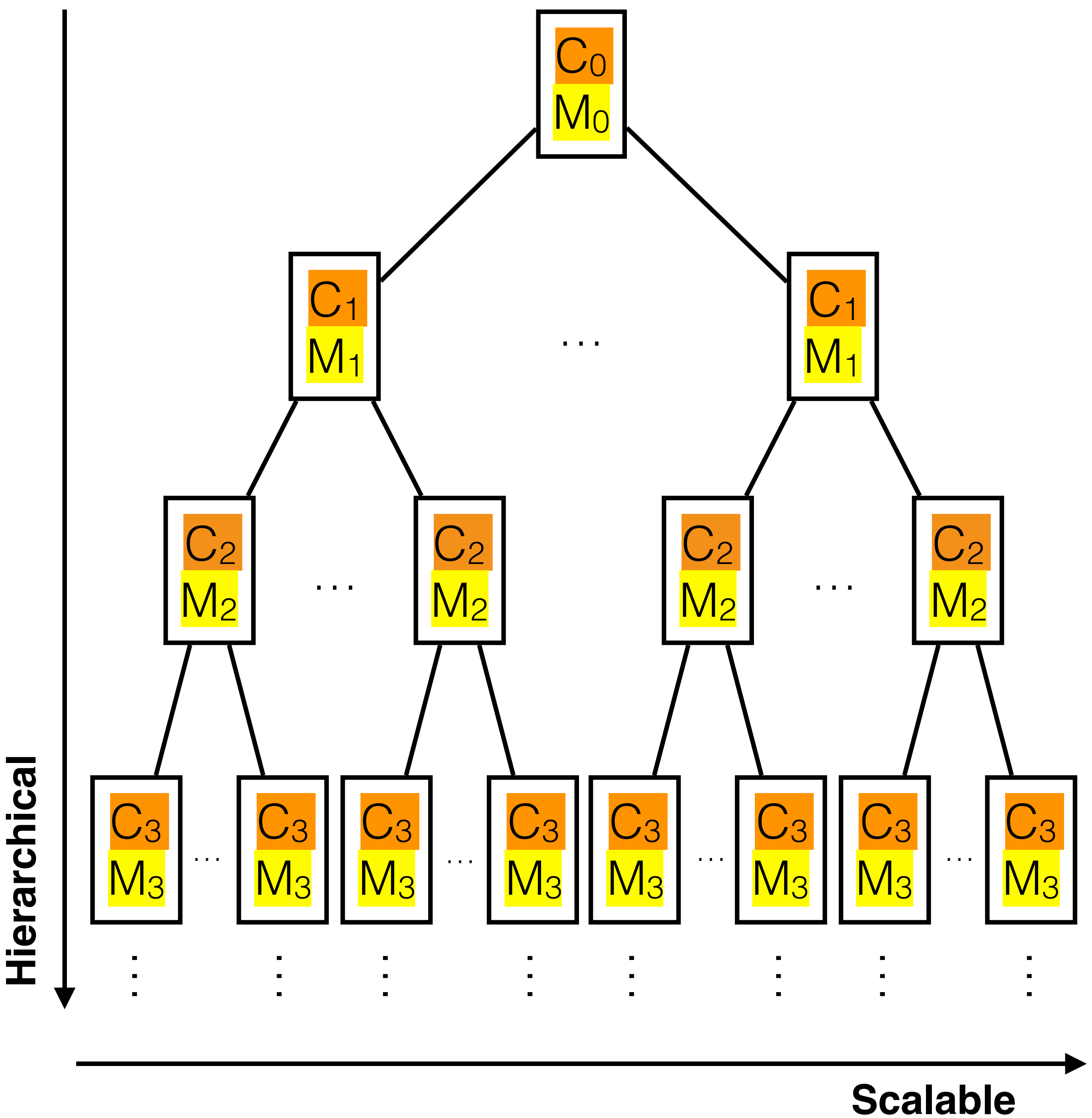
```
TC = md_hom( *, (++,…,++ , +,…,+) ) o view(…)
```

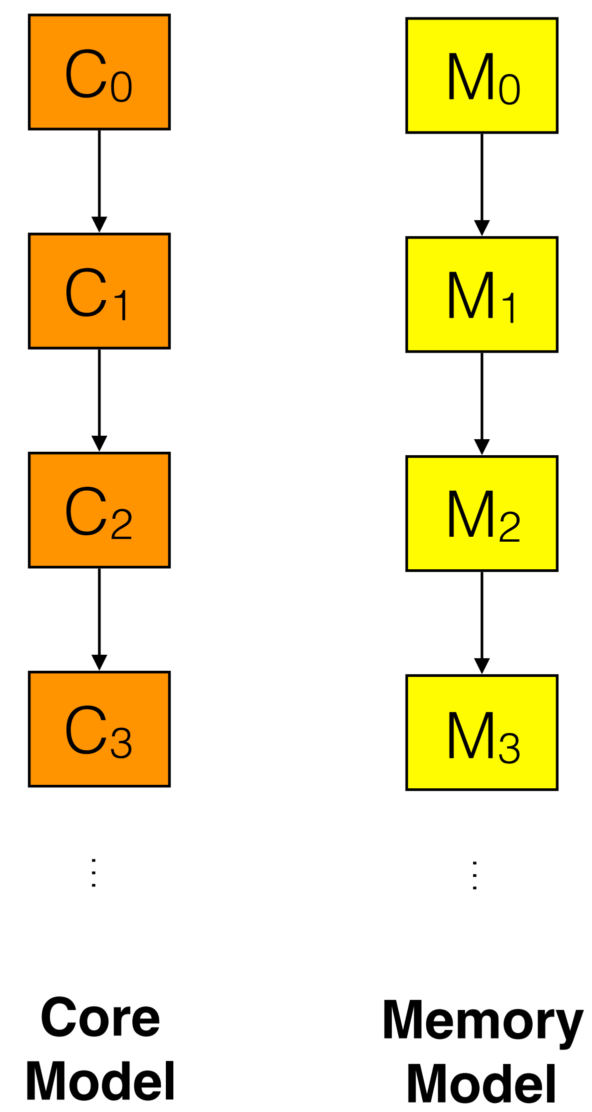
Further examples: MLP, SVM, ECC, …, Mandelbrot, Parallel Reduction, …

Our DSL needs only the patterns `md_hom(…)` and `view(…)`

MDH — Target Machine Model

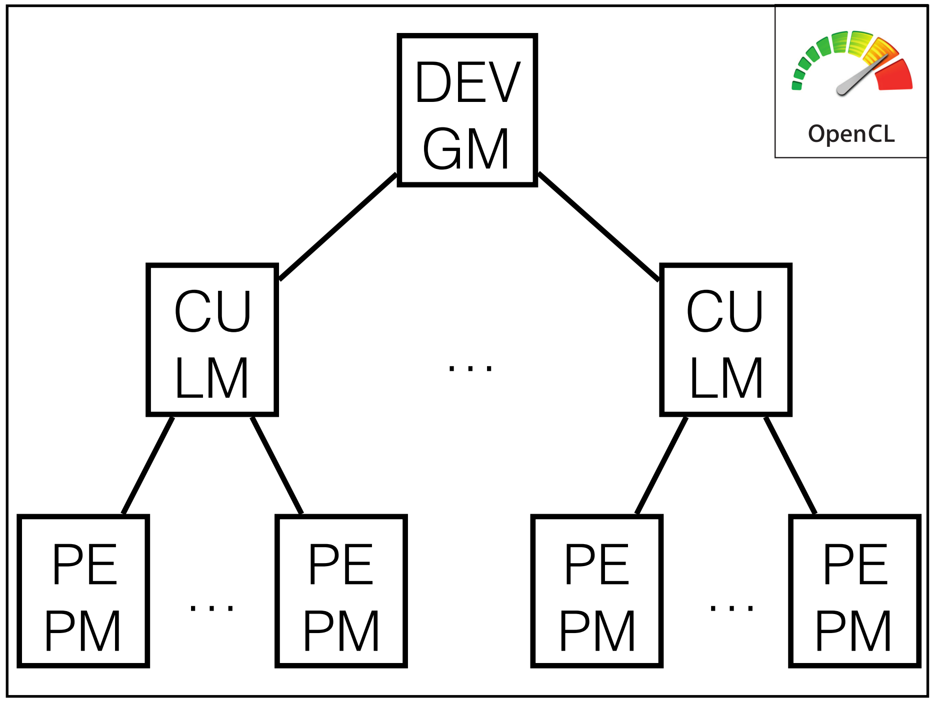
<u>We **generate** code for MDHs targeting ***hierarchical***, ***scalable*** *machine models*:</u>

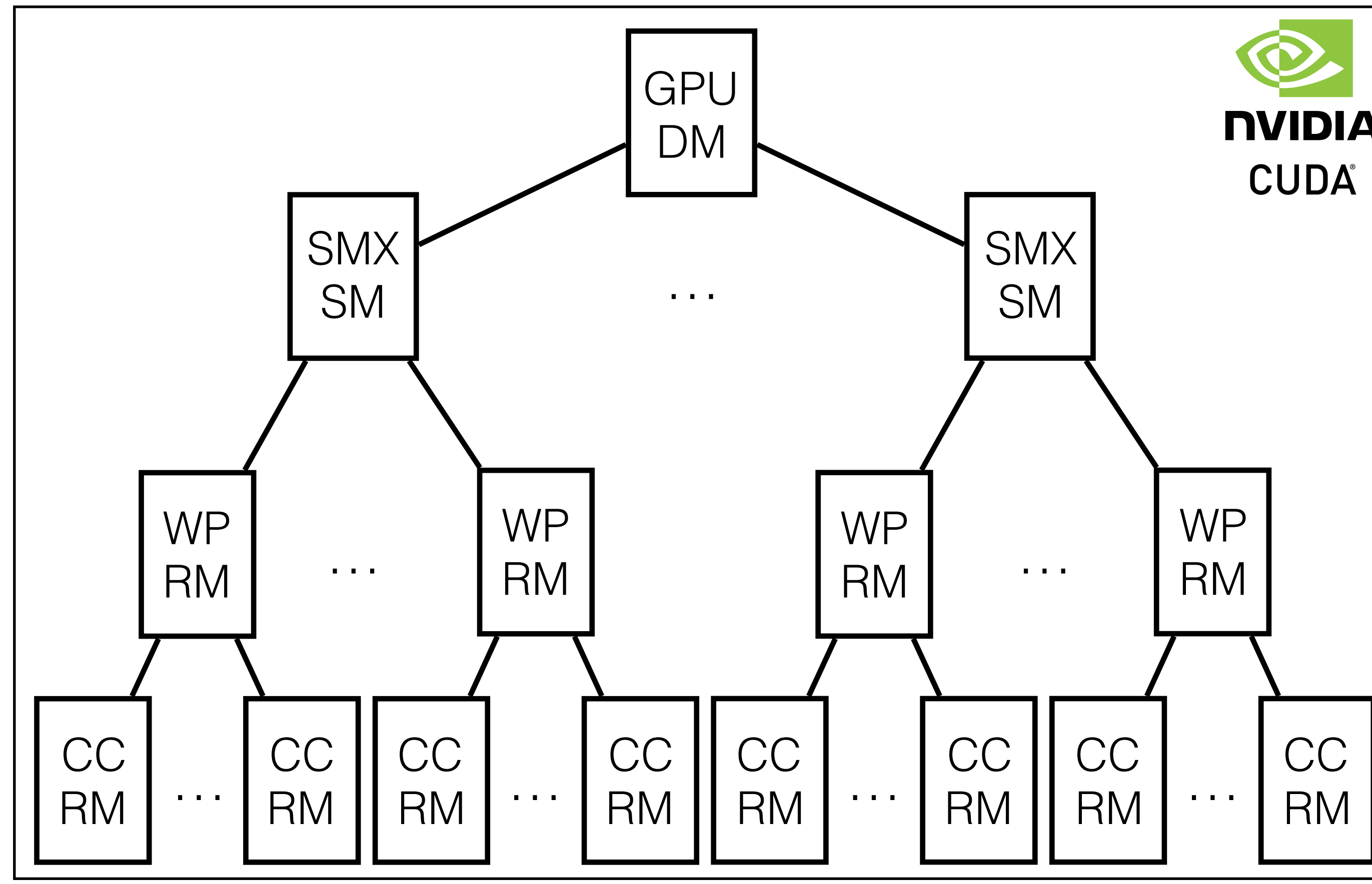


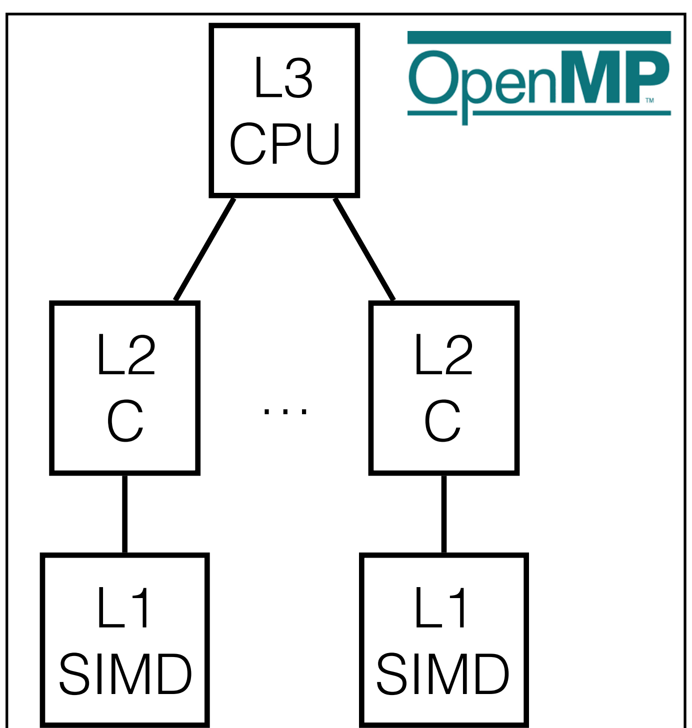


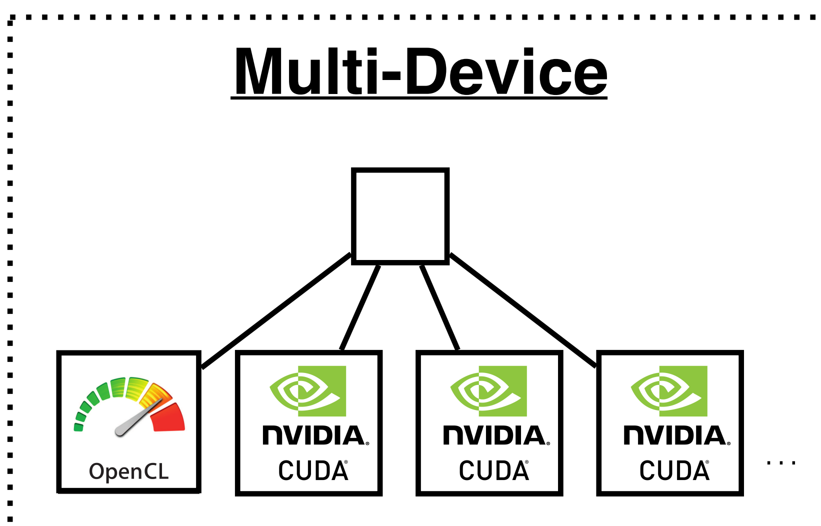
MDH — Target Machine Model

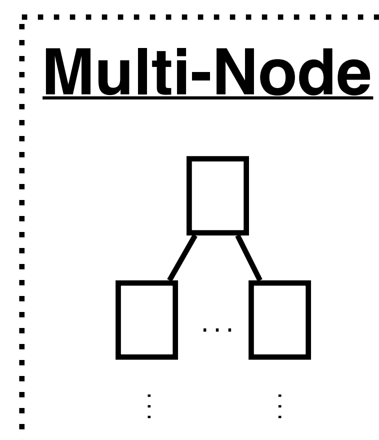
Examples of such *machine models*:







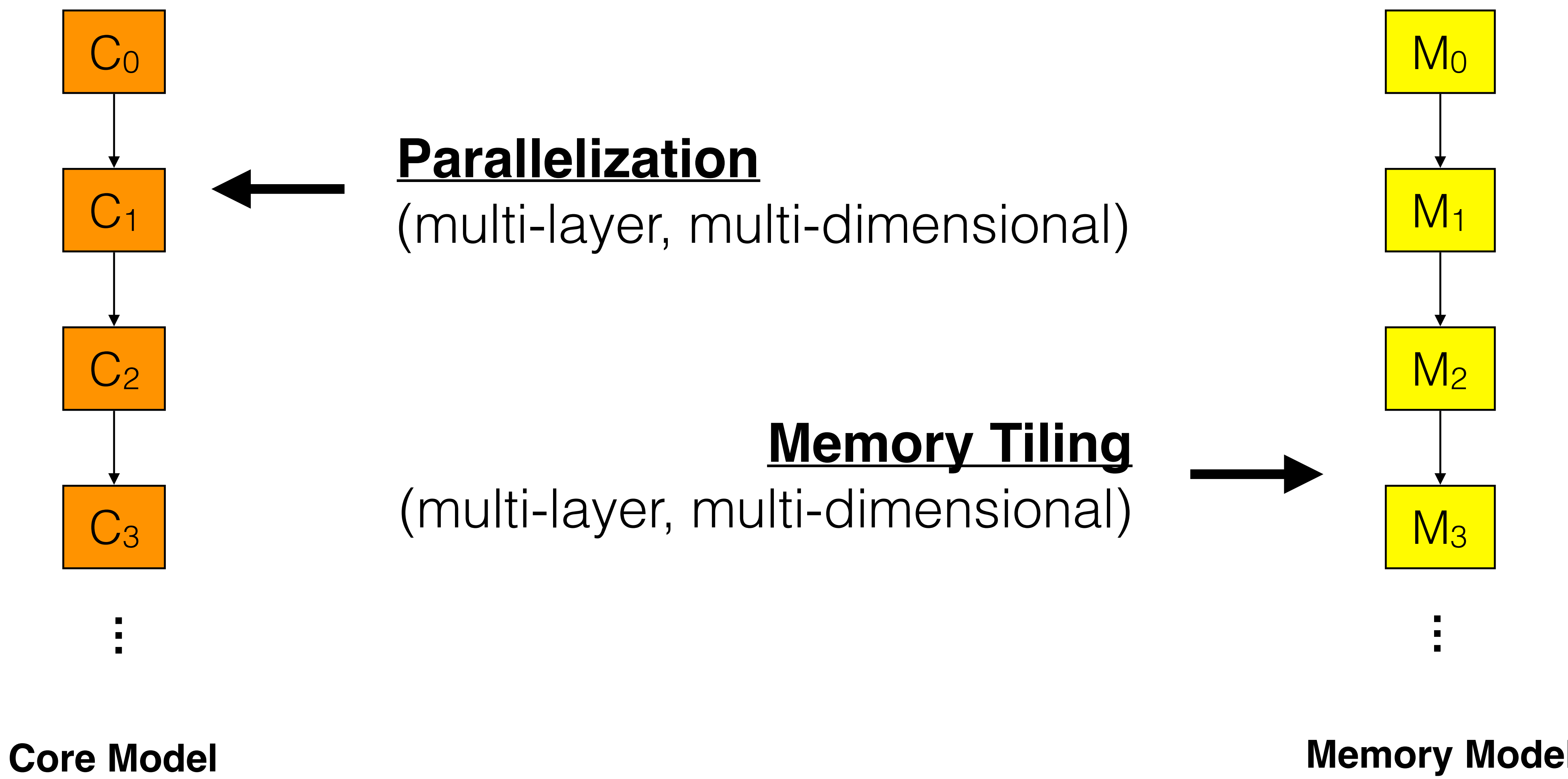




. . .

Code Generation for MDHs

Our **uniform `md_hom` representation** of MDHs enables **systematically generating code for such machine models** that can be **automatically optimized**:



In the following: Explain implementation at example of OpenCL.

Optimization of MDHs

1. Parallelization (multi-layer, multi-dimensional):

```
#reduce ⊛_1
for( i_1 = 1, ... , N_1 )
  ⋱
    #reduce ⊛_d
    for( i_d = 1, ... , N_d )
    {
      f( a[i_1]...[i_d] )
    }
```

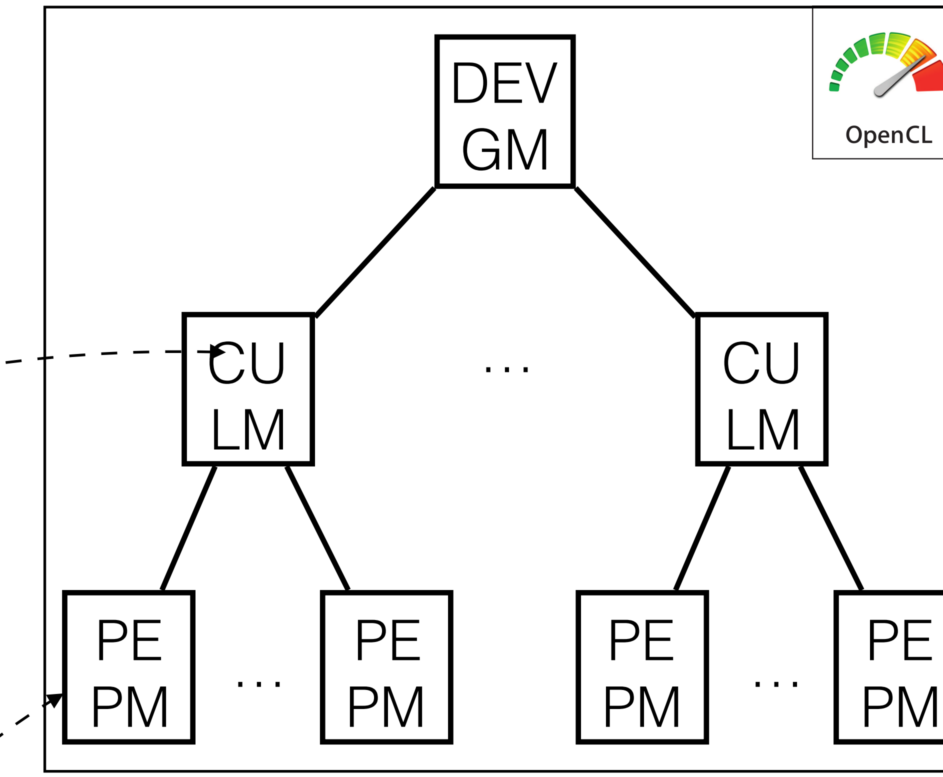
MDH pseudocode for
$\texttt{md\_hom}(f,\ (\circledast_1, \ldots, \circledast_d))$

```
#reduce ⊛_1
parallel_for( i_1 = 1, ... , NUM_WG_1 )
  ⋱
   #reduce ⊛_d
   parallel_for( i_d = 1, ... , NUM_WG_d )
```

Work-Groups

```
#reduce ⊛_1
parallel_for( ii_1 = 1, ... , NUM_WI_1 )
  ⋱
   #reduce ⊛_d
   parallel_for( ii_d = 1, ... , NUM_WI_d )
```

Work-Items



- We parallelize for each of OpenCL's **two parallel layers.**
- We parallelize **on each layer** in **all *d* dimensions** of the MDH.

▸ We **auto-tune** the number of threads **on each layer** and **in each dimension**.

Optimization of MDHs

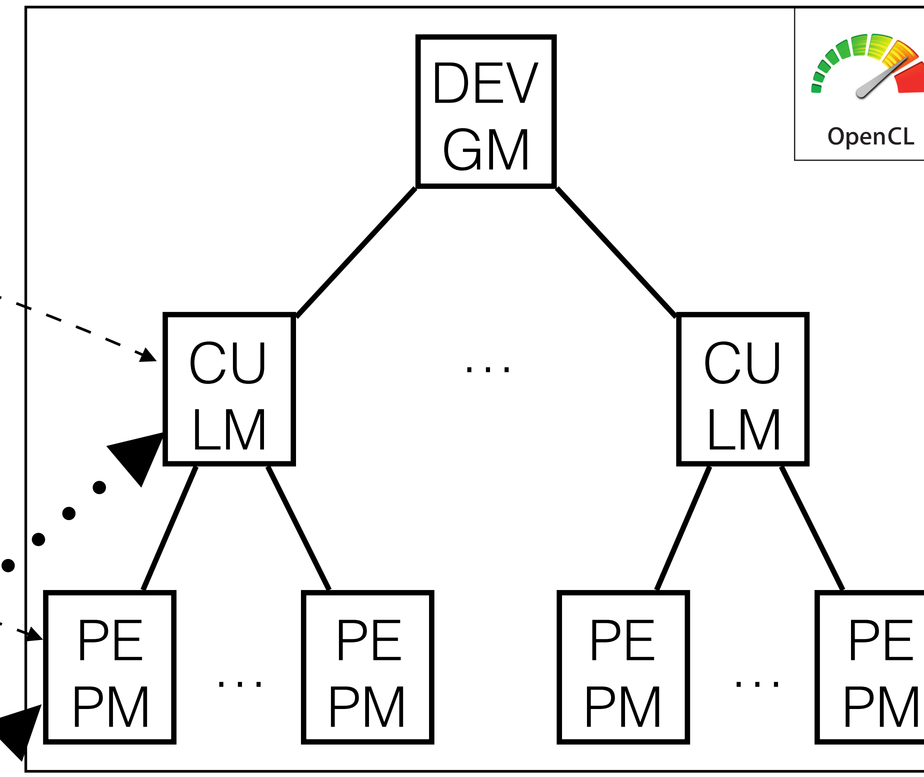
2. Memory Tiling (multi-layer, multi-dimensional):

```
#reduce ⊛_1
parallel_for( i_1 = 1, ... , NUM_WG_1 )
  ⋱
   #reduce ⊛_d
   parallel_for( i_d = 1, ... , NUM_WG_d )
```

```
#reduce ⊛_1
parallel_for( ii_1 = 1, ... , NUM_WI_1 )
  ⋱
   #reduce ⊛_d
   parallel_for( ii_d = 1, ... , NUM_WI_d)
```

```
for( j_1 = 1, ... , NUM_LM_TL_1 )
  ⋱
   for( j_d = 1, ... , NUM_LM_TL_d )
```

```
for( j_1 = 1, ... , NUM_PM_TL_1 )
  ⋱
   for( j_d = 1, ... , NUM_PM_TL_d )
{
  f( ...)
}
```



- We tile for each of OpenCL's **two memory layers.**
- We tile **on each layer** in **all dimensions** of the MDH.
- <u>We **auto-tune** the sizes of tiles **on each layer** and **in each dimension**.</u>

Automatic Performance Tuning

Our OpenCL implementation is generated as **generic in performance-critical parameters** (a.k.a. tuning parameters) of OpenCL's abstract device models:

	Name	Description
Platform Model	NUM_WG_i	number of Work-Groups
	NUM_WI_i	number of Work-Items
Memory Model	LM_TL_SIZE_i	local memory tile size
	PM_TL_SIZE_i	private memory tile size
	LM_CACHING	copying tiles to local memory on/off
	PM_CACHING	copying tiles to private memory on/off
	COMP_LM_TL	compute local tiles in: PM, LM, or GM
	COMP_PM_TL	compute private tiles in: PM, LM, or GM
Platform↔Memory Model	OCL_MDA_MAPPING	mapping OCL to MDA dimensions
	COMB_OP_ORDER	order of combine operators
	COMB_WG_RES_AFTER	when to combine WG results
	COMB_WI_RES_AFTER	when to combine WI results
	…	

All parameters are chosen independently of the target MDH, architecture, and input size!

Automatic Performance Tuning

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

	Domain-specific auto-tuning	OpenTuner	CLTune	ATF
Arbitrary Programming Language		✓		✓
Arbitrary Application Domain		✓	✓	✓
Arbitrary Tuning Objective	✓	✓		✓
Arbitrary Search Technique	✓	✓	✓	✓
Interdependent Parameters	✓		✓	✓
Large Parameter Ranges	✓	✓		✓
Directive-Based Auto-Tuning				✓
Automatic Cost Function Generation	✓		✓	✓

ATF combines major advantages over state-of-the-art auto-tuning approaches

[1] Rasch, Ari et al. "ATF: A Generic, Directive-Based Auto-Tuning Framework." *Concurrency and Computation: Practice and Experience, 2019*

Automatic Performance Tuning

<u>ATF usage:</u> We annotate our program code with easy-to-use *tuning directives.*

```
#atf::tp name       NUM_WG_1
         range      interval<int>( 1, N_1 )

#atf::tp name       NUM_WI_1
         range      interval<int>( 1, N_1 )

// ...

#atf::tp name       LM_SIZE_1
         range      interval<int>( 1, N_1 )
         constraint LM_SIZE_1 <= N_1

#atf::tp name       PM_SIZE_1
         range      interval<int>( 1, N_1 )
         constraint PM_SIZE_1 <= LM_SIZE_1

// …

// OpenCL kernel code
```

→ **ATF automatically determines optimized parameter values**

(ATF is also available as C++/Python programming library)

Execution

We execute our *generated* and *optimized* OpenCL code using our own ***dOCAL*** **[1]** framework which:

1. provides **high-level abstractions** for simplifying implementing **OpenCL host code**, especially for multi-device systems (e.g., by automatically performing memory allocations and synchronization);

2. provides **asynchronous computation efficiency** (e.g., overlapping data transfers and/or kernel computations) by generating and maintaining a data-dependency graph transparently from the user;

3. enables conveniently executing OpenCL kernels on **remote nodes** (via *Boost.Asio*).

[1] Rasch, Ari et al., "dOCAL: High-Level Distributed Programming with OpenCL and CUDA", *The Journal of Supercomputing, 2019*

Execution

<u>Illustration:</u> using dOCAL for executing OpenCL kernel on GPU.

```
#include "docal.hpp"

int main()
{
  // 1. choose device
  auto device = docal::get_device( "NVIDIA", "Tesla" );

  // 2. declare kernel
  docal::kernel GEMM = docal::source( /* OpenCL Code */ );

  // 3. prepare kernels' inputs
  docal::buffer<float> A( N*N );
  docal::buffer<float> B( N*N );
  docal::buffer<float> C( N*N );

  std::generate( A.begin(), A.end(), std::rand );
  std::generate( B.begin(), B.end(), std::rand );

  // 4. start device computations
  device( GEMM                                      )
        ( nd_range( /* GS */), nd_range( /* LS */ ) )
        ( read ( A ), read( B ), write( C )         );

  // 5. print result
  for( int i = 0 ; i < N*N ; ++i )
    std::cout << C[ i*N + j ];
}
```

Experimental Results

We compare our **automatically-generated and optimized code** using:

Applications

1. Linear Algebra Routines (GEMM, GEMV)
2. Stencil Computations (Gaussian Convolution 2D, Jacobi 3D)
3. Tensor Contractions

Competitors

- Performance-Portable approaches
- Domain-specific, hand-optimized approaches

Architectures

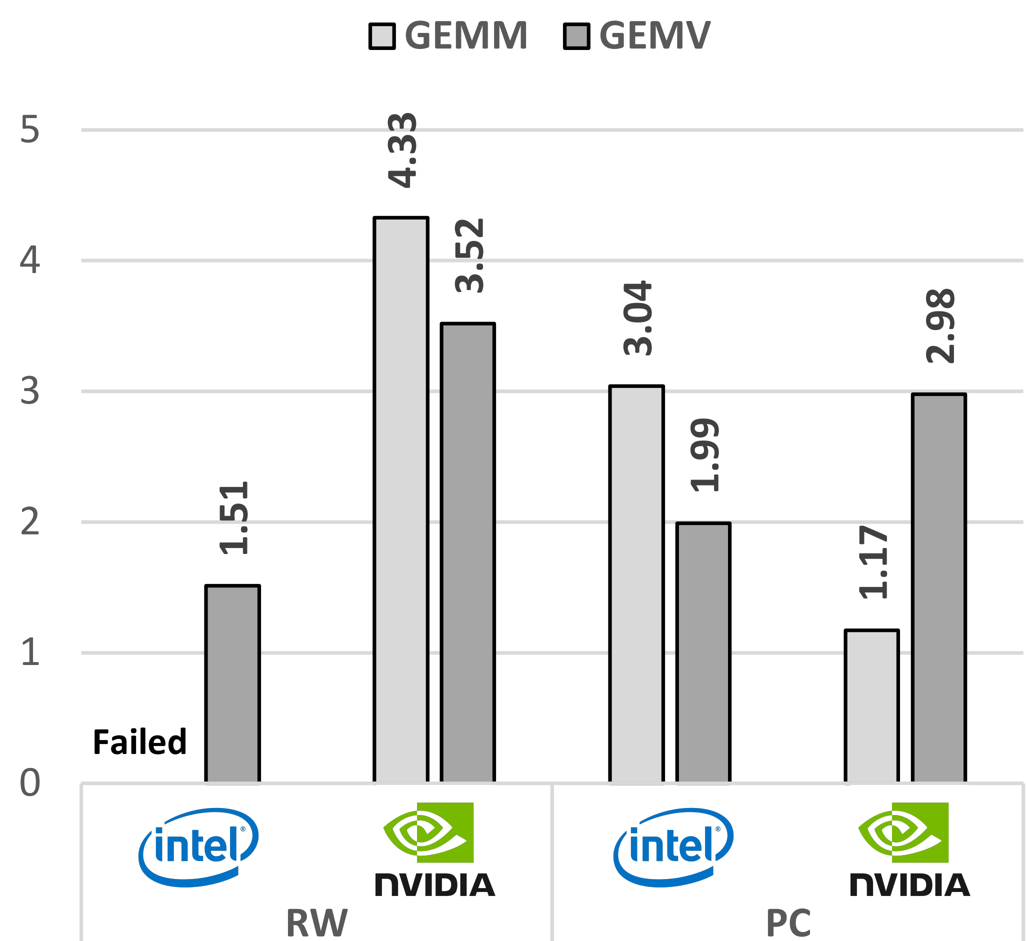
- Intel Xeon multi-core CPU (E5-2640)
- NVIDA Tesla V100 GPU (SMX2-16GB)

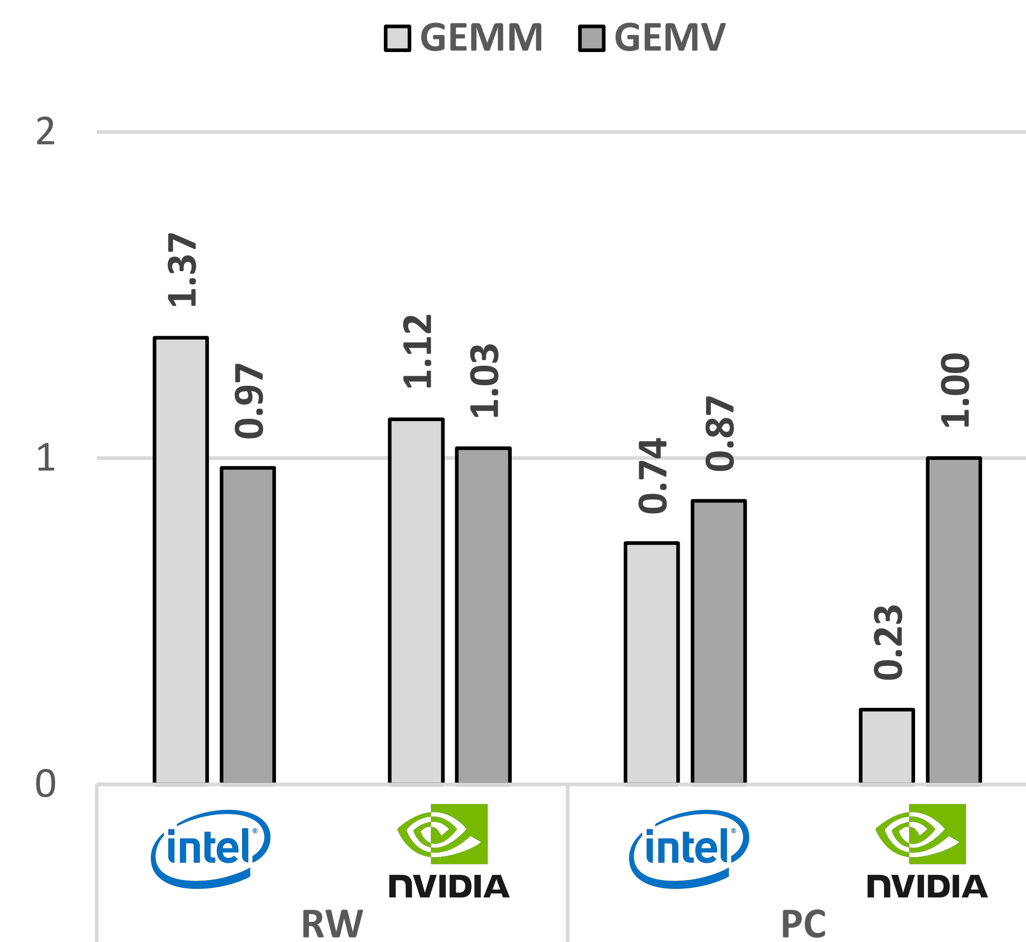
Data Sets

- RW: Real-world sizes from Deep Learning
- PC: Sizes that are preferable for our competitors

Experimental Results

1. Linear Algebra:

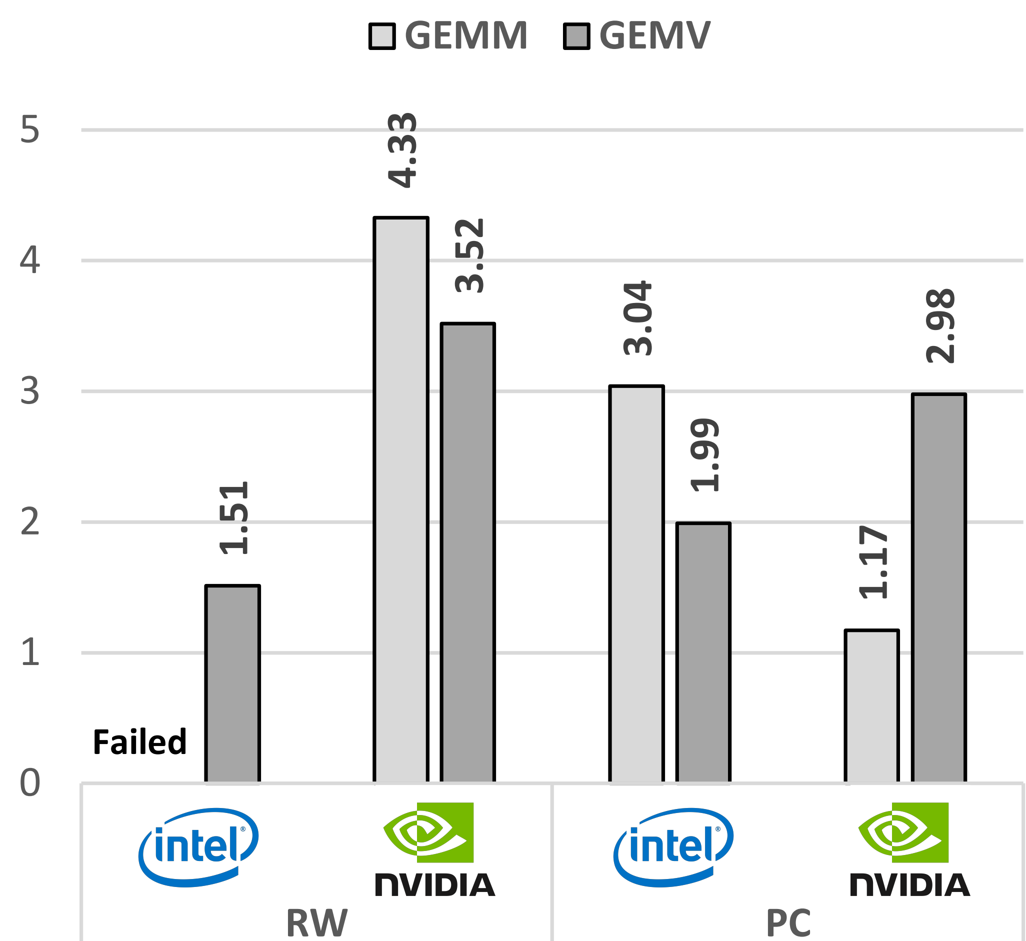


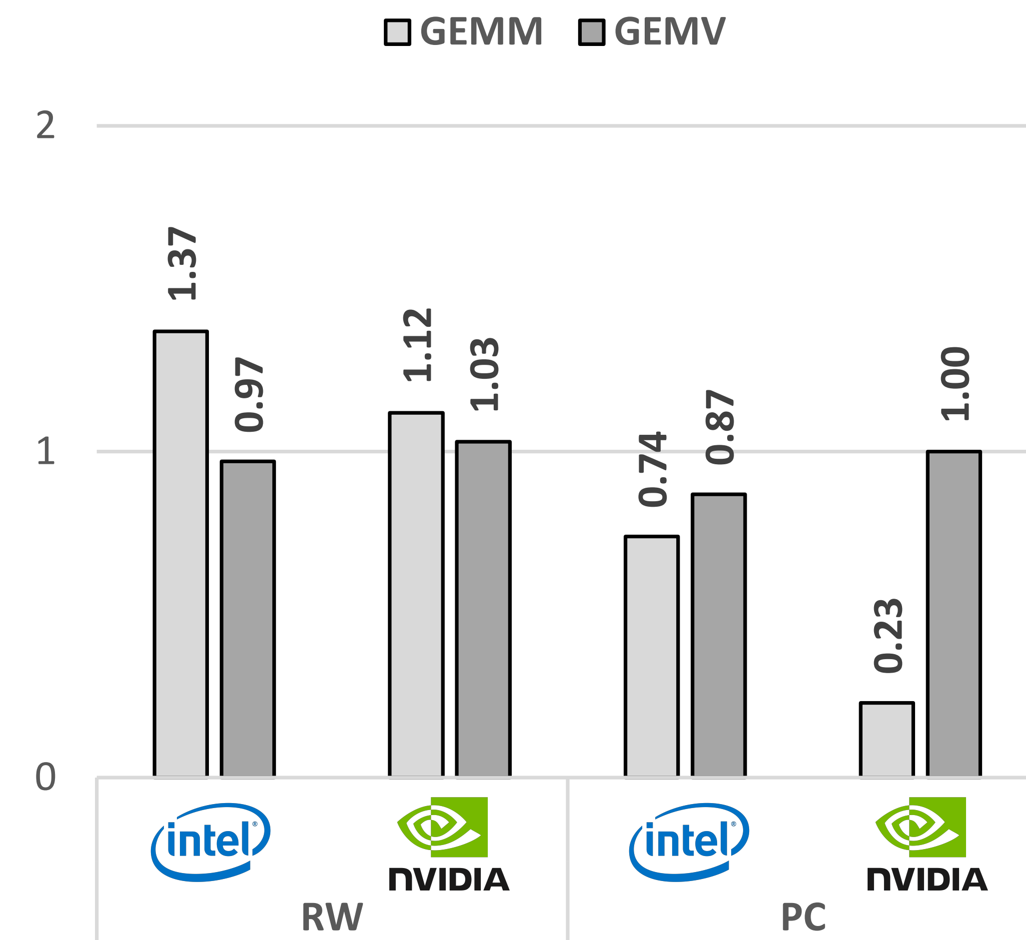


- We demonstrate speedup of our approach for both `GEMM` and `GEMV` over
  - *Lift* (left) — currently one of the best-performing performance-portable approaches (based on OpenCL)
  - *Intel MKL* & *NVIDIA cuBLAS* (right) — hand-optimized, BLAS-specific approaches for Intel/ NVIDIA hardware only
- Note: We use our `GEMM` implementation also for `GEMV` → demonstrates efficiency of our approach for very irregular sizes

Experimental Results

1. Linear Algebra:





Comparison to Lift [1,2]:

- Speedups up to **> 4x**.
- Our approach relies on generic optimizations + auto-tuning (rather than transformation rules).

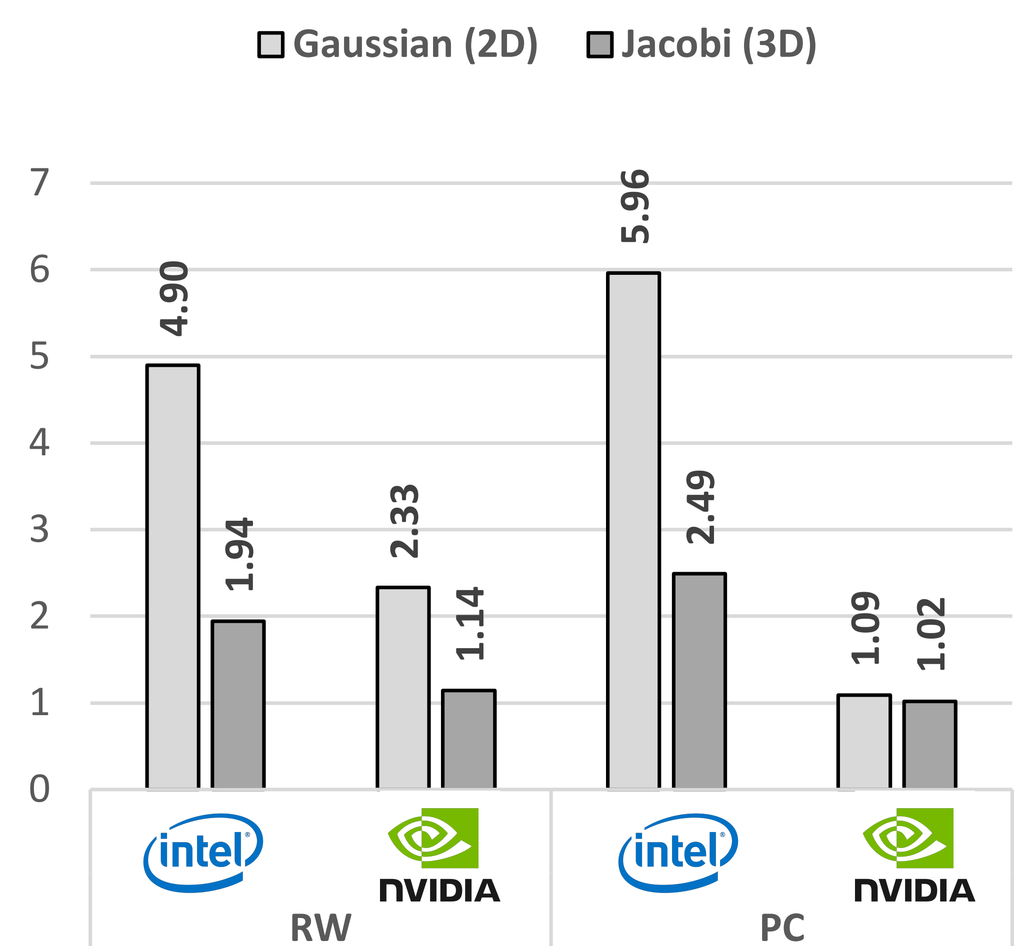
Comparison to MKL/cuBLAS:

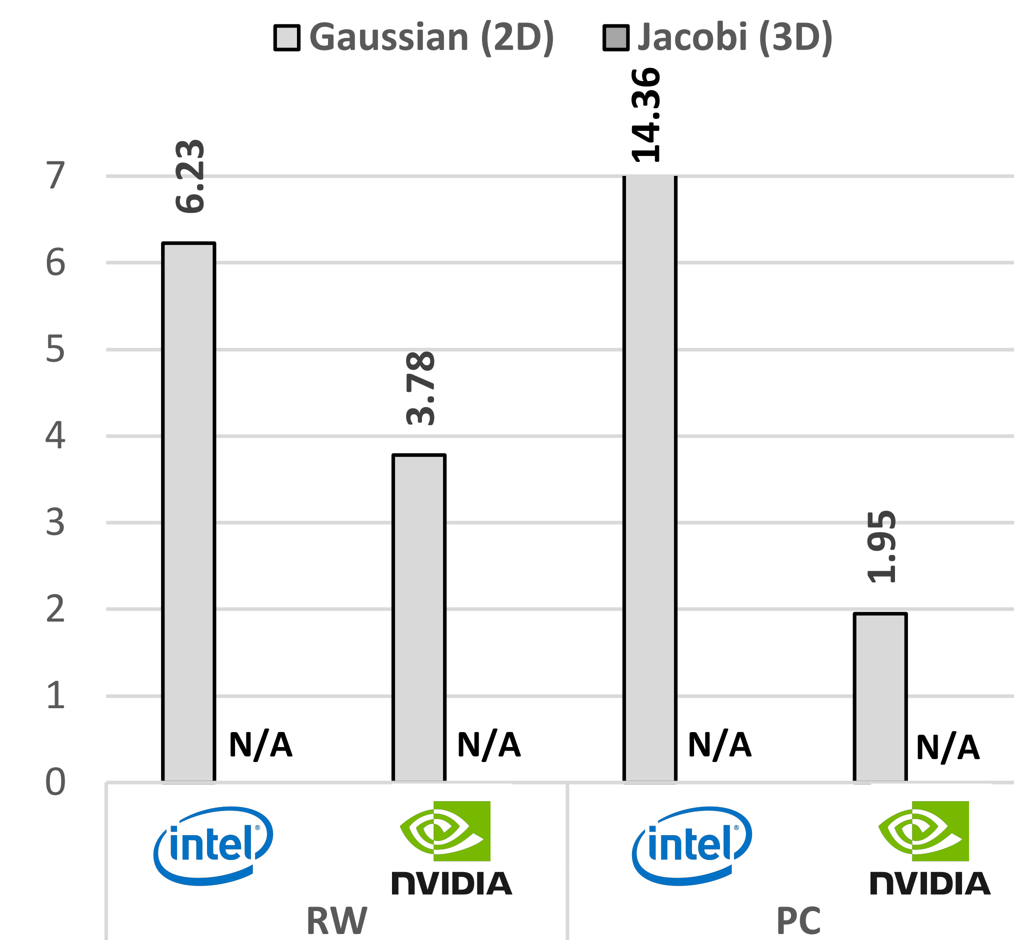
- Competitive performance for RW.
- Lower performance for PC because of assembly optimizations (e.g., CUDA tensor cores).

[1] Steuwer et. al, "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation", CGO'17.
[2] Steuwer et. al, "Matrix Multiplication Beyond Auto-Tuning: Rewrite-Based GPU Code Generation.", CASES'16.

Experimental Results

2. Stencil Computations





Comparison to Lift [3]:

- Speedups up to **>5x**
- Our approach relies on generic optimizations + auto-tuning (rather than transformation rules)

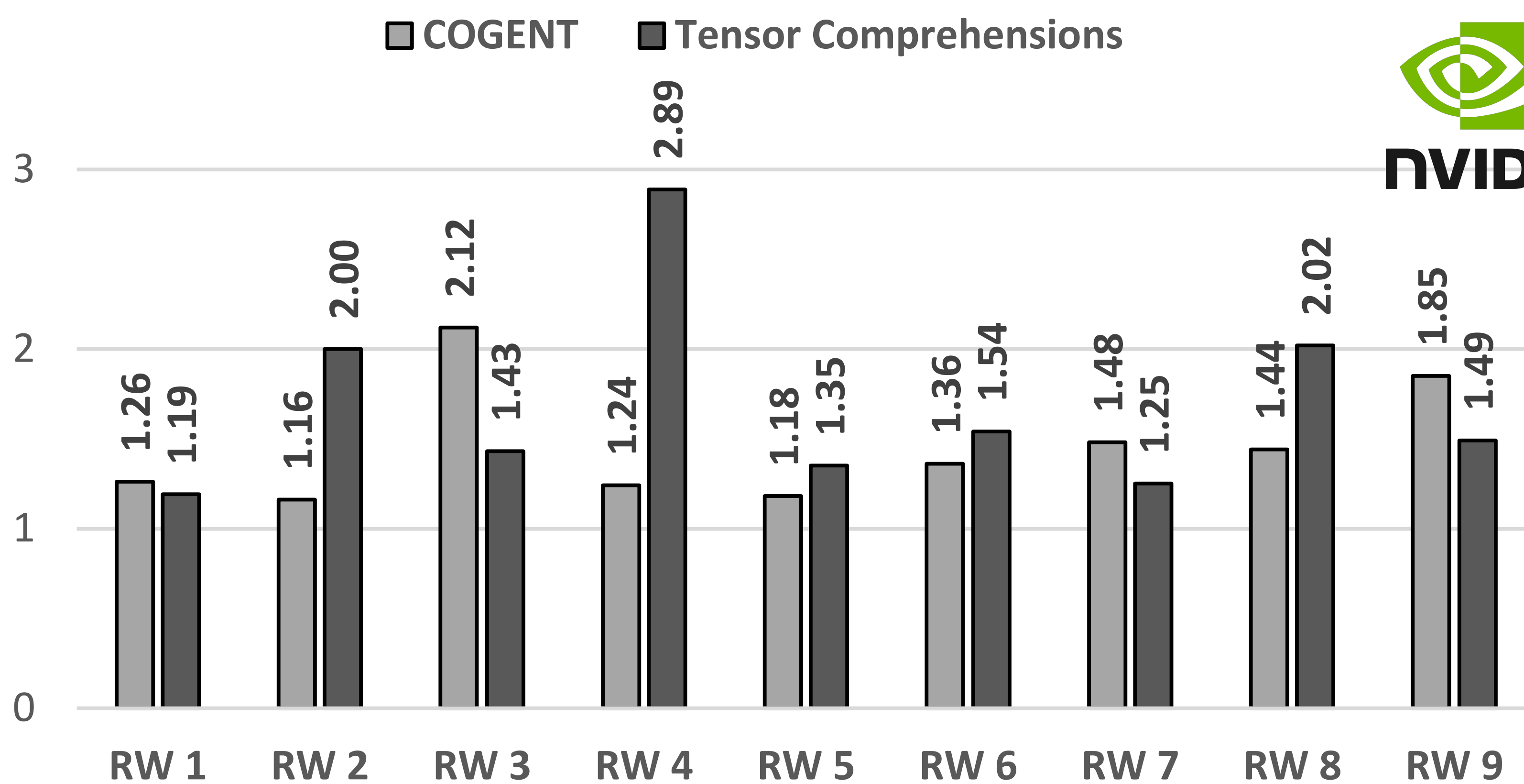
Comparison to MKL-DNN/cuDNN:

- Better performance → MCC!
- MCC: speedup >2x over MKL-DNN
- MCC: speedup 0.5x over cuDNN

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

Experimental Results

3. Tensor Contractions:



Tensor Contractions on 9 real-world samples

- Speedups up to **>2x** over both competitors.
- We provide a more flexible implementations, e.g., tile sizes of also >1 (COGENT [1]).

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

Conclusion

We provide **performance**, **portability**, and **productivity** for MDH functions:

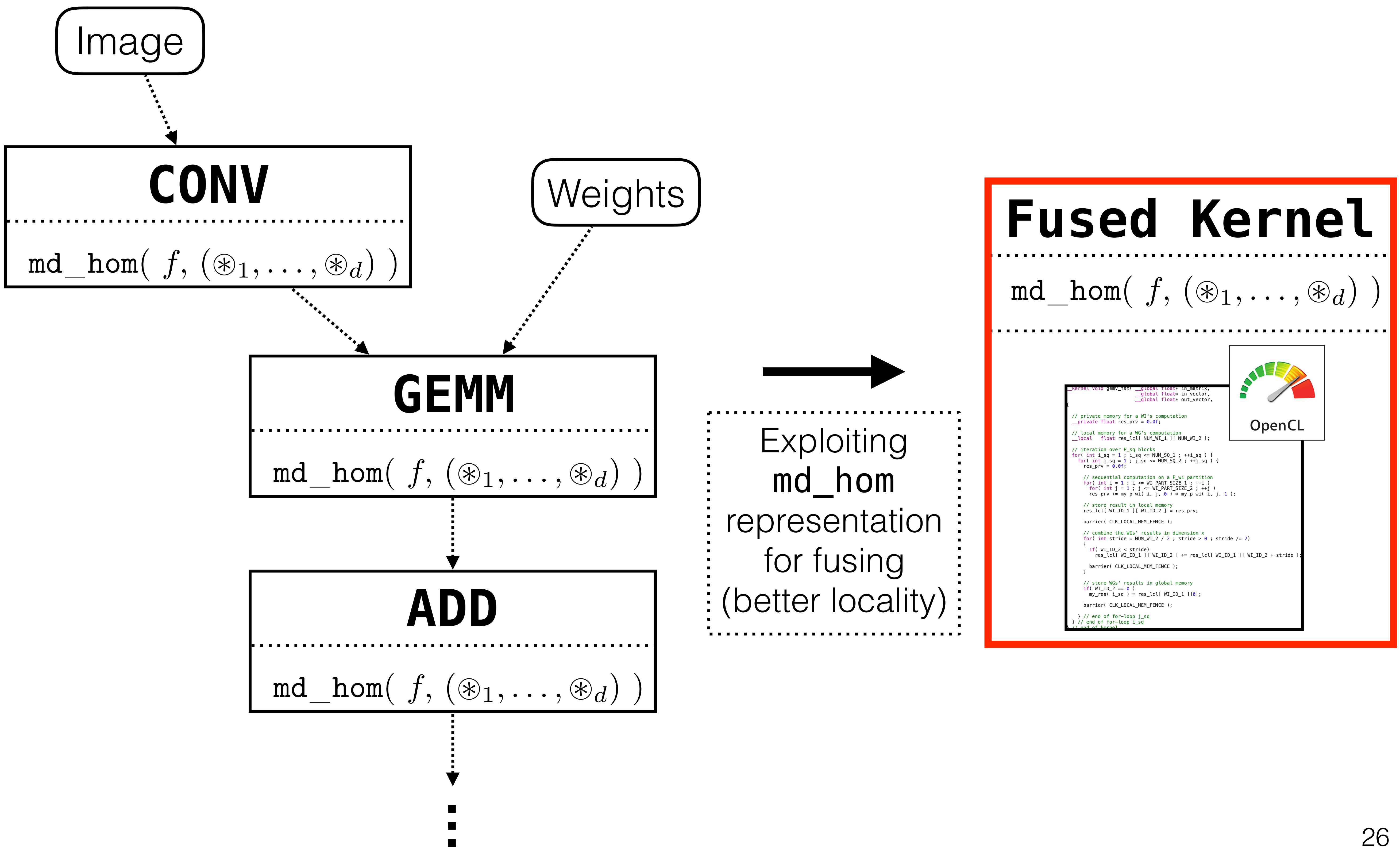
1. MDHs cover a **broad range of applications** (BLAS, Stencil, PRL, TC, …).
2. Our implementation of MDHs provides **high performance** (speedups up to >5) on both CPU and GPU
3. MDHs are functionally and performance **portable** over architectures and input sizes.
4. MDHs can be **conveniently implemented** using our `md_hom` high-level abstraction.

Moreover:

- Our **Auto-Tuning Framework (ATF)** is a **general-purpose approach** that **supports auto-tuning of interdependent tuning parameters.**
- We provide our **dOCAL** framework for **conveniently executing OpenCL kernels on multi-device/multi-node systems**, and it automatically **provides asynchronous computation efficiency**.

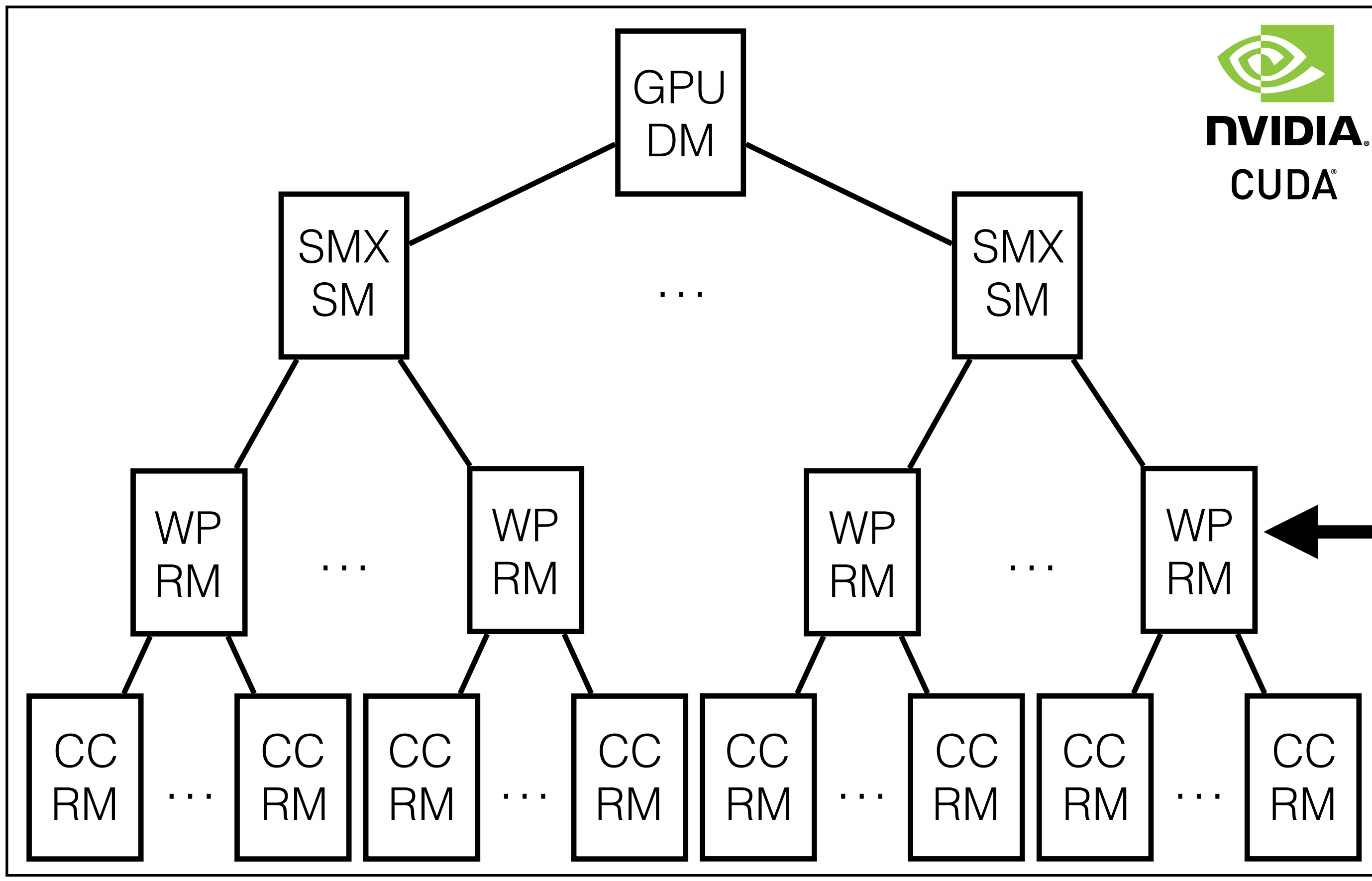
Current/Future Work

Graph-Level Optimizations (e.g. as required for deep learning):



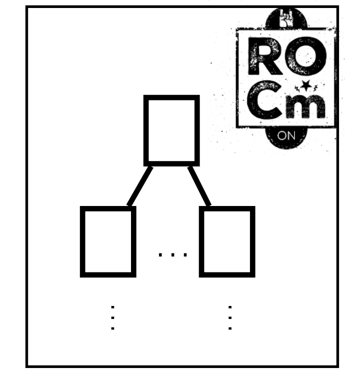
Current/Future Work

Further backends:



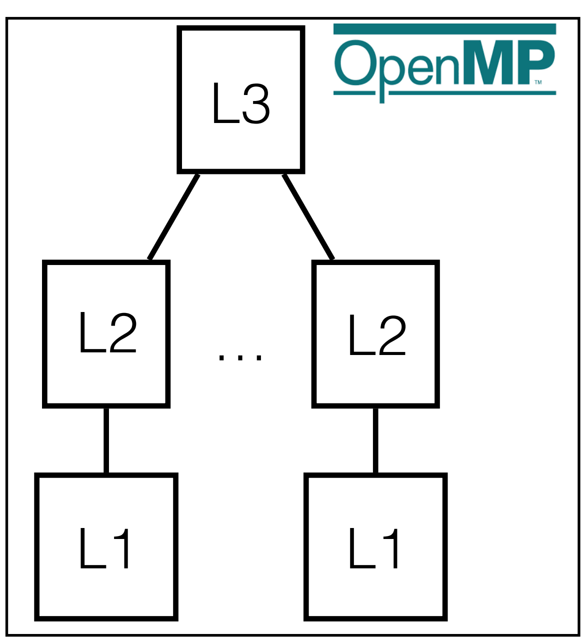
- **Shuffle Operations**
- **Tensor Cores**

→ comparable performance to cuBLAS

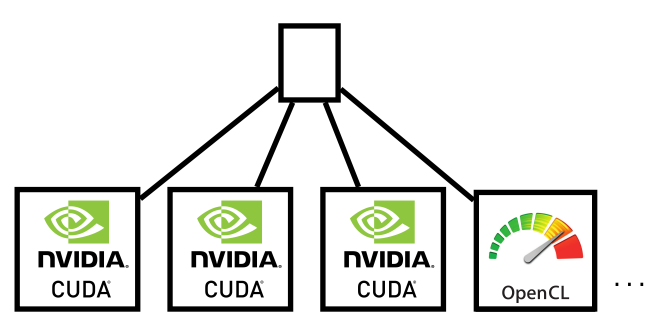


Assembler Backends

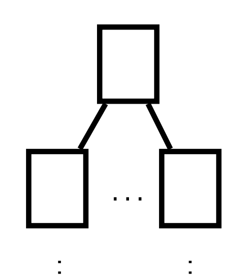
(e.g., NVIDIA PTX, Intel x86-64, …)



Multi-Device



Multi-Node



Current/Future Work

Parallelizing sequential C programs via MDH approach:

```
int main()
{
  // ...

  #pragma mdh ( ,, +:C[i][j] )
  for( int i = 0 ; i < M ; ++i )
    for( int j = 0 ; j < N ; ++j )
      for( int k = 0 ; k < K ; ++k )
      {
        C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
      }

  // ...
}
```

MDH Compiler

OpenCL

OpenMP

ROCm

NVIDIA CUDA

...

md_hom expression and **view** are automatically derived by MDH compiler:

```
md_hom( loop_body, (++, ++, +:C[i][j]) )
view( A,B )( i,j,k )=( A[i][k], B[k][j])
```

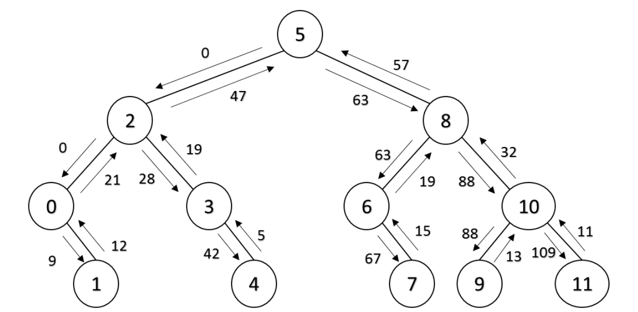
Annotating sequential C-Code with simple *MDH directives* *(similarly as in OpenMP/OpenACC).*

Current/Future Work

Demonstrating MDH approach's efficiency for more applications:

Benchmark	Typical Bottleneck of an Unoptimized Implementation	Optimizations Applied	Optimized Implementation Bottleneck	Potential Improvements
cutcp	Contention, Locality	Scatter-to-Gather, Binning, Regularization, Coarsening	Instruction Throughput	Minimizing Reads/Checks of Irrelevant Bin Data
mri-q	Poor Locality	Data Layout Transformation, Tiling, Coarsening	Instruction Throughput	
gridding	Contention, Load Imbalance	Scatter-to-Gather, Binning, Compaction, Regularization, Coarsening	Instruction Throughput	Minimizing Reads/Checks of Irrelevant Bin Data
sad	Locality	Tiling, Coarsening	Memory Bandwidth/Latency	Target Devices with Higher Register Capacities
stencil	Locality	Coarsening, Tiling	Bandwidth	
tpacf	Locality, Contention	Tiling, Privatization, Coarsening	Instruction Throughput	
lbm	Bandwidth	Data Layout Transformation	Bandwidth	
sgemm	Bandwidth	Coarsening, Tiling	Instruction Throughput	
spmv	Bandwidth	Data Layout Transformation	Bandwidth	
bfs	Contention, Load Imbalance	Privatization, Compaction, Regularization	Bandwidth	Avoiding Global Barriers / Better Kernels for Midsized Frontiers
histogram	Contention, Bandwidth	Privatization, Scatter-to-Gather	Bandwidth	Reducing Reads of Irrelevant Input (alleviated by cache)

Parboil



Rank	0	1	2	3	4	5	6	7	8	9	10	11
n	9	12	7	14	5	16	4	15	6	13	8	11
S	9	21	28	42	47	63	67	82	88	101	109	120

Prefix sum

TABLE I
RODINIA APPLICATIONS AND KERNELS (*DENOTES KERNEL).

Application / Kernel	Dwarf	Domain
K-means	Dense Linear Algebra	Data Mining
Needleman-Wunsch	Dynamic Programming	Bioinformatics
HotSpot*	Structured Grid	Physics Simulation
Back Propagation*	Unstructured Grid	Pattern Recognition
SRAD	Structured Grid	Image Processing
Leukocyte Tracking	Structured Grid	Medical Imaging
Breadth-First Search*	Graph Traversal	Graph Algorithms
Stream Cluster*	Dense Linear Algebra	Data Mining
Similarity Scores*	MapReduce	Web Mining

Rodinia

TensorFlow

Caffe2

Questions?