Fast Implementation of DGEMM on Fermi GPU*

Guangming Tan†, Linchuan Li†, Sean Triechle‡, Everett Phillips‡, Yungang Bao†, Ninghui Sun†
†Key Laboratory of Computer Architecture, Institute of Computing Technology,Chinese Academy of Science
‡Nvidia Corporation
{tgm,lilinchuan,baoyg,snh}@ncic.ac.cn,{striechler,ephillips}@nvidia.com

ABSTRACT

In this paper we present a thorough experience on tuning double-precision matrix-matrix multiplication (DGEMM) on the Fermi GPU architecture. We choose an optimal algorithm with blocking in both shared memory and registers to satisfy the constraints of the Fermi memory hierarchy. Our optimization strategy is further guided by a performance modeling based on micro-architecture benchmarks. Our optimizations include software pipelining, use of vector memory operations, and instruction scheduling. Our best CUDA algorithm achieves comparable performance with the latest CUBLAS library[1]. We further improve upon this with an implementation in the native machine language, leading to 20% increase in performance. That is, the achieved peak performance (efficiency) is improved from 302Gflop/s (58%) to 362Gflop/s (70%).

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

General Terms

Algorithms, Performance

Keywords

high performance computing, GPU, CUDA, matrix-matrix multiplication

*This work is supported by National 863 Program (2009AA01A129), the National Natural Science Foundation of China (60803030,61033009,60921002,60925009) and 973 Program (2011CB302500).

[1]CUBLAS3.2 was the latest version when this work has been done. Now CUBLAS4.0 is released, but its DGEMM performance stays the same level as before. Therefore, we still use CUBLAS3.2 for comparison in this paper

SC11 November 12-18, 2011, Seattle, Washington, USA

1. INTRODUCTION

Dense matrix operations are important problems in scientific and engineering computing applications. There have been a lot of works on developing high performance libraries for dense matrix operations. Basic Linear Algebra Subprograms (BLAS) [4] is a defacto application programming interface standard for publishing libraries to perform basic linear algebra operations such as vector and matrix multiplication. The first BLAS was released as a building block of LAPACK [2], which is a performance portable library for implementing dense linear algebra. Hardware vendors also provide BLAS libraries tuned on their own processors, i.e. MKL and ACML. It is well-known that the performance of BLAS depends on the underlying hardware [3, 5].

Recently, multi/many-core CPU processors have become mainstream since parallelism has shown strength in overcoming power and frequency limits of the latency optimized architecture. The GPU, on the other hand, evolved as a data parallel processor optimized for throughput. As a result the GPU outpaces modern CPUs in performance. Over time the GPU has become more programmable which lead to adoption of the GPU as an accelerator for general purpose computing including numerical computations. Since dense matrix operations are compute intensive and exhibit regular memory access patterns, they are especially well suited for the GPU architecture. NVIDIA has produced a series of GPU architectures: G80, GT200 and Fermi. For each generation it announced a corresponding matrix computation library CUBLAS. Undoubtedly, these libraries achieve higher performance than its counterparts (i.e. MKL, ACML) on general-purpose multi-core CPU. For example, CUBLAS3.2 on the latest Fermi GPU outperforms Intel's MKL on Xeon six-core processors (Westmere) by more than five times. According to the release note of CUBLAS3.2, it adopts a similar idea with MAGMA [9]. However, all available materials from both NVIDIA and MAGMA do not address following two unsolved problems:

- *What are the technical details behind the performance improvement?* As we know, CUBLAS3.0 achieves more than 90% efficiency of peak performance on GT200 architecture. However, CULBAS3.2 does not get portable efficiency on the new Fermi architecture although it is much faster than CUBLAS3.0. It is a definite fact that CUBLAS3.2 uses some different optimizations for better performance. Is the difference from algorithmic innovation or any optimization strategy specific to Fermi's new feature?

- *Is there any more room for further performance improvement?* The experiments show that CUBLAS3.2 achieves about 58% efficiency of peak performance. With respect to efficiency of floating-point calculation, it is much worse than both the previous generation and its counterpart on multi-core CPUs. NVIDIA claims that Fermi is enhanced with some architectural innovations for general-purpose computing. Does CUBLAS3.2 take full advantage of Fermi's new features? What is the limitation for higher performance?

In this paper, we address these issues by investigating Fermi's micro-architecture and going deeply into a fast DGEMM implementation. DGEMM is a pronoun of general double-precision matrix-matrix multiplication in BLAS [4]. It is a performance critical kernel in numerical computations including LU factorization, which is a benchmark for ranking supercomputers in the world. We take DGEMM as an example to illustrate our insight on Fermi's performance optimizations through this paper. Specifically, we make three main contributions in this paper:

- We formulate a performance model to select an optimal algorithm for blocking matrix multiplication on the Fermi architecture. We use benchmarking to identify a global memory access pattern for maximizing effective bandwidth. Combined with a software prefetching strategy, our algorithm achieves comparable performance with CUBLAS3.2.

- We discuss three optimization strategies to exploit the potential of Fermi's computing capability. The three incremental strategies include wider memory operations, double-buffering in shared memory, and instruction scheduling. Our experiments indicate that it is necessary to tune the code by hand for a maximal performance. Finally, the DGEMM program using all three strategies together achieves 20% higher performance than the latest CUBLAS3.2.

- We present an experimental experience on tuning DGEMM code on the Fermi architecture. A micro-benchmark analysis of Fermi architecture is used to guide program optimizations. The benchmark makes a connection between Fermi's architectural features and program implementation choice. These results are not disclosed in NVIDIA's programming manuals and may be instructive to both other math libraries and compiler optimization.

Overall, this paper presents a deep insight on optimizing the performance critical code DGEMM by taking full advantage of the emerging many-core architectures like NVIDIA's Fermi GPU. In fact, the optimized DGEMM code presented in this paper has been adopted as one part of NVIDIA's accelerated library for running Linpack on top ranking supercomputers including Tianhe-1A and Nebulae, who now are ranking the No.1, No.3 in the world, respectively. We demystify the technical details on Fermi architectural features and how to implement an extremely fast DGEMM code on Fermi. The rest of this paper is organized as follows: Section 2 gives a brief background on Fermi architecture and DGEMM. Section 3 and 4 detail the proposed parallel algorithm and performance tuning on Fermi. The performance experiments and analysis are reported in Section 5. We summarize the related work in Section 6, and conclude this paper in Section 7.

Table 1: Summary of NVIDIA GPU architecture

Parameters/GPU	GT200	Fermi
CUDA cores	240	448(512)
DP peak	30MAD/ops	256FMA/ops
SP peak	240MAD/ops	512FMA/ops
Warp scheduler/SM	1	2
Shared memory/SM	16KB	48KB or 16KB
L1 cache/SM	N/A	16KB or 48KB
L2 cache	N/A	768KB
DRAM	GDDR3 102GB/s	GDDR5+ECC 144GB/s

2. BACKGROUND

This paper focuses on performance optimization on Fermi GPUs, we first introduce its architecture and highlight its difference from the previous generation GT200. Since we use DGEMM as a running through example, we also briefly describe one of its implementation variants on the GPU as well.

2.1 Fermi GPU Architecture

The GPU achieves its impressive computing power through a large number of parallel SIMD engines. Fermi [11] from NVIDIA is equipped with 14 or 16 SIMD engines (448 or 512 cores), which offers 1.03Tflop/s single-precision operations and 515Gflop/s double-precision operations. Its competitor Cypress GPU [1] from AMD/ATI has 20 SIMD engines and offers a peak performance of 2.72Tflop/s single-precision operations and 544Gflop/s double-precision operations. In order to facilitate users to programming, NVIDIA abstracts a data parallel programming model called CUDA [12]. In the CUDA model, one SIMD engine is referred to as a streaming multi-processor (SM). The basic unit of execution flow in the SM is the warp, which is a collection of 32 threads. One warp is scheduled to execute in way of single-instruction multiple-thread (SIMT), which means that the 32 threads execute the same instruction while operating on different data in lockstep. One SM can concurrently execute multiple warps which are grouped into blocks. Each SM contains both per-block shared memory and register files, which are shared by all threads in a block. At the level of the CUDA model, these features are common to GT200 and Fermi. However, at micro-architectural level Fermi is a significant evolution from GT200.

Table 1 summarizes the comparison between GT200 and Fermi. Fermi is an evolvement from GT200 architecture. It improves throughput of double-precision floating-point operations. For example, NVIDIA C2050 with 1.1GHz Fermi chip provides a capability of 515Gflop/s (Note: all experiments reported in this paper are conducted in this machine). We highlight several important features related with applications' performance:

- The size of shared memory is increased to 48KB while it may be configured with 16KB. Usually, shared memory is used to exploit locality for a blocking algorithm. Its capacity has a direct impact on the choice of blocking factors. In addition, the number of banks of shared

memory increases to 32 so that a bank conflict may also happen for a warp of threads. These changes will lead to a modification to the existing algorithms for better performance.

- A cache hierarchy is added between DRAM and streaming cores. Caches can amplify bandwidth and reduce the penalty of global memory access. Unlike shared memory, caches automatically exploit locality by hardware. Since they cache parts of data from global memory, the overhead of a bad global memory access pattern (i.e. uncoalesced access) is mitigated.

- There are two warp schedulers for each SM. That means two instructions may be combined to be issued at the same time. Every cycle, the two schedulers can issue two warps, however, double precision instructions cannot be dual-issued with other instructions.

- Another important feature is the width of memory operations. In Fermi architecture, applications can use wider load/store operations than 32-bits per thread. It also supports both 64- and 128-bits memory operations. An intuitive idea is that we may take use of these new instructions to improve memory access efficiency. However, as discussed in the flowing context, the wide data transfer instructions incur higher latency.

Algorithm 1

The size of thread block: $vlx * vly$
Register: $accum[rx * ry]$, //$rx * ry$ is a factor of register blocking
$rA[rx]$,$rB[ry]$
Shared memory: $smA[bk][bm]$,$smB[bk][bn]$
//
$accum[0 \dots rx][0 \dots ry]$=0
load one $bm * bk$ block of A into $smA[bk][bm]$
load one $bk * bn$ block of B into $smB[bk][bn]$
synch
while (–K>0) {
  for (ki=0;ki<bk;ki++) {
    load one column of A in smA into $rA[0 \dots rx]$
    load one row of B in smB into $rB[0 \dots ry]$
    $accum[0 \dots rx][0 \dots ry] += rA[0 \dots rx] * rB[0 \dots ry]$
  } // end for
  load one $bm * bk$ block of A into $smA[bk][bm]$
  load one $bk * bn$ block of B into $smB[bk][bn]$
  synch
} // end while
Merge $accum[0 \dots rx][0 \dots ry]$ with $bm * bn$ block of C.

Figure 1: The basic framework of DGEMM routines

2.2 DGEMM on GPU

The BLAS specification [4] defines DGEMM as $C := alpha * A * B + beta * C$, where A, B and C are $m * k$, $k * n$, $m * n$ matrices, respectively. A straightforward implementation of DGEMM is three nested loops. It is well known that a blocking algorithm often has higher performance on a processor with a memory hierarchy because a blocking matrix-matrix multiplication exploits more data reuse and achieves a higher effective memory bandwidth.

There have been several published literatures describing optimizing DGEMM on CUDA. Previous works [3, 5, 9, 15, 13, 14, 8] have pointed out that both latency and bandwidth of global memory have a significant effect on DGEMM's performance. For a blocking DGEMM on the GPU, the three matrices are partitioned into blocks of $bm*bk$,$bk*bn$,$bm*bn$ and these blocks are laid out as grids of $M*K$,$K*N$,$M*N$, where $M = m/bm$, $N = n/bn$, $K = k/bk$. The computation is done on a two-dimensional grid of thread blocks, where one block of C is assigned to one thread block. That is, there are $M * N$ thread blocks, each of which requires fetching K blocks of A and B. Totally these fetches read $M*N*K*bm*bk+M*N*K*bk*bn = m*n*k*(1/bm+1/bn)$ words through the memory hierarchy. The blocking algorithm results in a bandwidth reduction of $2/(1/bm + 1/bn)$. Furthermore, since blocks of both A and B are first loaded into shared memory, data is reused multiple times, and the penalty of global memory access is reduced.

The memory hierarchy on the GPU is abstracted to be three levels: off-chip memory (global memory), on-chip memory (cache or shared memory) and register files. The bandwidth increases while the latency decreases through the memory hierarchy from global memory to registers. Therefore, a blocking algorithm usually contains two folds of blocking. For the instance of our blocking DGEMM, we refer to the level of blocking from global memory to shared memory as *shared memory blocking*. Since there are gaps of both bandwidth and latency between shared memory and register file, another level of blocking sub-matrices in shared memory is necessary to get reuse of the data in registers. We call this level of blocking *register blocking*. Further, in order to maximize efficiency of floating-point execution units, the utilization of shared memory and register files is another key factor. For example, the data layout and access patterns in shared memory should be carefully orchestrated to avoid bank conflicts because the conflict results in extra latency. Additionally, the number of required registers per thread should be balanced to maintain sufficient thread occupancy to hide latencies.

As indicated by previous works [15, 13, 14], the above concerns are non-trivial for a high performance DGEMM implementation on GPU. We here describe a basic framework of our DGEMM algorithm on GPU. Algorithm 1 in Figure 1 is the two-levels of blocking algorithm on a memory hierarchy with global and shared memory. The matrix A is transposed before the multiplication is actually performed. The experimental results presented in the paper include this overhead, which is about 1% of the total execution time. There are several undefined parameters (i.e. bm, $bn\dots$) in pseudo-code. These parameters are directly related to performance. In the next section we will show how to select optimal values and thread mapping on Fermi.

3. IMPLEMENTATION CHOICES AND ANALYSIS

In this section, we describe a DGEMM implementation on Fermi based on our micro-benchmarking results. The benchmark analysis identifies how to maximize efficiency (i.e. bandwidth or bank conflicts) of both global and shared memory. Although we achieved a comparable performance to CUBLAS3.2, we present additional analysis of potential optimizations.

3.1 Implementation

With respect to memory efficiency, both reduction of the required memory bandwidth and optimal memory access pattern should be determined. For the issue of memory bandwidth, we formulate a performance model to estimate the required memory bandwidth in both shared memory

and register blocking. Assume that there are two levels of memory hierarchy, i.e. level 2 is bigger but of high latency and level 1 is small but of low latency. Initially matrices A,B,C are stored in level 2. Let's denote DF and WB to be floating-point operations per second and the word size in bytes, respectively. A blocking algorithm loads blocks of three matrices from level 2 into level 1. As described in the previous section, a blocking algorithm reduces bandwidth requirement by $S = 2/(1/bm + 1/bn)$ times. The required memory bandwidth RB is computed as:

$$RB = DF * WB * 1/S = (DF * WB) * (1/bm + 1/bn)/2$$

where $DF * WB$ represents the required bandwidth for a peak floating-point performance.

The optimization is reduced to a problem of minimizing RB since memory bandwidth is often limited. Given a specific hardware architecture, the parameters DF,WB are constant. Thus, $1/bm + 1/bn$ should be minimized. The mathematical function reaches its minimal value when bm is equal to bn ($bm = bn$). Therefore, a blocking algorithm selects some appropriate values to satisfy:

$$RB = (DF * WB) * (1/bn) < BW$$

where WB is the theoretical peak bandwidth.

Shared memory blocking Firstly, we select proper values for bandwidth reduction. The global memory bandwidth of Fermi C2050 is 144GB/s. For double-precision the required bandwidth is computed as $RB = 515 * 8 * (1/bn) = 4120/bn$ GB/s. Thus, the values of bn (or bm) are selected to satisfy $4120/bn < 144$, that is, $bn = bm > 28$. In order to identify some restrictions on achieving maximal bandwidth, we develop a micro-benchmark program to measure the effective bandwidth. Figure 2 plots the achieved bandwidth under different modes. The experiment shows that block size should be the largest one with multiple of 16 for the highest bandwidth. Additionally, the CUDA programming manual suggests memory is aligned to 512 bytes, which would make 64 double-precision values a perfect fit.

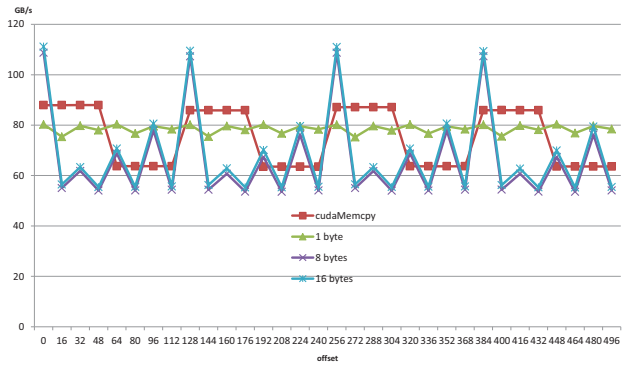

Figure 2: The effective bandwidth is affected by offset and width

Secondly, the optimal access pattern or thread-data mapping is identified for maximizing global memory efficiency. The benchmark program performs a device to device memory copy, comparing different per-thread word sizes while accessing memory from different offsets. The array occupies 480MB global memory (device memory) and is assigned to a grid of 30720 blocks of 1024 threads. The plotted bandwidth is calculated as the ratio of the copied memory size (480MB) to the kernel execution time. As a reference, we also compare our benchmark to the bandwidth test program in the CUDA SDK. Figure 2 shows the achieved effective bandwidth is determined by both offset and width. If each thread copies one double (64-bits) or two doubles (128-bits), performance is very sensitive to the offset. The mode of 128-bits is slightly better than that of 64-bits. An important observation is that *the effective bandwidth is maximized if threads in a group of 16 all use addresses in the same 128B-aligned region.*

Based on the above observation, one column of A or one row of B with 64 double words (512 bytes) may be assigned 64 or 32 threads. For the initial version presented in this section, we use 64-bits, since the CUDA programming manual indicates that 128-bit memory operations will result in at least a two-way bank-conflict in shared memory. Bank conflicts result in serialization that leads the 128-bits to be loaded in two passes of 64-bits each.

Register blocking Using the same philosophy for shared memory blocking, we first look for appropriate register blocking sizes (rx,ry). According to the CUDA programming manual, Fermi takes 2 cycles to issue an entire warp which fits with a 16-wide SIMD and a 32-wide warp. There are 32 banks, each with 4 bytes width. Therefore, the shared memory bandwidth is calculated as:

$$BW_{sm} = 4 * 32 * 0.5 * 1.15 * 14 = 1030GB/s$$

where 1.15 means core frequency 1.15GHz, 0.5 is the warp issue rate and means that it takes two cycles to issue a warp, 14 is the number of SMs. Based on our performance model, register block-ing sizes are selected to satisfy:

$$RB = (DF*WB)*(1/rx) = 4120/rx < BW_{sm} = 1030$$

which means that $rx(ry)$ should be 4 at least.

On the other hand, each SM has only 32768 registers. Our benchmarking experiments identify that there should be at least two concurrent thread blocks of 192 threads (6 warps) on an SM to hide independent instruction reissue latency. Furthermore, the maximum number of registers per thread is limited to 63. Since double precision values require 2 registers, and we require $rx * ry$ values for the accumulators of the C matrix, and $rx + ry$ values of A and B to be multiplied, the choice of rx and ry should meet $2(rx * ry + rx + ry) < 63$. If we assume $rx = ry$, then $rx = ry = 4$ is the maximum blocking factor. If we choose a larger blocking size there will not be enough registers which will lead to a performance penalty due to register spilling.

Until now, we have defined the values of parameters $bm = bn = 64$,$rx = ry = 4$. Obviously, bk is determined by the shared memory size on each SM. In

our configuration, each SM on Fermi has 48KB shared memory and run 2 thread blocks at least. We choose a maximal bk which satisfies $2*2*64*bk*8B < 48KB$, thus, $bk = 16$. Looking at Algorithm 1, the number of registers of each thread is $48 = (rx*ry+rx+ry)*2$ at least, where 2 means one double word occupying two 32-bits registers. To support more than two thread blocks on one SM, the size of each thread block should be less than $32768/(48*2) = 341$. Considering that the minimal number of threads (512), we choose thread block size to be $256 = 64*4$. For a block of size $bm*bk = 64*16$, each thread needs to load 4 elements.

```
Loop:
load.64 rA[0],smA[k][threadIdx.x]; load.64 rA[1],smA[k][threadIdx.x+16];
load.64 rA[2],smA[k][threadIdx.x+32]; load.64 rA[3],smA[k][threadIdx.x+48];
load.64 rB[0],smB[k][threadIdx.y]; load.64 rB[1],smB[k][threadIdx.y+16];
load.64 rB[2],smB[k][threadIdx.y+32]; load.64 rB[3],smB[k][threadIdx.y+48];
dfma accum[0][0],rA[0],rB[0],accum[0][0]; dfma accum[0][1],rA[0],rB[1],accum[0][1];
dfma accum[1][0],rA[1],rB[0],accum[1][0]; dfma accum[1][1],rA[1],rB[1],accum[1][1];
dfma accum[0][3],rA[0],rB[2],accum[0][2]; dfma accum[0][3],rA[0],rB[3],accum[0][3];
dfma accum[1][2],rA[1],rB[2],accum[1][2]; dfma accum[1][3],rA[1],rB[3],accum[1][3];
dfma accum[2][0],rA[2],rB[0],accum[2][0]; dfma accum[2][1],rA[2],rB[1],accum[2][1];
dfma accum[3][0],rA[3],rB[0],accum[3][0]; dfma accum[3][1],rA[3],rB[1],accum[3][1];
dfma accum[2][2],rA[2],rB[2],accum[2][2]; dfma accum[2][3],rA[2],rB[3],accum[2][3];
dfma accum[3][2],rA[3],rB[2],accum[3][2]; dfma accum[3][3],rA[3],rB[3],accum[3][3];
Repeat Loop for 16 times (k=0,1,...15)
```

Figure 3: A pseudo-code of the inner loop. *load*.64 **is defined to be loading a 64-bit word from shared memory to registers.** *dfma* **is the fused multiply-and-add .** rA, rB **and** *accum* **are register variables,** smA **and** smB **are shared memory addresses.**

3.2 Analysis

Based on the analysis in the previous subsection, the blocking algorithm is devised to maximize the utilization of memory bandwidth. In this subsection we discuss a few potential ways to improve performance. In Figure 5 the blue line plots the performance of this baseline version. The baseline program only achieves an efficiency of 54%, which is worse than CUBLAS3.2. Looking at the inner loop with data for blocks of A and B in shared memory, the performance is totally determined by the two nested loops in Figure 1. Figure 3 lists pseudo-assembly code for the inner *for*-loop. In each inner loop there are 8 memory instructions and 16 floating-point instructions. Totally there are 128 memory instructions and 256 floating-point instructions for all $bk = 16$ loops. In the outer *while*-loop there are 8 load instructions and 8 store instructions that read both matrices A and B from global memory, then store them into shared memory.

- Note that there are 8 global memory operations and 136 shared memory load/store operation. However, the cost (latency) of global memory operations is about 100 times (see more details in Table 3 in Section 4.3) more than that of shared memory operations so that the 8 global memory operations take more time to finish. Therefore, the first priority is hiding the latency of global memory operations. Note that there are two levels of cache between global memory and register. We may make use of software prefetching to hide the long latency. One way is to load the next block into register files just before current block is calculated. Figure 4 describes the software prefetching strategy. As shown by the green line in Figure 5, this optimization improves the efficiency to 57%, which is very close to the efficiency of CUBLAS3.2 (red line).

Algorithm 2
The size of thread block: $vlx * vly$
Register: $accum[rx*ry]$, //$rx*ry$ is a factor of register blocking
$rA[rx]$,$rB[ry]$,$nrA[rx]$,$nrB[ry]$
Shared memory: $smA[bk][bm]$,$smB[bk][bn]$
//
$accum[0...rx][0...ry]$=0
load one $bm*bk$ block of A into $smA[bk][bm]$
load one $bk*bn$ block of B into $smB[bk][bn]$
synch
while (–K>0) {
  prefetch one $bm*bk$ block of A into $nrA[rx]$
  prefetch one $bk*bn$ block of B into $nrB[ry]$
  for (ki=0;ki<bk;ki++) {
    load one column of A in smA into $rA[0...rx]$
    load one row of B in smB into $rB[0...ry]$
    $accum[0...rx][0...ry] += rA[0...rx]*rB[0...ry]$
  } //end for
  store $nrA[rx]$ into $smA[bk][bm]$
  store $nrB[ry]$ into $smB[bk][bn]$
  synch
} //end while
Merge $accum[0...rx][0...ry]$ with $bm*bn$ block of C.

Figure 4: The algorithm with software prefetching in registers. The red texts highlight the changes (the same way is used in Algorithm 3)

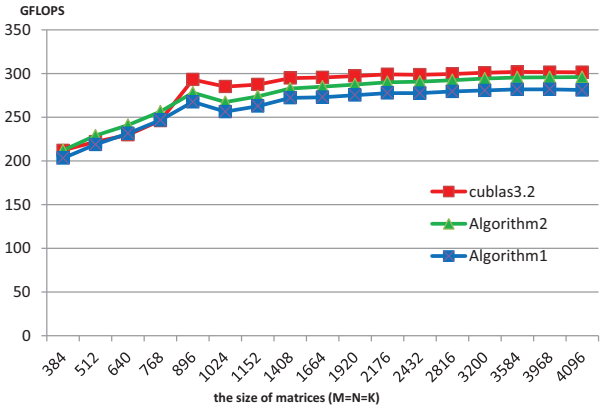

Figure 5: The performance of our initial DGEMM routines

- However, we note that a disadvantage of Algorithm2 in Figure 4 is the use of extra registers, i.e. additional 8 registers are temporarily used to store the next block of matrices A and B. The requirement of more registers leads to register spilling to local memory. Like global memory, local memory is also cached on Fermi. By profiling the execution, we compare the number of local memory access between optimization code (Prefetching) and CUBLAS3.2 in Figure 6. CUBLAS3.2 has no local memory access while there are amount of local memory access in our code. The analysis implies an optimization to *implement the software prefetching strategy with as less register as possible.*

- A rule of thumb tells us that the floating point

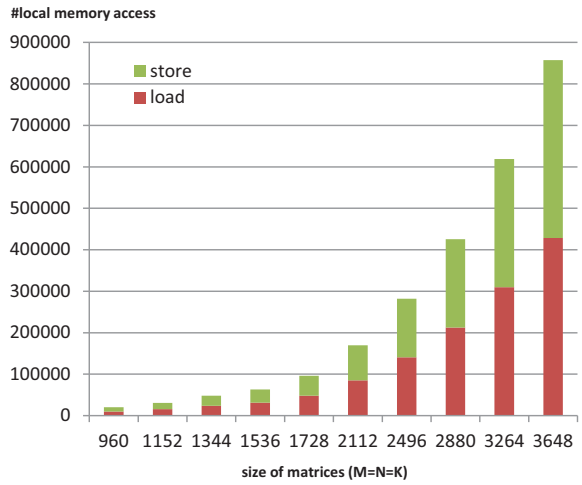

Figure 6: The number of local memory accesses

Table 2: Latency of floating-point load instructions. It excludes the latency of two independent instruction reissues, each of which takes 6 cycles (reissue latency)

Instruction	ld.32	ld.64	ld.128
Latency	26	38	46

throughput of an application largely depends on the percentage of the floating point instructions. This is especially true on Fermi since the peak instruction throughput is equal to the peak floating point throughput, thus the floating point efficiency of an application is limited by the percentage of floating point instructions. The ratio in Algorithm 1 is $256/(128+8+8+256) = 64\%$, which is the upper bound it can reach. Both CUDA3.2 and our optimized routine reach at about 58% and almost approach this theoretical value.

Note that CUDA3.2 on Fermi supports 128-bits load/store operations. Obviously, the use of 128-bits load/store instructions will increase the ratio of floating-point operations to $256/(64+4+4+256) = 78\%$, which means that we may achieve 78% efficiency of peak performance. However, CUDA programming manual points out that 128-bits load/store operations always result in at least 2-way conflicts in shared memory. We write a micro-benchmark program to measure latency of load instructions with different width. Table 2 summarizes the latency of each type of instructions. Due to long latency, the next step is to seek a way to *hide latency for the algorithm that uses 128-bit load/store instructions.*

4. OPTIMIZATION

The analysis above indicates that the use of 128-bit memory instructions improves the theoretical efficiency from 64% to 78%. In this section we present an aggressively optimized DGEMM algorithm, which combines both software prefetching without usages of extra registers and 128-bits load/store instructions. The challenges are no extra register requirement for prefetching and minimal effect of longer latency caused by 128-bits memory operations in shared memory. We first describe the optimizations at the level of shared memory blocking in section 4.1 and 4.2. The strategies include data layout, thread mapping and double-buffering. Then we focus on improving the second level of register blocking by a careful instruction scheduling in section 4.3.

4.1 Data-Thread Mapping

The use of 128-bits memory operations leads to another different data-thread mapping. As discussed before, two warps load A/B's one column/row of length 64 (bm or bn) and each thread loads one 64-bit double word. If the 128-bit load instructions are used, we only need 32 threads (one warp), each thread loading two doubles (128-bits). Thus we change the thread block of $64*4$ to $32*8$. Figure 7 depicts the shape of data-thread mapping using two modes of memory operations. The left picture shows the process of reading the data block of $64*16$ from global memory to shared memory using 64-bit load operations. The right one is the counterpart using 128-bit load operations.

Assume that matrices are stored in global memory with column-wise layout, but the blocks are stored in shared memory with row-wise layout. As shown in Figure 7, each thread in the 64-bit mode issues four load/store instructions to fetch data while one in the 128-bit mode issues two load/store instructions for a block of $64*16$. For example, by issuing four 64-bits memory instructions, thread $0\ldots63$ fetch columns $0,4,8,12$ (the blue columns), thread $64\ldots127$ fetch columns $1,5,9,13$ (the dark red columns), thread $128\ldots191$ fetch columns $2,6,10,14$ (the red columns), thread $192\ldots255$ fetch columns $3,7,11,15$ (the yellow columns). In shared memory each column of 64 elements (64-bit double words) is stored as a row, which is contiguously accessed by a column of 64 threads. The right picture illustrates the corresponding mapping with 128-bit memory operations. Thread $0\ldots31$ fetch columns $0,8$ (the blue columns) with two 128-bit memory instructions, thread $32\ldots63$ fetch columns $1,9$ (the dark red columns). In the same way, each column of threads fetches two columns (the same color) by issuing two 128-bit memory instructions.

4.2 Double-Buffering

If we abstract Fermi's memory hierarchy to be two levels composed of global memory and shared memory, it is very similar to other many-core architectures like IBM Cell and IBM Cyclops64 with explicitly software controlled memory architecture. It is proven that software pipelining using double-buffers in the low latency memory is an efficient method to overlap computation with communication through the memory hierarchy.

In order to implement a double-buffering strategy, we split the data block of $64*16$ into two halves of $64*8$. When the first half is being used to compute matrix C, the instructions of fetching the second half are issued. The thread scheduler is responsible for overlapping computing with memory operations. The double-buffering algorithm is outlined in Figure 8. In the pseudo-code we map $smA/B[0\ldots bk/2-1][bm]$ to buffer 0 in Figure 7 and $smA/B[bk/2-1\ldots bk-1][bm]$ to buffer 1. Within the *while*-loop, the grouped instructions are

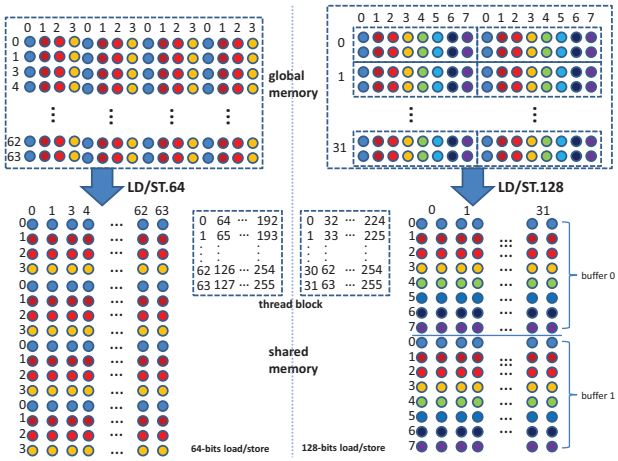

Figure 7: Data-thread mapping for data transfer between global memory and shared memory. This picture is split into two parts by the dashed line. The left part illustrates the mapping in Algorithm 2 using 64-bit memory operations. The right one illustrates the mapping in Algorithm 3 using 128-bit memory operations

Algorithm 3

The size of thread block: $vlx * vly$
Register: $accum[rx * ry]$,//$rx * ry$ is a factor of register blocking $rA[rx]$,$rB[ry]$
Shared memory: $smA[bk][bm]$,$smB[bk][bn]$
//
1.$accum[0 \dots rx][0 \dots ry]$=0
2.load one $bm * bk/2$ block of A into $smA[0 \dots bk/2-1][bm]$
3.load one $bk/2 * bn$ block of B into $smB[0 \dots bk/2-1][bn]$
4.**synch**
5.**while** (--K>0) {
6. load one $bm * bk/2$ block of A into $smA[bk/2 \dots bk-1][bm]$
7. load one $bk/2 * bn$ block of B into $smB[bk/2 \dots bk-1][bm]$
8. **for** (ki=0;ki<bk/2;ki++) {
9. load a col of A in $smA[0 \dots bk/2-1][bm]$ into $rA[0 \dots rx]$
10. load a row of B in $smB[0 \dots bk/2-1][bn]$ into $rB[0 \dots ry]$
11. $accum[0 \dots rx][0 \dots ry] += rA[0 \dots rx] * rB[0 \dots ry]$
12. } //end for
13. **synch**
14. load one $bm * bk/2$ block of A into $smA[0 \dots bk/2-1][bm]$
15. load one $bk/2 * bn$ block of B into $smB[0 \dots bk/2-1][bn]$
16. **for** (ki=bk/2;ki<bk;ki++) {
17. load a col of A in $smA[bk/2 \dots bk-1][bm]$ into $rA[0 \dots rx]$
18. load a row of B in $smB[bk/2 \dots bk-1][bn]$ into $rB[0 \dots ry]$
19. $accum[0 \dots rx][0 \dots ry] += rA[0 \dots rx] * rB[0 \dots ry]$
20. } //end for
21. **synch**
22.} //end while
23.Merge $accum[0 \dots rx][0 \dots ry]$ with $bm * bn$ block of C.

Figure 8: The algorithm with double-buffering strategy

separated by synchronization into memory and computing operations. Since they operate on different shared memory buffers, memory operations can proceed in parallel with computing. Without data dependency, there is further room for instruction scheduling optimizations to hide memory latency.

Comparing with Algorithm 2 in Figure 4, the new algorithm does not use an extra register to implement double-buffering strategy. In fact, one significant change is the use of 128-bits memory operations with longer latency. Besides, in order to make sure that data in the next block is totally loaded into shared memory before computation, the double-buffering forces us to use one more synchronization instruction in the *while*-loop. Obviously, in addition to 128-bit load/store instructions, the extra synchronization results in additional latency. Since the major penalty is extra latency, we therefore believe that there exists room for optimizing instruction scheduling to hide these latencies.

4.3 Instruction Scheduling

The *while*-loop in Algorithm 3 occupies most of the execution time. Our instruction scheduling focuses on this section of code. Before the algorithm for scheduling instructions is detailed, we first figure out the instruction mix for executing this code.

Table 3 summarizes the instruction mix. As explained before, each thread loads one element (quad-word. In the following context, we refer to one element as 128-bits) from global memory to one of two buffers in shared memory (see lines 6-7,14-15 in Algorithm 3). Since there are no memory-memory instructions, one data movement operation is translated into two memory instructions: one load ($ld.gm.128$) from global memory to registers and one store ($st.sm.128$) from registers to shared memory. For each *while*-loop, four memory instructions are issued to fill the double buffers. Now let's check the two inner iterations for each half buffer. In every *ki*-loop, each thread reads two elements of A and two elements of B into registers, then computes a four-by-four sub-block of C. The memory operations are composed from 4 load instructions ($ld.sm.128$) in shared memory, and the computing operations from 16 floating-point instructions ($dfma$). Totally, there are 64 $ld.sm.128$ instructions and 256 $dfma$ instructions for all $bk = 16$ loops. Additionally, we need 10 integer instructions and one branch instruction for address calculation and loop control.

In order to facilitate to scheduling sequence of instruction execution, we measure pipeline latency of the instructions used in the *while*-loop. Table 3 lists the instruction latencies. Due to data dependency, we classify latency into two types: read-after-write (RAW) and write-after-read (WAR). For example, assume that instruction y depends on instruction x, the RAW latency is the number of cycles from the time instruction x is issued to the time instruction y that reads the content of a register written by x can be issued. The WAR latency is the number of cycles from the time instruction x is issued to the time instruction y that writes the content of a register read by x can be issued.

The execution of an instruction stream is scheduled based on the measured latencies. Given a sequence of instructions, we scan through the code, and for each instruction, we calculate how long it takes to make the values of registers available. For an instance of $ld.gm.128$ $r2$,[$r1$] instruction, the values of registers $r2$,$r3$,$r4$,$r5$ are available after it is issued for several hundred cycles. In our scheduling algorithm we refer to this latency as an effective time of a register (i.e. $r2$'s effective time is 332∼1000 cycles, depending on whether data is cached or not). Thus, we dynamically construct two map tables $last\_raw$ and $last\_war$ to record the register name and its effective time for each instruction (i.e. $last\_raw[r2] = 1000$). At the same time, we track

Table 3: Instruction mix in each iteration of the *while*-loop. The "integer" contains addition and branch instructions

inst.	counts	ratio(%)	RAW latency	WAR latency
$ld.gm.128$	4	1.17	332∼1000	46
$ld.sm.128$	64	18.76	46	46
$dfma$	256	75.07	24	N/A
$integer$	13	3.80	18	N/A

the accumulated execution time (t_{curr}) until the current instruction is being scheduled. The registers of the current instruction are checked if they are in either $last\_raw$ or $last\_war$. If they are in neither table, its stall time for issue is zero because there is no data dependence. Otherwise, if one of the register hits in either one of two tables, the stall time of the current instruction for issue is calculated as the difference between the effective time and current time. For example, if instruction $st.sm$ $[smA]$,$r2$ after $ld.gm.128$ $r2$,$[r1]$ is currently scanned, its stall time equals to $last\_raw[r2] - t_{curr}$. For instance, assume that there are four independent instructions between $ld.gm$ and $st.sm$. Note that each of these four independent instructions has a 6-cycle no-stall issue-to-issue latency, the observed latency of $ld.gm$ to $st.sm$ (or say the extra number of cycles $st.sm$ stalls) is $1000 - (t_{curr} = 4 * 6 + 6) = 970$ cycles ($st.sm$ has a 6-cycle latency even without RAW delay). The objective of instruction scheduling is to minimize the accumulated stall time of all instructions.

```
//threadIdx.x=0...31
//Loop 0:
ld.sm.128 rA[0],smA[0][threadIdx.x];// for loop 0
ld.sm.128 rB[0],smB[0][threadIdx.x];// for loop 0
ld.sm.128 rB[2],smB[2][threadIdx.x];// for loop 0
dfma accum[0][0],rA[0],rB[0],accum[0][0]; dfma accum[0][1],rA[0],rB[1],accum[0][1];
dfma accum[1][0],rA[1],rB[0],accum[1][0]; dfma accum[1][1],rA[1],rB[1],accum[1][1];
ld.sm.128 rA[2],smA[2][threadIdx.x]; // for loop 0
dfma accum[0][3],rA[0],rB[2],accum[0][2]; dfma accum[0][3],rA[0],rB[3],accum[0][3];
dfma accum[1][2],rA[1],rB[2],accum[1][2]; dfma accum[1][3],rA[1],rB[3],accum[1][3];
ld.sm.128 rA[0],smA[0][threadIdx.x]; // for loop 1
dfma accum[2][0],rA[2],rB[0],accum[2][0]; dfma accum[2][1],rA[2],rB[1],accum[2][1];
dfma accum[3][0],rA[3],rB[0],accum[3][0]; dfma accum[3][1],rA[3],rB[1],accum[3][1];
ld.sm.128 rB[0],smA[0][threadIdx.x]; // for loop 1
dfma accum[2][2],rA[2],rB[2],accum[2][2]; dfma accum[2][3],rA[2],rB[3],accum[2][3];
dfma accum[3][2],rA[3],rB[2],accum[3][2]; dfma accum[3][3],rA[3],rB[3],accum[3][3];
//Loop 1:
ld.sm.128 rA[0],smA[0][threadIdx.x]; //for loop 1
dfma accum[0][0],rA[0],rB[0],accum[0][0]; dfma accum[0][1],rA[0],rB[1],accum[0][1];
dfma accum[1][0],rA[1],rB[0],accum[1][0]; dfma accum[1][1],rA[1],rB[1],accum[1][1];
ld.sm.128 rA[2],smA[2][threadIdx.x]; // for loop 1
...//omitted other instructions for space because they're
cloned from loop 0
```

Figure 9: Instruction scheduling of the inner loop

A general algorithm for instruction scheduling is out of scope of this paper. In this section, we present a specific algorithm for arranging the order of instruction execution in *while*-loop of Algorithm 3. Initially, lines 6-21 of Algorithm 3 are directly transformed into assembly instructions described in Table 3 and then the two inner *for*-loops are completely unrolled.

At first, we re-arrange the sequence of instructions of each inner *for*-loop. Figure 9 illustrates an example of instruction re-ordering for the first two loops in the inner *for*-loop. For each loop there are 4 $ld.sm.128$ instructions followed by 16 $dfma$ instructions. Intuitively the execution cannot be re-ordered due to data dependencies in registers $rA[0\ldots3]$ and $rB[0\ldots3]$. However, as shown in Figure 9, registers $rA[0]$,$rA[1]$ are free after the first 8 $dfma$ instructions are executed, registers $rB[0]$,$rB[1]$ are free after the first 12 $dfma$ instructions are executed. This observation indicates that it is feasible to search proper points to insert these load instructions in unrolled loop for minimizing stall time because there are 8 possible points at most for searching. After that, we added scheduling of the 4 load/store instructions that load data from global memory into shared memory. Thanks to no data dependence between these instructions and the other 160 instructions bounded by synchronization, we can exhaustively search for the best points among the 160 points for these 4 instructions. Finally, after the second one of two $st.sm.128$ instructions is scheduled, we search an insert point starting from this store instruction for synchronization instructions.

5. EXPERIMENTAL RESULTS AND ANALYSIS

The proposed optimization strategies are involved with the exact selection and scheduling of instructions, they cannot be achieved at the level of either CUDA C or PTX language because the programs of CUDA C/PTX are transformed to the native machine instructions by compiler. With NVIDIA's internal tool-chain, we first implemented Algorithm 3 using Fermi's native machine language on NVIDIA Tesla C2050. Thanks to the regular access pattern in matrix, the memory operations are implemented by the corresponding texture memory instructions. In this section we first report our achieved performance with the proposed optimizations, then discuss some insights on general optimization and architectural impact. The benchmark programs are put in a public web site: http://asl.ncic.ac.cn/dgemm/dgemm_nv.html.

5.1 Performance

We initially expect that both double-buffering and instruction scheduling derive the performance improvement because these strategies save the use of registers and hide latency as well. Figure 10 reports their overall improvement of performance. It plots performance of our final version of DGEMM routine. Comparing with CUDA3.2, it improves floating-point performance by 20%, and reaches a peak of 362Gflop/s or floating-point efficiency of 70%. Although a combination of our optimization strategies indeed improves performance, a major premise of this success is that we have to implement the optimized program in assemble code by hand. In fact, as indicated by our analysis before, the performance bottleneck of DGEMM routine is latency after we choose an optimal implementation of blocking algorithm. Especially, the use of 128-bits memory operations makes the problem even worse. However, our optimization strategies mitigate the effect of instruction latency and improve performance.

We are also interested in figuring out which part is the main source of performance improvement. Note that the instruction scheduling algorithm actually finds proper insert points (before or after some floating-point arithmetic instruction) for all memory instructions. Our scheduling strategy is composed of two major steps: insert memory instructions of loading data in shared memory (the inner *for*-loop) and insert memory instructions of loading data from global memory then storing to shared memory. Therefore, in

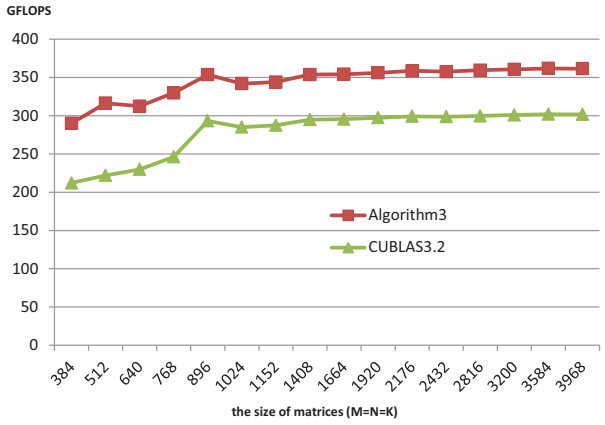

Figure 10: The performance of our final version of DGEMM

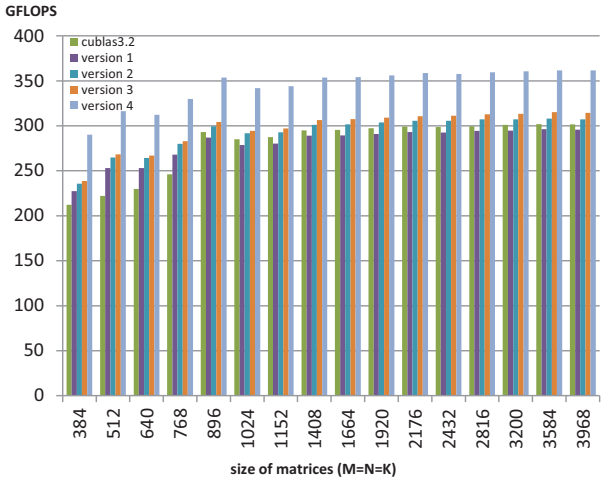

Figure 11: The incremental improvement by the optimization strategies. They are also compared to CUBLAS3.2

the experimental evaluation, there are four versions written in assemble code:

- *version 1*: Based on Algorithm 3, we modify it to only use 128-bits memory operations and do not implement double-buffering. The implementation eliminates one synchronization instruction in *while*-loop.
- *version 2*: Algorithm 3 is directly translated into assemble code without any instruction scheduling optimization.
- *version 3*: Based on *version 2*, the instructions in all inner *for*-loops are reordered by instruction scheduling optimization. That is, we only optimized latency hiding for shared memory accesses.
- *version 4*: Based on *version 3*, we further optimized the latency hiding for global memory access using instruction scheduling optimization. This is our final version.

Figure 11 plots an incremental improvement of the four versions. The green bars represent the performance of CUBLAS3.2. From the experiments, we conclude several observations:

- The cache hierarchy newly introduced by Fermi mitigates the overhead of register spilling. A fact we have to note is that the assembly codes of all the four versions use less than 63 registers by carefully scheduling instructions for more register reuse. However, not to speak of better than CUBLAS3.2, *version 1* does not achieve better performance than Algorithm 2 which uses more than 63 registers. Although register spilling results in local memory accesses, unlike the previous GT200, Fermi uses cache to hold most of local memory accesses. Therefore, the latency caused by register spilling may not be a main performance bottleneck to some extent. That is why the double-buffering strategy without extra register just slightly improves performance (see the blue bars for *version 2* in Figure 11).
- The latency of global memory access is always the first class of citizenship under the consideration of performance optimization. The comparison of *version 3* and *version 4* shows that scheduling global memory operations is the main source of performance improvement. *Version 3* improves the peak performance of floating-point by 5% comparing to CUBLAS3.2 while *version 4* further improves the peak performance by 15% compared to *version 3*.

As mentioned before, the ratio of double-precision floating-point instructions determines the upper bound of performance. The instruction mix in Table 3 shows that the theoretical efficiency of our algorithm on Fermi is 75%. We achieve a peak efficiency of 70%, which is close to the limitation. For example, although there are two warp schedulers, double-precision instructions cannot be combined with any other instructions for dual issue. This is the reason why the theoretical performance is estimated only by the ratio of double-precision floating-point instructions. Surprisingly the scheduling of global memory operations contributes the most to our performance improvement. We thought the compiler should perform well in this case because it is a significant bottleneck. Algorithm 2 with 64-bit memory operations approaches its performance limitation of 64% without any special optimization to its assembly code by hand. It is another story for Algorithm 3 with 128-bit memory operations. This indicates that there may be room for improvements in the compiler for dealing with 128-bit memory operations.

5.2 Discussion on General Optimization

In this paper we only present the optimization of DGEMM on Fermi. The proposed optimization strategies are closely related to Fermi's hardware architecture. However, we note that the ideas are adaptive to other dense matrix operations and performance critical libraries as well. As a matter of fact, DGEMM has been extensively used as a motivating example or benchmark for micro-architecture and compiler research since it lays stress on both computing capability

and memory access. A retrospect to the previous research shows that many optimizations on DGEMM were easily applied to other applications. It is expected that some ideas may be generalized and adopted by either hand or compiler for optimizing math libraries.

Firstly, a key observation in this work is the use of 128-bit memory instructions. The wider memory instructions result in less memory instructions in the kernel loop so that the peak floating-point efficiency is potentially improved, i.e. from 64% to 75% for DGEMM's efficiency. Such wide memory operations can be used in more scientific and engineering applications because most of them exhibit contiguous memory access pattern as that in DGEMM. The compiler can easily provide an option to users for choosing such wider memory operations, or automatically make the optimal choice for users.

Secondly, we implement the double-buffering strategy in shared memory instead of registers. It has been proven that double-buffering is a general approach to hide long memory latency. CUBLAS3.2 adopted MAGMA's idea to implement double-buffering (or software prefetching) by using more registers. In Figure 11 *version 2* represents our double-buffering algorithm in shared memory. The experimental results show that our implementation is a little better than CUBLAS3.2. In fact, due to the long latency of 128-bits memory instructions, *version 1* is worse than CUBLAS3.2 in some cases. The improvement of *version 2* indicates that double-buffering in shared memory provides more benefits thanks to less pressure of register usages.

Finally, the experiments show that the major performance improvement is achieved by instruction scheduling. The scheduling strategy is totally derived from instructions dependency, which is not specific to DGEMM. Although we use a method of exhaustion to search the optimal scheduling, we argue that the idea may be applicable to general optimizations because small piece of code often occupy most of the execution time in a lot of scientific and engineering applications. As an alternative way, auto-tuning techniques [3, 17] can be taken to find a better scheduling order. The instruction scheduling optimization requires support of assemble language.

Note that the experience on optimizing DGEMM shows that only use of 128-bit memory instruction does not improve performance in practice. It has to be combined with other latency hidden strategies including both double-buffering and instruction scheduling.

5.3 Architectural Impact

Based on our optimization and performance evaluation, we summarize several possibilities for improving the GPU architecture for higher performance on DGEMM and other dense linear algebra operations:

- *Increasing the number of registers per thread.* Although the overhead of register spilling may be mitigated by the cache hierarchy, the theoretical limit of performance based on the ratio of floating-point instructions can be increased with more registers because of larger blocking in registers. For example, our algorithm uses a register blocking of 4x4, the ratio of floating-point instructions within the inner-loop is $16/(16+4) = 80\%$ (or $16/(16+8) = 66\%$ using 64-bit memory operations). If the blocking size increases to 8x8, for example, the ratio is increased to $64/(64+8) = 88\%$ (or $64/(64+16) = 80\%$ using 64-bit memory operation).

- *Increasing shared memory bandwidth.* In fact, the shared memory bandwidth relative to floating point throughput has decreased from GT200 as well. Blocking is proven to be an efficient way to reduce the bandwidth requirement. However, larger blocking puts more pressure on the register file. Therefore, an increase in shared memory bandwidth will require lower blocking factors in registers.

- *Increasing instruction throughput.* The ability to perform memory and math operations simultaneously would allow our algorithms to reach much closer to the theoretical peak performance. We have seen this trend already with the GTX460 GPU which can issue twice as many instructions per clock compared to the original Fermi design.

- *More efficient instruction scheduler.* We have shown that a careful instruction scheduling by hand can improve performance. However, we are limited to instruction ordering within a warp. The actual instructions that are executed depend largely on the hardware scheduling of warps. Thus, there may be room for improvements either by more efficient hardware, or by software if the scheduler were made programmable.

6. RELATED WORKS

Since the GPU provides powerful capability in floating-point computing, it is not surprising that there have been a lot of works on optimizing applications on the GPU [10]. DGEMM is a compute intensive kernel in high performance computing applications. It is also an important benchmark for high performance processors. Volkov et.al. [15] presented a fast implementation on G80 architecture. A blocking algorithm was used to exploit locality in shared memory. They also presented detailed benchmarks of the GPU memory system, kernel start-up costs, and arithmetic throughput. However, we present a performance modeling and a detailed analysis on the rationality of choosing an optimal blocking factor, and an experience on instruction scheduling as well. Nath et.al. [9] has developed a dense linear algebra library MAGMA for heterogeneous architectures of multi-core+GPU. The library contains a DGEMM routines optimized for Fermi. MAGMA adopts a similar idea of double-buffering by prefetching data into extra registers instead of shared memory. MAGMA's performance is almost the same as CUBLAS3.2, but is lower than ours. Ryoo et.al.'s work [13, 14] provided some valuable findings on ratio of floating-point instructions and tiling. They discussed tiling algorithms under the constraint of registers per thread, but not memory bandwidth and instruction latency. Nakasato [8] implemented a fast DGEMM on ATI Cybress GPU. We share the common idea of blocking algorithm, but the micro-architecture is different between ATI Cybress and Nvida Fermi, leading to different optimization strategies. There are several works on auto-tuning DGEMM [7, 6] for searching the optimal blocking algorithm on the GPU. Our findings on optimal blocking and instruction scheduling may be complements to them.

An important part of this work is that we develop a set of micro-benchmarks to identify Fermi's micro-architecture

features. Wong et.al. [16] performed a comprehensive benchmarking work on GT200. However, we further use the benchmarking results to guide the selection of optimization strategies. Our work illustrates an example of connecting micro-benchmarking to program optimization. The experience may be an optimization guideline for other applications.

7. CONCLUSIONS

We have presented the fastest implementation of DGEMM on a single Fermi GPU. The optimized DGEMM routine attains 70% of peak speeds, which is higher than the 58% of peak speeds attained by CUBLAS3.2. The performance boost is achieved by carefully chosen optimizations to match the capabilities of the hardware. For example, a specific blocking algorithm is adaptive to parameters (capacities, banks, bandwidths) of both global memory and shared memory for wider memory operation instructions. Both double-buffering and an aggressive instruction scheduling are used to hide memory latencies. Our micro-benchmarking results also disclose some interesting architectural features, i.e. a rule to maximize DRAM efficiency and latencies of 128-bits memory instructions.

Actually, this work shows an example of how to use the newly introduced wider memory operations (128-bits load/store) for program optimization. There is a tradeoff to use these wider memory operations. They can reduce the number of load/store instructions, but incur more latency. This tradeoff puts a heavy burden on instruction scheduling. The experimental results in this paper indicate that either current CUDA compiler or warp scheduler (or both) is not well ready for this new feature. We also point out that the limitation for issuing double-precision floating-point arithmetic instructions really limits the performance of floating-point intensive programs like DGEMM.

8. REFERENCES

[1] AMD. *ATI Stream Computing OpenCL Programming Guide, rev1.05*, August 2010.

[2] E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Technical Report 20, LAPACK Working Note, May 1990.

[3] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. In *Proceedings of the IEEE*, volume 93, pages 293–312, 2005.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16:1–17, March 1990.

[5] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34:12:1–12:25, May 2008.

[6] C. Jang. Gatlas gpu automatically tuned linear algebra software. http://golem5.org/gatlas/.

[7] 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: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] N. Nakasato. A fast gemm implementation on the cypress gpu. *SIGMETRICS Perform. Eval. Rev.*, 38:50–55, March 2011.

[9] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi gpus. Technical Report 227, LAPACK Working Note, July 2010.

[10] NVIDIA. Cuda Community Showcase. http://www.nvidia.com/object/cuda_apps_flash_new.html.

[11] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/object/fermi_architecture.html, 2009.

[12] NVIDIA. *CUDA C Programming Guide, Version 3.2*, 2010.

[13] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[14] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[15] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[16] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software,*, ISPASS '10, pages 235–246, 2010.

[17] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? In *Proceedings of the IEEE*, volume 93, pages 358–386, 2005.