A Fast GEMM Implementation On a Cypress GPU

Naohito Nakasato
University of Aizu
Aizu-Wakamatsu
Fukushima, Japan
nakasato@u-aizu.ac.jp

ABSTRACT

We present benchmark results of optimized dense matrix multiplication kernels for Cypress GPU. We write general matrix multiply (GEMM) kernels for single (SP), double (DP) and double-double (DDP) precision. Our SGEMM and DGEMM kernels show ~ 2 Tflop/s and ~ 470 Glop/s, respectively. These results for SP and DP correspond to 73% and 87% of the theoretical performance of the GPU, respectively. Currently, our SGEMM and DGEMM kernels are fastest with one GPU chip to our knowledge. Furthermore, the performance of our matrix multiply kernel in DDP is 31 Gflop/s. This performance in DDP is more than 200 times faster than the performance results in DDP on single core of a recent CPU (with mpack version 0.6.5). We describe our GEMM kernels with main focus on the SGEMM implementation since all GEMM kernels share common programming and optimization techniques. While a conventional wisdom of GPU programming recommends us to heavily use shared memory on GPUs, we show that texture cache is very effective on the Cypress architecture.

Categories and Subject Descriptors

F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Single-instruction-stream, multiple-data-stream processors

1. INTRODUCTION

Dense matrix computation is an important problem in computer science and engineering. Since it is compute intensive and exhibit regular memory access, it is well suited for acceleration by a many-core architecture like GPUs. There were previous works to implement and try to speed-up dense matrix computation especially dense matrix multiplication with GPUs [7, 18, 13]. Many important numerical algorithms such as BLAS-3 itself and LU factorization rely on high speed GEMM implementations [11, 5, 18]. In [18], it has shown that after detailed and precise analysis of GPUs from NVIDIA they could significantly accelerate one sided factorizations in SP.

In this paper, we present our effort to implement GEMM kernels for GPUs from AMD. Our focus on this paper is how we program GPUs in an optimal way for GEMM computations. We present benchmark results for SGEMM and DGEMM. Furthermore, for the first time, we show GEMM in DDP (DDGEMM) is very fast on GPUs and present benchmark results. We believe DDGEMM will be rather critical in many applications with a trend such that we will require ever larger problem size.

This paper organized as follows. In Section 2, we briefly summarize architecture of a GPU that we use in the present work and show comparisons with a competing GPU. We refresh a basic fact on a blocking GEMM algorithm in Section 3. Section 4 is our main contribution that describe details and analysis of our GEMM kernels for the GPU. In this section, we also present the benchmark results of SGEMM, DGEMM and DDGEMM. Finally, we conclude in Section 6 with a mention to future work.

2. GPU ARCHITECTURE

In this section, we briefly summarize the Cypress GPU architecture[3].

2.1 Cypress Architecture

The Cypress GPU from AMD is the company's latest GPU (as of time of writing) with many enhancements for general purpose computing on GPU (GPGPU). It has 1600 arithmetic units. Each arithmetic unit called a stream core (SC) is capable of executing SP floating-point (FP) multiply-add. Stream cores are organized hierarchically as follows. At one level higher from the stream cores, a five-way very long instruction word (VLIW) unit called a thread processor (TP) that consists of four simple stream cores and one transcendental stream core. Therefore, one Cypress processor has 320 TPs. A TP can execute either at most five SP/integer operations, four simple SP/integer operations with one transcendental operation, or DP operations by combinations of the four stream cores. Each TP has a register file of 1024 words where one word is 128 bit (4 SP words or 2 DP words). 16 TPs are grouped into to consist of an unit called a SIMD engine. A SIMD engine is an unit of kernel execution in the Cypress GPU. All TPs in the SIMD engine work in single-instruction-multiple-thread (SIMT) way. At the top level of the GPU, there are 20 SIMD engines, a controller unit

Architecture	Cypress	Fermi
Board Name	Radeon 5870	Tesla C2050
# of SP cores	1600	448
# of DP cores	320	224
# of vector cores	20	14
registers/core	256 KB	128 KB
tex. cache/core	8 KB	12 KB
shared mem./core	32 KB	64 KB
2nd cache	512 KB	768 KB
core clock(GHz)	0.85	1.15
SP peak(Tflop/s)	2.72	1.03
DP peak(Gflop/s)	544	515
memory clock(GHz)	1.2	0.75
memory bus	256	384
memory size(GB)	1	3
memory BW (GB/s)	153.6	144
tex. cache BW(GB/s)	54.5	

Table 1: Comparison between Cypress and Fermi GPU boards. Register, texture cache and shared memory size is per vector core.

called an ultra-threaded dispatch processor, and other units such as units for graphic processing, memory controllers and DMA engines. An external memory attached to the Cypress GPU is 1 GB GDDR5 memory with a bus width of 256 bit. It has a data clock rate at 4800 MHz and offers us a bandwidth of 153.6 GB/s.

At the time of writing, the fastest Cypress processor is running at 850 MHz and offers a peak performance of $1600 \times 2 \times 850 \times 10^6 = 2.72$ Tflop/s in SP operations. With DP operations, the four stream cores in each TP are working together to compute either two DP addition, one DP multiply, or one DP fused-multiply-add (FMA). Note the transcendental stream core is not used in DP operations. Accordingly, we have $320 \times 2 \times 850 \times 10^6 = 544$ Gflop/s in DP operations.

In the present work, we have programed the Cypress GPU through an assembly like language called IL (Intermediate Language). The IL is like a virtual instruction set for GPUs from AMD. With IL, we have full control of every VLIW instructions. The device driver for a given GPU compiles IL instructions into the corresponding code in the instruction set architecture when we load a kernel written in IL.

2.2 Comparison with Other Architecture

Table 1 shows comparison between two latest GPU architectures: Cypress and Fermi [2].

Both architectures have introduced in 2009. They are somehow in similar performance range. A list of notable differences between two architectures is as follows: (a) A vector core in Cypress is 16 VLIW (5-way) processors working in SIMT. (b) A vector core in Fermi is 32 scalar processors working in SIMT. (c) Accordingly, effective vector length in SP per vector core is $16 \times 5 = 80$ for Cypress and 32 for Fermi, respectively. (d) Fermi has writable cache and ECC feature.

3. BLOCKING MATRIX MULTIPLICATION

To effectively utilize respectable computing power available by GPUs (see Table 1), a blocking algorithm is necessary to accommodate with massive bandwidth requirement. In this section, we present analysis of the blocking algorithm for matrix multiplication.

In the following, we only deal with square $N \times N$ matrix. And in this section, we consider to compute matrix multiplication as $C = AB$ where A, B, C are all square $N \times N$ matrices. Note GEMM is defined as $C \leftarrow \alpha AB + \beta C$ however the computation of $C = AB$ is dominant in a GEMM implementaion.

Let divide a square $N \times N$ matrix into blocks of $b \times b$ square matrices[1]. The input matrices A and B are divided into $(N/b)^2$ blocks. To compute matrix multiplication of the blocked (divided) matrices, we read $3b^2$ words from A, B and C, do $2b^3$ floating-point operations, and write b^2 words for C. If we can save a block of C on registers, we need to read $2b^2$ (two blocks from each A and B) to update b^2 words on registers in each iteration. In this case, required memory words per a floating-point operation is $W = 2b^2/2b^3 = 1/b$ words/flop. The required memory bandwidth is computed as

$$B_{\mathrm{GEMM}} = W \times F \times S \ \text{ byte/s} \tag{1}$$

where F is floating-point operations per second and S is the word size in bytes. Clearly, larger b we need relaxed bandwidth requirement at a factor of $1/b$.

Suppose we implement DGEMM on Cypress GPU, i.e., $S = 8$. It has the theoretical peak speed of $F = 544$ Gflop/s. If we assume that b is small enough to put a block of C on each thread processor's register file, we need memory bandwidth of $B_{\mathrm{DGEMM}} = 0.544(\text{Tflop/s}) \times 8/b = 4.352/b$ TB/s. With $b = 1$ (non-blocking algorithm), we need massive bandwidth of > 4.3 TB/s while the memory bandwidth of the GPU is only ~ 150 GB/s. With $b = 4$, the required bandwidth is reduced to be $B_{\mathrm{DGEMM}} = 1.088$ TB/s. This is exactly match to aggregate bandwidth of texture cache units on Cypress GPU since the texture cache unit on each SIMD engine can fetch data at 54.4 GB/s and we have 20 SIMD engines on a chip, i.e., 54.4 (GB/s) $\times 20 = 1.088$ TB/s. For SGEMM, we need bandwidth of $B_{\mathrm{SGEMM}} = 10.88/b$ TB/s where $F = 2.72$ Tflop/s and $S = 4$ hence $b \geq 10$ will be good choice. On the other hand, memory access in Cypress architecture is done in 128 bit hence it is desirable that b is multiple of 4 in SP. We will use $b = 8$ in the present work as explained in later sections. For DDGEMM, we estimate the theoretical performance in DD operations is roughly $F = 37.5$ Gflop/s (see Appendix A) so that we have $B_{\mathrm{DDGEMM}} = 0.6/b$ TB/s where $S = 16$. In case of DDGEMM, we might not need the blocking algorithm at all.

4. GEMM ON CYPRESS GPU

In this section, we describe detailed of our GEMM implementations.

4.1 Implementation Choices

Even we force to use the blocking algorithm, there are many alternative implementations with a given GPU architecture. Here, we summarize three critical decisions we made.

[1] b is also called the blocking factor.

4.1.1 How large is block size?

A software development kit (SDK) for GPUs from AMD is shipped with sample applications including many variants of GEMM written in IL and OpenCL. In addition, ACML-GPU ver.1.1, that includes SGEMM, DGEMM, ZGEMM and CGEMM, is available with the source code. All implementations supplied by AMD adopt 4×8 block (in SGEMM) because they use the output stream to store matrix C. We have up to 8 output streams in the Cypress GPU. Each thread writes 128 bit data (4 SP/2 DP words) on a given output stream so that in total a thread can output 32 SP/16 DP words at most. With this constraints, the maximum possible blocked matrix is 4×8 for SGEMM. This block size is too small to relax the memory bandwidth requirement. For DGEMM, we can arrange a blocked matrix in either 2×8 or 4×4.

Alternatively, we can store matrix C on the global memory. A good thing with the global memory is that we can access arbitrary position with linear address while with the output stream we write data into a predetermined position for each thread. A further distinction between the global memory and output stream is that the global memory is read/write memory but the output stream is write-only memory. So, the global memory is much more flexible but we need to explicitly calculate linear address before we access it. With the output stream, no address calculation is required.

In the present work, we adopt the global memory to store matrix C since 4×8 block is not enough to relax the bandwidth constraints for SGEMM implementation. Also, if we implement not just a simple matrix multiplication as $C = AB$ but real GEMM operations as $C \leftarrow \alpha AB + \beta C$, we require read access to C but it is tricky to use the *output* stream to implement GEMM operations if possible. Another drawback of the output stream is that we have to split matrix C into multiple stripes since each output stream corresponds to a block of memory on a host memory if we use multiple output streams. It requires additional memory arrangement and copy work on host because input matrices to a GEMM kernel is usually one block of memory region. With the global memory, we can preserve the given memory view of input matrices.

4.1.2 Pixel shader or compute shader?

In programing IL, we have two types of compute modes. *Pixel shader* is usually used to implement graphics pipelines with presumably graphics specific optimizations in hardware level. Even with the pixel shader, we can write a kernel that do general computations. In this mode, the system assign each thread with two-dimensional thread numbers. This reflect a fact that a pixel shader works on group of pixels, i.e., an image, a texture or a frame buffer.

Compute shader is specially prepared for GPGPU. In this mode, we have more flexibility such that we can use shared memory only in this mode. Also, the system assign each thread with one-dimensional (flat) thread numbers. Another critical difference is that with the compute shader we have control on how many active threads for a given kernel. However, in the pixel shader, we have no control of a number of threads. A number of threads in the pixel shader is determined by the system presumably depending on a number of live registers as we will describe later.

We can implement GEMM in both modes but after some experiments, we have chosen the pixel shader in the present work because it shows better performance. This decision is closely related to a next decision that we do not use shared memory in our implementation.

4.1.3 Which memory type do we put blocks of A, B, C?

In a fast GEMM implementation for GPUs [18], they put blocks of matrices A and C on the register file and a block from B on shared memory. According to analysis described in [18], shared memory on the GPU they have used has not enough memory bandwidth to make vector cores busy. They have estimated that a theoretical throughput of GEMM kernel using shared memory is 66% of the peak performance at most. With their SGEMM implementation, they have obtained 60% of the peak performance. It seems that a situation is not changed with Fermi GPU [17]. In [17], they put matrices A and B on shared memory and matrix C on registers with additional tuning. Their DGEMM kernel for Fermi GPUs shows roughly 60% of the peak performance.

In the present work, we put whole matrix C on the global memory and put blocks of matrix C on the register file. For blocks from matrices A and B, we have two options: (1) following [18], first a group of thread load the blocks into shared memory (local data store (LDS)) and then fetch data from the LDS into registers in each iteration. (2) we directly read the blocks from GPU main memory through texture cache. In either case, we put whole matrices A and B on the input stream data.

On Cypress, both LDS and cache units have aggregate bandwidth of 1.088 TB/s for fetching data[2]. Texture cache is managed by the hardware with no strict control by a programmer while normally LDS is used as *software cache*. With LDS, we have to manage memory allocation by ourself and calculate address for every read/write access. Also, we need to care about possible read conflicts between threads or coalesced memory access. We must say it is not easy although a fairly large fraction of optimization efforts in previous studies on GPGPU has been devoted to how to use *software cache* effectively [3] Obviously, hardware cache is easier to cope with from a programmer's view. Since memory access pattern of GEMM is regular, we suppose it is a good bet to rely on the hardware cache system.

4.2 SGEMM Implementation

According to the analysis of the blocking algorithm, $b \geq 8$ is desirable to implement SGEMM on Cypress GPU. If we fix the block size and decide to choose the pixel shader, we have no large room for optimizations so that automatic tuning techniques [13] is not very effective for a simple kernel like GEMM. One consideration is that a unit of memory

[2]According to the specification, LDS also has a capacity of 1.088 TB/s for write access in parallel.

[3]Fermi's 64KB shared memory on each vector core is configurable to choose either (a) 16KB for cache and 48 KB for shared memory or (b) 48KB for cache and 16 KB for shared memory. It is not clear [17] adopted either mode and whether they use the this configurable cache.

access in Cypress architecture is 128 bit (4 SP words). Accordingly, it is better to make a block size multiple of 4 in SGEMM implementation. In the present work, we choose $b = 8$ for our SGEMM implementation and we obtain a good performance as we will show. As we already mentioned in section 2, $b = 8$ is not optimal since we need bandwidth of $B_{\mathrm{SGEMM}} = 1.36$ TB/s while the bandwidth of texture cache unit on Cypress GPU is 1.088 TB/s. With larger b, hopefully we will have better performance while we will face severe register pressure if we set $b = 12$ for instance.

In the present paper, we used the following system for all benchmarks. CPU we used is Intel Core i7 920 (2.67GHz) with 3GB of DDR3-1066 memory. The motherboard is ASUS P6T (Intel X58 chipset). The GPU board is Sapphire Radeon HD5870 with the reference setting. We used Linux kernel 2.6.31 (x86_64) and Catalyst 10.7 for the device driver and the Stream SDK version 2.1.

4.2.1 Kernels for SGEMM

In the present work, we assume that matrices are in row-major format. In the following sections, we use i and j as indexes for a whole matrix. We use I and J to indicate indexes for each block. Namely, with $N \times N$ square matrix A, $A(j, i)$ represents a element (1 word) of the matrix A where i and j are running from 0 to $N - 1$. With the block size b, we have $(N/b)^2$ blocks and each block is identified with I and J, e.g., $A[J, I]$ represents a block (b^2 words) of the matrix A where I and J are running from 0 to $N/b - 1$.

On Cypress GPU, a kernel is executed by a group of threads. The total number of spawned threads equals to $(N/b)^2$ if N is multiple of b. Note we only consider that this assumption is always hold in the present work. In Figure 1, we present a structure of our SGEMM implementation. In the current implementation, each thread is responsible for $C[J, I]$ that is resided on registers and we adopt the rank-1 update (i.e., outer product based) algorithm. The kernel consists of three parts; (1) initialization of registers for accumulation, (2) rank-1 update loop and (3) merging of results.

At k-th iteration of the main loop, the kernel fetch a column of data ($A(bJ : bJ + 7, k)$; 8 SP words) from matrix A and a row of data ($B(k, bI : bI + 7)$) from matrix B, and compute the outer product between the column and the row as shown in Figure 2. Note this figure corresponds to SGEMM with both matrices A and B in non-transpose format (hereafter we call this "NN" kernel). If input flags to GEMM is to specify transpose matrix A, memory access pattern is different as shown in Figure 3 (we call this "TN" kernel). With 128 bit memory access mode in mind, we expect that the TN kernel is desirable for Cypress GPU. At the moment, we do not implement all patterns of a kernel (e.g., NN, TN, NT, TT). For SGEMM, we only implement the TN kernel. In the following, we analyze the SGEMM TN kernel.

4.2.2 Analysis of the SGEMM TN kernel

As noted previously, we write our GEMM kernels in IL. As far as we understood, there are one-to-one mapping between most of IL instructions and real ISA for a given hardware. The device driver of GPU hardware compile IL instructions into real ISA when we load a kernel written in IL. At the same time, the device driver does register allocation for a

```
// declare A and B as input streams
// declare C as global memory
float c[8][8], a[8], b[8]; // on registers

// zero clear accumulator registers
c[0:7][0:7] = 0.0

for k = 0 to N-1
   // fetch a column from A
  load a[0:3] <- A(bJ:bJ+3, k)
  load a[4:7] <- A(bJ+4:bJ+7, k)
   // fetch a row from B
  load b[0:3] <- B(k, bI:bI+3)
  load b[4:7] <- B(k, bI+4:bI+7)

  // 8x8 rank-1 update
  c[0][0:3] += a[0]*b[0:3]; // 8 flops
  c[0][4:7] += a[0]*b[4:7];

  c[1][0:3] += a[1]*b[0:3];
  c[1][4:7] += a[1]*b[4:7];
  ...
  c[7][0:3] += a[7]*b[0:3];
  c[7][4:7] += a[7]*b[4:7];
end
Merge c[][] with C[J,I]
```

Figure 1: A pseudo code of our SGEMM implementation that shows a structure of the code in IL.

given GPU architecture. We found the register allocation is critical to get an optimal performance as explained below.

With pixel shader, a number of active threads per TP is automatically controlled at the system level as far as we understand. In [3] (Table 4.8), they present a relation between the active number of threads and the number of register used in a given kernel. This relation is approximately expressed in a following equation:

$$(\#\text{ of active threads}) = \lfloor 256/(\#\text{ of registers}) \rfloor \times 4. \quad (2)$$

A more number of active threads, there is a better chance to hide memory access latency hence we could have better performance. For the SGEMM implementaion with our strategy described above, we need 16 registers (64 SP words) to store a block from matrix C and 4 registers to fetch two stripes, each of which is 8 SP words, from matrices A and B. Accordingly, we need at least $16 + 4 = 20$ registers to implement a SGEMM kernel. Note we certainly need additional registers for address calculations and loop counter.

To see the importance of register allocation, we show two variants of the SGEMM kernel as (a) highly optimized kernel and (b) less optimized kernel. We arrange the latter kernel by artificially tweaking the highly optimized kernel to make requiring more live registers. The highly optimized kernel needs 25 registers but the less optimized kernel consumes 37 registers. With our formula (Equation (2)), we would have 40 and 24 threads for the two kernel, respectively. Figure 4 shows the performance of the two kernels as a function matrix size N. In this plot we compare the highly optimized kernel (in red solid circles) and the less optimized

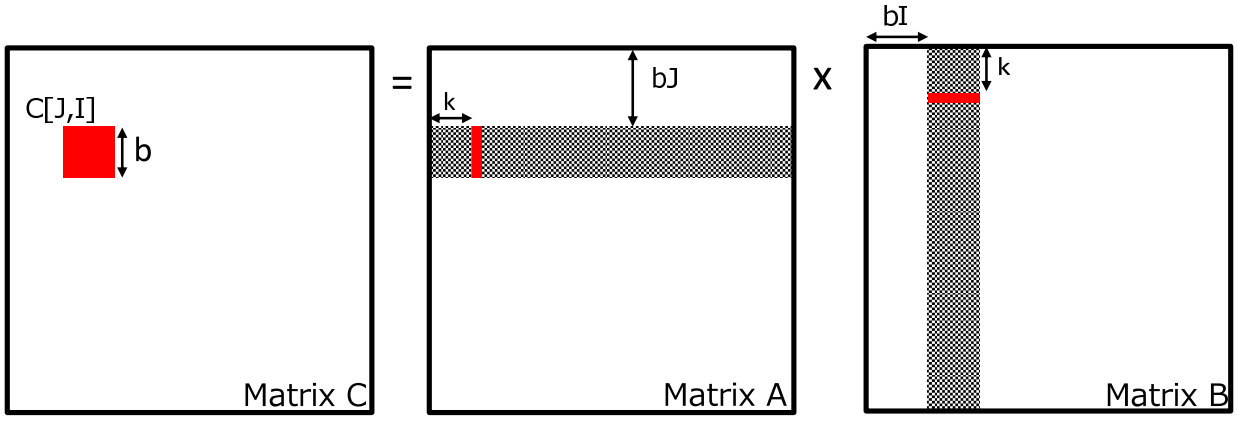

Figure 2: Schematic presentation of our GEMM implementation on Cypress GPU with GEMM flags "N" for both matrices A and B. Each thread computes a block of matrix C shown in the red square $C[J, I]$. In each iteration of the main loop, each thread reads the red stripes from matrices A and B to do the rank-1 update on the block. In total, we need N iterations to complete.

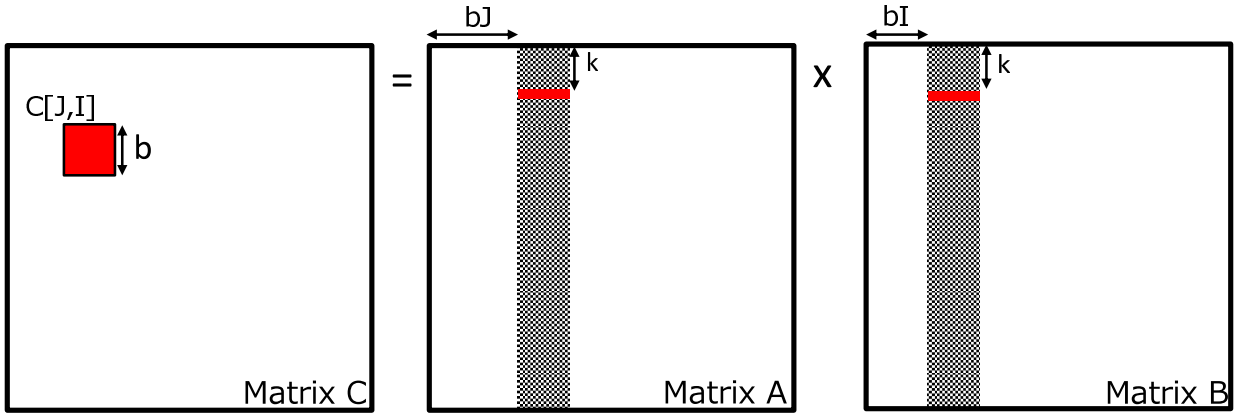

Figure 3: Schematic presentation of our GEMM implementation on Cypress GPU with GEMM flags "T" for matrix A and "N" for matrix B. Notation is same as Figure 2 except in this case each thread reads a stripe from matrix A in transposed order.

kernel (in green squares). It is clear that two kernels show rather different performance with the ratio of the computing speed between two kernels ~ 1.3. We conclude that a critical optimization technique in GEMM kernels on Cypress GPU is to reduce the number of live registers as much as possible.

4.3 DGEMM and DDGEMM Implementation

We implement DGEMM and DDGEMM kernels as a straitforward extension from the SGEMM kernel. A difference from the SGEMM kernel is the blocking factor b. We set $b = 4$ and $b = 2$ for DGEMM and DDGEMM, respectively. For DDGEMM, we have implemented the DD emulation scheme in IL as described in Appendix A.

For DGEMM, we have implemented TN and NN kernels. We present the performance of our DGEMM kernels in Figure 5. In this Figure, we compare our results with ACML-GPU 1.1 running on Cypress, MAGMA BLAS 0.3 running on Fermi [17], and GotoBLAS2 [8] on the host CPU. Note that our results on upper half (A^tB(TN) and AB(NN)) and the result with MAGMA BLAS (AB(Fermi)) [4] show results that do not take into account data transfer time between CPU and GPU. While our result in lower half ($A^tB_{\mathrm{I/O}}$) and the result with ACML-GPU ($A^tB_{\mathrm{I/O}}$(ACML)) show results that take into account the data transfer. As we compare our results with MAGMA BLAS 0.3 both of our kernels show better performance despite the theoretical peak performance of Cypress and Fermi GPUs is comparable. The result with GotoBLAS2 is $\sim$ 43 Gflop/s on our system. Our GEMM kernels for Cypress GPU are 10 times faster than a 4 core CPU.

The performance of NN kernel is slower than TN kernel. We speculate a reason that two kernels do memory access to matrix A in different way as described in Figures 2 and 3. Another reason is that since the memory access is done with 128 bit in row-major formant, NN kernel first fetches four 128 bit words (8 DP words) and then only uses half of the

[4] This result corresponds to NN in our definition

	SGEMM TN	DGEMM TN	DGEMM NN	DDGEMM TN (FMA)	DDGEMM TN (no FMA)
Pmax	2014	472	359	31	23
Nmax	4352	1664	3712	1408	768
# of reg.	25	25	25	18	29
4 slots(%)	25.8	94.1	94.1	90.9	90.9
5 slots(%)	58.1	0	0	0	0

Table 2: Performance analysis of our GEMM variants. Pmax and Nmax rows show the peak Gflop/s and the corresponding the matrix size that shows a peak performance. 3rd row presents a number of register used in a given kernel. 4th and 5th rows indicate a fraction of occupied VLIW slots in a given kernel. For instance, in SGEMM TN kernel, 25.6% and 51.8% of all instructions consume 4 and 5 VLIW slots on each TP, respectively. It means our SGEMM kernel highly (close to 84%) utilize the available computing resource. DGEMM and DDGEMM kernels show even higher utilization.

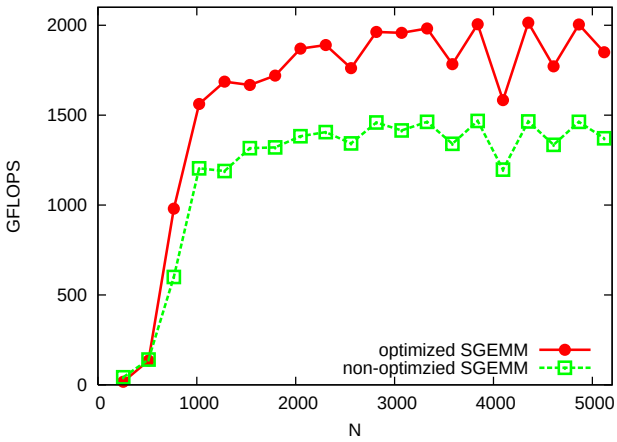

Figure 4: SGEMM performance as function of N for two variants. The optimized SGEMM uses 25 registers while the naive SGEMM uses 37 registers.

data just fetched as depicted in Figure 6. Note other half of the data is used in the next iteration. On the other hand, with TN kernel, the memory access is row-major in nature hence we need to fetch two 128 bit words (4 DP words) and use all data in the current iteration. The memory access pattern in NN kernel seems to be not match to execution model on Cypress GPU since for the pixel shader a group of 2x2 threads called *quad* is always processed together [3].

For DDGEMM, we have only implemented TN kernel at the moment. Overall trend in performance as a function of N looks similar to SGEMM and DGEMM kernels. With $N \geq 512$, our DGEMM TN kernel shows the peak performance of $\sim$ 31 Gflop/s. This result is obtained with the kernel that takes advantages of FMA instructions. Without FMA instructions, the performance drops to $\sim$ 23 Gflop/s. See Appendix A for effectiveness of FMA instruction in the DD emulation scheme. We have compared the performance of our DDGEMM kernel with performance obtained with the mpack version 0.6.5 [16]. The performance of mpack running on single core of Core i7 920 (2.67 GHz) is $\sim$ 140 Mflop/s. Although there seems to be a large room for possible optimization for mpack, our DDGEMM kernel running on Cypress GPU is more than 200 times faster.

Table 2 summarize the all benchmark results of our GEMM kernels. 4th and 5th rows show a fraction of occupied VLIW slots in a given kernel. Sum of two rows is not 100% because a residual is the fraction of instructions where either 1, 2 or 3 slots are occupied. Those low utilized VLIW instructions are necessary for computation of address, updating the loop counter, and other calculations. With SGEMM kernel, roughly 60% of VLIW instructions occupy 5 slots that use all 5 SCs on each TP while other 26% occupy 4 slots that use 4 SCs. With DGEMM and DDGEMM kernels, more than 90% of VLIW instructions occupy 4 slots that is maximum in DP operations.

4.4 Timing Model of Data Transfer

In Figure 5, we plot the benchmark result with taking into account data transfer time between host and GPU memory. It is shown in lower region of the Figure with "I/O". The result for ACML-GPU is also obtained with the same way. To implement GEMM, we need to communicate $O(N^2)$ data through PCI-Express bus and we do $O(N^3)$ floating-point operations on Cypress GPU. With small N, the time for data transfer dominates the computing time on the GPU as shown in Figure 5.

Here, we construct a model for an elapsed time (T_{DGEMM}) of one DGEMM call as follows.

$$T_{\mathrm{DGEMM}} = T_{\mathrm{comm}} + T_{\mathrm{kernel}}, \tag{3}$$

where T_{comm} and T_{kernel} represent the elapsed time for data transfer and the elapsed time for the kernel execution on the GPU. T_{comm} are simply defined as

$$T_{\mathrm{comm}} = \frac{32N^2}{B_{\mathrm{PCIe}}}, \tag{4}$$

where $\mathrm{B_{PCIe}}$ is transfer speed between CPU memory and GPU memory in byte/s since we send three matrices and receive one matrix, i.e., $32 = 4 \times 8$ bytes. The elapsed time of the kernel execution on GPU is expressed as

$$T_{\mathrm{kernel}} = \frac{2N^3}{F}, \tag{5}$$

where F is floating-point operations per second of a given kernel. From these equations, we can write the performance in Gflop/s as $2N^3/T_{\mathrm{DGEMM}}$.

Figure 7 shows comparison between the performance of our DGEMM TN kernel and our timing model. Here we set parameters as $\mathrm{B_{PCIe}} = 3$ GB/s and $F = 450$ Gflop/s. Our

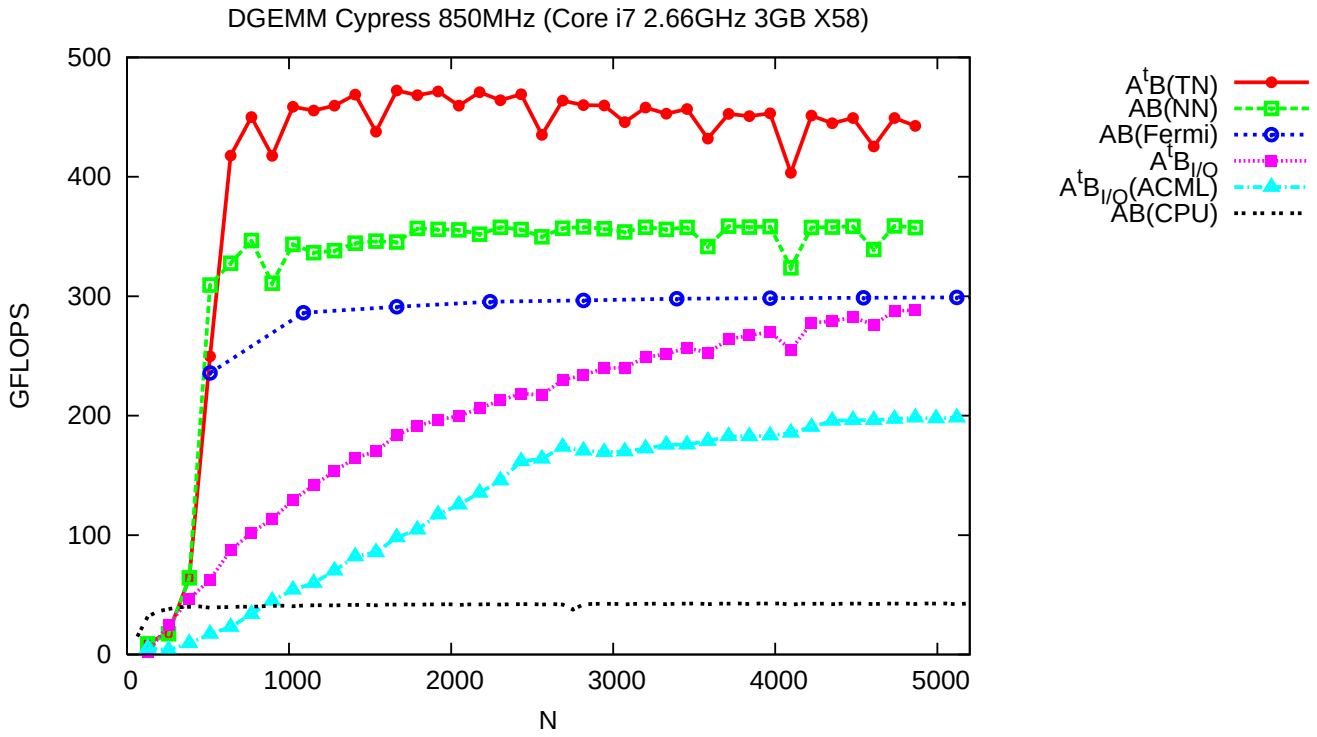

Figure 5: DGEMM performance as function of N. We compare our results with ACML-GPU 1.1 and MAGMA BLAS 0.3 running on Fermi [17]. Note the lines without I/O show the performance that dose not take into account data transfer time between CPU and GPU.

timing model works rather well. In all range of N we have benchmarked, a limiting factor in our current DGEMM implementation is slow data transfer between host and GPU memory with effective bandwidth of only $\sim 2-3$ GB/s. On the other hand, the theoretical bandwidth of PCI-Express bus Gen.2 is 8 GB/s. If we have a hypothetically better I/O bus with $B_{\mathrm{PCIe}} = 10$ GBs^{-1}, the predicted performance of our DGEMM kernel is shown in the blue dashed line. In this case, $N \geq 2048$ is enough to reach potential performance of our DGEMM kernel. Another possible optimization for data transfer is to overlap communications and computations. That will be our future task for real applications such as LU factorization.

4.5 Comparison to Other Work

[18] has presented an optimized GEMM implementation for GeForce GPUs. Their SGEMM runs at $\sim 60\%$ of the peak performance while CUBLAS 1.1 runs at $\sim$ 40% with the same GPUs. [13] have presented auto-tuning techniques for GEMM on GeForce GPUs and shown that their auto-tuned SGEMM and DGEMM are slightly faster than those of CUBLAS 2.0, that adopted the GEMM implementation by [18]. [10] have presented another auto-tuning framework in OpenCL for GEMM. [17] presented DGEMM implementations for Fermi GPU with $\sim 60\%$ of the peak.

In this paper, we present GEMM implementations for Cypress GPU. Our GEMM implementations are fastest with one GPU chip and show very high performance efficiency compared to the peak performance. Our SGEMM TN kernel runs at 55 - 73% of the peak performance. Even at the lowest efficiency, our SGEMM is faster than [18, 13]. Our DGEMM TN and NN kernels are even more efficient as 74 - 87% of the peak. With the same hardware, we did the benchmark of the DGEMM kernel in ACML-GPU where we take into account data transfer between host and GPU memory. We found that our DGEMM substantially outperforms ACML-GPU even though we did not optimize data transfer at all. A critical difference between our DGEMM and that of ACML-GPU is the blocking factor b. Finally, if we compare the performance of our GEMM kernels with an advanced implementaion [8] for a multicore CPU (4 core running at 2.67 GHz), our TN kernel roughly 10 times faster as shown in Figure 5.

There are BLAS implementations with higher precision than DP [1, 16]. [1] supports SP, DP and DDP and only implements a part of BALS routines while [16] supports various high precision arithmetic libraries not only DDP and implements even a part of LAPACK routines. Our new contribution is that for the first time we show DDGEMM on GPU is very fast and efficient. Our DDGEMM shows 83% of the estimated peak performance in DDP. The computing speed of 31 Gflop/s in DDP is more than 200 times faster than the performance of mpack in DDP.

5. CONCLUSION

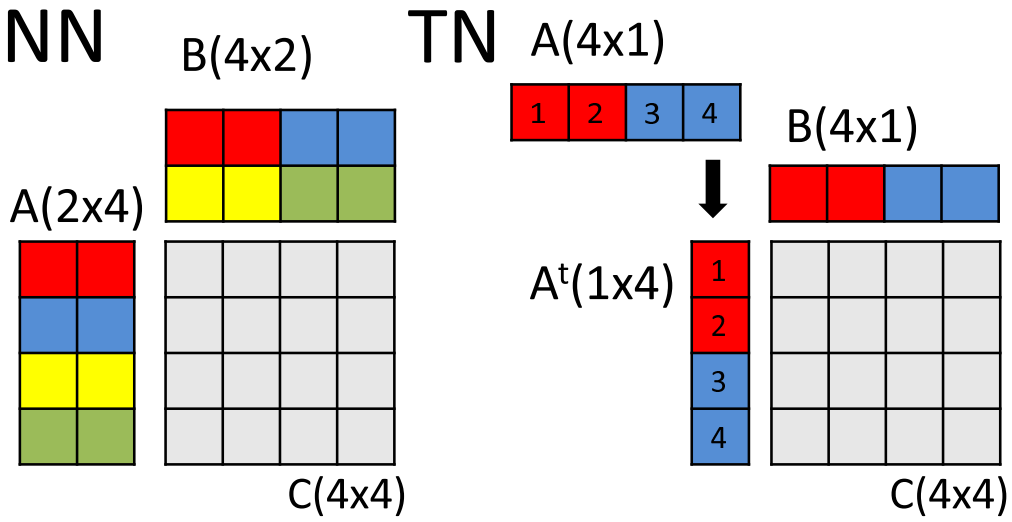

Figure 6: Precise memory access patterns in NN (left) and TN (right) kernels in DP. Each box represent one DP word and boxes with same color correspond to one 128 register. In NN kernel, we read 8 DP words from matrices A and B then compute the rank-1 update two times. In TN kernel, we read 4 DP words from matrix A and transpose it and read 4 DP from matrix B then compute the rank-1 update one time.

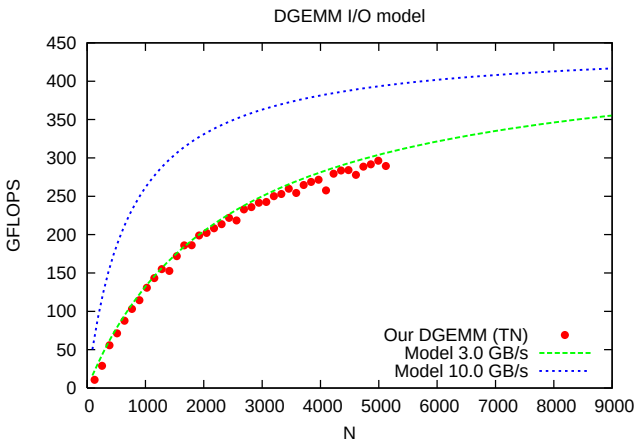

Figure 7: Comparison between the performance our DGEMM TN kernel (shown in red squares) and our timing model (shown in green dashed line). The blue dashed line shows the performance prediction with better communication bandwidth.

We described our GEMM kernels for Cypress GPU. With three critical decisions to design GEMM kernels for Cypress, we presented detailed description of SGEMM kernels. DGEMM and DDGEMM kernels are implemented as a natural extension from the SGEMM kernel. Our GEMM kernels are fastest at the moment without too complicated tuning efforts thanks to the texture cache on Cypress architecture. In our opinion, a large fraction of previous work regarding GPGPU has been dealing with how to use shared memory effectively as software cache. In contrast, the texture cache is implemented as hardware so that we can skip those efforts that were thought to be necessary. Even with our insight, it will be a possible future work to utilize LDS for GEMM kernels. Also, the performance of SGEMM and DGEMM with $N > 1024$ is not flat as in other work but fluctuating and slightly decreasing at large $N > 2048$. To address reasons for such trend and try to remove performance decrease will be another future work.

Acknowledgments

I would like to thank Prof. S.Sedukhin and K.Matsumoto for discussions and their comments on our paper and Dr. M.Nakata for a help with a part of benchmark and mpack library. This work was supported in part by the Grant-in-Aid of the Ministry of Education (No. 21244020).

6. REFERENCES

[1] XBLAS - Extra Precise Basic Linear Algebra Subroutines: http://www.netlib.org/xblas/.

[2] NVIDIA's Next Generation CUDA Compute Architecture: Fermi, http://www.nvidia.com/object/fermi_architecture.html, 2009.

[3] ATI Stream Computing OpenCL Programming Guide, rev1.05, August 2010.

[4] Bailey, D. QD Library: http://crd.lbl.gov/ dhbailey/mpdist/.

[5] BARRACHINA, S., CASTILLO, M., IGUAL, F., MAYO, R., and QUINTANA-ORTI, E. Solving Dense Linear Systems on Graphics Processors (ICC 02-02-2008). Tech. rep., Universidat Jaime I, 2008.

[6] Dekker, T. A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik 18* (1971), 224–242.

[7] Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the Efficiency of GPU Algorithms For Matrix-matrix Multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (Newyork, NY, USA, 2004), pp. 133–137.

[8] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (2008), 1–25.

[9] Hida, Y., Li, X., and Bailey, D. Algorithms for Quad-Double Precision Floating Point Arithmetic. In *Proceedings of the 15th Symposium on Computer Arithmetic* (2001), pp. 155–162.

[10] Jang, C. GATLAS GPU Automatically Tuned Linear Algebra Software: http://golem5.org/gatlas/.

[11] Kågström, B., and Van Loan, C. GEMM-Based Level-3 BLAS (CTC91TR47). Tech. rep., Department of Computer Science, Cornell University, 1989.

[12] Kunuth, D. *The Art of Computer Programming vol.2 Seminumerical Algorithms*, first ed. Addison Wesley, Reading, Massachusetts, 1998.

[13] Li, Y., Dongarra, J., and Tomov, S. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of ICCS'09* (Baton Rouge, LA, USA, 2009).

[14] Nakasato, N. Application of many-core accelerators for problems in astronomy and physics. In *13th International Workshop on Advanced Computing Analysis Techniques in Physics Research* (2010).

[15] Nakasato, N., and Makino, J. A compiler for high performance computing with many-core accelerators. In *IEEE International Conference on Cluster Computing and Workshops* (2009), pp. 1–9.

[16] Nakata, M. The MPACK (MBLAS/MLAPACK); A Multiple Precision Arithmetic Version of BLAS and LAPACK: http://mplapack.sourceforge.net/, 2010.

[17] Nath, R., Tomov, S., and Dongarra, J. An Improved MAGMA GEMM for Fermi GPUs (UT-CS-10-655 also LAPACK working note 227). Tech. rep., Universiyt of Tennessee Computer Science, July 2010.

[18] Volkov, V., and Demmel, J. Benchmarking GPUs to Tune Dense Linear Algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA, 2008), pp. 1–11.

APPENDIX

A. DD EMULATION ON CYPRESS GPU

To implement DDGEMM on Cypress GPU, we have written the DD emulation code in IL. The emulation scheme was originally proposed by [12] and [6]. [9] presented quad-double precision scheme as an extension. We have implemented DDP arithmetic routines in IL by referring the algorithms used in [9, 4]. We have used our DDP routines in our compiler system for many-core accelerators as described in [15, 14].

In this scheme, a DD variable is expressed as a sum of two DP variables and we utilizes DP arithmetic units to emulate DDP arithmetic operations. One addition and multiply in DDP requires 21 and 25 DP instructions, respectively. Note this is an implementation with naive algorithm without FMA instructions. Multiply in DDP is efficiently implemented with FMA instructions. In this case, we only need 8 DP instructions for multiply. We define a peak performance of Cypress GPU in DD operations as follows. In the present work, we are dealing with GEMM so that we require equal number of addition and multiply operations. namely, we count an average number of instructions in this case as $(21 + 25)/2 = 23$. Accordingly, we estimate a peak performance in DDP is 544 (Gflop/s) /23 = 23.7 Gflop/s. With FMA instructions, we have a better average instructions as $(21 + 8)/2 = 14.5$. In this case, a peak performance in DDP is 544 (Gflop/s) /14.5 = 37.5 Gflop/s.

It is obvious that the DD emulation scheme is well-suited for GPU architecture since the scheme is compute intensive by definition. More precisely, to compute addition in DDP, we read 4 DP words as inputs and write 2 DP words as a result while we do 21 DP operations hence we have $W = 6/21 \sim 0.29$ words/flop for addition and $W = 6/25 \sim 0.24$ words/flop for multiply (without FMA). These numbers are 4 times compute intensive than a simple add/mul in DP. With DGEMM with FMA instructions, we have $W = 6/14.5 \sim 0.41$. It is still *easier* for GPUs with the limited memory bandwidth.