Technical Report 2014-001

Implementing Level-3 BLAS Routines in OpenCL on Different Processing Units

Kazuya Matsumoto, Naohito Nakasato, and Stanislav Sedukhin

October 22, 2014

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

Technical Report 2014-001

Title:
Implementing Level-3 BLAS Routines in OpenCL on Different Processing Units

Authors:
Kazuya Matsumoto, Naohito Nakasato, and Stanislav Sedukhin

Key Words and Phrases:
Level-3 BLAS, GPU, multi-core CPU, many-core processor, OpenCL, performance porting, auto-tuning

Abstract:
This paper presents an implementation of different matrix-matrix multiplication routines in OpenCL. We utilize the high-performance GEMM (GEneral Matrix-Matrix Multiply) implemen- tation from our previous work for the present implementation of other matrix-matrix multiply routines in Level-3 BLAS (Basic Linear Algebra Subprograms). The other routines include SYMM (Symmetric Matrix-Matrix Multiply), SYRK (Symmetric Rank-K Update), SYR2K (Symmetric Rank-2K Update), and TRMM (Triangular Matrix-Matrix Multiply). A key in our approach is to copy given matrix data by copying OpenCL kernels into a form such that a high-performance GEMM kernel can be utilized for computation. We use a previously developed auto-tuning system for the highly optimized copying kernels as well as for GEMM kernel. The performance evaluation of our implementation is conducted on four different GPUs (AMD Radeon R9 290X, FirePro W9100, Radeon HD 7970, and NVIDIA GeForce GTX Titan), a many-core processor (Intel Xeon Phi 5110P), and a multi-core processor (Core i7 3960X). The evaluation results show that the tuning on the copying kernels is effective and contributes to develop high-performance BLAS3 routines.

Report Date:
10/22/2014

Written Language:
English

Any Other Identifying Information of this Report:

Distribution Statement:
First Issue: 10 copies

Supplementary Notes:

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

1 Introduction

As an interface of numerical software library, Basic Linear Algebra Subprograms (BLAS) is widely used in scientific and engineering communities. The BLAS consists of three levels. The Level-1 BLAS is a collection of vector-vector routines, the Level-2 BLAS is dedicated to matrix-vector routines, and the Level-3 BLAS (BLAS3) includes matrix-matrix routines. Processor manufactures, universities, and research institutions have struggled to develop highly-tuned BLAS libraries since the publication of BLAS standard [9, 4].

The General Matrix-Matrix Multiply (GEMM) routine is considered to be the most fundamental BLAS3 routine. Utilizing a tuned GEMM routine as the core computation is generally accepted approach to implement other BLAS3 matrix-matrix routines by considering each of the other routines as a combination of GEMM and a small amount of Level-1 and Level-2 BLAS computations [9]. Igual et al. [6, 5] presented high-performance BLAS3 based on cuBLAS library. They made experiments on three different GEMM-based algorithms for each BLAS3 routine. Their implementation showed better performance without touching underlining CUDA kernels than simply using BLAS3 routine of cuBLAS.

Auto-tuning is an important technique to resolve the problem of performance portability, and it is a well accepted solution for developing high-performance BLAS implementation. ATLAS (Automatically Tuned Linear Algebra Software) [15] is famous projects for auto-tuned BLAS routines on CPUs. Also, several auto-tuning systems for GEMM have been developed [2, 7, 12, 10, 3, 11]. In CUDA, an auto-tuning framework for NVIDIA GPUs has been implemented [10]. In OpenCL, Du et al. [3] presented auto-tuned GEMM routines for GPUs. In our previous work [11], we have also implemented an auto-tuning system for a high-performance GEMM kernel in OpenCL. The GEMM routine is realized with an approach that copies all matrix data into buffers and then runs the auto-tuned GEMM kernel. This present work extends the approach with the matrix data copying for other BLAS3 routines including Symmetric Matrix-Matrix multiply (SYMM), Symmetric Rank-K update (SYRK), Symmetric Rank-2K update (SYR2K), and Triangular Matrix-Matrix multiply (TRMM). We consider that this approach is effective to develop highly efficient BLAS3 routines especially on many-core processors like GPUs.

Another contribution of this work is that we use the developed auto-tuning system to tune copying kernels as well as GEMM kernel. Copying kernels are also an important component in our implementation. Particularly for small problem sizes, the time spending on the data copying is a performance bottleneck in our BLAS3 implementation. We show that the auto-tuning makes it possible to develop high-performance copying kernels.

In this work, our BLAS3 is implemented in OpenCL. OpenCL is a standard framework for parallel programming [8, 1]. Programs written in OpenCL are functionally portable across various processors that include CPUs, GPUs, and other computing devices such as Intel Xeon Phi co-processor and FPGA. This paper presents results of

performance evaluation on the following different processing units:

1. AMD Radeon R9 290X (Volcanic Islands);
2. AMD FirePro W9100 (Volcanic Islands);
3. AMD Radeon HD 7970 (Southern Islands);
4. NVIDIA GeForce GTX Titan (Kepler);
5. Intel Xeon Phi 5110P (MIC);
6. Intel Core i7 3960X (Sandy Bridge).

The rest of this paper is organized as follows: Section 2 presents the basic information on OpenCL. Section 3 describes our approach for high-performance BLAS3 implementation. Section 4 shows the performance evaluation of the BLAS3 implementation and provides comparison with the existing vendor provided BLAS libraries. Finally, Section 6 gives the concluding remarks of this study.

2 OpenCL Basics

OpenCL (Open Computing Language) is an open standard for general purpose parallel programming on heterogeneous computing platforms. The OpenCL framework includes a C99-based programming language for writing parallel functions called *kernels*, and runtime APIs for controlling OpenCL platforms and devices.

An OpenCL platform consists of one or more OpenCL devices. An OpenCL device comprises multiple *compute units* (CUs), each of which has multiple *processing elements* (PEs). When an OpenCL kernel is executed on the device, an N-dimensional index space, which is called *NDRange*, is defined. In this work, we consider a two-dimensional index space only, which is suitable for dealing with matrix data. Each instance in an NDRange, is called a *work-item*. Every work-item has a unique ID. Several work-items organize a *work-group*. A work-item runs on one or more PEs. A task on a work-group is processed by all the PEs of a CU.

In an OpenCL kernel, four distinct memory regions are accessible to work-items.

1. *Global memory* (gm) is a memory region in which data can be read/written by all work-items. It is impossible to synchronize work-items during kernel execution in this memory.
2. *Constant memory* (cm) is a read-only region of global memory. Data in this region are not changed during execution.
3. *Local memory* (lm) is a specific memory region to a work-group. Work-items in a work-group can share data in local memory.

4. *Private memory* (pm) is a specific memory region to a work-item. Data in private memory of a work-item is not visible to other work-items. On most OpenCL devices, private memory is in the register file.

3 Implementation

We use a common strategy in our implementation of different matrix-matrix multiply routines. The strategy is to copy matrix data into a storage layout such that a high-performance GEMM kernel can be utilized for computation.

3.1 GEMM

Let us review the GEMM implementation presented in our previous paper [11]. We focus on the matrix multiply-add $C \leftarrow C + AB$ operation. The current implementation supposes column-major as the initial matrix storage order for consistency with BLAS standard[1].

Our GEMM routine is implemented with a GEMM ($AB^T + C$) kernel and a data copying kernel in OpenCL. The copying kernel is not simply replicating matrix data into additionally allocated memory space (buffer). For computing $AB + C$, transposing the matrix B is necessary to utilize the auto-tuned $AB^T + C$ kernel. Moreover, the tuned GEMM kernel assumes that matrix data are aligned in a block-major order for high performance, and zeros are padded if a matrix dimension is not in multiples of blocking factor. In summary, the copying kernel conducts matrix transposition, storage layout changing and zero padding if necessary. For simplicity, the following descriptions assume cases where matrices are square, all matrix dimensions are multiples of any blocking factors, and matrix data are aligned in column-major order if not specified.

The copying kernel as well as GEMM kernel uses two levels of blocking for efficiently utilizing caches of processing units. In this paper, we describe values on the first level of blocking for work-item processing as M_{wi}, N_{wi}, K_{wi} and values on the other level of blocking for work-group processing as M_{wg}, N_{wg}, K_{wg}. Fig. 1 delineates a kernel which copies matrix A data in column-major to a buffer in a form of *column-block column-major* (CBC) order for our GEMM routine. The matrix-matrix multiply operation part is generally represented as `GEMM_BODY` in Fig. 2. After copying matrices A and B to buffer, our GEMM routine calls the GEMM kernel (see Fig. 3).

3.2 SYMM

For SYMM routine, we focus on $C \leftarrow AB + C$, where A is a symmetric matrix and only the lower triangular part stores the matrix data. The difference between GEMM and SYMM is only in an operation on the matrix A: our SYMM routine are implemented by only changing an OpenCL kernel which copies the matrix A from the GEMM routine.

[1] In our previous work, the GEMM OpenCL kernel fundamentally supposes row-major order.

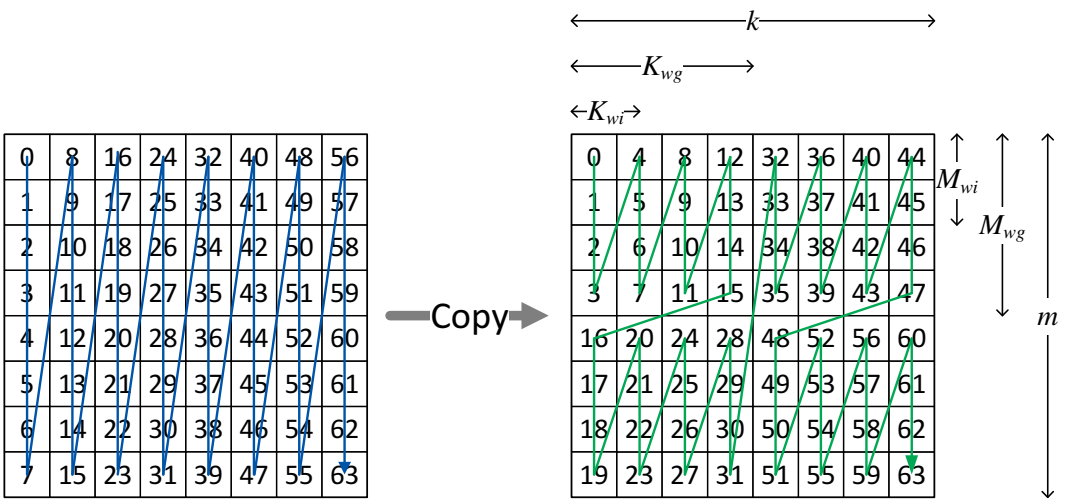

Figure 1: Copying matrix data from column-major order into a column-block column-major order with blocking factors $M_{wg}, K_{wg}, M_{wi}, K_{wi}$

`GEMM_BODY`$(A, B, Start, End)$

1: $C_{pm} := 0$
2: **for** $p = Start$ **to** $End - K_{wi}$ **step** K_{wi} **do**
3: Load $M_{wi} \cdot K_{wi}$ elements of A from global memory into private memory (A_{pm})
4: Load $K_{wi} \cdot N_{wi}$ elements of B from global memory into private memory (B_{pm})
5: $C_{pm} += A_{pm} \times B_{pm}$
6: **end for**
7: Return C_{pm}

Figure 2: Body of GEMM kernel

`GEMM_KERNEL`(m, n, k, A, B, C)

1: $C_{pm} :=$ `GEMM_BODY`$(A, B, 0, k)$
2: // Accumulating the product C_{pm} to the corresponding $M_{wi} \cdot N_{wi}$ data in global memory (C_{gm})
3: $C_{gm} := C_{pm} + C_{gm}$

Figure 3: GEMM kernel algorithm

The newly prepared kernel copies data of A into a buffer A' in a form of symmetric matrix. This means that the lower triangular and the diagonal parts of A' is filled with the lower triangular matrix A straightforwardly and the upper triangular part of A' is filled with transposed data of the lower triangular matrix A:

$$a'(i,p) = \begin{cases} a(i,p) & \text{if } i \geq p; \\ a(p,i) & \text{if } i < p, \end{cases}$$

where $a(i,p)$ and $a'(i,p)$ represents the element of A and A' in (i,p) position, respectively. Since we can treat SYMM as GEMM after the copying, the SYMM routine simply calls the `GEMM_KERNEL`.

3.3 SYRK

For SYRK routine, we select $C \leftarrow C + AA^T$, where only the lower triangular part of C is stored and updated. Because of the lower triangular property, we are required to prepare a different matrix multiplication kernel for SYRK based on the GEMM kernel. Fig. 4 gives the SYRK kernel algorithm. The auto-tuned GEMM kernel (GEMM_BODY in Fig. 2) is called inside the SYRK algorithm.

The newly prepared SYRK kernel only updates the lower triangular part. To realize the operation, an work-group containing global work-item ID (y,x) updates the responsible $M_{wg} \cdot N_{wg}$ elements of a block if the block is in lower triangular part of C, otherwise, it returns immediately. All the work-items of a work-group assigned for updating a diagonal block are responsible to compute the block in a parallel fashion redundantly to avoid conditional statements which yield large latencies.

`SYRK_KERNEL`(n, k, A, B, C)
1: Get a two-dimensional global ID (y,x)
2: **if** $y \cdot (M_{wi}/M_{wg} + 1) \cdot M_{wi} < x \cdot N_{wi}/N_{wg}$ **then**
3: Return // Assigned block for (y,x) is not in the lower triangular part
4: **end if**
5: C_{pm} := `GEMM_BODY`$(A, B, 0, k)$
6: Merge C_{pm} with $M_{wi} \cdot N_{wi}$ elements only of the lower triangular part of C

Figure 4: SYRK ($C := C + AA^T$) kernel algorithm

3.4 SYR2K

For SYR2K routine, we focus on $C := C + AB^T + BA^T$, where only the lower triangular part of C is stored and updated. Our SYR2K implementation simply calls the above SYRK routine twice, i.e., $C := C + AB^T$ and $C := C + BA^T$ computations are conducted in a consecutive order.

3.5 TRMM

For TRMM routine, we focus on $B := AB$, where A is a upper triangular matrix. Like the SYRK case, our TRMM OpenCL kernel is a slightly different from the GEMM kernel. Since a multiplication of the upper triangular part of A and the general matrix B is necessary, the TRMM kernel shifts the starting indexes of the multiplication in a work-group in units of the work-group blocking factor K_{wg}. Fig. 5 schematically

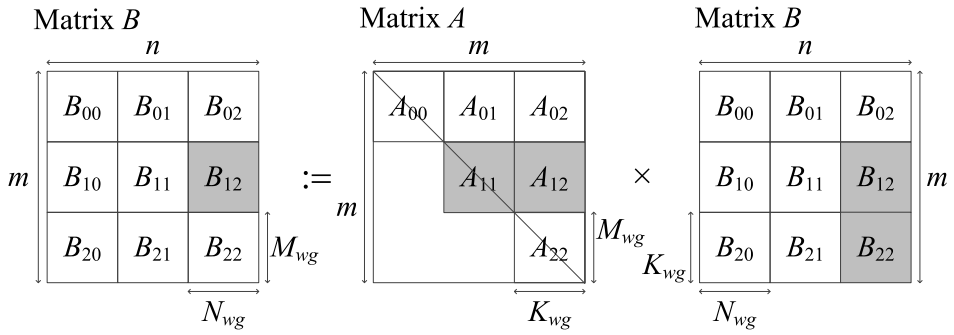

Figure 5: TRMM ($B := AB$) scheme

illustrates our TRMM implementation. The following statement shows the operations:

$$B_{IJ} := \sum_{P=I}^{K-1} A_{IP} B_{PJ} \quad (0 \leq I, J \leq m/M_{wg} - 1 \text{ and } K = m/K_{wg}).$$

In addition to the TRMM kernel, we have prepared a copying kernel to make the TRMM computation possible. Like in the SYMM case, the prepared kernel copies data of A into a buffer A'. This indicates that data elements of the upper triangular and diagonal parts of A' are filled with the corresponding data of A, and all elements of the lower triangular part of A' are initialized with zero:

$$a'(i,p) = \begin{cases} a(i,p) & \text{if } i \leq p; \\ 0 & \text{if } i > p. \end{cases}$$

3.6 Performance Tuning of Copying Kernels

As mentioned above, several copying kernels are required in addition to the matrix-matrix multiplication kernels. The amount of matrix data copying is $O(n^2)$ for general square $(n \times n)$ matrices while the amount of the computations (multiply-adds) is $O(n^3)$ in case $m = n = k$. The larger the matrix size is, the copying time becomes less critical in the total computation time. Nevertheless, we try to tune copying kernels as well to avoid using these kernels whose performance is low.

Among BLAS3 routines implemented in this work, GEMM, SYMM, and TRMM routines require respectively different copying kernels. We have newly implemented code generators which produce copying kernels for the auto-tuning. Each code generator takes a set of six parameters which are two work-group blocking factors M_{wg}, N_{wg}, two work-item blocking factors M_{wi}, N_{wi}, and two loop unrolling degrees M_{ud}, N_{ud}. Fig. 6 represents a sample code algorithm of copying kernel (for GEMM without any data converting operations like matrix transposition). Note that the work-group blocking factors do not appear in the code; these appear in a routine interface code. We utilize the developed auto-tuning system for tuning the copying kernels in the same manner as for the GEMM kernel.

```
COPY_KERNEL(m, k, A_src, A_dst)
1: Get a two-dimensional global ID (y, x)
2: #pragma unroll (M_ud)
3: for i = y to y + M_wi − 1 do
4:    #pragma unroll (N_ud)
5:    for j = x to x + N_wi − 1 do
6:       A_dst(i, j) := A_src(i, j)
7:    end for
8: end for
```

Figure 6: Sample algorithm of copying kernel

An advantage of the present approach is that the necessary time for tuning copying kernels are much smaller than the necessary time for GEMM kernel. A search space for each copying kernel is very small compared to a search space for the GEMM kernel. The number of tunable parameters on a copying kernel is six; hence, to find a near-optimal parameter setting, we are required to measure the performance of hundreds of kernels only. On the GEMM kernel, the number of tunable parameters is about 20 and we are required to test tens of thousands of kernels, which takes us much time for the auto-tuning. A disadvantage of the approach is that every routine requires additional memory space as destination buffers of data copying. In case of a BLAS3 computation on square ($n \times n$) matrices, additional memory space for temporarily storing two matrices ($2n^2$ data elements) is needed.

4 Experimental Results

We evaluate our implementation of BLAS3 routines on four different GPUs (AMD Radeon R9 290X, FirePro W9100, HD 7970, and NVIDIA GeForce GTX Titan), a many-core co-processor (Intel Xeon Phi 5110P), and a multi-core processor (Intel Xeon Core i7 3960X). Specifications of the processors are shown in Table 1. This paper describes the performance of each BLAS3 routine, and does not take into account data communication time between the host and OpenCL device. Notable characteristics of each device are as follows:

- The AMD Radeon R9 290X is one of the AMD Volcanic Islands series that adopts the Graphic Core Next (GCN) architecture. The GCN is a RISC (Reduced Instruction Set Computer) SIMD (Single-Instruction Multiple-Data) architecture. The R9 290X contains 44 compute units (CUs). All CUs share a 1 MB L2 cache. Each CU has four vector units, a scalar unit, a 16 KB read/write L1 cache, and a 64 KB local data share (32 KB is available to a single wavefront). Each vector unit consists of 16 processing elements (PEs); thus, the R9 290X is equipped with 2816 ($= 44 \cdot 4 \cdot 16$) PEs. The performance in double-precision is limited to one-eighth that in single precision.

- The AMD FirePro W9100 is one of the AMD Volcanic Islands series for usage on workstations. The W9100 and R9 290X have the same GPU codename ("Hawaii") and characteristics of W9100 are mostly same as R9 290X. An important difference between the two GPUs is that the performance in double precision is half that in single precision (no limitation).

- The AMD Radeon HD 7970 is one of the AMD Southern Islands series that also adopts the GCN architecture. The HD 7970 contains 32 compute units (CUs). All CUs share a 768 KB L2 cache. Each CU has four vector units, a scalar unit, a 16 KB read/write L1 cache, and a 64 KB local data share (32 KB is available to a single wavefront). Each vector unit consists of 16 processing elements (PEs); thus, the HD 7970 is equipped with 2048 ($= 32 \cdot 4 \cdot 16$) PEs.

- The GeForce GTX Titan is a NVIDIA Kepler GF110 GPU. The GTX Titan contains 14 Streaming Multiprocessor eXtremes (SMXs). All SMXs share a 1.5 MB L2 cache. Each SMX has a 64 KB L1 cache, a 64 KB shared memory, 192 CUDA cores; thus, the GTX Titan is equipped with 2688 ($= 14cdot192$). The performance in double precision is one-third of the single precision.

- The Xeon Phi 5110P is one of the Intel's first commercial products in MIC (Many Integrated Core) architecture. The Xeon Phi features 60 in-order cores (59 cores available for computation) connected by a high-performance bidirectional ring interconnect. Each core supports four-way hyper-threading to hide memory and instruction latencies. The computation unit of the core is a vector unit which has a 512-bit wide register file (32 registers per thread) and can execute 8-way double-precision or 16-way single-precision floating-point SIMD instructions in a single clock. Additionally, each core has a 32 KB L1 data cache, a 32 KB L1 instruction cache, and a 512 KB L2 cache.

- The Core i7 3960X is a six-core CPU of Intel's Sandy Bridge architecture. The CPU has 15 MB L3 cache. The processing core has a 32 KB L1 cache and 256 KB L2 cache. Also, the core has a 256-bit wide register file and can execute four-way double-precision or eight-way single-precision floating-point SIMD (AVX) instructions in a single clock.

Table 1: OpenCL device specifications

Device name	Radeon R9 290X	FirePro W9100	Radeon HD 7970	GeForce GTX Titan	Xeon Phi 5110P	Core i7 3960X
Vendor	AMD			NVIDIA	Intel	
Architecture	Volcanic Islands		Southern Islands	Kepler	MIC	Sandy Bridge
Codename	Hawaii		Tahiti	GK110	Knights Corner	-
Core clock speed [MHz]	1000	930	925	876	1053	3300
Maximum SP operations per clock	5632	5632	4096	5376	1888	96
Maximum DP operations per clock	704	2816	1024	1792	944	48
Peak SP performance [GFlop/s]	5632	5238	3789	4709	1988	316.8
Peak DP performance [GFlop/s]	704	2619	947	1570	994	158.4
Global memory size [GBytes]	4	16	3	6	8	-
Global memory bandwidth [GByte/s]	320	320	264	288	320	-
Local memory size [KBytes]	32	32	32	48	32	32
Local memory type	Scratchpad	Scratchpad	Scratchpad	Scratchpad	Global	Global
OpenCL SDK	AMD APP SDK 2.9			CUDA 5.5	Intel SDK 2013	
Display driver version	14.4 [a]	13.352.1014 [b]	14.4 [a]	319.21[c]	-	

SP: Single-precision
DP: Double-precision

[a]: Catalyst driver version
[b]: FirePro Unified Driver version
[c]: CUDA driver version

4.1 Performance on Radeon R9 290X

This section presents performance measurement results on the Radeon R9 290X GPU. We compare our implementation with clMathLibraries clBLAS 2.2.0. The clBLAS is an open-source BLAS library in OpenCL and is being developed mainly by AMD employees. In the clBLAS, a program for tuning BLAS routines is provided, and the performance results of clBLAS are measured after applying the tuning program.

Fig. 7 shows the performance of SSYMM routine. Our implementation demonstrates higher performance for most of matrix sizes than clBLAS, and the performance is 2.8 times higher on average. Relative performances of BLAS3 implementation to clBLAS are shown in Fig. 8 (higher is better). The STRMM and SGEMM routines achieves better performance for large matrix sizes. We do not see a large difference in performances of the SSYRK and SSYR2K routines.

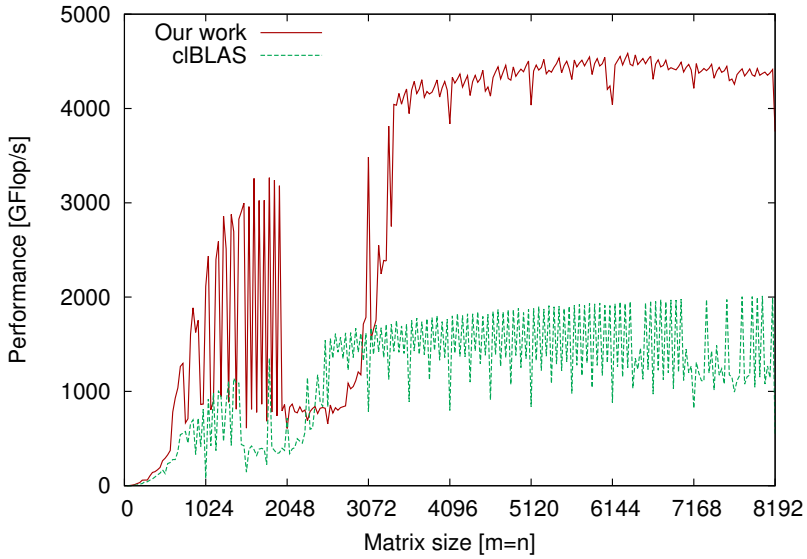

Figure 7: SSYMM performance on Radeon R9 290X

Fig. 9 shows a measured bandwidth of copying kernel for the SSYMM routine, and it compares the bandwidth of auto-tuned copying kernel with that of non-tuned copying kernel[2]. The auto-tuning enables to develop a copying kernel whose performance is much higher than the non-tuned kernel. The bandwidth of tuned kernel is up to 110 GB/s while the bandwidth of non-tuned kernel is up to 2 GB/s. To show another evidence why tuning of a copying kernel is necessary, we present the ratio of copying time to total computation time for the SSYMM routine in Fig. 10. When we do not tune the copying kernel, the copying time dominates over 65% of total computation time even for matrix sizes of around 8000. By the auto-tuning, the ratio decreases to less than 5% for these sizes. The auto-tuning of the copying kernel contributes to develop high-performance routines.

[2]All parameters $(M_{wg}, N_{wg}, M_{wi}, N_{wi}, M_{ud}, N_{ud})$ of the non-tuned copying kernel are set as 1.

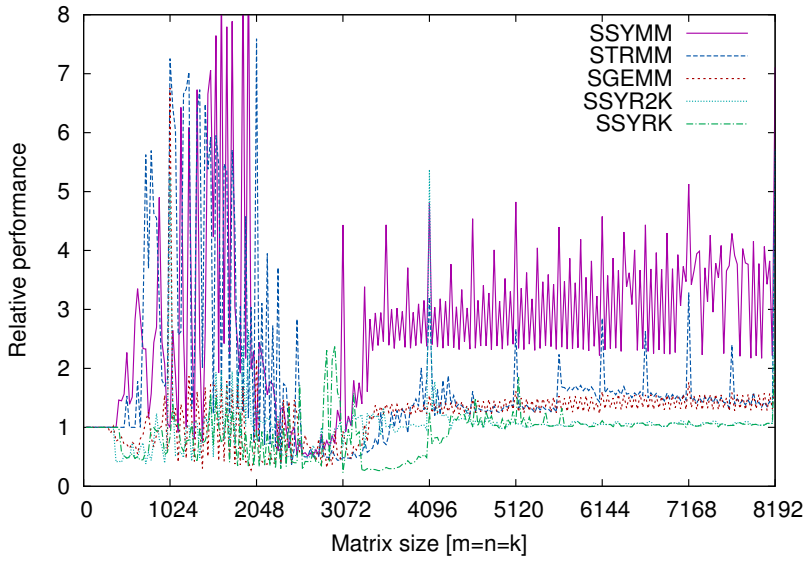

Figure 8: Relative performance of our BLAS3 implementation to clBLAS

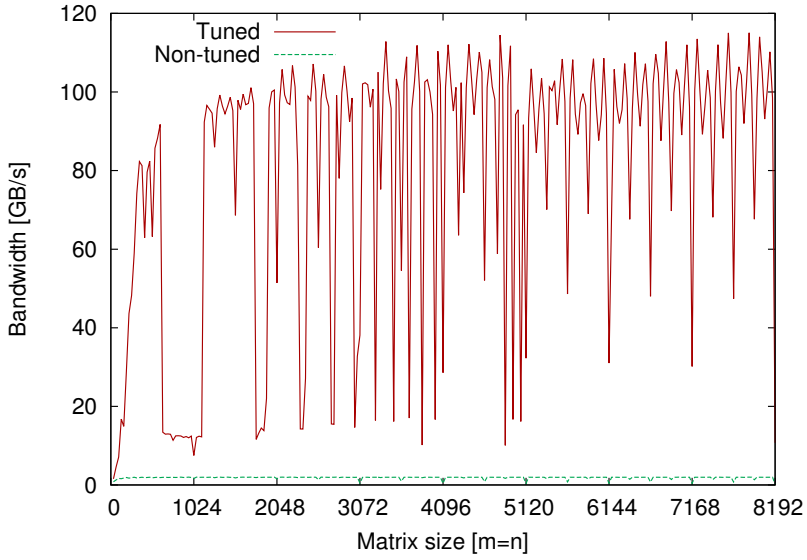

Figure 9: Bandwidth of copying kernel for SSYMM routine

Our SSYMM routine demonstrates a similar performance to our DGEMM routine since we use the identical GEMM kernel usage in both routines. However, the other three routines (SSYRK, SSYR2K, and STRMM) requires small modifications in GEMM kernel usage to implement as described in Section 3.3-3.5. Fig. 11 depicts the relative performance of each BLAS3 routine to the performance of the SGEMM routine. The performance of the three routines is lower than that of SSYMM. The SSYRK and SSYR2K routines demonstrate an almost identical performance since the two routines

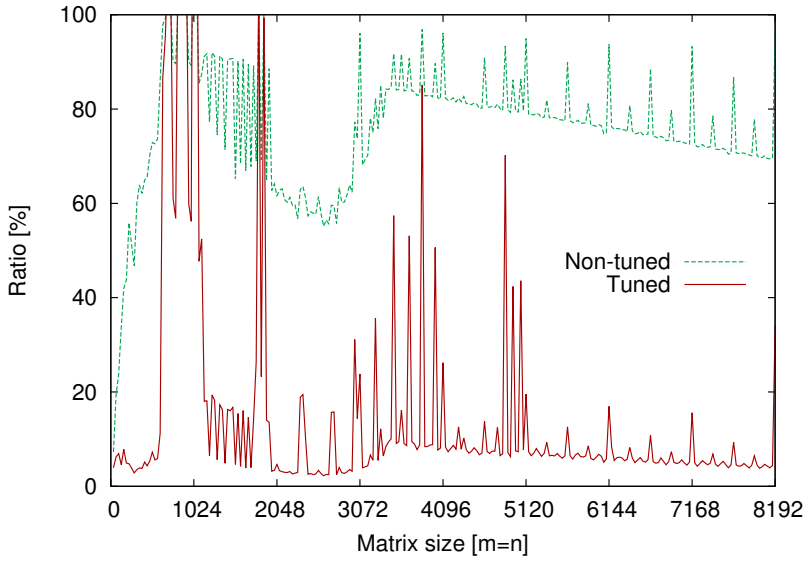

Figure 10: Retio of copying time to total computation time for routine

share utilized OpenCL kernels. The performance of STRMM routine is the lowest among all our BLAS3 routines on the GPU.

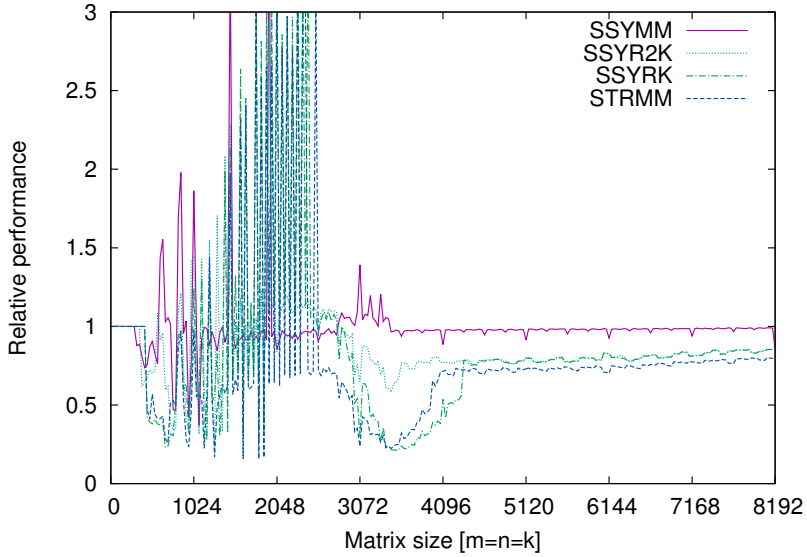

Figure 11: Relative performance of each BLAS3 routine to SGEMM routine

4.2 Performance on Different Processing Units

This section presents experimental results on different processing units (see Table 1). Table 2 describes the maximum measured performance of our implementation as well

as a vendor BLAS library on the processing units. The R9 290X and W9100 show a little different performance's behaviour though the GPU architecture is same (Volcanic Islands). The STRMM routine works more efficiently than SSYRK and SSYR2K on the W9100 while it is opposite on the R9 290X. Among the present devices, W9100 is the fastest for the double-precision routines while R9 290X is the fastest for the single-precision routines. For the GTX Titan, our implementation in OpenCL works slower than cuBLAS. We suppose that the OpenCL runtime in CUDA toolkit is less tuned than the CUDA runtime. Our implementation on the Xeon Phi and Core i7 is worse than MKL. MKL is highly tuned in an assembly language level, and this result shows another evidence that it is hard to outperform the program written in assembly level using a high-level language.

Table 2: Maximum performance in GFlop/s on different OpenCL devices

Device	R9 290X		W9100		HD 7970		GTX Titan		Xeon Phi		Core i7	
Impl.	Ours	#1	Ours	#1	Ours	#1	Ours	#2	Ours	#3	Ours	#3
SGEMM	4650	3617	4223	3341	2913	2472	2300	3374	545	1701	97	277
SSYMM	4584	2013	4113	1698	2871	1047	2292	2620	544	1188	97	273
SSYRK	3853	3619	2295	3325	2096	2448	2183	3248	91	768	95	256
SSYR2K	3822	3623	2223	3307	2774	2448	2211	3164	89	205	95	254
STRMM	3623	2649	3157	2463	2758	1779	2226	2908	13	510	91	262
DGEMM	670	654	1778	1019	823	662	958	1349	269	841	60	136
DSYMM	665	488	1708	584	818	407	955	1229	266	649	59	131
DSYRK	657	664	1138	1008	734	739	929	1332	34	646	55	125
DSYR2K	655	665	1096	991	796	740	910	1307	32	277	55	117
DTRMM	643	576	1576	732	789	538	820	1189	10	281	54	128

#1: clBLAS (clMathLibraries clBLAS) 2.2.0
#2: cuBLAS (CUDA BLAS in CUDA) 5.5 Toolkit
#3: MKL (Math Kernel Library) 11.0 update 1

5 Related Work

Igual et al. [6, 5] designed and evaluated a high-performance implementation of different Level-3 BLAS routines. Their implementation takes advantage of existing the NVIDIA cuBLAS library and attains huge speedups over the cuBLAS. The cuBLAS at that time does not show an equal performance for all Level-3 BLAS routines (the GEMM routine runs much more efficiently than the other routines). For their BLAS3 implementation, they presented matrix-panel (MP), panel-matrix (PM), panel-panel (PP) product based variants, where both matrix dimensions is large in the "matrix" and one of the dimensions is small in the "panel". In this point of view, our approach, presented in this paper, is considered to be a matrix-matrix (MM) product variant.

As for SYRK routine implementation on a GPU, Nath et al.[14] have used a thread-block reordering technique to limit the computation only to data blocks that are on the diagonal or in the in the lower/upper triangular part of the matrix. All the threads in diagonal blocks compute half of the block in a parallel fashion redundantly to avoid conditional statements that have been required otherwise. Their SYRK implementation showed higher performance than cuBLAS at that time [13]. Our approach for SYRK routine also limits the computation in the lower triangular part and implements the redundant computation for the diagonal blocks.

6 Conclusion

In this paper, we have presented the implementation of different Level-3 BLAS (BLAS3) routines in OpenCL. A key in the implementation is to utilize data copying kernels to realize each BLAS3 routine. In the present work, the copying kernels as well as GEMM kernel are highly tuned by an auto-tuning technique. We have parameterized each copying kernel and implemented code generators which produce copying kernels for the auto-tuning. Experimental results show that tuning the copying kernels greatly contributes to developing of high-performance BLAS3 routines.

The proposed approach utilizes a highly-tuned GEMM kernel for other existing BLAS3 routines. We found that the performance of the SYMM routine is almost equivalent to that of GEMM routine while the performance of the SYRK, SYR2K, and TRMM routines is a little lower than that of GEMM. A future work is to investigate the reason of such low performance for a better BLAS implementation.

References

[1] AMD Inc.: AMD Accelerated Parallel Processing OpenCL Programming Guide, rev2.4 (Dec 2012)

[2] Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing matrix multiply using PHiPAC : a portable, high-performance, ANSI C Coding methodology. Tech. rep., Computer Science Department, University of Tennessee (May 1996), `http://www.netlib.org/lapack/lawnspdf/lawn111.pdf`

[3] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 38(8), 391–407 (Oct 2011)

[4] Goto, K., van De Geijn, R.: High-performance implementation of the level-3 BLAS. ACM Transactions on Mathematical Software (TOMS) 35(1), 4:1–4:14 (Jul 2008)

[5] Igual, F.D.: Matrix computations on graphics processors and clusters of GPUs. Ph.D. thesis, Universitat Jaume I. Castellón (2011)

[6] Igual, F.D., Quintana-Qrtí, G., van De Geijn, R.A.: Level-3 BLAS on a GPU: Picking the low hanging fruit. Tech. Rep. DICC 2009-04-01, Depto. de Ingenieraíy Ciencia de Computadores, Universidad Jaime I de Castellón (2009)

[7] Jiang, C., Snir, M.: Automatic tuning matrix multiplication performance on graphics hardware. pp. 185–194. IEEE (Sep 2005)

[8] Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems (Accessed Sep 18, 2014), `http://www.khronos.org/opencl`

[9] Kågström, B., Ling, P., Van Loan, C.: GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. ACM Transactions on Mathematical Software 24(3), 268–302 (1998)

[10] Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems 23(11), 2045–2057 (Nov 2012)

[11] Matsumoto, K., Nakasato, N., Sedukhin, S.G.: Performance tuning of matrix multiplication in OpenCL on different GPUs and CPUs. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC 2012). pp. 396–405. IEEE CS's Conference Publishing Service, Salt Lake City, Utah, USA (Nov 2012)

[12] Nath, R., Tomov, S., Dongarra, J.: An improved MAGMA GEMM for Fermi graphics processing units. International Journal of High Performance Computing Applications 24(4), 511–515 (2010)

[13] Nath, R., Tomov, S., Dongarra, J.: BLAS for GPUs. In: Bailey, D.H., Lucas, R.F., Williams, S. (eds.) Performance Tuning of Scientific Applications, chap. 4. CRC Press (2010)

[14] Nath, R., Tomov, S., Dongarra, J.: Accelerating GPU kernels for dense linear algebra. In: Proceedings of the 9th International Meeting High Performance Computing for Computational Science (VECPAR 2010). LNCS, vol. 6449, pp. 83–92. Springer (2011)

[15] Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1-2), 3–35 (Jan 2001)