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

1 MOTIVATION

Providing performance, portability, and productivity for data-parallel computations on state-of-the-art parallel architectures and varying input data sizes is challenging. For example, for high performance, the programmer has to optimize its source code for the 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 a consistent level of high performance over different architectures – the programmer has to consider that architectures may differ significantly in their characteristics, e.g., the number of cores and size of caches. Moreover, performance portability also has to be ensured over different input sizes: for example, a high-performance implementation of matrix multiplication on traditional big, power-of-two input sizes is programmed fundamentally differently as compared to matrix multiplication on small, irregularly-shaped input matrices as currently used in deep learning. Furthermore, modern architectures are usually programmed on a low level, e.g., in OpenCL – an emerging de-facto standard for uniformly programming different architectures, such as CPU and GPU – which severely decreases programming productivity for such architectures: the programmer has to explicitly deal with complex index computations, manage synchronization, mange thread ids and memory indices on different core/memory layers, etc.

We provide an approach to address all the aforementioned challenges – performance, portability, and productivity – for our class of data-parallel computations to which we refer as *Multi-Dimensional Homomorphisms (MDHs)*:

(1) We define MDHs formally [13] as a class of functions that cover important data-parallel computations, e.g., linear algebra routines (BLAS), stencil computations, data mining algorithms, and tensor contractions.
(2) We enable conveniently expressing MDHs by introducing a high-level Domain-Specific Language (DSL) for them [13].
(3) We provide a DSL compiler to generate OpenCL program code from expressions in our DSL [2]. We generate our OpenCL code as fully automatically optimizable (auto-tunable) – for each arbitrary combination of an MDH function, target architecture, and input size – by generating the code as targeted to the OpenCL's abstract device models (rather than a particular architecture) and as parametrized in the models' performance-critical parameters, e.g., the number of threads and size of tiles on different core/memory layers.
(4) We provide our own general-purpose *Auto-Tuning Framework (ATF)* [12] to automatically choose optimized values of our code's parameters – for any arbitrary combination of an architecture and input size.

We show that with our MDH+ATF approach, we often reach significantly better performance than state-of-the-practice performance-portable approaches (e.g., the popular Lift framework [15]) and competitive or even better performance than hand-optimized approaches, e.g., the assembly-optimized, vendor-provided libraries Intel MKL and NVIDIA cuBLAS for linear algebra routines (BLAS).

2 APPROACH

In our approach, we generate code for MDHs [13] which are formally defined as follows: Let T and T' be two arbitrary data types. A function $h : T[\,N_1\,]\ldots[\,N_d\,] \to T'$ on d-dimensional arrays of size $N_1 \times \ldots \times N_d$ and with elements in T 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 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 combining the results afterwards by using combine operator $\circledast_k$.

We express MDHs in our DSL using our `md_hom` parallel pattern [13] which we define as follows. Every MDH h is uniquely determined by its combine operators $\circledast_1, \ldots, \circledast_d$ and its behavior f on scalar values (i.e., $f(\, a[0]\ldots[0]\,) = h(a)$ for every $a \in T[1]\ldots[1]$). This enables expressing h using our `md_hom` higher-order function (a.k.a. parallel pattern) which takes these functions as parameters:

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

For example, matrix multiplication `MatMul` is expressed using `md_hom` as follows:

```
md_hom( *, (++, ++, +) ) ∘ 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 `MatMul`, function `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 GEMM's scalar function $f = *$.

We generate OpenCL code for MDHs which are expressed in our DSL via patterns `md_hom` and `view` (we have embedded both in C++ in form of functions of a programming library [2]). A major feature of our OpenCL code is that we generate it as targeted to

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the OpenCL's abstract device models and as parameterized in these models' performance-critical parameters; this allows automatically optimizing our code for any particular combination of an architecture and input size using classical auto-tuning. For example, our code is parameterized in the number of threads and the size of tiles – on *both* layers of OpenCL's core and memory layers and in *all* dimensions of the multi-dimensional input.

To determine optimized values of our code's performance-critical parameters (a.k.a. *tuning parameters*), we provide our own general-purpose *Auto-Tuning Framework (ATF)* [12] which has a major-advantage over the state-of-the-art general-purpose auto-tuning approaches [1, 9, 11, 17]: it enables auto-tuning programs with *interdependent tuning parameters*. For example, in our generated code, we auto-tune the size 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 our ATF approach, and ATF can efficiently generate, store, and explore the search space of such interdependent tuning parameters, which is not possible with the other state-of-the-art general-purpose auto-tuning frameworks.

3 EXPERIMENTAL EVALUATION

We experimentally evaluate our MDH+ATF approach using computations from four important areas: 1) GEneral Matrix-Matrix multiplication (GEMM) and GEneral Matrix-Vector multiplication (GEMV) from the area of linear algebra (BLAS), 2) Gaussian2D convolution and Jacobi3D which are stencil computations, 3) Probabilistic Record Linkage (PRL) which is important in data mining, and 4) tensor contractions which are essential in machine learning. For evaluation, we use a dual-socket system equipped with two Intel Xeon E5-2640 v2 8-core CPUs and an NVIDIA Tesla V100-SXM2-16GB GPU.

We compare the performance of our automatically generated and auto-tuned OpenCL code on Intel CPU and NVIDIA GPU against several state-of-the-practice approaches: 1) Lift [15] – an academic framework – which is closely related to our approach and has proven to provide high, portable performance for BLAS and stencil computations [4, 16]; 2) Intel MKL/MKL-DNN [6, 7] and NVIDIA cuBLAS/cuDNN [5, 10] – vendor-provided BLAS libraries – which are optimized by hand at the assembly level to provide highest performance for BLAS (MKL and cuBLAS) or Gaussian stencils (MKL-DNN and cuDNN) on Intel or NVIDIA hardware, correspondingly; 3) EKR [14] – the original JAVA implementation from the largest cancer registry in Europe for Probabilistic Record Linkage – an important application in data mining; 4) COGENT [8] and Facebook's Tensor Comprehensions [18] for high-performance tensor contractions on NVIDIA GPUs.

For a challenging comparison, we use i) real-world input sizes (abbreviated by RW in the following), e.g., taken from the state-of-the-art deep-learning framework Caffe [19]; for example, we use for GEMM's $M \times K$ and $K \times N$ input matrices a size of $(M, N, K) = (10, 500, 64)$ which is repeatedly called in Caffe's siamese sample, and ii) input sizes that are preferable for our competitors (abbreviated by PC), e.g., big, square, power-of-two input matrices of size 1024×1024 in case of GEMM for which Lift, MKL and cuBLAS are highly optimized. For EKR, we use different power-of-two input sizes, and for tensor contractions, we use the 9 original RW sizes from [8].

For a fair comparison, in all experiments, we use exactly the same auto-tuning time of our competitors. For example, we auto-tune our stencil kernels for 5h which is exactly the same auto-tuning time that Lift uses for its stencil kernels.

Our experimental results are depicted in Figures 1-4. Our good results are because our generated OpenCL code can be auto-tuned for *both* core and memory layers of OpenCL's device models and in *all* dimensions of the multi-dimensional input.

A detailed discussions of our results can be found in [2, 3].

	CPU			
	GEMM		GEMV	
	RW	PC	RW	PC
MDH	1.00	1.00	1.00	1.00
Lift	fails	3.04	1.51	1.99
MKL	4.22	0.74	1.05	0.87

	GPU			
	GEMM		GEMV	
	RW	PC	RW	PC
MDH	1.00	1.00	1.00	1.00
Lift	4.33	1.17	3.52	2.98
cuBLS	2.91	0.83	1.03	1.00

Figure 1: Speedup of our automatically generated and auto-tuned code for linear algebra routines (BLAS) on Intel CPU (left) and NVIDIA GPU (right).

	CPU			
	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
MDH	1.00	1.00	1.00	1.00
Lift	4.90	5.96	1.94	2.49
MKL	6.99	14.31	N/A	N/A

	GPU			
	Gaussian (2D)		Jacobi (3D)	
	RW	PC	RW	PC
MDH	1.00	1.00	1.00	1.00
Lift	2.33	1.09	1.14	1.02
cuDNN	3.78	19.11	N/A	N/A

Figure 2: Speedup of our automatically generated and auto-tuned code for stencil computations on Intel CPU (left) and NVIDIA GPU (right).

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

Figure 3: Speedup of our automatically generated and auto-tuned code for Probabilistic Record Linkage (PRL) on Intel multi-core CPU.

	Tensor Contractions								
	RW 1	RW 2	RW 3	RW 4	RW 5	RW 6	RW 7	RW 8	RW 9
MDH	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
CGNT	1.26	1.16	2.12	1.24	1.18	1.36	1.48	1.44	1.85
F-TC	1.19	2.00	1.43	2.89	1.35	1.54	1.25	2.02	1.49

Figure 4: Speedup of our automatically generated and auto-tuned code for Tensor Contractions on NVIDIA GPU.

REFERENCES

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

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

[3] Artifact Implementation. 2019. https://github.com/mdh-project/sc19-md-hom.

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

[5] Intel. 2018. CUDA® Deep Neural Network library. https://developer.nvidia.com/cudnn

[6] 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

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

[8] Jinsung Kim et al. 2019. A Code Generator for High-performance Tensor Contractions on GPUs *(CGO)*. 85–95.

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

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

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

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

[13] Ari Rasch and Sergei Gorlatch. 2018. Multi-dimensional Homomorphisms and Their Implementation in OpenCL. *International Journal of Parallel Programming*, 101–119.

[14] Ari Rasch, Richard Schulze, Waldemar Gorus, Jan Hiller, Sebastian Bartholomäus, and Sergei Gorlatch. 2019. High-performance Probabilistic Record Linkage via Multi-dimensional Homomorphisms. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. ACM, New York, NY, USA, 526–533. https://doi.org/10.1145/3297280.3297330

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

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

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

[18] Nicolas Vasilache et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730

[19] Yangqing Jia et al. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 4 pp.