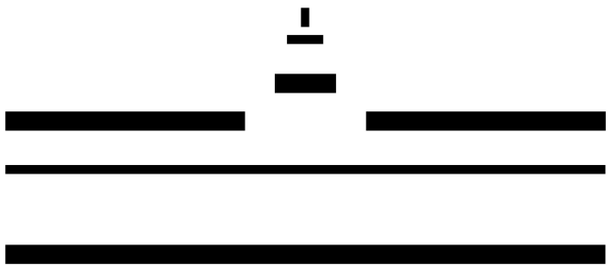


WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

Multi-Dimensional Homomorphisms and Their Implementation in OpenCL

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

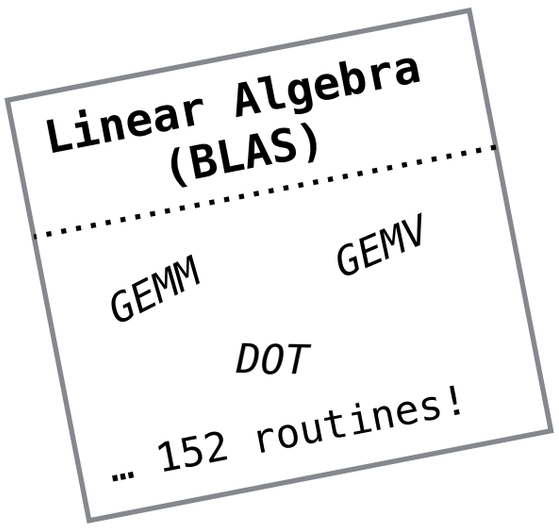
Ari Rasch, Richard Schulze, Sergei Gorlatch

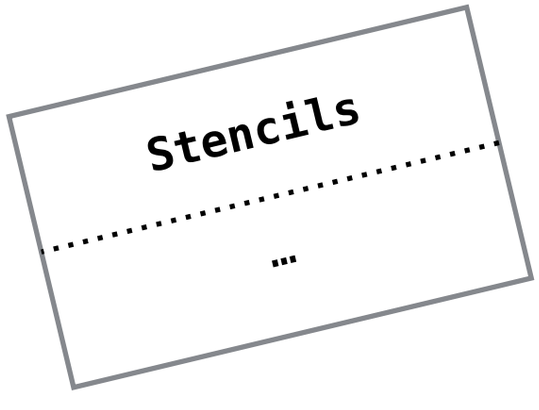
University of Münster, Germany

Motivation

Observation:

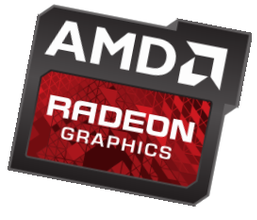
Applications





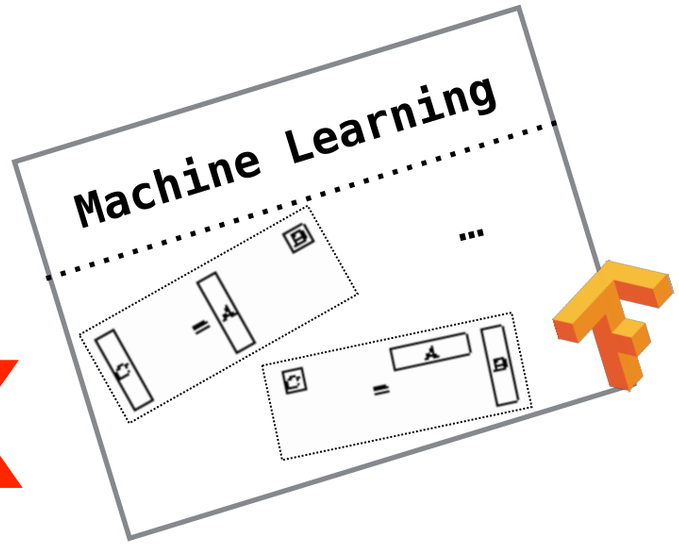
X

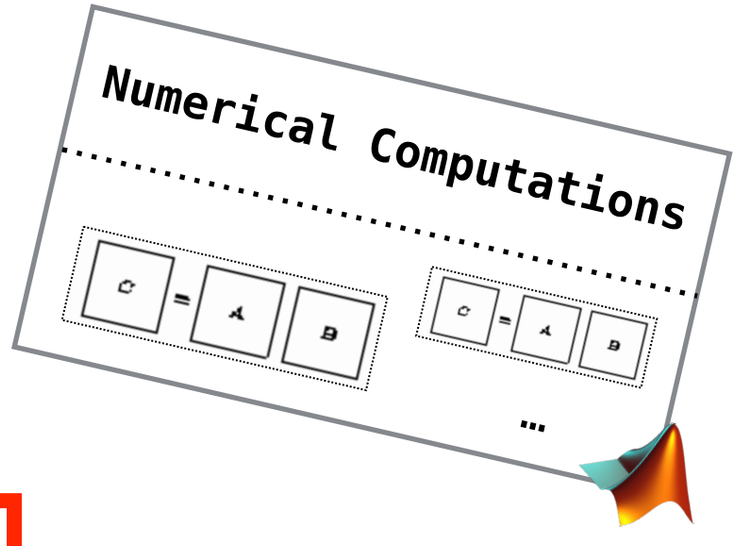
Architectures



X

Input/Output Characteristics

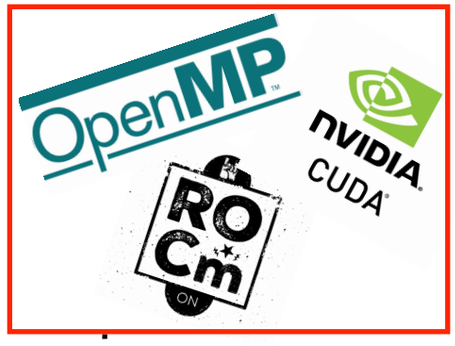


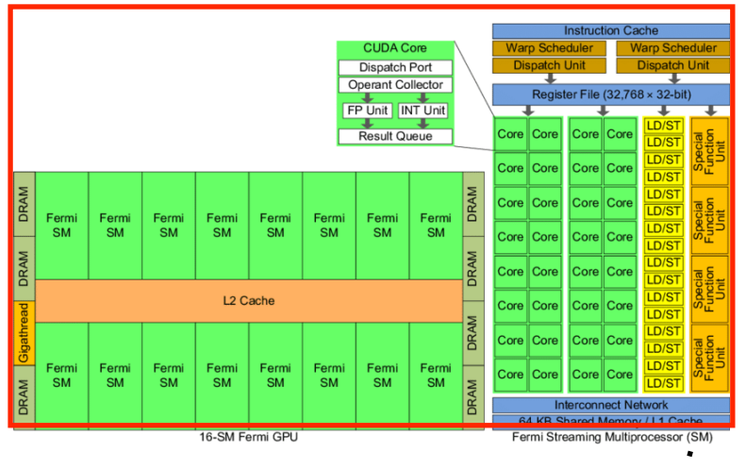


Combinatorial Explosion

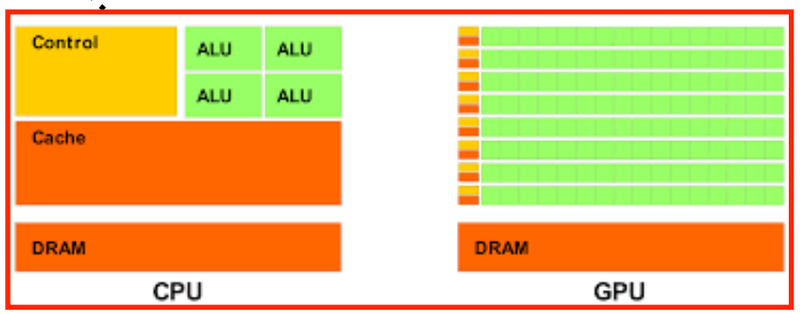
Motivation

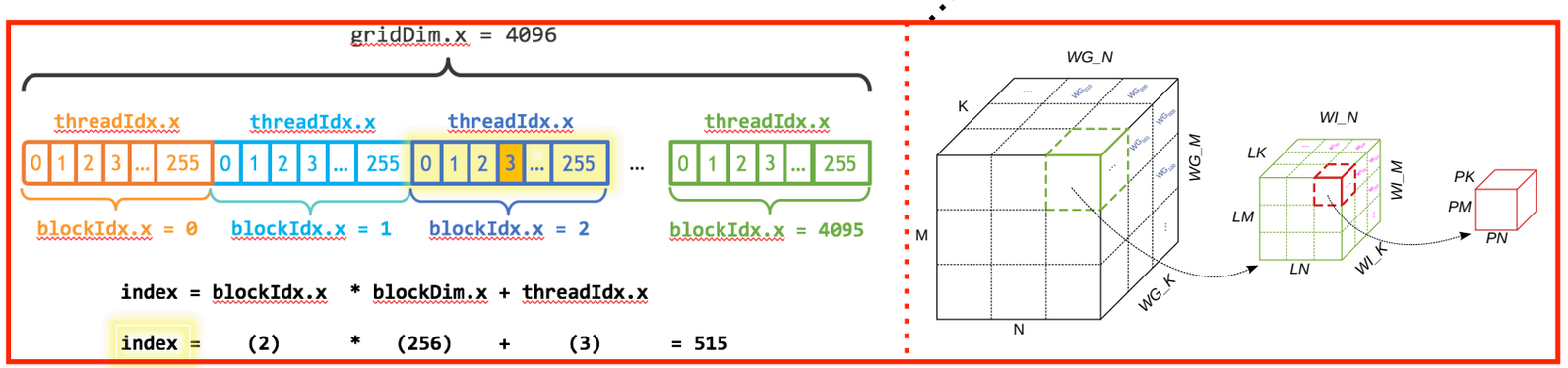
In a perfect world:





Write **one piece of code** for our application that provides **high performance**, is **performance-portable over different architectures** and **input/output characteristics** 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 a holistic approach to automatic code *generation*, *optimization*, and *execution* to address all aforementioned challenges for our algebraic class of *Multi-Dimensional Homomorphisms (MDHs)*:

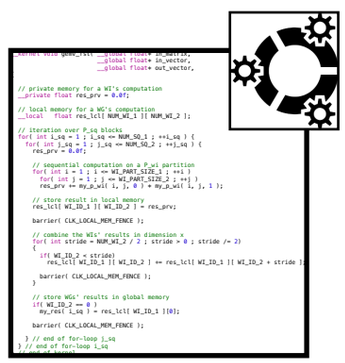
- **Multi-Dimensional Homomorphisms (MDHs)** provide a **formal notion of data parallelism**; thereby, they **cover important data-parallel computations**, e.g.: linear algebra (BLAS), stencils computations, …
- We enable **conveniently** and **uniformly implementing MDHs** by providing a **high-level DSL** for them.
- We provide a **DSL compiler** that **automatically generates OpenCL code** — the standard for uniformly programming different parallel architectures (e.g., CPU and GPU).
- Our OpenCL code is **fully automatically optimizable** (auto-tunable) — for any combination of a target **architecture** and **input/output characteristics** — by being generated as targeted to OpenCL's **abstract device model** and as **parametrized in this abstract model's performance-critical parameters.**

Our approach consists of three major steps:

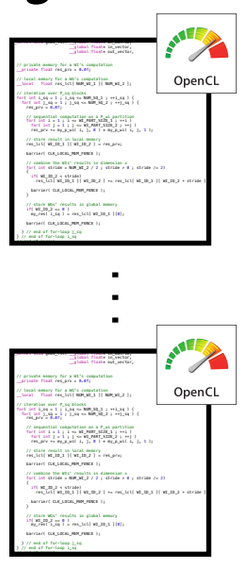
High-level parallel programming abstractions

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

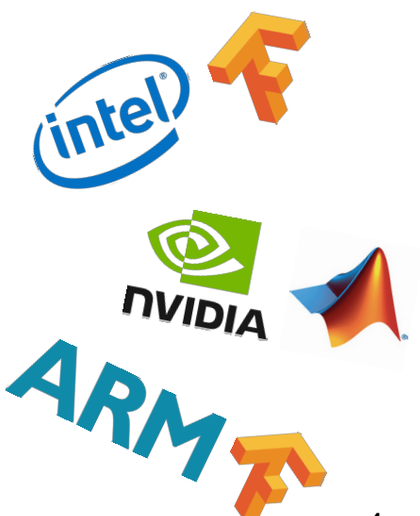
Generic program code



Executable program code



Different architectures and input/output characteristics



(1) Generation *[PACT'19, IJPP'18]*

(2) Optimization *[CCPE'18, HPCC'17]*

(3) Execution *[JOS'19, ICPADS'18]*

Agenda

We discuss the three major steps of our approach:

1. **Generation**

2. **Optimization**

3. **Execution**

Afterwards:

4. **Experimental Results**

5. **Current WIP / Future Work**

Generation

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, Schulze, et al. "Multi-Dimensional Homomorphisms and Their Implementation in OpenCL.", *International Journal of Parallel Programming, 2018*

MDH — Examples

Important data-parallel functions are MDHs — we can express them conveniently in our DSL:

Linear Algebra

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

```
for( int i = 0; i < M ; ++i )
  for( int j = 0; i < N ; ++j )
    for( int k = 0; i < K ; ++k )
      C[i][j] += A[i][k] * B[k][j];
```

GEMM in C

What's happening?

1. Prepare the domain-specific input uniformly for **`md_hom`**; for this, our DSL provides pattern **`view`**.
   - here: fuse matrices A and B to 3-dimensional array of pairs consisting of the elements in 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 dimensions `i` and `j` by concatenation (**++**).

MDH — Examples

Important functions are MDHs — we can express them conveniently in our DSL:

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

Access neighboring elements within their input buffer

```
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(…)
```

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

Machine Learning

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

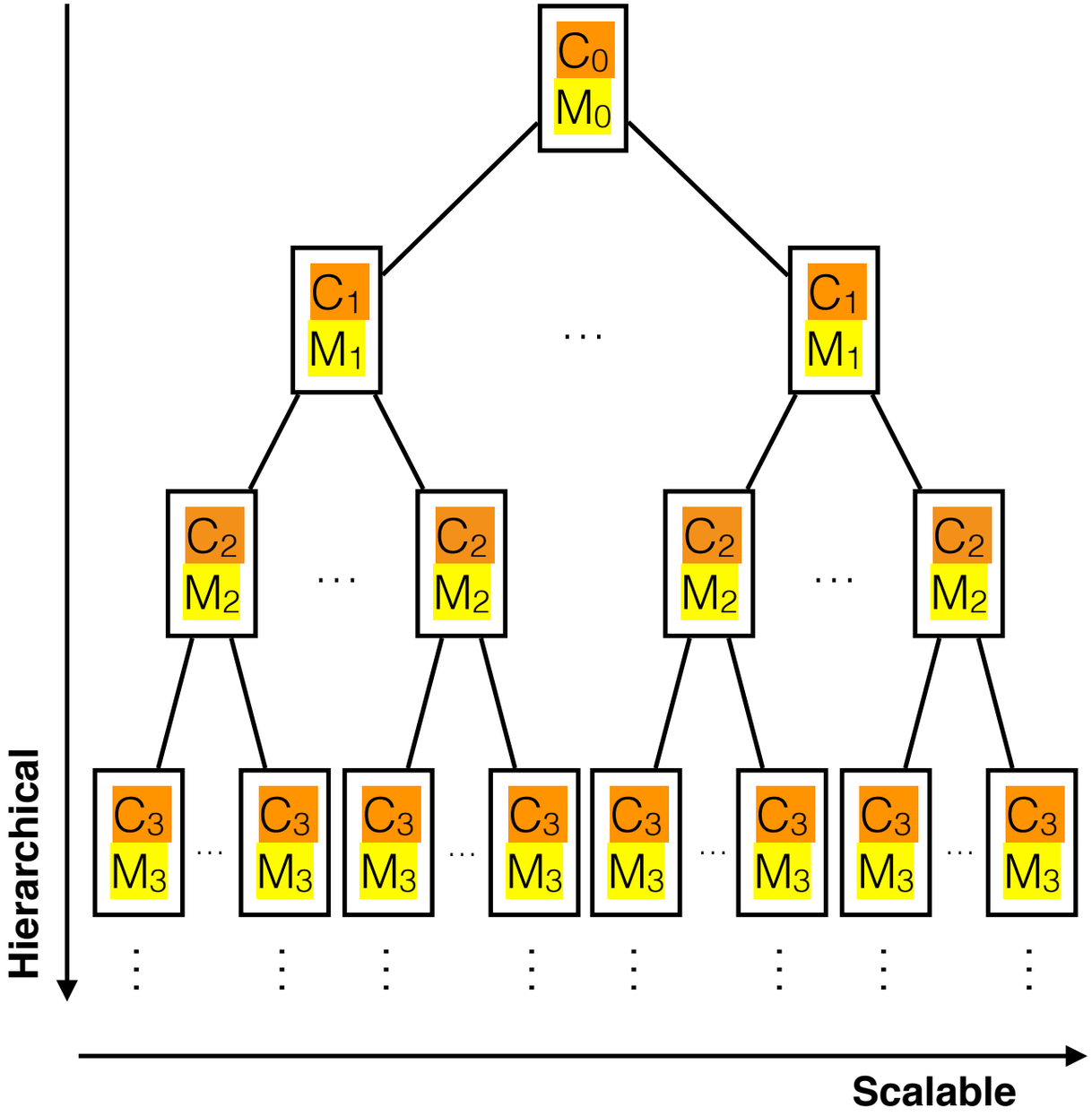
```
TC = md_hom( *, (++,…,++ , +,…,+) ) o view(…)
```

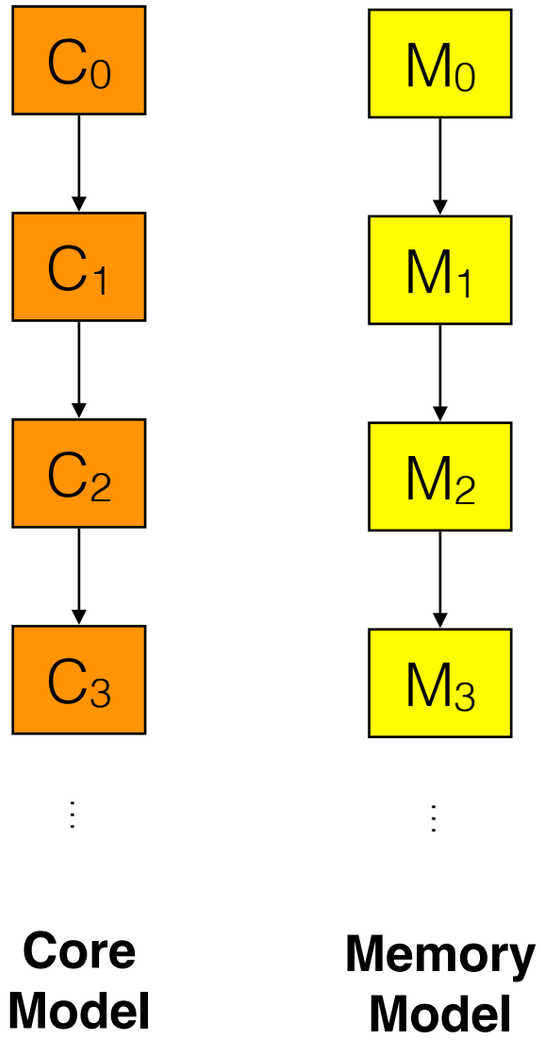
Further examples: MLP, SVM, ECC, …, Mandelbrot, Parallel Reduction, …

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

MDH — Target Machine Model

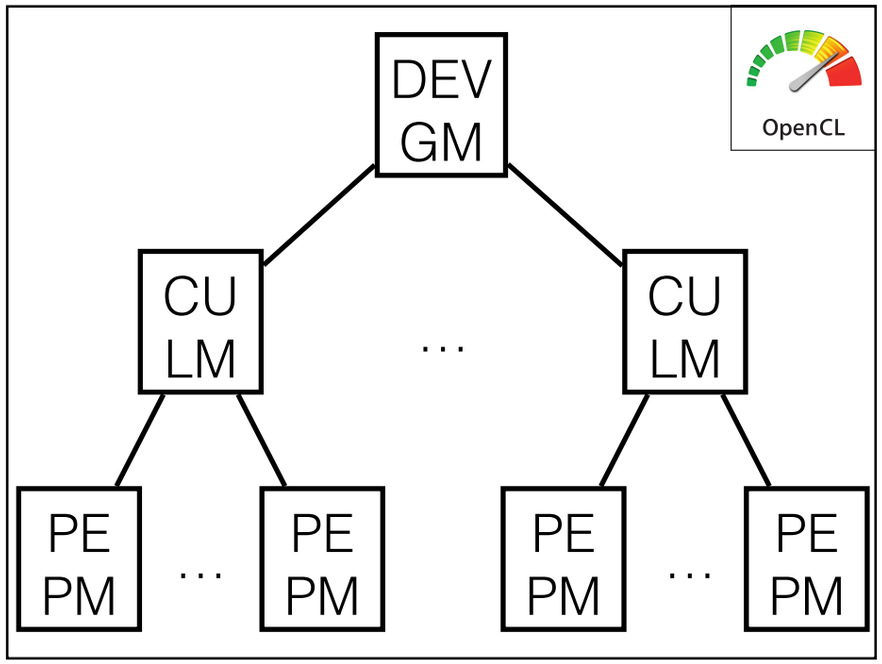
With MDHs, we can target arbitrary ***hierarchical***, ***scalable*** *machine models*:

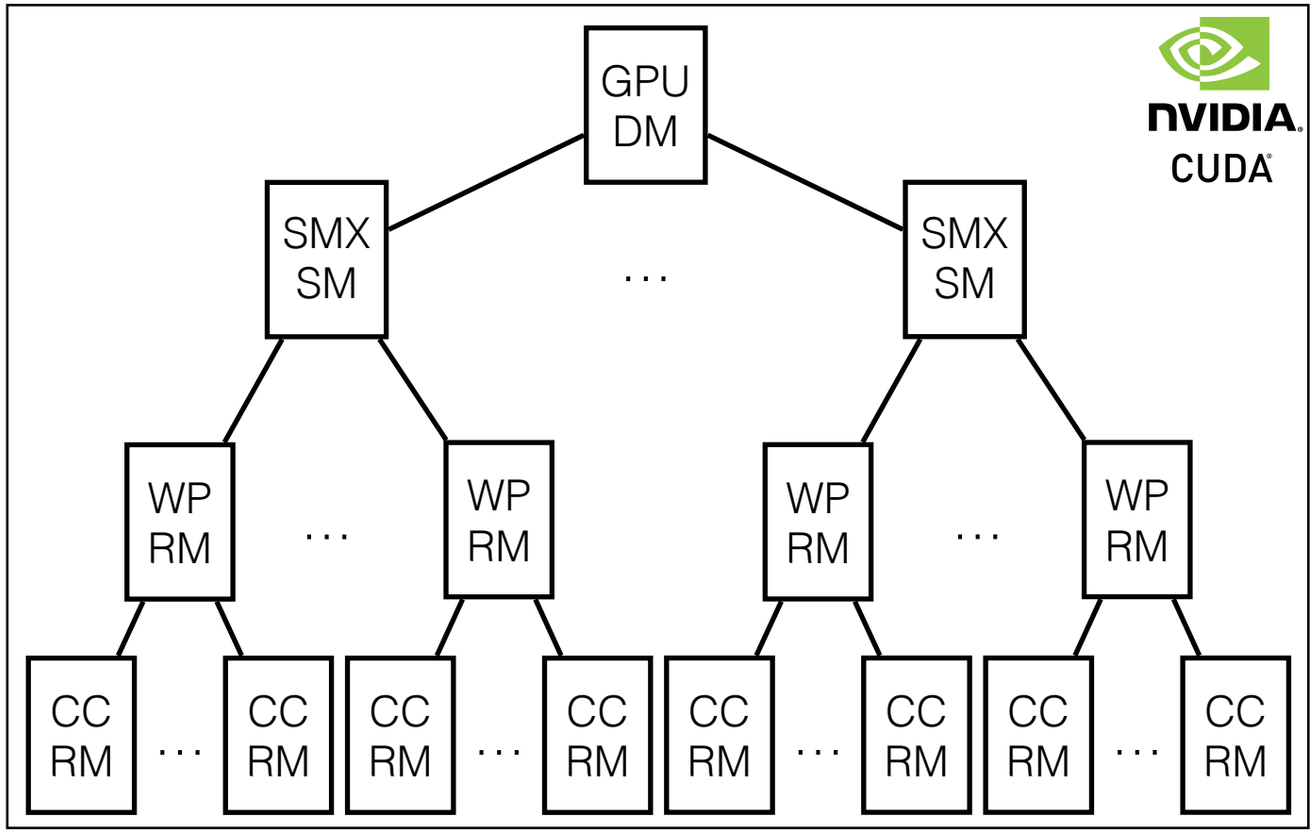


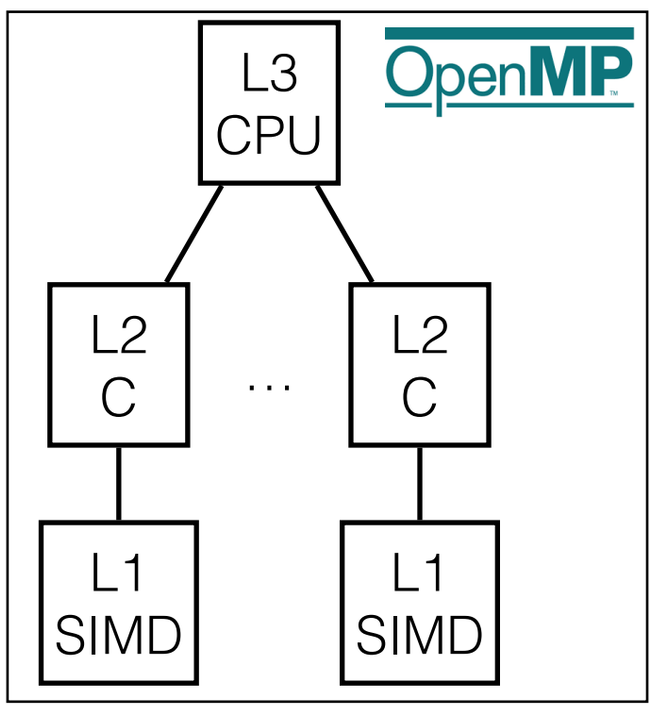


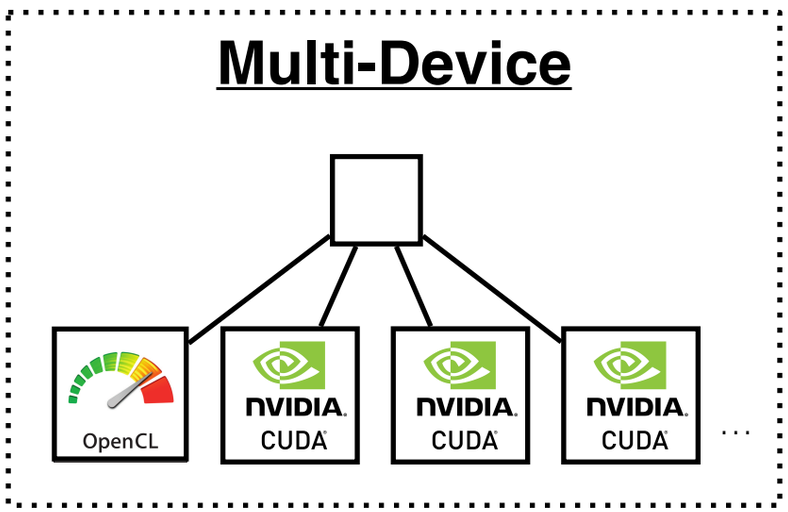
MDH — Target Machine Model

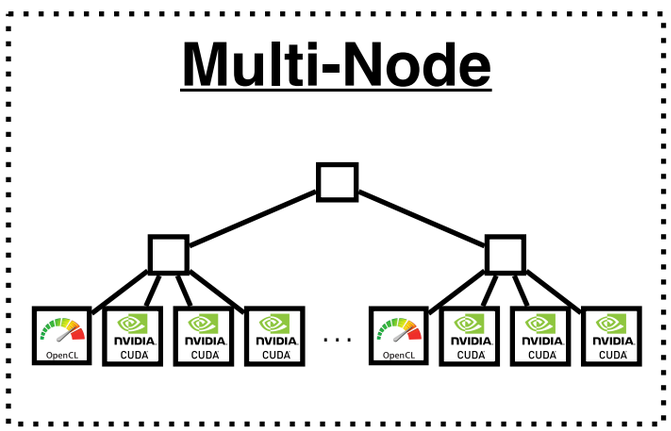
Examples of such *machine models*:





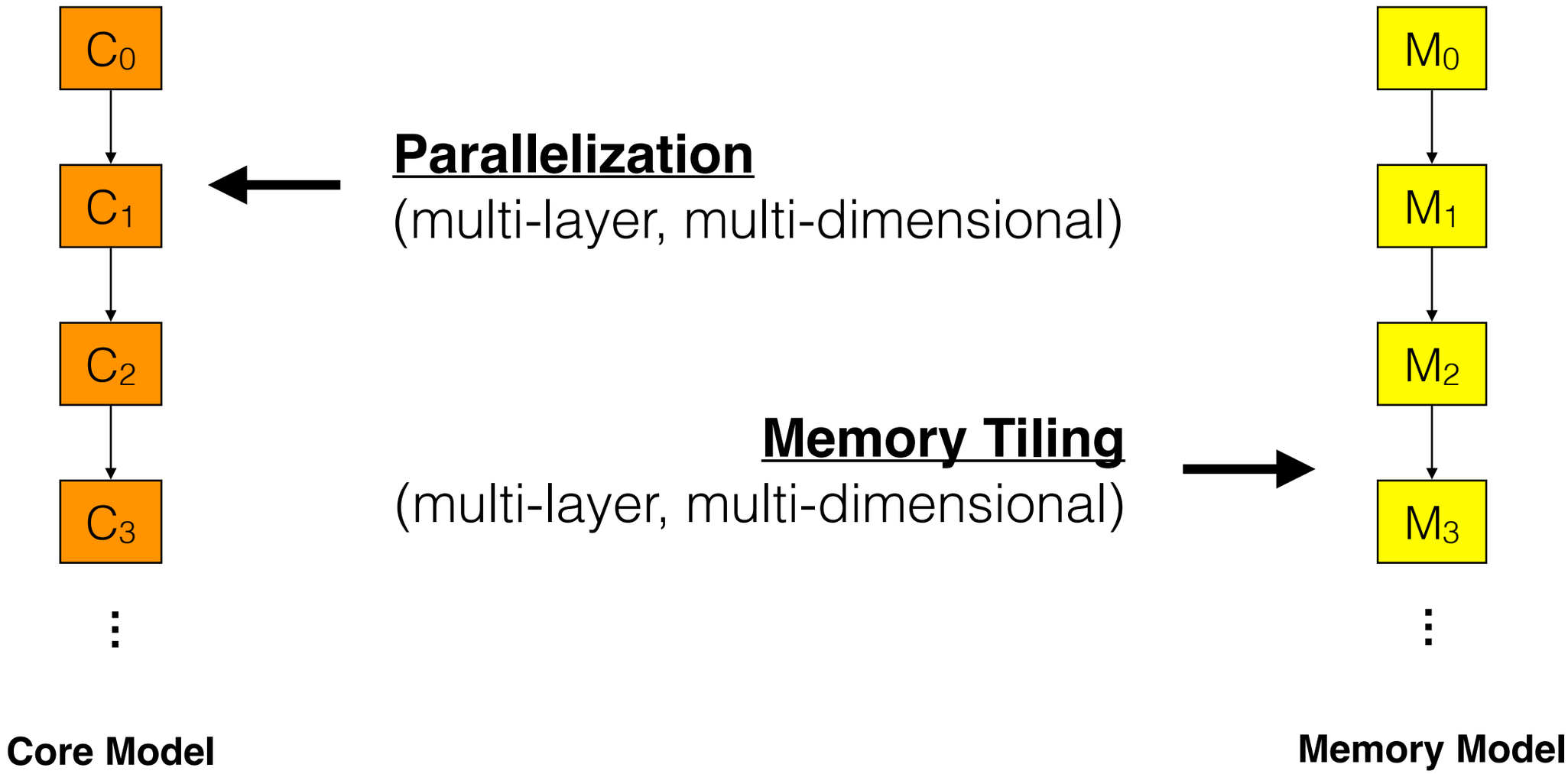






Code Generation for MDHs

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



In the following: Explain code generation at example of OpenCL [1].

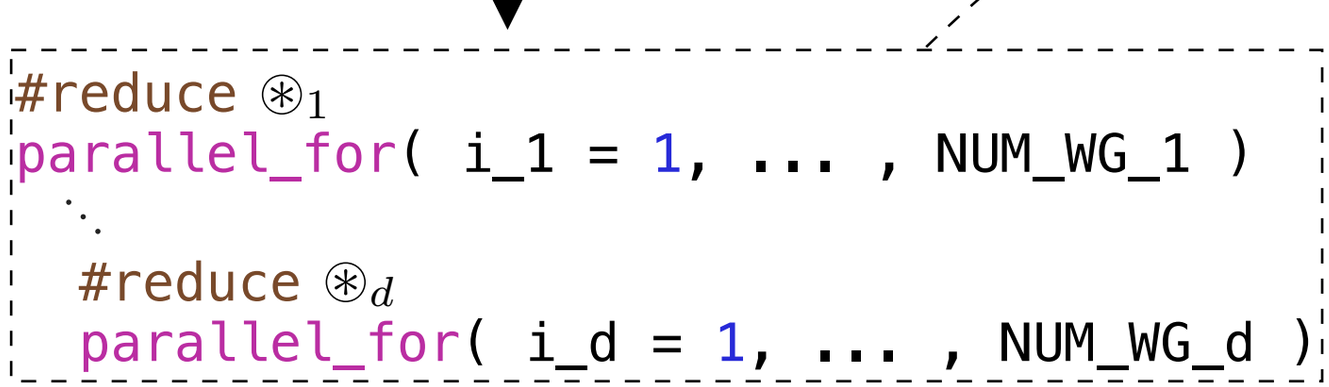
[1] Rasch, Schulze, et al. “Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms.”, *PACT’19.*

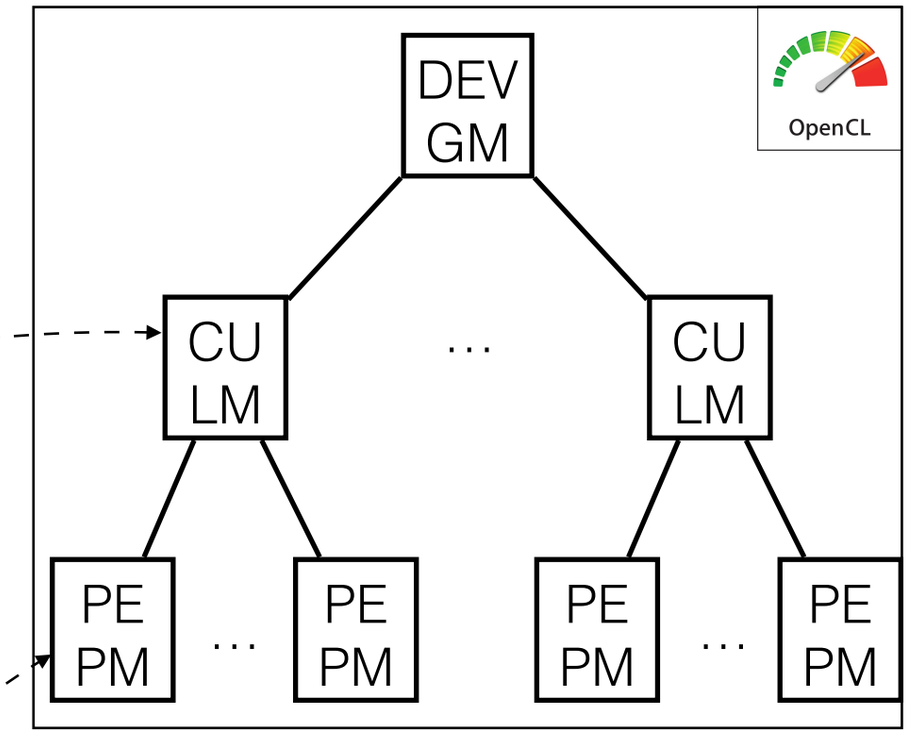
Code Generation for MDHs

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





- 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**.

Code Generation for MDHs

2. Memory Tiling (multiple layers, multiple dimensions):

```
#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)
```

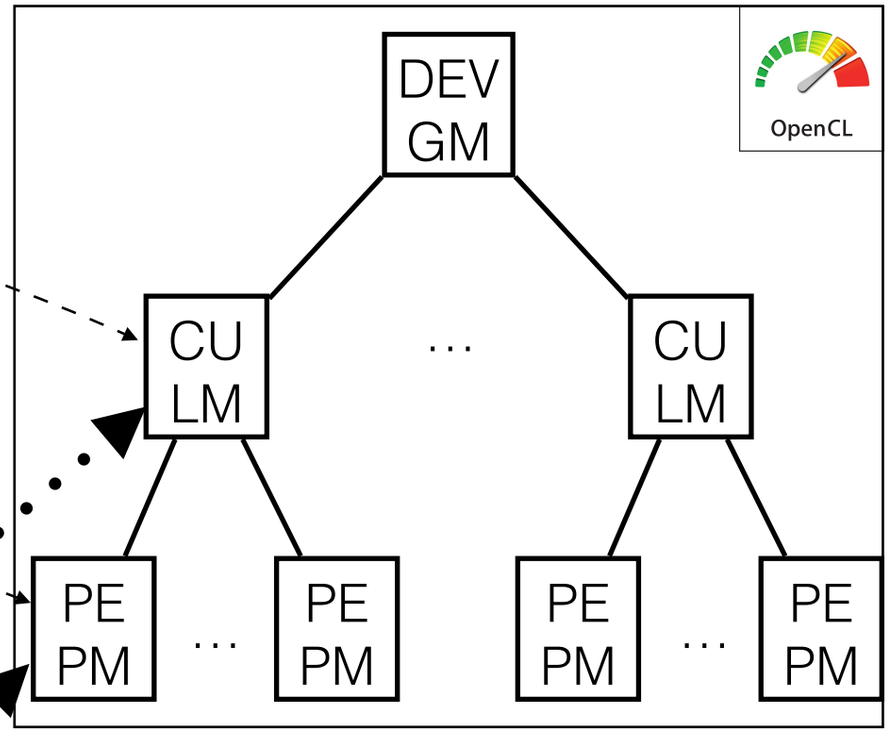
Work-Items

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

LM-Tiles

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

PM-Tiles



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

Code Generation for MDHs

Our OpenCL implementation is generated as **parametrized in performance-critical parameters** (a.k.a. tuning parameters):

No.	Name	Description
1	`NUM_WG`$^{\langle i \rangle}$	number of Work-Groups
2	`NUM_WI`$^{\langle i \rangle}$	number of Work-Items
3	`LT_SIZE`$^{\langle i \rangle}$	local tile size
4	`PT_SIZE`$^{\langle i \rangle}$	private tile size
5	`MEM_INP`$^{\langle LYR,b,i \rangle}$	memory regions for caching input
6	`MEM_RES`$^{\langle LYR,b,i \rangle}$	memory regions for comp. results
7	$\sigma^{\langle LYR \rangle}_{\texttt{arr}\rightarrow\texttt{ocl}}$	mapping array to OpenCL dimensions
8	$\sigma^{\langle LYR,b \rangle}_{\texttt{buff-do}}$	buffer dimension order
9	$\sigma^{\langle LYR \rangle}_{\texttt{mdh-do}}$	MDH dimension order
10	`CMB_RES`	layer to combine results on

`<i>` → dimension ; `<LYR>` → layer ; `<b>` → buffer

All parameters are chosen as optimized for:

→ ***abstract device model;***
→ ***arbitrary MDH;***
→ ***arbitrary input/output characteristics.***

Optimization

Auto-Tuning Framework (ATF)

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

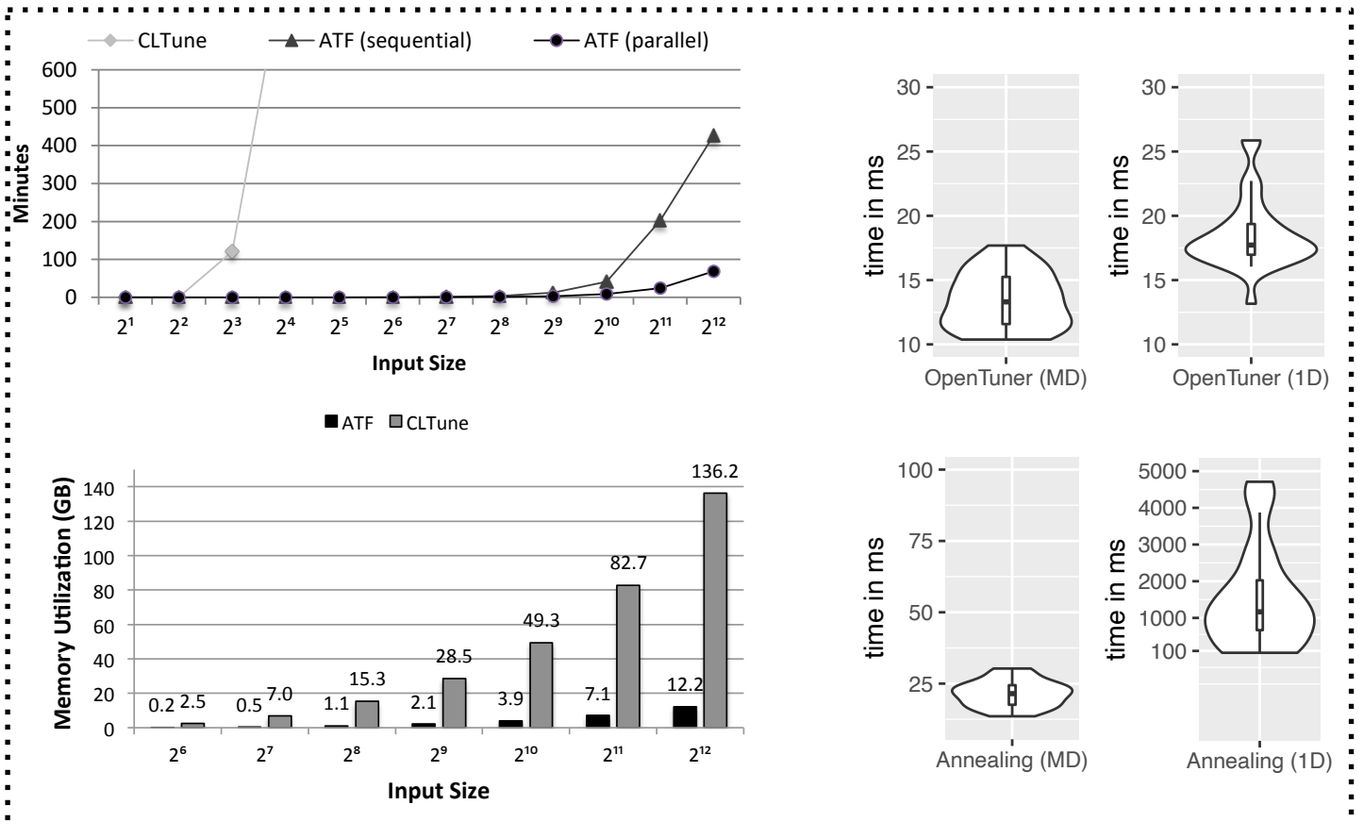
[1] Rasch, et al. “ATF: A Generic, Directive-Based Auto-Tuning Framework”, Concurrency and Computation: Practice and Experience, 2019

ATF - Internals

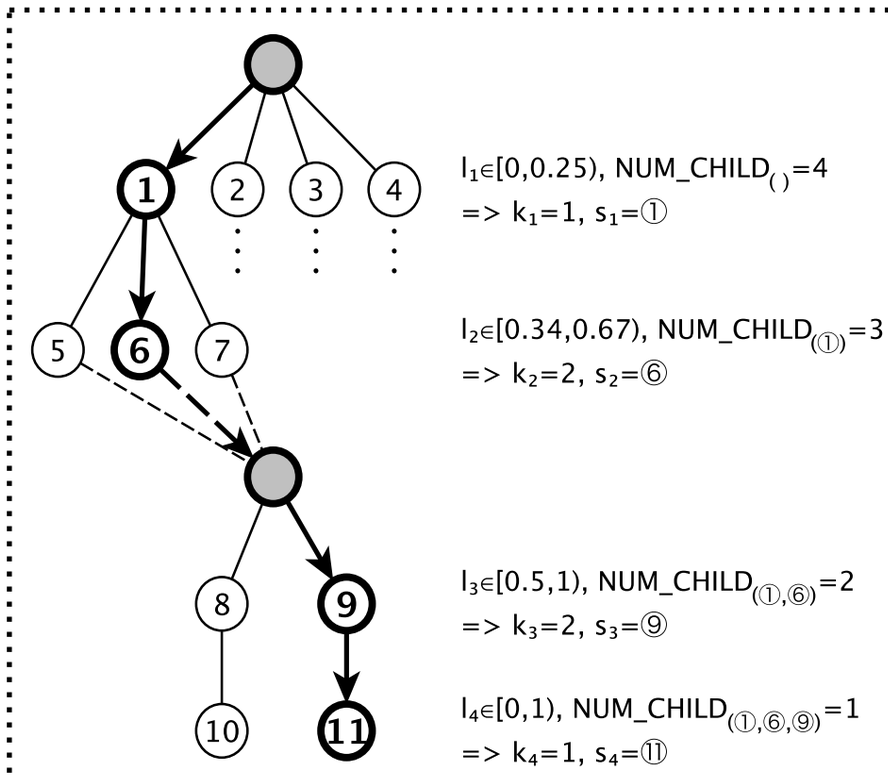
ATF efficiently **generates / stores / explores** the search spaces of **interdependent tuning parameters**:

```
#atf::tp name         /* name         */
         range        /* range        */
         constraint /* constraint */
```

We extend the traditional definition of *tuning parameters* by a ***parameter's constraint***.



ATF **generates** constrained search spaces with high performance (→ **parameter constraints**) and **stores** and **explores** these spaces efficiently (→**chain-of-trees structure**).



We provide a novel ***chain-of-trees*** *search space structure* for interdependent tuning parameters.

Under Review @ TACO

ATF — Usage

ATF usage [1]: We annotate our generated 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 values of parameters

(ATF is also available as C++ [2] / Python (WIP) programming library — for online auto-tuning.)

[1] Rasch, et al. “ATF: A Generic, Directive-Based Auto-Tuning Framework”, Concurrency and Computation: Practice and Experience, 2019
[2] Rasch, et al. “ATF: A Generic Auto-Tuning Framework”, HPCC, 2017

Execution

Execution

We execute our *generated* and *optimized* OpenCL code using our own ***dOCAL*** **[1,2]** framework which:

1. provides **high-level abstractions** for simplifying implementing **OpenCL (and CUDA) 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;

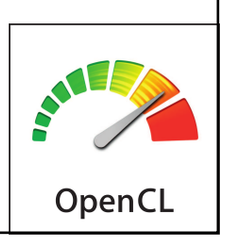
3. enables conveniently executing OpenCL kernels on **remote nodes** (via *Boost.Asio*).

[1] Rasch et al., “dOCAL: High-Level Distributed Programming with OpenCL and CUDA”, *The Journal of Supercomputing, 2019*
[2] Rasch et al., “OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA”, *ICPADS’18*

Execution

Illustration: using dOCAL for executing OpenCL code on GPU (single device, single node)

```
#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 evaluate or **automatically-generated** and **auto-tuned code** using:

Applications

1. Linear Algebra Routines (BLAS): GEMM, GEMV
2. Stencils: Gaussian Convolution 2D, Jacobi 3D
3. Data Mining: Probabilistic Record Linkage
4. Machine Learning: Tensor Contractions

Competitors

- Generic Approaches:
  - Lift: *BLAS **[CGO'17]***, *Stencils **[CGO'18 — Best Paper]***
- Domain/Hardware-Specific Approaches:
  - Intel MKL, NVIDIA cuBLAS: *BLAS*
  - Intel MKL-DNN, NVIDIA cuDNN: *Stencils*
  - EKR: *Data Mining **[HFSL'13]***
  - COGENT, Facebook Tensor Comprehensions: *Machine Learning **[CGO'19, TACO'19]***

Architectures

- CPU: Intel Xeon multi-core E5-2640
- GPU: NVIDIA Tesla V100 SMX2-16GB

Data Sets

- RW: Real-world sizes, e.g., from deep learning
- PC: Sizes that are preferable for our competitors

Experimental Results

Linear Algebra

CPU	GEMM		GEMV	
	RW	PC	RW	PC
Lift [1]	fails	3.04	1.51	1.99
MKL	4.22	0.74	1.05	0.87

GPU	GEMM		GEMV	
	RW	PC	RW	PC
Lift [1]	4.33	1.17	3.52	2.98
cuBLAS	2.91	0.83	1.03	1.00

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

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 both generic- and also domain-specific approaches.

Rasch, Schulze, et al. "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***

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)**.

Experimental Results

Summary:

- **Lift:** Speedups of up to
  - **1.02x - 5.96x**, including own **Stencil** samples presented at **[CGO'18 — Best Paper]**;
  - **1.17x - 4.33x**, including own **BLAS** samples presented at **[CGO'17]**.

- **COGENT / Facebook Tensor Comprehensions:** Speedups of
  - **1.19x - 2.89x** on own **Tensor Contraction** samples **[CGO'19, TACO'20]**.

- **Intel MKL / NVIDIA cuBLAS:** Speedups of
  - **0.74x-1.00x** for **BLAS** on **their preferable sizes**;
  - **1.03x-4.22x** for **BLAS** on **real-world input sizes**.

- **Intel MKL-DNN / NVIDIA cuDNN:** Speedups of
  - **14.31x-19.11x** for **Stencils** on **PC sizes**;
  - **3.78x-6.99x** for **Stencils** on **real-world input sizes**.

Experimental Results

Our better results are because:

We generate OpenCL code that is auto-tunable for any combination of a device and input/output characteristics.

Lift

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

Intel MKL/MKL-DNN & NVIDIA cuBLAS/cuDNN

Rely on hand-optimized kernels optimized for average high performance over input sizes.

EKR Java

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

Tensor Comprehensions & COGENT

No parallelization in summation dimensions; use smaller optimizations spaces.

Conclusion

We provide a holistic approach to automatic code *generation*, *optimization*, and *execution* toward **performance**, **portability**, and **productivity** for data-parallel computations:

1. MDHs **uniformly cover data-parallel computations** (BLAS, Stencil, PRL, TC, …).
2. MDHs can be computed with **high performance** (speedups up to 4x over domain-specific, hand-optimized approaches).
3. MDHs are **portable over architectures** and **input/output characteristics**.
4. MDHs can be **conveniently implemented** using our DSL for MDHs.

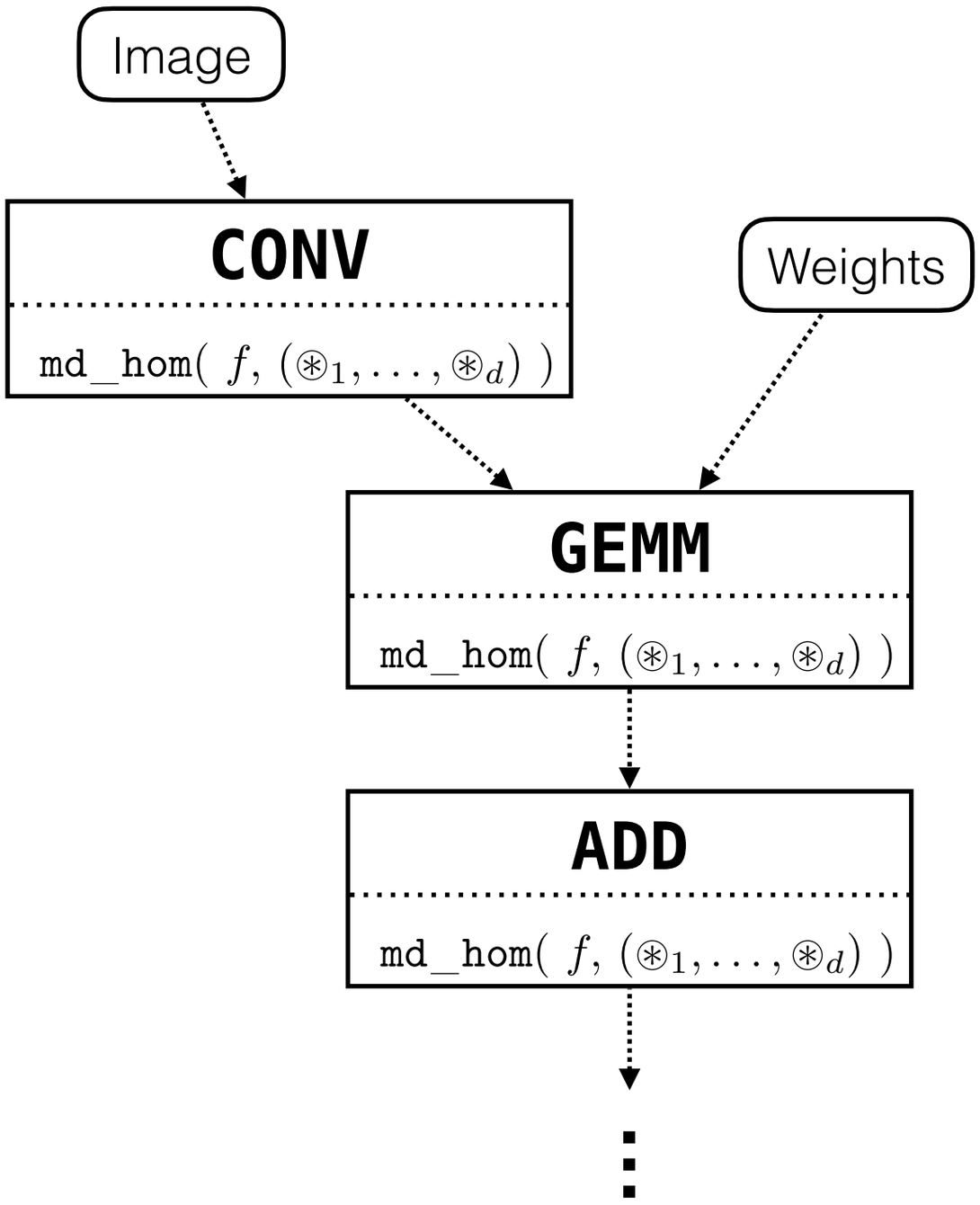
Moreover:

- Our **Auto-Tuning Framework (ATF)** is a **general-purpose approach** that **supports auto-tuning** of **interdependent tuning parameters.**
- We provide our **dOCAL** framework — it enables **conveniently executing OpenCL (and CUDA) kernels on multi-device/multi-node systems**, and it automatically performs **host code optimizations** (e.g., overlapping data transfers with computations).

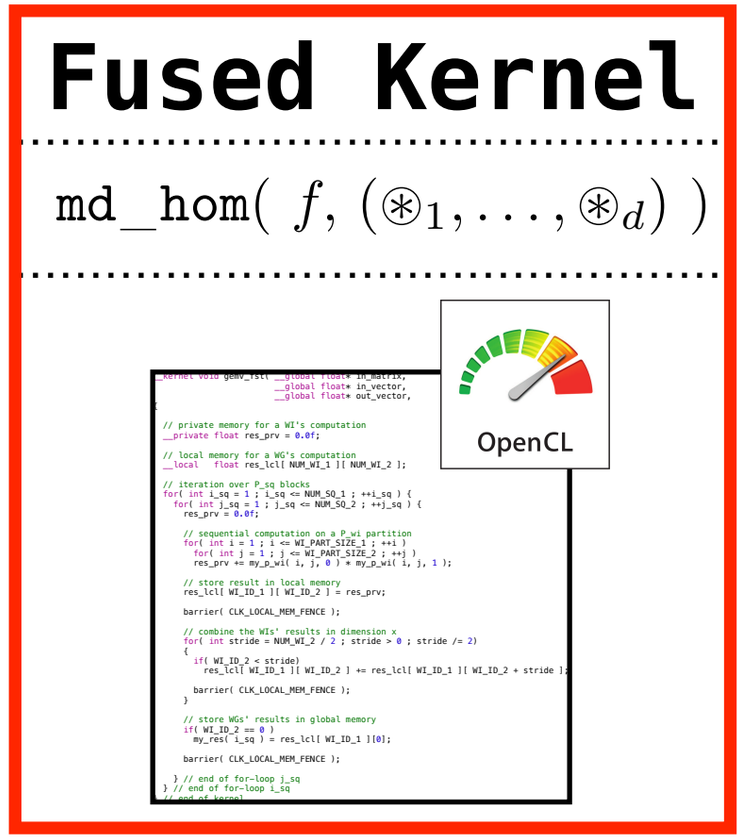
Current WIP / Future Work

WIP Results

Code generation for **composed `md_hom` expressions** (e.g. as in deep learning graphs):



Exploiting **`md_hom`** representation for optimizing composed computations



Current WIP / Future Work

WIP Results

Simplifying code generation via *Directives (rather than using DSL)*:

```
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

RO Cm

NVIDIA CUDA

. . .

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

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

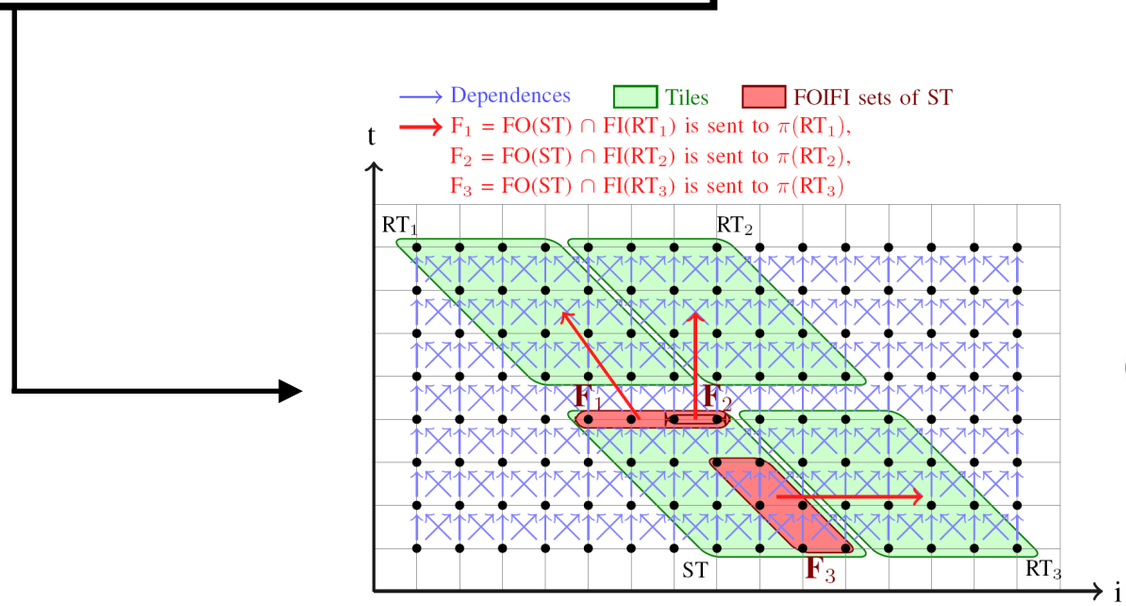
Annotating sequential C-Code with simple *MDH directives* *(similarly as in OpenMP/OpenACC).*

Current WIP / Future Work

WIP Results

Automatic parallelization of sequential C programs:

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

   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 ];
      }

  // ...
}
```



Polyhedral Model

MDH
Code Generation

. . .

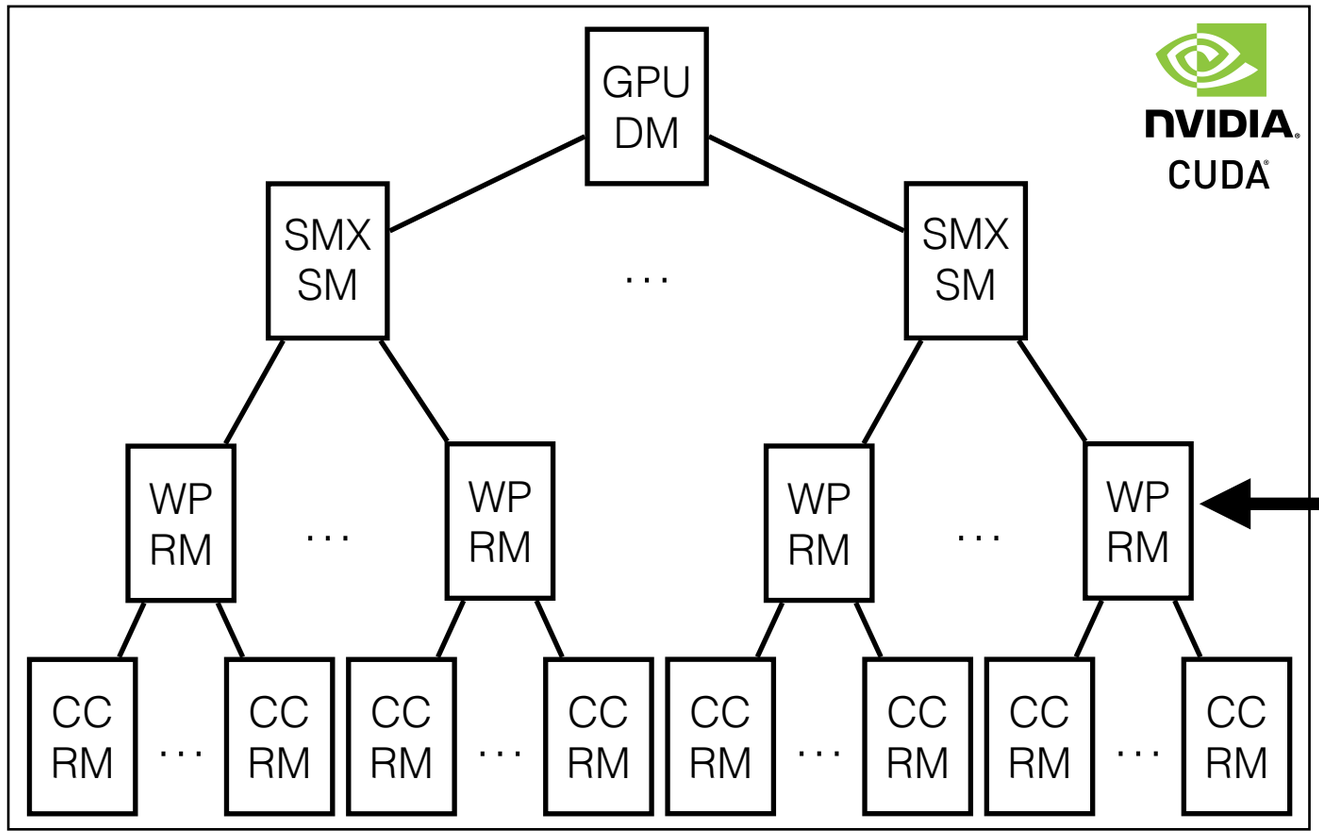
Polyhedral Front End for MDH Code Generation

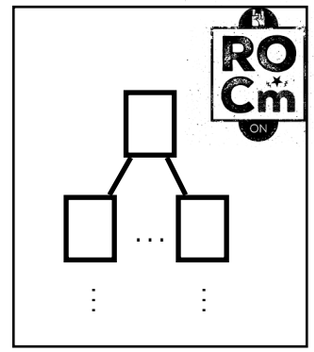
Rasch, Schulze, Gorlatch. md_poly: A Performance-Portable Polyhedral Compiler Based on Multi-Dimensional Homomorphisms. IMPACT'20 (WIP)

Current WIP / Future Work

WIP Results

Further backends:

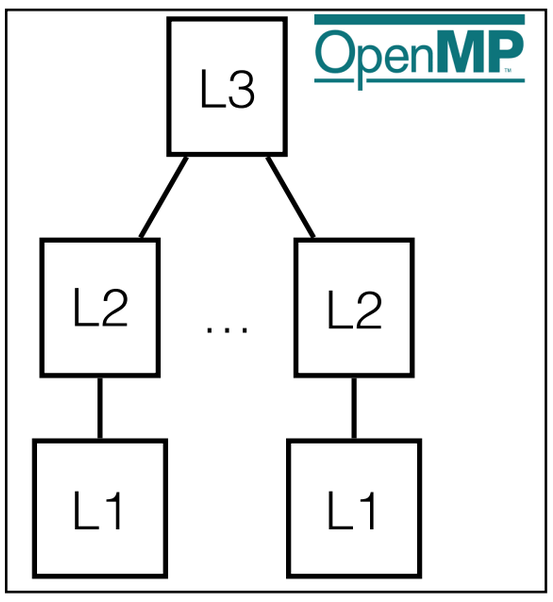




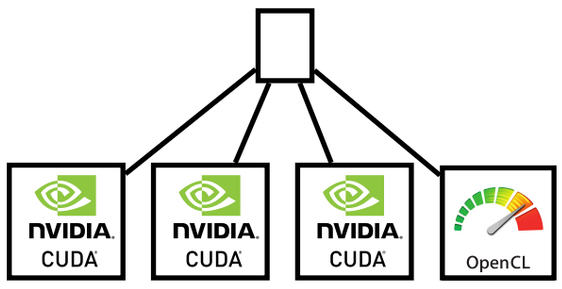
- Tensor Cores
- Shuffle Operations

Assembler Backends

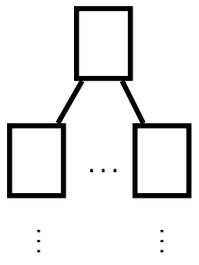
(e.g., NVIDIA PTX, Intel x86-64, …)



Multi-Device



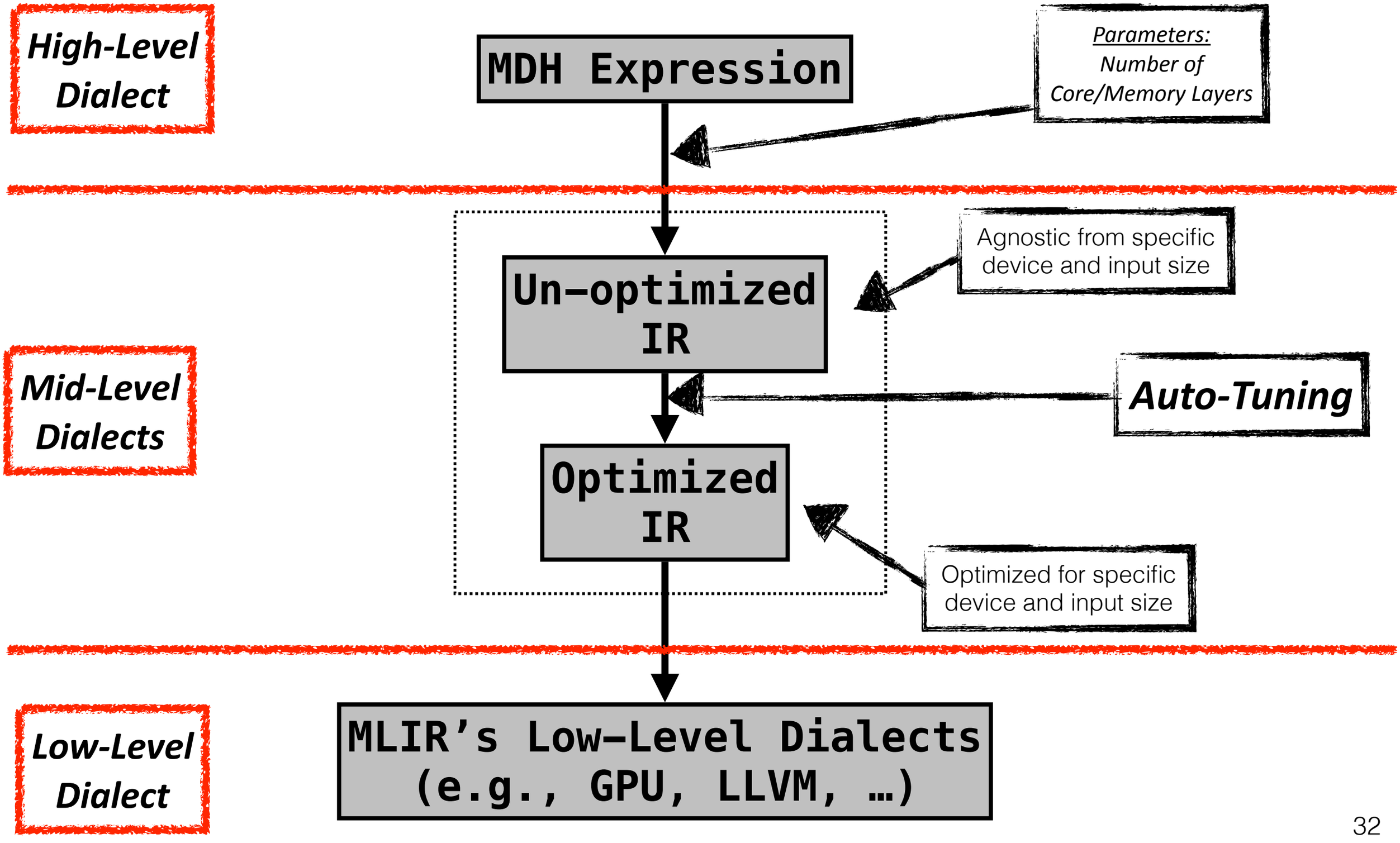
Multi-Node



Current WIP / Future Work

MLIR

Integrating MDH in MLIR:



Current WIP / Future Work

WIP Results

Analyze efficiency of our approach for further case studies:

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

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

Benchmark	Description
`2mm`	2 Matrix Multiplications (D=A.B; E=C.D)
`3mm`	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
`adi`	Alternating Direction Implicit solver
`atax`	Matrix Transpose and Vector Multiplication
`bicg`	BiCG Sub Kernel of BiCGStab Linear Solver
`cholesky`	Cholesky Decomposition
`correlation`	Correlation Computation
`covariance`	Covariance Computation
`doitgen`	Multiresolution analysis kernel (MADNESS)
`durbin`	Toeplitz system solver
`dynprog`	Dynamic programming (2D)
`fdtd-2d`	2-D Finite Different Time Domain Kernel
`fdtd-apml`	FDTD using Anisotropic Perfectly Matched Layer
`gauss-filter`	Gaussian Filter
`gemm`	Matrix-multiply C=alpha.A.B+beta.C
`gemver`	Vector Multiplication and Matrix Addition
`gesummv`	Scalar, Vector and Matrix Multiplication
`gramschmidt`	Gram-Schmidt decomposition
`jacobi-1D`	1-D Jacobi stencil computation
`jacobi-2D`	2-D Jacobi stencil computation
`lu`	LU decomposition
`ludcmp`	LU decomposition
`mvt`	Matrix Vector Product and Transpose
`reg-detect`	2-D Image processing
`seidel`	2-D Seidel stencil computation
`symm`	Symmetric matrix-multiply
`syr2k`	Symmetric rank-2k operations
`syrk`	Symmetric rank-k operations
`trisolv`	Triangular solver
`trmm`	Triangular matrix-multiply

PolyBench

Questions?

Experimental Results

		GPU			
		GEMM		GEMV	
		RW	PC	RW	PC
MDH	GF/s	*99*	*8996*	*415*	*440*
	SP	***1.00***	***1.00***	***1.00***	***1.00***
Lift	GF/s	23	7713	118	148
	SP	**4.33**	**1.17**	**3.52**	**2.98**
cuBLAS	GF/s	34	10820	404	438
	SP	**2.91**	**0.83**	**1.03**	**1.00**
cuBLASLt	GF/s	84	36474	N/A	N/A
	SP	**1.18**	**0.25**	**N/A**	**N/A**

		CPU			
		GEMM		GEMV	
		RW	PC	RW	PC
MDH	GF/s	*31*	*339*	*23*	*30*
	SP	***1.00***	***1.00***	***1.00***	***1.00***
Lift	GF/s	fails	111	15	15
	SP	**fails**	**3.04**	**1.51**	**1.99**
MKL	GF/s	7	456	22	35
	SP	**4.22**	**0.74**	**1.05**	**0.87**
MKL-JIT	GF/s	23	N/A	N/A	N/A
	SP	**1.37**	**N/A**	**N/A**	**N/A**

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

		Tensor Contractions								
		RW 1	RW 2	RW 3	RW 4	RW 5	RW 6	RW 7	RW 8	RW 9
MDH	GF/s	5771	5322	7824	5739	5500	5029	6760	6628	6777
	SP	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***
COGENT	GF/s	4573	4584	3697	4631	4647	3685	4573	4604	3670
	SP	**1.26**	**1.16**	**2.12**	**1.24**	**1.18**	**1.36**	**1.48**	**1.44**	**1.85**
F-TC	GF/s	4834	2660	5476	1987	4088	3273	5426	3286	4561
	SP	**1.19**	**2.00**	**1.43**	**2.89**	**1.35**	**1.54**	**1.25**	**2.02**	**1.49**

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

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

		Probabilistic Record Linkage					
		2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
MDH	SP	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***	***1.00***
EKR	SP	**1.87**	**2.06**	**4.98**	**13.86**	**28.34**	**39.36**

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

		GPU			
		Gaussian (2D)		Jacobi (3D)	
		RW	PC	RW	PC
MDH	GF/s	*386*	*4195*	*1137*	*1005*
	SP	***1.00***	***1.00***	***1.00***	***1.00***
Lift	GF/s	165	3852	995	984
	SP	**2.33**	**1.09**	**1.14**	**1.02**
cuDNN	GF/s	102	219	N/A	N/A
	SP	**3.78**	**19.11**	**N/A**	**N/A**

		CPU			
		Gaussian (2D)		Jacobi (3D)	
		RW	PC	RW	PC
MDH	GF/s	*73*	*208*	*49*	*60*
	SP	***1.00***	***1.00***	***1.00***	***1.00***
Lift	GF/s	15	35	25	24
	SP	**4.90**	**5.96**	**1.94**	**2.49**
MKL-DNN	GF/s	10	15	N/A	N/A
	SP	**6.99**	**14.31**	**N/A**	**N/A**

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

MDH — Examples

Gaussian 2D (standard)

```
md_hom( G_func, (++,++) ) o view( image )( i,j )( image[i,j] ,..., image[i+4,j+4] )

G_func( image_0_0 ,..., image_4_4 ) = ( 2 * image_0_0 + 4 * image_0_1 + ... + 2 * image_4_4 )
```

Gaussian 2D (optimized)

```
md_hom( *, (++,++) ) o view( image, filter )( i,j,r,s )( image[ i+r, j+s ], filter[ r,s ] )
```

Jacobi 3D

```
md_hom( J_func, (++,++,++, )) o view( image )( i,j,k )( image[i+1,j+1,k+2] ,..., image[i+1,j+1,k+1] )

J_func( image_1_1_2, ..., image_1_1_1 ) = 0.161f * image_1_1_2 + ... - 1.67f * image_1_1_1
```

Tensor Contraction

```
md_hom( *, (++,++,++,++,++,++,+,) ) o view( lhs, rhs )( a,b,c,d,e,f,g )( lhs[g,d,a,b], rhs[e,f,g,c] )
```

MCC (standard)

```
md_hom( f, (++,++,++,++,+) ) o
view( images, filter )( n,k,p,q,c )( images[n,c,p,q], ..., images[n,c,p+2,q+2], filter[k,c,0,0], ..., filter[k,c,2,2] )

f( images_0_0 ,..., images_2_2 , filter_0_0 ,..., filter_2_2) = filter_0_0 * images_0_0 + ... + filter_2_2 * images_2_2
```

MCC (optimized)

```
md_hom( *, (++,++,++,++,+,+,+) ) o view( images, filter )( n,k,p,q,c,r,s )( images[n,c,p+r,q+s], filter[k,c,r,s] )
```