Portable Parallel Performance via Multi-Dimensional Homomorphisms

Ari Rasch
University of Münster, Germany
a.rasch@wwu.de

Richard Schulze
University of Münster, Germany
r.schulze@wwu.de

Sergei Gorlatch
University of Münster, Germany
gorlatch@wwu.de

ACM Reference format:
Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2018. Portable Parallel Performance via Multi-Dimensional Homomorphisms. In *Proceedings of Supercomputing, Dallas, Texas USA, November 2018 (SC'18),* 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MOTIVATION AND RELATED WORK

Achieving portable program performance on modern architectures, e.g., multi-core CPU and Graphics Processing Unit (GPU) is hard: while approaches such as OpenCL [7] provide portability of program code across a range of hardware architectures, they do not guarantee portability of performance, e.g., a parallel program yielding high performance on a multi-core CPU can yield poor performance on a GPU and vice versa. This is because different hardware architectures usually differ significantly in their characteristics, e.g., GPUs provide a high number of cores but small caches while multi-core CPUs have a low number of cores and big caches.

Performance differs also across input sizes [17]. For example, a high-performance implementation of *GEneral Matrix-Matrix Multiplication (GEMM)* targeting big square input matrices – the usual input of numerical applications [8] – differs significantly from a GEMM implementation optimized for small matrices, e.g., as used in the important application area of deep learning [18]. This is because high performance on big square input matrices is achieved by computing each element of the result matrix in parallel [6]; however, for high performance on small matrix sizes, the computation of each result element itself has to be parallelized as well [17] to increase the degree of parallelism and consequently to better utilize modern parallel hardware with many cores.

Several state-of-the-art approaches aim at providing portability of performance. However, they are mostly limited to restricted combinations of: 1) applications, 2) hardware architectures, and/or 3) input sizes. For example, the popular libraries NVIDIA cuBLAS [13] and Intel MKL [5] provide high performance for linear algebra routines on NVIDIA GPUs or Intel multi-core CPUs, respectively. However, these libraries cannot be used for application classes that are different from linear algebra and for hardware architectures that are not NVIDIA GPU or Intel CPU, e.g., AMD CPU/GPU and ARM mobile processors. In contrast, the CLBlast library [12] targets linear algebra routines on various architectures, but it is only optimized for large input sizes and usually provides lower performance than MKL und cuBLAS on Intel and NVIDIA hardware. Lift [11] is a novel approach that targets various application classes, hardware architectures and input sizes. For example, it has been recently shown that Lift provides high portable performance for stencil applications on GPU architectures [3]. However, the Lift approach is based on a vast search space of differently-optimized implementations, where search space's efficient exploration requires artificial search space pruning by a Lift expert for each combination of target application, hardware architecture, and input sizes.

Multi-dimensional Homomorphisms (MDHs) [15] are a recently-defined class of parallelizable functions; they cover application areas such as linear algebra routines (BLAS) and stencil computations. MDHs can be efficiently executed on modern parallel architectures and on a broad range of input sizes. An OpenCL-implementation schema for MDHs is presented in [15]; it addresses the performance-portability issue by being generic in the performance-critical parameters of the OpenCL *platform model* – the number of *work-items* (the OpenCL term for thread) and the number of *work-groups* (groups of work-items). The generality of the schema enables automatically choosing optimized values of these parameters for a target architecture and input size by exploiting the *auto-tuning* approach [14]. However, many MDHs (e.g., the popular GEMM routine [6]) rely on fast data accesses for high performance, and thus, an efficient implementation of MDHs has to efficiently utilize also the OpenCL's *memory model*.

In this paper, we fundamentally extend the MDHs' OpenCL implementation schema by making it generic also in the performance-critical parameters of the OpenCL's memory model: the *local* and *private memory* sizes. This enables auto-tuning our novel schema for the specific memory requirements of a target parallel device (and not only for device's thread hierarchy, as in [15]), thereby significantly contributing to portability of performance. Our preliminary results demonstrate for applications from linear algebra (BLAS) and stencil computations that with our novel implementation schema, we reach competitive and often even significantly better performance than the related work on modern architectures and for important input sizes, e.g., as used in the arising application area of deep learning.

2 MULTI-DIMENSIONAL HOMOMORPHISMS AND THE `MD_HOM` PARALLEL PATTERN

Multi-dimensional homomorphisms [15] are defined as follows. Let T and T' be two arbitrary data types. e.g., `float`. 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 $a +\!\!+_k b$ in dimension k:

$$h(\, a +\!\!+_k b\,) \;=\; h(a) \;\circledast_k\; h(b)$$

SC'18, November 2018, Dallas, Texas USA

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

In words: the value of h on a concatenated array in dimension k can be computed by applying h to the MDA's chunks a and b and combining the results afterwards by using the combine operator $\circledast_k$. Since the computations of $h(a)$ and $h(b)$ are independent of each other, they can be performed in parallel.

According to [15], every MDH h can be computed as

$$h(\, a[N_1]\ldots[N_d]\,) = \underset{i_1\in[1,N_1]}{\circledast_1} \cdots \underset{i_d\in[1,N_d]}{\circledast_d} f(\, a[\,i_1\,]\ldots[\,i_d\,]\,)$$

where f represents the behavior of h on scalar values, i.e., $f(\, a[0]\ldots[0]\,) = h(a)$ for each d-dimensional array a comprising only one element (i.e., a has size 1 in each of its d dimensions). This enables expressing h also as:

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

As `md_hom` represents a higher-order function that can be computed in parallel, it represents a *parallel pattern* (a.k.a. *algorithmic skeleton* [1, 2]).

3 THE OPENCL IMPLEMENTATION OF THE `MD_HOM` PARALLEL PATTERN

We provide a high-performance portable OpenCL implementation schema for the `md_hom` parallel pattern. In comparison to our initial schema in [15], our novel approach efficiently utilizes also the OpenCL's memory model (and not only its platform model). For this, we develop our OpenCL implementation as parametrized in the performance-critical parameters of both models. For the platform model, we parametrize in the number of work-groups `NUM_WG_i` and the number of work-items per work group `NUM_WI_i`, where $1 \le i \le d$ for an d-dimensional input MDA. In addition, we introduce novel parameters for OpenCL's memory model: the sizes of the OpenCL *local* and *private memory* `LM_SIZE_i` and `PM_SIZE_i`. The OpenCL local and private memory regions are fast but scarce memory resources that, for high performance, have to be efficiently utilized by the programmer as explicitly managed caches [6].

Our implementation schema is as follows. We split the input MDA into d-dimensional chunks of size `LM_SIZE_i` in dimension i (a.k.a. LM-chunks), $1 \le i \le d$, and we split each LM-chunk further in chunks of size `PM_SIZE_i` (PM-chunks). We cache the LM-chunks in local memory and the PM-chunks in private memory. For computing the chunks, we start `NUM_WG_i` work-groups each comprising `NUM_WI_i` work-items. The work-groups process the LM-chunks and the work-items process the PM-chunks, correspondingly.

We enable portability of performance by automatically choosing optimized values of the performance-critical parameters for each new target hardware architecture and input size using the auto-tuning approach. As concrete auto-tuner, we use the Auto-Tuning-Framework (ATF) [14] – it is well suited to complex parallel applications that target modern hardware architectures with many cores.

4 EXPERIMENTAL EVALUATION

We experimentally evaluate our approach to performance portability using two important samples: 1) GEMM – the most prominent BLAS routine, and 2) Gradient – a popular stencil application [16].

In Figure 1, we demonstrate the speedup of `md_hom` for GEMM and Gradient – both expressed according to [15] – on Intel Xeon E5 CPU and NVIDIA Tesla V100 GPU over state-of-the-art approaches: i) Intel MKL [5] and NVIDIA cuBLAS [13] for GEMM, and ii) the prominent Lift approach [3] in case of Gradient. For each sample, we use i) a small input size taken from the application area of deep learning, and ii) a large input size, e.g., as used in numerical computation (in case of BLAS) or image processing (Stencil), correspondingly. In case of GEMM, the small input matrices are of size 10×64 and 64×500 [18] and the large matrices are both of size 1024×1024 [8]. For Gradient, we use a small input image size of 224×224 [4] and a large image of size 4096×4096 [10].

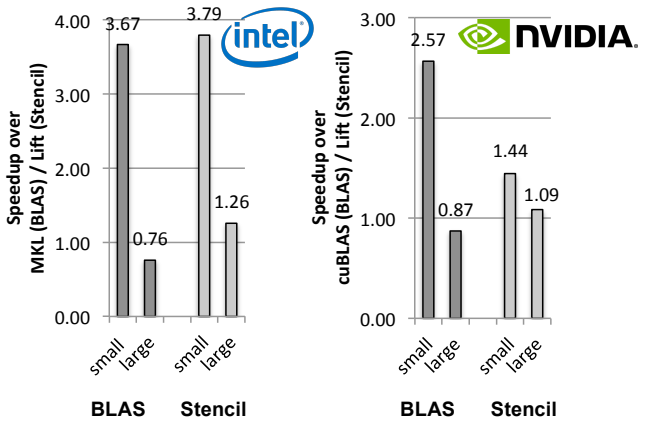

Figure 1: Speedup of `md_hom` (higher is better) on Intel CPU (left) and NVIDIA GPU (right) for the samples GEMM (BLAS) and Gradient (Stencil) over state-of-the-art approaches Intel MKL, NVIDIA cuBLAS (BLAS) and Lift (Stencil). We use input sizes as used in deep learning (small), and numerical computations and image processing (large).

We observe that `md_hom` provides competitive and often even significantly better performance than the references. The reason is that Intel MKL and NVIDIA cuBLAS are highly hand-optimized libraries, but only for large input sizes. The better performance of the MKL and cuBLAS for large input sizes is because they perform optimizations at the assembly level which cannot be implemented using OpenCL [9] . In case of Gradient, the Lift's search space is pruned by the Lift experts with focus on GPU architectures [3], thereby missing optimal solutions for multi-core CPUs. In contrast, `md_hom` has a smaller search space for its tuning parameters `NUM_WG_i`, `NUM_WI_i`, `LM_SIZE_i`, and `PM_SIZE_i`. This enables high performance for both CPU and GPU, because we do not rely on artificial search space pruning for its efficient exploration – we have an average tuning time of < 15h – and thus, we do not miss optimal solutions.

We compare our novel `md_hom`'s implementation schema also with its initial schema in [15]: we obtain a speedup of up to 5.3 for GEMM on the CPU and 2.6 on the GPU; for Gradient, our speedups are 2.0 (CPU) and 1.3 (GPU). This is because, in comparison to our novel schema, `md_hom`'s initial schema is not optimized for OpenCL's memory model which is crucial for achieving high performance for GEMM and Gradient.

REFERENCES

[1] Sergei Gorlatch. 1999. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Science of Computer Programming*, 27 pp.

[2] Sergei Gorlatch and Murray Cole. 2011. Parallel Skeletons. *Encyclopedia of Parallel Computing*, 1417–1422.

[3] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. *International Symposium on Code Generation and Optimization*, 100–112.

[4] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <1MB Model Size. *CoRR*, 13 pp.

[5] Intel. 2018. MKL. (2018). https://software.intel.com/en-us/mkl

[6] K. Matsumoto et al. 2012. Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 396–405.

[7] Khronos. 2018. OpenCL. (2018). https://www.khronos.org/opencl/

[8] Jens Krüger and Rüdiger Westermann. 2003. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Transactions on Graphics*, 908–916.

[9] Junjie Lai and Andre Seznec. 2013. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. *International Symposium on Code Generation and Optimization*, 1–10.

[10] Thibaut Lutz, Christian Fensch, and Murray Cole. 2013. PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 59–82.

[11] Michel Steuwer et al. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. *International Conference on Functional Programming*, 13 pp.

[12] Cedric Nugteren. 2017. CLBlast: A Tuned OpenCL BLAS Library. *CoRR*, 8 pp.

[13] NVIDIA. 2018. cuBLAS. (2018). https://developer.nvidia.com/cublas

[14] Ari Rasch and Sergei Gorlatch. 2018. ATF: A Generic, Directive-Based Auto-Tuning Framework. *Concurrency and Computation: Practice and Experience*, 13 pp. https://doi.org/10.1002/cpe.4423

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

[16] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. 2016. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs. *Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, 92–102.

[17] Philippe Tillet and David Cox. 2017. Input-aware Auto-tuning of Compute-bound HPC Kernels. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 12 pp.

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