Array Programming via Multi-Dimensional Homomorphisms

Ari Rasch
a.rasch@uni-muenster.de
University of Muenster, Germany

Richard Schulze
r.schulze@uni-muenster.de
University of Muenster, Germany

Sergei Gorlatch
gorlatch@uni-muenster.de
University of Muenster, Germany

1 Introduction

The approach of *Multi-Dimensional Homomorphisms (MDH)* [5] offers a formalism for array programming: data-parallel computations on arrays, such as linear algebra routines and stencil computations, are expressed via higher-order functions (a.k.a. *patterns* or *skeletons* in programming terminology [3]), and optimized program code is fully automatically generated from these functions in the MDH approach [6], e.g., in CUDA for GPU or OpenCL for CPUs.

In this extended abstract, we present the three major contributions of the MDH approach[1]: 1) a *high-level program representation* for expressing data-parallel array computations in a convenient, hardware- and optimization-agnostic manner, based on formally defined higher-order functions; 2) a *low-level program representation* to formally reason about optimizations and from which we can straightforwardly generate executable program code (e.g., in CUDA or OpenCL); 3) a *fully automatic process* for lowering a program expressed in MDH's high-level program representation to an hardware-optimized MDH program in its low-level representation, based on auto-tuning [7]. Our preliminary experimental results confirm that MDH achieves higher performance on GPU and CPU than well-performing state-of-practice approaches, including hand-optimized vendor libraries from NVIDIA and Intel.

2 High-Level Program Representation

We explain MDH's high-level program representation by presenting and discussing the example of *Matrix-Vector Multiplication (*`MatMul`*)*, shown in Figure 1.

```
MatVec<T∈TYPE|I,K∈ℕ> :=
  out_view<T>( w:(i,k)↦(i) ) ∘
   md_hom<I,K>( *, (++,+) ) ∘
    inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

Figure 1. MDH high-level expression for `MatVec`

Computation `MatVec` takes as input a matrix $M \in T^{I\times K}$ and vector $v \in T^K$ of arbitrary scalar type T and sizes $I \times K$ (matrix) and K (vector), for arbitrary but fixed positive natural numbers $I, K \in \mathbb{N}$. In the figure, based on index function $(i,k) \mapsto (i,k)$ and $(i,k) \mapsto (k)$, higher-order function `inp_view` computes a function that takes M and v as input and maps them to a 2-dimensional array of size $I \times K$ – referred to as *input Multi-Dimensional Array (MDA)* in MDH. The MDA contains at each point (i,k) the pair $(M_{i,k}, v_k) \in T \times T$ comprising element $M_{i,k}$ of matrix M (first component) and element v_k of vector v (second component). The input MDA is then mapped via function `md_hom` to an output MDA of size $I \times 1$, by applying multiplication $*$ to each pair $(M_{i,k}, v_k)$ within the input MDA, and combining elements in i dimension via $+\!\!+$ (concatenation) and in k-dimension via $+$ (point-wise addition). Finally, function `out_view` computes a function that straightforwardly maps the output MDA of size $I \times 1$ to `MatVec`'s result vector $w \in T^I$, which has scalar type T and is of size I. For the example of `MatVec`, the output view is trivial, but it can be used in other computations (such as matrix multiplication) to conveniently express more advanced variants of computations (e.g., computing the result matrix of matrix multiplication as transposed).

MDH's high-level program representation can be used for expressing various kinds of data-parallel computations from important areas [4] (not presented for brevity), including linear algebra, quantum chemistry, data mining, and deep learning.

3 Low-Level Program Representation

Figure 2 shows a particular instance of MDH's low-level representation for `MatVec`. For simplicity, we consider a straightforward target architecture that has 2 memory layers (`HM` and `L1`) and 1 core layer (`COR`) only. In the right part of the figure (steps 1-7), the `MatVec` computation is de-composed (a.k.a. *tiling* in programming) for each memory and core layer of the target architecture (indicated by superscripts of p variables) and for each of the two dimension of `MatVec`'s iteration space (indicated by subscripts). For each tile, the low-level representation indicates the memory region for the corresponding parts of the M input matrix and v input vector to be used – *Host Memory (*`HM`*)* or `L1` cache – as well as the parts' corresponding memory layouts, e.g., $[1,2]$ (standard layout or $[2,1]$ (transposed layout). Similarly, the re-composition phase (steps 9-15) combines computed tiles to the final result. $+\!\!+$ and $+$.

[1] This extended abstract briefly summarizes some highlights presented in the following work-in-progress paper [4].

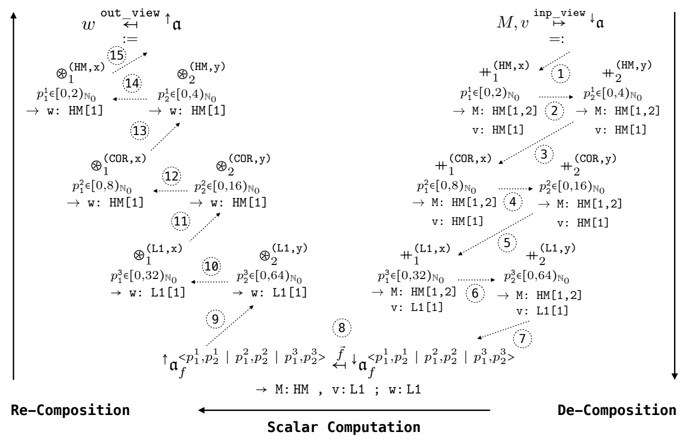

Figure 2. MDH low-level expression for `MatVec`

4 Lowering: From High Level to Low Level

We lower programs in our high-level representation (Figure 1) to our low-level representation (Figure 2) based on formally defined *tuning parameters*. These parameters determine, e.g., the numbers of tiles, the memory regions for the input/output data to be used (e.g., *device*, *shared* or *register memory* for CUDA devices), etc.[2]

We fully automatically choose device-optimized values of tuning parameters using the *Auto-Tuning Framework (ATF)* [7]. Consequently, based on the high-level representation of the target computation (Figure 1) and a set of tuning parameter values (chosen via ATF), we fully automatically generate a low-level representation for the target computation (Figure 2) that is optimized for the particular target architecture. From this representation (Figure 2), we generate executable program code (e.g., in CUDA or OpenCL). Our low-level representation is designed such that code generation becomes trivial – all optimization decisions already are made in the low-level program representation based on auto-tuning.

5 Experimental Results

We achieve encouraging experimental results on GPU and CPU as compared to well-performing competitors for case studies from popular areas, including: linear algebra, stencil computations, quantum chemistry, data mining, and deep learning. For brevity, we focus in this section on presenting some notable highlights from our extensive experiments presented and thoroughly discussed in [4].

Figure 3 reports the speedup of our approach over deep learning compiler TVM+Ansor [2, 9], polyhedral compilers PPCG [8] and Pluto [1], as well as the hand-optimized vendor libraries NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN for *Matrix Multiplication (*`MatMul`*)* and *Multi-Channel Convolution (*`MCC`*)* on real-world input sizes as used in the training and inference phases of the `ResNet-50` neural network.

[2] Figure 2 shows an example for already chosen parameters for simplicity.

Deep Learning	NVIDIA Ampere GPU			
	ResNet-50			
	Training		Inference	
	MCC	MatMul	MCC	MatMul
TVM+Ansor	1,00	1,26	1,05	2,22
PPCG	3456,16	8,26	–	7,89
PPCG+ATF	3,28	2,58	13,76	5,44
cuDNN	0,92	–	1,85	–
cuBLAS	–	1,58	–	2,67

Deep Learning	Intel Skylake CPU			
	ResNet-50			
	Training		Inference	
	MCC	MatMul	MCC	MatMul
TVM+Ansor	1,53	1,05	1,14	1,20
Pluto	355,81	49,57	364,43	13,93
Pluto+ATF	13,08	19,70	170,69	6,57
oneDNN	0,39	–	5,07	–
oneMKL	–	0,44	–	1,09

Figure 3. Speedup (higher is better) of MDH over state-of-practice approaches

We observe often higher performance of our approach over competitors. This is because MDH's optimization space incorporates more optimization opportunities than TVM and polyhedral compilers, e.g., parallelizing also reduction dimensions. Vendor libraries sometimes achieve higher performance than our approach, because the libraries are optimized at the assembly level which offers more optimization opportunities than the CUDA/OpenCL level of abstraction.

References

[1] U. Bondhugula et al. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI*.
[2] T. Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*.
[3] S. Gorlatch et al. 2011. Parallel skeletons. In *EPC*.
[4] A. Rasch. 2023. Formal (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms. *(under review at ACM TOPLAS)* (2023). https://www.dropbox.com/s/nmfu8cshd9uvcje/mdh_toplas.pdf?dl=0
[5] A. Rasch et al. 2018. Multi-dimensional Homomorphisms and Their Implementation in OpenCL. *IJPP* (2018).
[6] A. Rasch et al. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *PACT*. 354–369.
[7] A. Rasch and other. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *TACO* 18, 1, Article 1 (jan 2021), 26 pages.
[8] S. Verdoolaege et al. 2013. Polyhedral Parallel Code Generation for CUDA. *TACO* 9, 4, Article 54 (Jan. 2013), 23 pages.
[9] L. Zheng et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *OSDI*.