Performance, Portability, and Productivity for Data-Parallel Applications on Multi- and Many-Core Architectures

Ari Rasch
a.rasch@wwu.de
University of Muenster (Germany)

Abstract

We present a novel approach to performance, portability, and productivity of data-parallel computations on multi- and many-core architectures. Our approach is based on Multi-Dimensional Homomorphisms (MDHs) – a formally defined class of functions that cover important data-parallel computations, e.g., linear algebra routines (BLAS) and stencil computations. For MDHs, we present a high-level Domain-Specific Language (DSL) that contributes to high user productivity, and we propose a corresponding DSL compiler which automatically generates optimized (auto-tuned) OpenCL code, thereby providing high, portable performance, over different architectures and input sizes, for programs in our DSL. Our experimental results, on Intel CPU and NVIDIA GPU, demonstrate competitive and often significantly better performance of our approach as compared to state-of-practice approaches, e.g., Intel MKL/MKL-DNN and NVIDIA cuBLAS/cuDNN.

CCS Concepts • **General and reference → Performance**; • **Computer systems organization → Parallel architectures**; • **Software and its engineering → Parallel programming languages**.

Keywords Multi-Dimensional Homomorphisms, OpenCL, Auto-Tuning, GPU, multi-core CPU, BLAS, Stencils

ACM Reference Format:
Ari Rasch. 2019. Performance, Portability, and Productivity for Data-Parallel Applications on Multi- and Many-Core Architectures. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3359061.3361072

SPLASH Companion '19, October 20–25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6992-3/19/10…$15.00
https://doi.org/10.1145/3359061.3361072

1 Motivation

Achieving performance, portability, and productivity for data-parallel applications targeting state-of-the-art parallel architectures and varying input sizes is challenging. For example, for high performance, the programmer has to optimize its source code for the complex hardware of modern parallel devices, e.g., Intel multi-core CPU or NVIDIA many-core GPU which are characterized by deep and complex core and memory hierarchies. For portable performance over such architectures – i.e., the same source code achieves consistent high performance over different architectures – the programmer has to consider that architectures may differ significantly in their characteristics, e.g., the number of cores, cache sizes, automatically managed caches (as in CPUs) vs. manually manages caches (GPUs), etc. Achieving portable performance over different input sizes is even more challenging: for example, a high-performance implementation of matrix multiplication on big power-of-two input sizes, as often used in the traditional field of numerical computations, is programmed fundamentally differently as compared to matrix multiplication on small, irregularly shaped input matrices, e.g., as currently occurring in deep learning. Furthermore, modern architectures are usually programmed on a low level, e.g., in OpenCL – a popular standard for uniformly programming different architectures (e.g., CPU and GPU) – making programming such architectures tedious and cumbersome, e.g., because of complex index computations and synchronization which severely affect programmer's productivity. Moreover, state-of-the-art parallel programming approaches (such as OpenCL) require so-called *host code* for their execution, which is cumbersome and error-prone to program: for example, in host code, the programmer has to explicitly perform data transfers between host and device memory and exploit asynchronous computations efficiency for high performance (e.g., overlapping data transfers and/or device computations).

We provide an approach to address all the aforementioned challenges – performance, portability, and productivity – for our *Multi-Dimensional Homomorphisms (MDHs)*:

1. We define MDHs [12] formally as a class of functions that cover data-parallel computations, e.g., linear algebra routines (BLAS) and stencil computations.
2. We enable conveniently expressing MDHs by providing a high-level Domain-Specific Language (DSL) for them [12].
3. We provide a DSL compiler [15] to automatically generate OpenCL code from expressions in our DSL – by relying on OpenCL, we target various parallel architectures, e.g., Intel CPU and NVIDIA GPU. We generate our OpenCL code as fully automatically optimizable (auto-tunable) – for any combination of a target device and input size – by generating our code as targeted to the OpenCL's abstract device models and as parametrized in these models' performance-critical parameters, e.g., the number of threads and sizes of tiles on different core/memory layers.
4. We provide our *Auto-Tuning Framework (ATF)* [10, 11] – a general-purpose auto-tuning approach – which we use to automatically choose optimized values of performance-critical parameters in our generated code – for any combination of a target device and input size.
5. We provide our *dOCAL* [13, 14] library – a high-level programming abstraction for OpenCL's low-level host code – which we use to conveniently execute our generated and auto-tuned OpenCL code on the devices of a distributed, heterogeneous system.

We show that we reach with our MDH+ATF+dOCAL approach often significantly better performance than Lift [16] – a popular, performance-portable approach which is closely related to our work. We also show that we reach competitive and sometimes even better performance than hand-optimized approaches, e.g., vendor libraries Intel MKL [5] and NVIDIA cuBLAS [8] for linear algebra routines (BLAS) – both libraries are optimized at the assembly-level for highest performance on Intel or NVIDIA hardware, respectively,

2 Approach

Our approach consists of three major steps, as follows.

2.1 Generation

We generate code for MDHs which are formally defined as follows [12]: Let T and T' be two arbitrary data types (e.g., `float`). A function $h : T[N_1]\dots[N_d] \to T'$ on d-dimensional arrays of size $N_1 \times \dots \times N_d$ and with elements in T is called a *Multi-Dimensional Homomorphism (MDH)* iff there exist *combine operators* $\circledast_1, \dots, \circledast_d : T' \times T' \to T'$, such that for each integer $k \in [1, d]$ and arbitrary, concatenated input array $a +\!\!+_k b$ in dimension k, the homomorphic property is satisfied: $h(\, a +\!\!+_k b\,) = h(\,a\,) \circledast_k h(\,b\,)$.
In words: the value of h on a concatenated array in dimension k can be computed by applying h to the array's parts a and b and then combining the results by combine operator $\circledast_k$.

We express MDHs in our DSL using our `md_hom` higher-order function [12] (a.k.a. parallel pattern) which we define as follows. Every MDH h is uniquely determined by its combine operators $\circledast_1, \dots, \circledast_d$ and its behavior f on scalar values (i.e., $f(\,a[0]\dots[0]\,) = h(a)$ for every $a \in T[1]\dots[1]$). This enables expressing h using our pattern `md_hom` which takes these functions as parameters: $h =$ `md_hom`$(\, f,\ (\circledast_1, \dots, \circledast_d)\,)$. For example, matrix multiplication is expressed using `md_hom` as follows: `md_hom`$(\,*,\ (+\!\!+,\ +\!\!+,\ +)\,) \circ$ `view(A,B)(i,j,k)(A[i,k], B[k,j])`. Here, `view` is the second pattern of our DSL, which we use to uniformly prepare a domain-specific input for `md_hom`. For example, for matrix multiplication, pattern `view` takes as input the two matrices `A,B` and the array indices `i,j,k`; it yields the pair `(A[i][k], B[k][j])` which is used as input for `md_hom`'s scalar function $f = *$.

We generate OpenCL code for MDHs expressed using patterns `md_hom` and `view` (which we both have embedded in C++ as functions of a programming library). A major feature of our code is that we generate it as targeted to the OpenCL's abstract platform and memory models (which uniformly represent the core/memory hierarchy of different architectures, e.g., CPU and GPU), and as parametrized in the performance-critical parameters of these two models; this enables fully automatically optimizing our generated code, for a particular target device and input size, using auto-tuning. For example, our code is parametrized in the number of threads and the sizes of tiles – on both layers of OpenCL's two models, and in all dimensions of the MDH's multi-dimensional input; these sizes and numbers are very critical for high performance.

Our MDH theory and our DSL for MDHs are described in detail in [12], and our OpenCL code generation approach for MDHs is presented in [15].

2.2 Optimization

To determine optimized values of performance-critical parameters in our generated code, we provide our general-purpose *Auto-Tuning Framework (ATF)* [10, 11] which combines major advantages over other general-purpose auto-tuning approaches, e.g., [1, 6, 9, 18]; for example, using ATF, we can auto-tune programs written in arbitrary programming languages and from arbitrary application domains.

A major feature of ATF is that it supports auto-tuning of programs that have interdependent tuning parameters. For example, in our generated code, we auto-tune the sizes of tiles on different memory layers, and a tile size on a lower memory layer has to be smaller or equal than a tile size on an upper layer, because a lower-layer tile is a chunk of an upper-layer tile – this can be conveniently expressed in ATF, and ATF efficiently generates, stores, and explores the search space of such interdependent tuning parameters, which is currently not possible using other state-of-the-art auto-tuning frameworks.

Our ATF framework is described in detail in [10, 11].

2.3 Execution

To execute our generated and auto-tuned OpenCL code, we provide our novel *dOCAL* [14] approach. dOCAL is a high-level programming abstraction – implemented as a C++ programming library – that allows conveniently executing OpenCL code on the devices of a distributed, heterogeneous system. For this, dOCAL simplifies implementing OpenCL's host code which consists of boilerplate low-level commands, e.g., for data transfers between host and device memory, and synchronization. For example, dOCAL automatically initializes OpenCL, manages the low-level host-code objects (e.g., the so-called OpenCL contexts and command queues), uses low-level functions for allocating and managing memory on the devices and the host, and it performs synchronization between data transfers and/or computations on the device/host. Moreover, dOCAL automatically provides asynchronous computation efficiency (e.g., overlapping data transfers and/or device computations) by automatically generating and maintaining a data-dependency graph. dOCAL also enables conveniently executing OpenCL programs on the devices of remote nodes by relying on the Boost.Asio library.

Our dOCAL approach is described in detail in [13, 14].

3 Evaluation Methodology

All experiments described in this section can be reproduced using our artifact implementation in [2].

We evaluate our MDH+ATF+dOCAL approach using computations from two prominent areas: 1) GEneral Matrix-Matrix multiplication (GEMM) and GEneral Matrix-Vector multiplication (GEMV) from the area of linear algebra (BLAS), and 2) Gaussian 2D convolution and Jacobi 3D from the area of stencil computations.

We compare the performance of our automatically generated and auto-tuned OpenCL code on CPU and GPU against well-performing competitors: 1) Lift – an academic framework – which is closely related to our approach and has proven to provide high, portable performance for BLAS [17] and stencil computations [3], and 2) Intel MKL/MKL-DNN [4, 5] and NVIDIA cuBLAS/cuDNN [7, 8] – vendor libraries that are optimized by hand at the assembly level to provide high performance for BLAS and stencil computations on Intel or NVIDIA hardware, correspondingly.

Figure 1 shows our experimental results. We observe better performance of our approach as compared to our competitors, because we generate a flexible OpenCL implementation whose parallelization and tiling strategies can be auto-tuned for *both* layers of OpenCL's platform and memory model and in *all* dimensions of the multi-dimensional input.

Our results (also for further computations, e.g., from the area of data mining and tensor contractions) are discussed in detail in [15].

CPU	GEMM		GEMV	
	RW	PC	RW	PC
AF	fails	3.04	1.51	1.99
VL	4.22	0.74	1.05	0.87

GPU	GEMM		GEMV	
	RW	PC	RW	PC
AF	4.33	1.17	3.52	2.98
VL	2.91	0.83	1.03	1.00

CPU	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
AF	4.90	5.96	1.94	2.49
VL	6.99	14.31	N/A	N/A

GPU	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
AF	2.33	1.09	1.14	1.02
VL	3.78	19.11	N/A	N/A

Figure 1. Speedups/slowdowns (higher is better) of our automatically generated and auto-tuned code over: i) academic framework (AF) Lift, and ii) vendor libraries (VL) Intel MKL/MKL-DNN and NVIDIA cuBLAS/cuDNN. We use for evaluation both Intel Xeon E5-2640 v2 8-core CPU (left) and NVIDIA Tesla V100-SXM2-16GB GPU (right), and we use input sizes from real-world (RW) applications, as well as sizes that are preferable for our competitors (PC).

References

[1] J. Ansel et al. 2014. OpenTuner: An Extensible Framework for Program Autotuning *(PACT)*. 303–316.

[2] Artifact Implementation. 2019. https://gitlab.com/mdh-project/pact_2019_artifact.

[3] B. Hagedorn et al. 2018. High Performance Stencil Code Generation with Lift *(CGO)*. 100–112.

[4] Intel. 2018. Math Kernel Library for Deep Learning Networks. https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation

[5] Intel. 2019. Math Kernel Library. https://software.intel.com/en-us/mkl

[6] C. Nugteren et al. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels *(MCSOC)*. 195–202.

[7] NVIDIA. 2018. CUDA Deep Neural Network library. https://developer.nvidia.com/cudnn

[8] NVIDIA. 2019. cuBLAS library. https://developer.nvidia.com/cublas

[9] P. Pfaffe et al. 2019. Efficient Hierarchical Online-autotuning: A Case Study on Polyhedral Accelerator Mapping *(ICS)*. 354–366.

[10] A. Rasch et al. 2017. ATF: A Generic Auto-Tuning Framework. In *IEEE 19th International Conference on High Performance Computing and Communications (HPCC)*. 64–71.

[11] A. Rasch et al. 2018. ATF: A Generic, Directive-Based Auto-Tuning Framework. *Concurrency and Computation: Practice and Experience*, 13 pp.

[12] A. Rasch et al. 2018. Multi-Dimensional Homomorphisms and Their Implementation in OpenCL. *International Journal of Parallel Programming*, 101–119.

[13] A. Rasch et al. 2018. OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA. In *IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 408–416.

[14] A. Rasch et al. 2019. dOCAL: High-Level Distributed Programming with OpenCL and CUDA. *The Journal of Supercomputing*, 22 pp.

[15] A. Rasch et al. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. (accepted).

[16] M. Steuwer et al. 2015. Generating Performance Portable Code Using Rewrite Rules *(ICFP)*. 205–217.

[17] M. Steuwer et al. 2016. Matrix Multiplication Beyond Auto-tuning: Rewrite-based GPU Code Generation *(CASES)*. 15 pp.

[18] B. Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* (2019), 347 – 358.