Technical Report 2012-002

Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU

Kazuya Matsumoto, Naohito Nakasato, Stanislav G. Sedukhin

July 2, 2012

Graduate School of Computer Science and Enginering
The University of Aizu
Tsuruga, Ikki-Machi, Aizu-Wakamatsu City
Fukushima, 965-8580 Japan

Technical Report 2012-002

Title:

Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU

Authors:
Kazuya Matsumoto, Naohito Nakasato, Stanislav G. Sedukhin

Key Words and Phrases:
matrix multiplication, OpenCL, GPU, auto-tuning

Abstract:

This paper presents results of an implementation of code generator for fast general matrix multiply (GEMM) kernels. When a set of parameters is given, the code generator produces the corresponding GEMM kernel written in OpenCL. The produced kernels are optimized for high-performance implementation on GPUs from AMD. Access latencies to GPU global memory is the main drawback for high performance. This study shows that storing matrix data in a block-major layout increases the performance and stability of GEMM kernels. On the Tahiti GPU (Radeon HD 7970), our DGEMM (double-precision GEMM) and SGEMM (single-precision GEMM) kernels achieve the performance up to 848 GFlop/s (90% of the peak) and 2646 GFlop/s (70%), respectively.

Report Date:
7/2/2012

Written Language:
English

Any Other Identifying Information of this Report:
Manuscript has been accepted for Special Session: Auto-Tuning for Multicore and GPU (ATMG) and is to appear in the proceedings of the IEEE 6th International Symposium on Embedded Multicore SoCs (MCSoC-12), Aizu-Wakamatsu City, Fukushima, Japan, September 20-22, 2012.

Distribution Statement:
First Issue: 10 copies

Supplementary Notes:

Distributed Parallel Processing Laboratory
The University of Aizu
Aizu-Wakamatsu
Fukushima 965-8580
Japan

Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU

Kazuya Matsumoto, Naohito Nakasato, and Stanislav G. Sedukhin
Graduate School of Computer Science and Engineering
The University of Aizu
Aizu-Wakamatsu City, Fukushima, 965-8580 Japan
Email: {d8121101, nakasato, sedukhin}@u-aizu.ac.jp

***Abstract*—This paper presents results of an implementation of code generator for fast general matrix multiply (GEMM) kernels. When a set of parameters is given, the code generator produces the corresponding GEMM kernel written in OpenCL. The produced kernels are optimized for high-performance implementation on GPUs from AMD. Access latencies to GPU global memory is the main drawback for high performance. This study shows that storing matrix data in a block-major layout increases the performance and stability of GEMM kernels. On the Tahiti GPU (Radeon HD 7970), our DGEMM (double-precision GEMM) and SGEMM (single-precision GEMM) kernels achieve the performance up to 848 GFlop/s (90% of the peak) and 2646 GFlop/s (70%), respectively.**

***Keywords*-matrix multiplication; OpenCL; GPU; auto-tuning**

I. Introduction

Matrix-matrix multiply-add is a fundamental routine in linear algebra, which is referred to as GEMM (GEneral Matrix Multiply) in BLAS (Basic Linear Algebra Subroutines) standard [1]. GEMM appears in a lot of important numerical algorithms and is a building block of LAPACK (Linear Algebra PACKage) [2] and other Level-3 BLAS routines [3]. The computational intensity and the regularity of GEMM algorithms make them good candidates for performance acceleration.

There are a lot of work on GEMM performance acceleration on CPUs [4]–[6] and GPUs [7]–[10]. Automatic performance tuning (or auto-tuning in short) is an important technique to generate near-optimal GEMM implementations. ATLAS (Automatically Tuned Linear Algebra Software) [4] is a prominent auto-tuning software library for BLAS routines running on CPUs. An auto-tuning system was also developed for GEMM kernels in CUDA (Compute Unified Device Architecture) for NVIDIA GPUs [7], [11], [12]. In addition, there is a report on auto-tuning methods for developing near-optimal GEMM kernels in OpenCL (Open Compute Language) for AMD GPUs [13].

Auto-tuning system needs two core components: code generator and heuristic search engine. Code generator produces codes parameterized from a pre-defined code template. Heuristic search engine runs the generated codes and searches the best set of parameters for increasing performance using a feedback loop. This study proposes a code generator for fast GEMM kernels written in OpenCL as a preliminary work for an auto-tuning system. The main difference from the similar works [9], [13] is that our code generator supports generating GEMM kernels where matrix data are stored in memory not only in a standard row-/column-major layout, but also in a block-major layout.

The rest of this paper is organized as follows. Section II describes our GEMM code generator. This section also explains relations between parameters of the generator and the OpenCL execution model. Section III shows results of performance evaluation on the AMD Tahiti GPU (Radeon HD 7970). Finally, Section IV concludes the paper.

II. Code Generator

In BLAS [1], GEMM is defined as

$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C,$$

where both α and β are scalar values, and op(A), op(B) and C are $m \times k$, $k \times n$ and $m \times n$ matrices, respectively. The op(X) takes X (non-transposed matrix) or X^T (transposed matrix); thus, there are four multiplication types:

(a) $C \leftarrow \alpha AB + \beta C$,
(b) $C \leftarrow \alpha AB^T + \beta C$,
(c) $C \leftarrow \alpha A^T B + \beta C$,
(d) $C \leftarrow \alpha A^T B^T + \beta C$,

in each real (double or single) precision. In the following description, if not specified, we assume that matrix data are stored in a row-major layout, which is standard in C Programming Language.

Our code generator takes a set of parameters as the input. When the input is given, the code generator produces the corresponding GEMM kernel code written in OpenCL as the output. We can set different parameters to the generator for each GEMM type (14 parameters for (c) type and 12 parameters for the other (a), (b), (d) types). Six parameters define the blocking factors and the other parameters are related to optimization of a way for accessing matrix data. The followings describe the meaning of every parameter.

A. Blocking

A straightforward implementation of GEMM is a three-nested-loop program. Blocking a GEMM algorithm is necessary for getting high performance on a processor with a multi-level memory hierarchy. It is because the blocking increases the data reuse ratio which is required to tolerate memory access latencies.

We can consider that the GPU has three levels of memory spaces: off-chip memory (global memory), on-chip memory (cache or local memory), and private memory (register file). Roughly speaking, the off-chip memory is the largest memory space with the lowest bandwidth, the private memory is the smallest memory space with the highest bandwidth, and the on-chip memory is in the middle of them.

OpenCL is an open standard for general-purpose parallel programming on heterogeneous platforms [14], [15]. The OpenCL execution model covers the three memory spaces. OpenCL specifies a C99-based language that allows to write parallel functions called *kernels*. When a kernel is submitted for execution on the GPU, an index space, which is called *NDRange*, is defined. Each instance in NDRange is named *work-item*, and several work-items are organized into a *work-group*. Every work-item can access its own private memory which is not visible from other work-items. Work-items in a single work-group concurrently run on the processing elements of a compute unit, and share their own on-chip memory. Global memory can be accessed from all work-items, though there is no way for synchronization of work-items in different work-groups during a kernel execution.

We use a two-level (larger and smaller) blocking in our GEMM kernel template since it is effective for high-performance implementation to utilize the three levels of memory and execution model. Let m_l, n_l, k_l be the larger blocking factors. The blocking divides the three matrices A, B, C into $(m/m_l) \times (k/k_l)$, $(k/k_l) \times (n/n_l)$ and $(m/m_l) \times (n/n_l)$ grids of $m_l \times k_l$, $k_l \times n_l$ and $m_l \times n_l$ blocks, respectively. Fig. 1 depicts a matrix multiply-add partitioned with the blocking factors. Workloads for a single $m_l \times n_l$ block of C are allocated to a work-group. The work-group is associated with multiplication of a $m_l \times k$ stripe of A with a $k \times n_l$ stripe of B and addition of the product to an $m_l \times n_l$ block of C for the final result.

An algorithm for the stripe-by-stripe multiplication iterates k/k_l times in the outermost loop of our GEMM algorithm (let us consider matrix sizes m, n, k as multiples of m_l, n_l, k_l, respectively). In every iteration of the outermost loop, the work-group produces the partial result of $m_l \times n_l$ block by multiplying $m_l \times k_l$ block on $k_l \times n_l$ block and adding the product to the $m_l \times n_l$ block. Fig. 2 shows the block-by-block multiplication. Each block is further divided with the smaller blocking factor (m_s, n_s, k_s) such that a work-item in the work-group is in charge of multiplication of $m_s \times k_s$ sub-block of A by $k_s \times n_s$ sub-block of B and accumulation of the product on an $m_s \times n_s$ sub-block of C.

B. Way for accessing matrix data

For storing matrix data in OpenCL, we can choose either of two types of memory objects: buffer objects and image objects. Buffer objects are like one-dimensional arrays in C Language and buffer data are sequentially stored in global memory. Image objects are intended to contain pixel data for image processing, and image data are stored in special global memory called *texture memory*. Using images may take advantage of spatial memory locality for increasing performance. Our code generator has a parameter to designate using either buffer or image for reading data from matrices A and B. For data related to matrix C, buffer is always used.

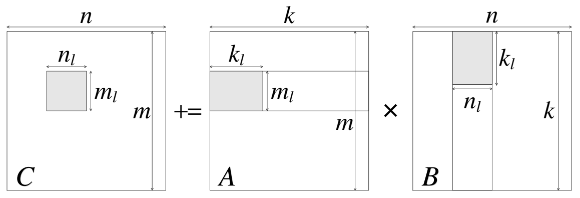


Figure 1. Matrix-matrix multiply-add with the larger blocking factor (m_l, n_l, k_l) in an OpenCL kernel on a GPU

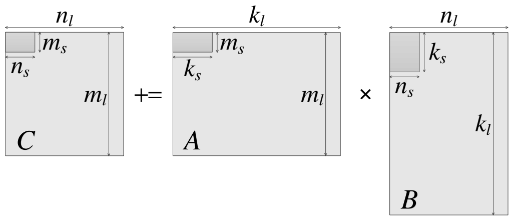


Figure 2. Block-block multiply-add with the smaller blocking factor (m_s, n_s, k_s) in a work-group on a compute unit

As another parameter, we can designate a width of vector variables to the generator. Kernels using different widths show different performance. Note that, when a kernel uses image objects, the width is limited to two in double-precision (`double2`) and four in single-precision (`float4`).

GPUs have a local memory[1] to share data between work-items in a single work-group. The bandwidth of the local memory is higher than that of the L1 cache. For instance, in case of the Tahiti GPU (Radeon HD 7970), the peak bandwidth of the local memory is 3789 GB/s while that of the L1 cache is 1894 GB/s [15]. A drawback of using the local memory is that it requires a barrier synchronization between the work-items, which costs a certain amount of time. The drawback leads to the fact that using the local memory is not always good for higher performance; hence, we have added to the code generator a parameter for either using the local memory for each matrix of A and B or not.

It is known that $A^TB + C$ kernel is the fastest among all four GEMM types in row-major on the Cypress GPU, because of its efficient memory access patterns [10], [16]. For example, the maximum performance of $A^TB + C$ double-precision kernel is 472 GFlop/s while that of $AB + C$ kernel is 359 GFlop/s on the Cypress GPU (Radeon HD 5870) [10]. $A^TB + C$ kernel is the fastest also on the Tahiti GPU (experimental results are shown in Section III). This means that using the $A^TB + C$

[1]In GPUs from AMD, the local memory is named *local data store (LDS)*.

Table I
TAHITI GPU SPECIFICATIONS

Codename Product name	Tahiti Radeon HD 7970
Core clock speed [MHz]	925
Number of compute units (CUs)	32
Number of double-precision (DP) units	512
Number of single-precision (SP) units	2048
Peak DP performance [GFlop/s]	947
Peak SP performance [GFlop/s]	3789
Memory clock speed [MHz]	1375
Global memory size / GPU [GB]	3
L2 cache size / GPU [kB]	768
L1 cache size / CU [kB]	16
Local memory size / CU [kB]	64
Global memory peak bandwidth [GB/s]	264
L2 cache peak bandwidth [GB/s]	710
L1 cache peak bandwidth [GB/s]	1894
Local memory peak bandwidth [GB/s]	3789

kernel also for the other three GEMM types will contribute performance improvement. A solution for the kernel utilization is to copy matrix data in a transposed form if necessary. Such optimization was used previously by Du et al. [9]. They showed that the optimization works effectively for large matrices in which the $O(n^2)$ time of the copying is negligible compared with $O(n^3)$ time of execution of GEMM kernel.

We extend the solution with matrix transposition such that matrix data are stored in global memory in a block-major layout to increase a spatial locality. Currently, as an option, the code generator supports producing $A^TB + C$ kernels where matrix data are supposed to be aligned in either of two block-major layouts shown in Fig. 3 (on an $m \times k$ transposed matrix A with the blocking factor m_l, k_l). Fig. 3(a) shows a column-block major layout and data in each $k \times m_l$ column-block are sequentially stored in a row-major order. We name this layout *column block layout (CBL)*. In CBL, matrix data needed for a multiplication of $k \times m_l$ stripe and $k \times n_l$ stripe are aligned in contiguous memory locations. Fig. 3(b) depicts a row-block major layout and data in each $k_l \times m_l$ sub-block of a $k_l \times m$ row-block are stored in a row-major order. Let us name this layout *row block layout (RBL)*. In RBL, matrix data required for a multiplication of $k_l \times m_l$ block and $k_l \times n_l$ block are aligned in memory. There are past studies on the benefits of using block major layouts to improve memory utilization in CPUs [17], [18].

III. PERFORMANCE EVALUATION

We evaluate the performance of DGEMM (double-precision GEMM) and SGEMM (single-precision GEMM) kernels generated by our code generator on the AMD Tahiti GPU (Radeon HD 7970). The GPU specifications are shown in Table I. The GPU system runs on Ubuntu 10.04. The installed display driver is AMD Catalyst 12.3. We use AMD Accelerated Parallel Processing (APP) SDK v2.6 for OpenCL software development. The present performance evaluation does not take into account data transfer time between host (CPU) and GPU.

The implemented code generator has been used to search the best set of parameters which produces the fastest GEMM kernel. We tested at least ten thousand kernel variants per GEMM type. Such large number of variants were heuristically chosen. Our heuristic search engine measures the performance of generated kernels in order according to a feedback of the measured performance. Note that the search engine is not matured. It takes around three hours to find the best set of parameters for each GEMM type and takes 24 hours to find all the fastest DGEMM and SGEMM kernels shown in this paper. The following shows the procedure for selecting the fastest kernel:

1) Measuring the performance in GFlop/s of every generated GEMM kernel for two problem sizes $n = 1536$ and 4096.
2) Further measuring the performance of the fastest 50 kernels for problems sizes n ($256 \leq n \leq 8192$ in multiples of 256) among a large number of kernels tested in 1).
3) Selecting the fastest kernel among the 50 kernels tested in 2).

The performance of the fastest DGEMM and SGEMM kernels is shown in Fig. 4. Table II shows sets of parameters and the observed maximum performance of the kernels. In row-major, the maximum DGEMM performance (790 GFlop/s) of $A^TB + C$ kernel is higher than that (689 GFlop/s) of $AB + C$ kernel, though the $A^TB + C$ performance fluctuates depending on matrix sizes probably due to memory bank conflicts. It was found that there exist several $A^TB + C$ kernels demonstrating a stable (non-fluctuated) performance tendency with around 620 GFlop/s. The $A^TB + C$ kernel in CBL is most superior to the other shown kernels in terms of the performance and the stability; the achieved performance for this kernel is up to 848 GFlop/s (90% of the peak performance) in double-precision and 2646 GFlop/s (70%) in single-precision. The present code generator limits the size of all blocking factors to a power of two. Therefore, there may be some room for improving the performance.

To make use of the $A^TB + C$ kernel in CBL, matrix data have to be copied into CBL before executing the kernel. We need to prepare two kinds of copying kernels: a kernel copying data into CBL without matrix transposition and a kernel copying data into CBL with transposition. For instance, in case of $AB + C$, the matrix A has to be copied with transposition and the matrix B has to be copied without transposition. Every copying kernel for a square matrix-matrix multiplication (i.e., $m = n = k$) reads $n \times n$ matrix from a memory space in global memory and then, writes the $n \times n$ matrix to another memory space in global memory. The memory bandwidth of every $n \times n$ matrix-to-matrix copying kernel can be calculated as

$$BW(n) \text{ [Bytes / sec]} = \frac{2n^2 \cdot S \text{ [Bytes]}}{\text{Time for copying [sec]}},$$

where $S = 8$ for the double precision and $S = 4$ for the single precision. The measured memory bandwidth of the copying kernels is shown in Fig. 5. The copying kernels

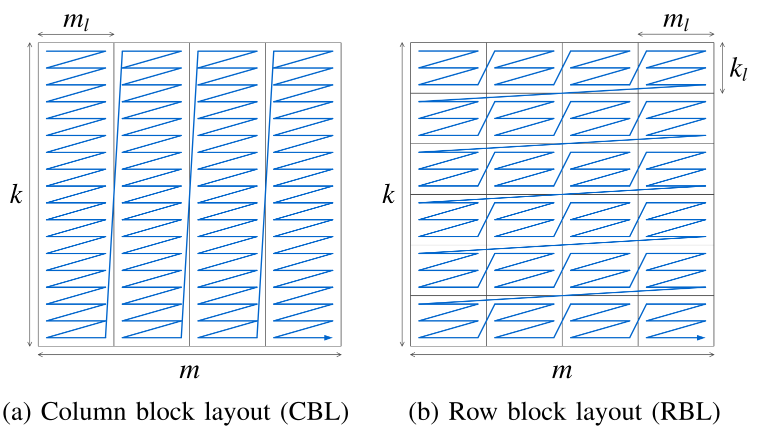


Figure 3. Block-major layouts on an $m \times k$ transposed matrix with the blocking factor m_l, k_l

Table II
PARAMETERS AND PERFORMANCE OF THE FASTEST GEMM KERNELS

Kernel type		m_l, n_l, k_l	m_s, n_s, k_s	Vector[a]	Shared[b]	Image[c]	Performance [Gflop/s]
DGEMM	$A^T B + C$ in CBL	64,16,16	4,4,2	2	B	-	848
	$A^T B + C$ in RBL	64,16,16	4,4,2	2	B	-	812
	$A^T B + C$ in row-major	256,8,16	4,4,2	2	-	-	790
	$AB + C$ in row-major	32,128,128	4,4,4	4	A	-	689
SGEMM	$A^T B + C$ in CBL	128,128,256	16,8,4	4	-	-	2646
	$A^T B + C$ in RBL	128,32,8	8,8,8	2	B	-	2577
	$A^T B + C$ in row-major	128,64,4	8,4,4	4	-	-	2488
	$AB + C$ in row-major	128,64,256	8,4,4	4	-	A	2382

a. Width of vector variables
b. Matrix whose data are shared in local memory
c. Matrix whose data are loaded from image

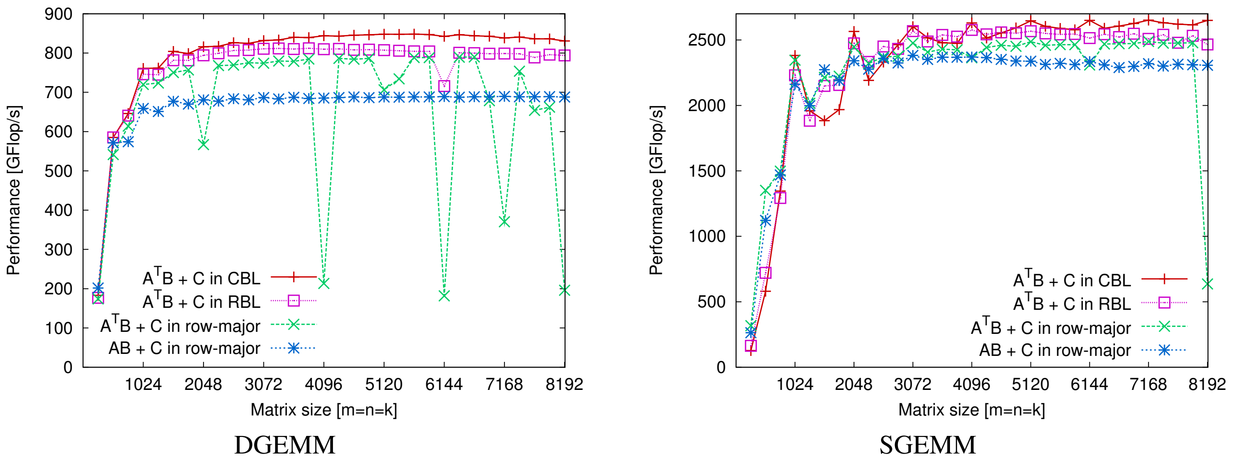


Figure 4. Performance of the fastest GEMM kernels produced by our code generator on the Radeon HD 7970. Matrix sizes are in multiples of 256.

are not deeply tuned; hence, the measured bandwidth of 20-160 GB/s is not high compared with the peak bandwidth (264 GB/s).

Fig. 6 shows the performance of our GEMM implementation with the fastest $A^T B + C$ kernel in CBL supported by the coying kernels. In the figure, the performance is compared with that of the $AB + C$ kernel in row-major and GEMM routines from AMD Accelerated Parallel Processing Math Libraries (APPML) clBLAS 1.8.269 [19]. The time for the copying is amortized when matrix sizes become larger, and our implementation shows higher performance than the $AB + C$ kernel in row-major when $n \geq 1536$ in DGEMM and $n \geq 4096$ in SGEMM. Also, the performance of our implementation is higher than that of APPML. The maximum performance of our DGEMM and SGEMM $AB + C$ implementations is 823 GFlop/s (87% of the peak) and 2541 GFlop/s (67%), respectively. The performance difference among all four GEMM types is not large and within 3% for DGEMM and 5% for SGEMM.

When a matrix size is not in multiples of a blocking factor, we use a zero padding technique. In the padding technique, the values of padded portion in matrices A, B are initialized as zero such that our GEMM implementa-

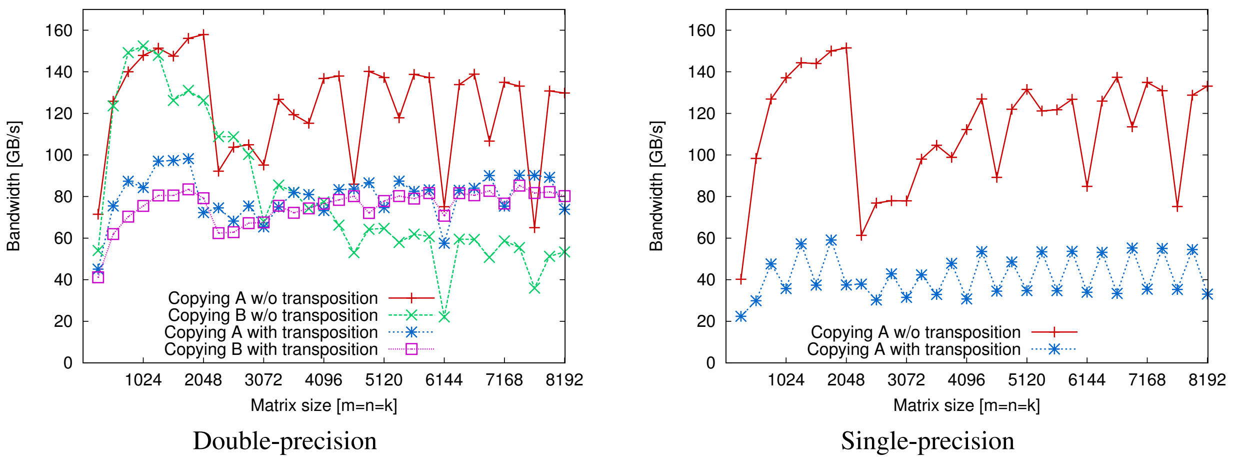


Figure 5. Measured memory bandwidth of copying kernels for utilizing the A^TB+C kernel in CBL. In single-precision case, since the corresponding bandwidth for A and B is identical, the bandwidth for A is only shown.

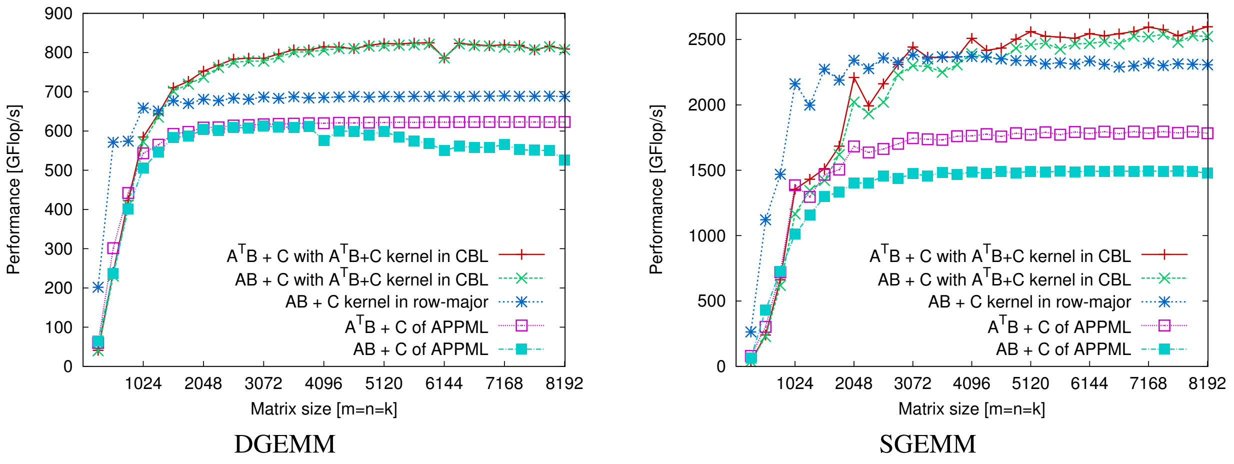


Figure 6. Performance of our GEMM implementation using the fastest A^TB+C kernel in CBL supported by the copying kernels, and the performance comparison with the $AB+C$ kernel in row-major and GEMM routines of AMD APPML clBLAS 1.8.269 [19].

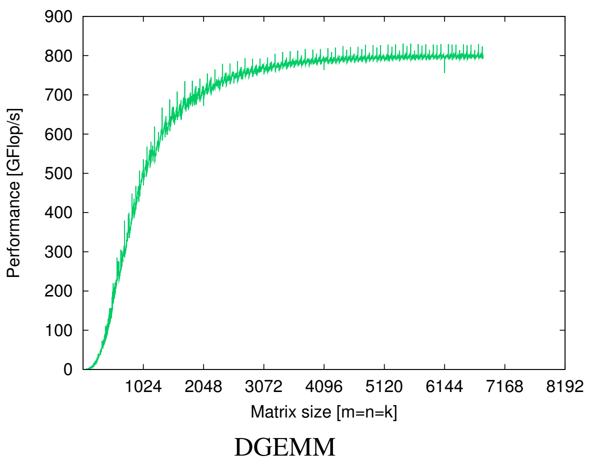
tion can utilize the fastest GEMM kernel in CBL, which is optimized for matrix sizes in multiples of the blocking factor. The performance of the GEMM implementation with the padding is shown in Fig. 7. One disadvantage of the padding is that it causes a performance degradation due to the extra computation for padded portions. The performance degradation is small for relatively large problem sizes. Another side effect of the padding is that it requires three extra buffers for matrices, and as a result, the DGEMM computation becomes impossible in the case of $m = n = k > 6800$ on the 3 GB memory.

In the SGEMM case of Fig. 7, two performance results are shown. The green line indicates the performance of the implementation with the already mentioned A^TB+C kernel in CBL which uses 256 as one of the blocking factors. In the padding, the larger a blocking factor is, the bigger the performance deterioration becomes. To see effects of using different blocking factors on the performance, we test another SGEMM implementation with an A^TB+C kernel in CBL having 64 as the largest blocking factor[2]. The gray line of Fig. 7 represents the performance. As shown, using the smaller blocking factor refrains a drastic performance deterioration although the maximum performance (2430 GFlop/s) of the implementation is a little lower.

IV. Conclusion

This paper has presented our code generator for searching fast GEMM kernels. The performance of GEMM kernels running on GPUs is determined by many factors and it is extremely difficult and time-consuming to develop highly optimized kernels by hand tuning. We think that searching a number of kernel pattens with an auto-tuning system is a good solution for quick development of fast GEMM kernels. Currently, the heuristic search engine in our implemented system is not matured and our system is not a perfect auto-tuning system; however, results of this study show that searching exhaustive kernel patterns by the code generator still contributes development of fast

[2]Parameters of the SGEMM kernel: $\{m_l, n_l, k_l\} = \{64, 32, 32\}$, $\{m_s, n_s, k_s\} = \{4, 8, 4\}$, the width of vector variables is 4, and common data of the matrix B is shared in local memory.



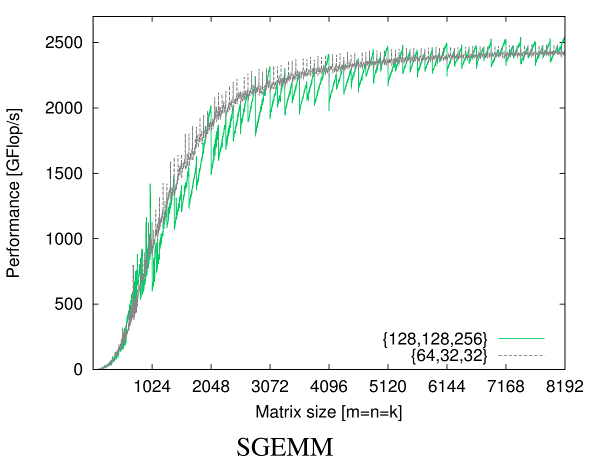


Figure 7. Performance of our $AB + C$ GEMM implementation with a padding technique. In SGEMM, two performance results are shown in different blocking factors $\{m_l, n_l, k_l\}$.

GEMM kernels with modest development time. The code generator supports production of A^TB+C GEMM kernels where matrix data are sequentially stored in a block-major layout. It was shown that using the best kernel in the block-major layout results in achieving higher maximum performance and performance stability than using a kernel in a row-major layout on the Tahiti GPU (Radeon HD 7970).

Since programs written in OpenCL are portable across OpenCL supported devices including multi-core CPUs and GPUs from different vendors, a possible future work is to evaluate GEMM kernels produced by the code generator on several different architectures of CPUs and GPUs. Implementing a code generator for other BLAS routines is another future work.

References

[1] Basic Linear Algebra Subprograms Technical Forum, “Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard,” Aug. 2001. [Online]. Available: http://www.netlib.org/blas/blast-forum/blas-report.pdf

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User’s Guide*, 3rd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[3] B. Kågström, P. Ling, and C. van Loan, “GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark,” *ACM Transactions on Mathematical Software*, vol. 24, no. 3, pp. 268–302, 1998.

[4] R. C. Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, Jan. 2001.

[5] K. Goto and R. A. van de Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, May 2008.

[6] J. Kurzak, W. Alvaro, and J. Dongarra, “Optimizing matrix multiplication for a short-vector SIMD architecture CELL processor,” *Parallel Computing*, vol. 35, no. 3, pp. 138–150, Mar. 2009.

[7] J. Kurzak, S. Tomov, and J. Dongarra, “Autotuning GEMM kernels for the Fermi GPU,” *IEEE Transactions on Parallel and Distributed Systems*, 2012 (in press). [Online]. Available: http://dx.doi.org/10.1109/TPDS.2011.311

[8] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, “Fast implementation of DGEMM on Fermi GPU,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’11)*. Seattle, WA, USA: ACM, Nov. 2011.

[9] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” *Parallel Computing*, vol. 38, no. 8, pp. 391–407, Oct. 2011.

[10] N. Nakasato, “A fast GEMM implementation on the Cypress GPU,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 50–55, Mar. 2011.

[11] J. Kurzak, P. Luszczek, S. Tomov, and J. Dongarra, “Preliminary results of autotuning GEMM kernels for the NVIDIA Kepler architecture - GeForce GTX 680,” 2012, LAPACK Working Note 267. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn267.pdf

[12] Y. Li, J. Dongarra, and S. Tomov, “A note on auto-tuning GEMM for GPUs,” in *Proceedings of the 9th International Conference on Computational Science (ICCS ’09)*, ser. LNCS, vol. 5544. Baton Rouge, LA: Springer-Verlag, May 2009, pp. 884–892.

[13] C. Jang. (Accessed Jun. 27, 2012) GATLAS GPU Automatically Tuned Linear Algebra Software. [Online]. Available: http://golem5.org/gatlas

[14] Khronos Group. (Accessed Jun. 27, 2012) OpenCL - The open standard for parallel programming of heterogeneous systems. [Online]. Available: http://www.khronos.org/opencl

[15] AMD Inc., “AMD Accelerated Parallel Processing OpenCL Programming Guide, rev2.2,” Jun. 2012.

[16] K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, and S. G. Sedukhin, “Multi-level optimization of matrix multiplication for GPU-equipped systems,” in *Procedia Computer Science*, vol. 4. Elsevier B.V., Jun. 2011, pp. 342–351.

[17] V. K. Prasanna, N. Park, and B. Hong, “Tiling, block data layout, and memory hierarchy performance,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 640–654, Jul. 2003.

[18] F. G. Gustavson, “New generalized data structures for matrices lead to a variety of high performance algorithms,” in *Proceedings of the 4th International Conference on Parallel Processing and Applied Mathematics (PPAM ’01)*, ser. LNCS, vol. 2328, 2002, pp. 418–436.

[19] AMD Inc. (Accessed Jun. 27, 2012) AMD Accelerated Parallel Processing Math Libraries (APPML). [Online]. Available: http://developer.amd.com/libraries/appmathlibs