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
Observation:

Applications

- Linear Algebra (BLAS)
  - GEMM
  - GEMV
  - DOT
  - ... 152 routines!

- Tensor Contractions
  - ...

- Stencils
  - ...

Architectures

- NVIDIA TESLA
- TITAN X
- ARM
- Intel Xeon Phi
- IBM
- AMD GRAPHICS

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**.

- **OpenMP**
- **CUDA**
- **ROCm**

### Dimensions

<table>
<thead>
<tr>
<th>dimensions</th>
<th>$C = AB$</th>
<th>$C = A^T B^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
<td>k</td>
</tr>
<tr>
<td>large</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td>large</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>large</td>
<td>small</td>
<td>large</td>
</tr>
<tr>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
<tr>
<td>small</td>
<td>small</td>
<td>large</td>
</tr>
<tr>
<td>small</td>
<td>small</td>
<td>small</td>
</tr>
</tbody>
</table>
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:

1. **Generation**
   
   **High-level DSL expression**
   
   \[ \text{md\_hom}( f, (\circ_1, \ldots, \circ_k) ) \]

2. **Optimization**
   
   **Generic program code**
   
   **Executable program code**
   
   (auto-tuning)

3. **Execution**
   
   Different architectures and input sizes

---

*Note: The diagram illustrates the process flow from high-level DSL expression to executable code for different architectures and input sizes.*
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* $\otimes_1, \ldots, \otimes_d : T' \times T' \to T'$, such that for each $k \in [1, d]$ and arbitrary, concatenated input MDA $a \oplus_k b$:

$$h( a \oplus_k b ) = h(a) \otimes_k h(b)$$

**Definition: [ md_hom ]**

We write

$$\text{md_hom}( f, (\otimes_1, \ldots, \otimes_d) )$$

for the unique $d$-dimensional homomorphism with combine operators $\otimes_1, \ldots, \otimes_d$ and action $f$ on singleton arrays.

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: \( i,j,k \mapsto (A[i,k], B[k,j]) \)

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**

\[
\text{GEMM} = \text{md.hom}( *, (++, ++, +) ) \circ \text{view}( A, B )( i,j,k )( A[i,k], B[k,j] )
\]

\[
\text{GEMV} = \text{md.hom}( *, (++, +) ) \circ \text{view}( A, B )( i, k )( A[i,k], B[k] )
\]

\[
\text{DOT} = \text{md.hom}( *, ( + ) ) \circ \text{view}( A, B )( k )( A[k], B[k] )
\]

- **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

\[
\text{GEMM} = \text{md\_hom}( *, (++, ++, +) ) \circ \text{view}( A, B )( i, j, k )( A[i,k], B[k,j] )
\]
\[
\text{GEMV} = \text{md\_hom}( *, (++, +) ) \circ \text{view}( A, B )( i, k )( A[i,k], B[k] )
\]
\[
\text{DOT} = \text{md\_hom}( *, (+) ) \circ \text{view}( A, B )( k )( A[k], B[k] )
\]

### Stencil Computations

\[
\text{Gaussian\_2D} = \text{md\_hom}( \text{G\_func}, (++,++) ) \circ \text{view}(\ldots)
\]
\[
\text{Jacobi\_3D} = \text{md\_hom}( \text{J\_func}, (++;++;++) ) \circ \text{view}(\ldots)
\]

### Data Mining

\[
\text{PRL} = \text{md\_hom}( \text{weight}, (++, \text{max}) ) \circ \text{view}(\ldots)
\]

### Machine Learning

\[
\text{TC} = \text{md\_hom}( *, (++,\ldots,++ , +,\ldots,+) ) \circ \text{view}(\ldots)
\]

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

Our DSL needs only the patterns `md_hom(\ldots)` and `view(\ldots)`
We generate code for MDHs targeting **hierarchical, scalable** machine models:

MDH — Target Machine Model
Examples of such *machine models*:
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):

   - 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.

\[
\begin{align*}
    &\text{MDH pseudocode for} \\
    &\text{md\_hom}(f, (\oplus_1, \ldots, \oplus_d)) \\
    \end{align*}
\]
Optimization of MDHs

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

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.
**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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform Model</td>
<td></td>
</tr>
<tr>
<td>NUM_WG_i</td>
<td>number of Work-Groups</td>
</tr>
<tr>
<td>NUM_WI_i</td>
<td>number of Work-Items</td>
</tr>
<tr>
<td>Memory Model</td>
<td></td>
</tr>
<tr>
<td>LM_TL_SIZE_i</td>
<td>local memory tile size</td>
</tr>
<tr>
<td>PM_TL_SIZE_i</td>
<td>private memory tile size</td>
</tr>
<tr>
<td>LM_CACHING</td>
<td>copying tiles to local memory on/off</td>
</tr>
<tr>
<td>PM_CACHING</td>
<td>copying tiles to private memory on/off</td>
</tr>
<tr>
<td>Platform→Memory Model</td>
<td></td>
</tr>
<tr>
<td>OCL_MDA_MAPPING</td>
<td>mapping OCL to MDA dimensions</td>
</tr>
<tr>
<td>COMB_OP_ORDER</td>
<td>order of combine operators</td>
</tr>
<tr>
<td>COMB_WG_RES_AFTER</td>
<td>when to combine WG results</td>
</tr>
<tr>
<td>COMB_WI_RES_AFTER</td>
<td>when to combine WI results</td>
</tr>
</tbody>
</table>

All parameters are chosen independently of the target MDH, architecture, and input size!
We use our **Auto-Tuning Framework (ATF)** [1] to automatically choose optimized values of our performance-critical parameters.

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Domain-specific auto-tuning</th>
<th>OpenTuner</th>
<th>CLTune</th>
<th>ATF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arbitrary Programming Language</td>
<td></td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Arbitrary Application Domain</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Arbitrary Tuning Objective</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Arbitrary Search Technique</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Interdependent Parameters</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Large Parameter Ranges</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Directive-Based Auto-Tuning</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td>Automatic Cost Function Generation</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Automatic Performance Tuning

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

```cpp
#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 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).

Execution

Illustration: using dOCAL for executing OpenCL kernel on GPU.

```cpp
#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)
- NVIDIA 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).

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

Experimental Results

3. Tensor Contractions:

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

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):

Image

\[ \text{CONV} \]

\[
\text{md\_hom}( f, (\ast_1, \ldots, \ast_d) )
\]

Weights

\[ \text{GEMM} \]

\[
\text{md\_hom}( f, (\ast_1, \ldots, \ast_d) )
\]

\[ \text{ADD} \]

\[
\text{md\_hom}( f, (\ast_1, \ldots, \ast_d) )
\]

Fused Kernel

\[
\text{md\_hom}( f, (\ast_1, \ldots, \ast_d) )
\]

Exploiting \text{md\_hom} representation for fusing (better locality)
Current/Future Work

Further backends:

- Shuffle Operations
- Tensor Cores
  → comparable performance to cuBLAS

Assembler Backends
(e.g., NVIDIA PTX, Intel x86-64, …)
Parallelizing sequential C programs via MDH approach:

```c
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 ];
                }
    // ...
}
```

**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:

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

### Parboil

![Parboil Logo](image)

### TensorFlow

![TensorFlow Logo](image)

### Caffe 2

![Caffe 2 Logo](image)

### Rodinia

![Rodinia Logo](image)

---

**TABLE 1**

<table>
<thead>
<tr>
<th>Application / Kernel</th>
<th>Dwarf</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means</td>
<td>Dense Linear Algebra</td>
<td>Data Mining</td>
</tr>
<tr>
<td>Needleman-Wunsch</td>
<td>Dynamic Programming</td>
<td>Bioinformatics</td>
</tr>
<tr>
<td>HotSpot*</td>
<td>Structured Grid</td>
<td>Physics Simulation</td>
</tr>
<tr>
<td>Back Propagation*</td>
<td>Unstructured Grid</td>
<td>Pattern Recognition</td>
</tr>
<tr>
<td>SRAD</td>
<td>Structured Grid</td>
<td>Image Processing</td>
</tr>
<tr>
<td>Leukocyte Tracking</td>
<td>Structured Grid</td>
<td>Medical Imaging</td>
</tr>
<tr>
<td>Breath-First Search*</td>
<td>Graph Traversal</td>
<td>Graph Algorithms</td>
</tr>
<tr>
<td>Stream Cluster*</td>
<td>Dense Linear Algebra</td>
<td>Data Mining</td>
</tr>
<tr>
<td>Similarity Scores*</td>
<td>MapReduce</td>
<td>Web Mining</td>
</tr>
</tbody>
</table>
Questions?