An Improved MAGMA GEMM for Fermi GPUs

Rajib Nath[1], Stanimire Tomov[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee (USA)
[2] Oak Ridge National Laboratory (USA)
[3] University of Manchester (UK)

July 20, 2010

Abstract. We present an improved matrix-matrix multiplication routine (GEMM) in the MAGMA BLAS library that targets the Fermi GPUs. We show how to modify the previous MAGMA GEMM kernels in order to make a more efficient use of the Fermi's new architectural features, most notably their extended memory hierarchy and sizes. The improved kernels run at up to 300 GFlop/s in double and up to 600 GFlop/s in single precision arithmetic (on a C2050), which is 58% of the theoretical peak. We compare the improved kernels with the currently available in CUBLAS 3.1. Further, we show the effect of the new kernels on higher level dense linear algebra (DLA) routines such as the one-sided matrix factorizations, and compare their performances with corresponding, currently available routines running on homogeneous multicore systems. A general conclusion is that DLA has become a better fit for the new GPU architectures, to the point where DLA can run more efficiently on GPUs than on current, high-end homogeneous multicore-based systems.

1 Introduction

GEMM is a fundamental linear algebra routine. Many numerical algorithms can be expressed in terms of GEMM, or at least designed to partially use GEMM. Numerous examples from the area of DLA can be seen in the LAPACK library [3]. The technique to achieve that in DLA is based on *delayed updates* – the application of basic linear transformations, e.g., expressed in terms of matrix-vector multiplications, are delayed and accumulated so that they are applied later at once, e.g., as a matrix-matrix multiplication.

The importance of having algorithms rich in GEMM is because the computational intensity of GEMMs can be increased by increasing the sizes of the matrices involved, which in turn is crucial for the performance on modern architectures with memory hierarchy. Major hardware vendors such as Intel, IBM, AMD, and NVIDIA maintain their own highly optimized GEMM routines, e.g., included into their BLAS implementation libraries – MKL [15], ESSL [14], ACML [11], and CUBLAS [12] correspondingly. Non-vendor optimized implementations for various architectures are also available, e.g., ATLAS [16] and GotoBLAS [13].

In the area of GPU computing, "high" performance GEMM implementations were not possible in the "early" GPUs [4]. The reason is that they did

not have memory hierarchy and therefore, the GEMM's performance peak was memory bound. With the introduction of memory hierarchy, e.g., the shared memory in the NVIDIA CUDA GPU architectures [1], this has changed. Algorithms that would reuse data brought into the shared memory were developed to achieve high, compute bound performance [10]. The performance of these algorithms relied on a number of very well selected parameters and optimizations [17]. Subsequent work in the area managed to "automate" the selection of these parameters and optimizations used, to quickly find the best performing implementations for particular cases of GEMM [5, 6].

The Fermi architecture introduced new features to CUDA [7]. Although old code would run on the Fermi architecture and would often be somehow faster, further optimizations, exploring the new features, could significantly accelerate it. We show that this is the case with the previous state-of-the-art GEMM implementations. Moreover, we have found that even the auto-tuning frameworks can not find the new "optimal" implementations, simply because their search space did not consider the newly introduced features.

Section 2 is an overview of the GEMM for the previous generation GPUs. Section 3 gives the main contribution of this paper – an improved GEMM for the Fermi architecture. Section 4 shows the effect of the improved kernels on higher level dense linear algebra (DLA) routines such as the one-sided matrix factorizations from the MAGMA library [9], and compares their performances with corresponding, currently available routines running on homogeneous multicore systems. Finally, Section 5 is on conclusions and future work.

2 GEMM for GTX280

This section gives an overview of the GEMM targeting the old generation of GPUs, e.g., the GTX280 GPU. We consider a GEMM algorithm [10], parametrized to facilitate auto-tuning [5, 6] for the case $C := \alpha AB + \beta C$. The computation is done on a two-dimensional grid of thread blocks (TBs) of size $N_{TBX} \times N_{TBY}$. Each TB is assigned to $N_T := N_{TX} \times N_{TY}$ threads.

For simplicity, take $N_T := N_{TBX}$. Then, each thread is coded to compute a row of the sub-matrix of C. To do that, it accesses the corresponding row of A (as indicated by an arrow in Figure 1), and uses the $K \times N_{TBY}$ sub-matrix of B for computing the final result. The TB computation is blocked, which is crucial for obtaining high performance. In particular, sub-matrices of B of size $nb \times N_{TBY}$ are loaded into shared memory and multiplied nb times by the corresponding $N_{TBX} \times 1$ sub-matrices of A. The $N_{TBX} \times 1$ elements are loaded and kept in registers while multiplying them with the $nb \times N_{TBY}$ part of B. The result is accumulated to the resulting $N_{TBX} \times N_{TBY}$ sub-matrix of C, which is kept in registers throughout the TB computation. All memory accesses are coalesced.

Kernels for various $N_{TBX}, N_{TBY}, N_{TX}, N_{TY}$, and nb are automatically generated in MAGMA BLAS to select the best performing for a particular architecture and for particular GEMM parameters [6]. The theoretical peak of the

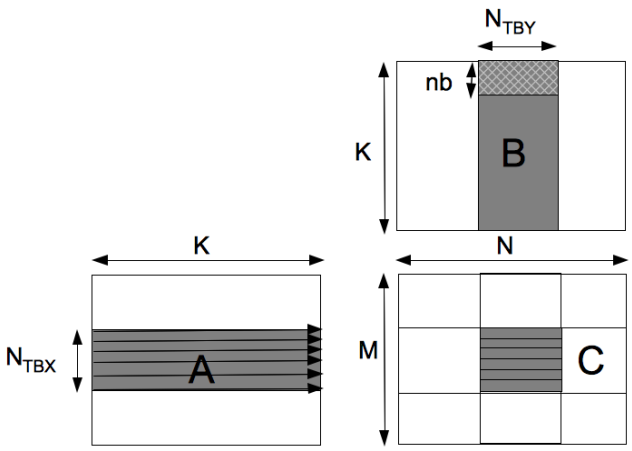


Fig. 1. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for GTX280.

GTX280 is 936 GFlop/s in single precision (240 cores × 1.30 GHz × 3 instructions per cycle). The kernel described achieves up to 40% of that peak.

3 GEMM for Fermi

Many of the architectural changes in Fermi are related to scaling up the compute capabilities of the 1.x generation of GPUs [8], e.g., increased shared memory, number of registers, number of CUDA cores in a multiprocessor, etc. The algorithm from Section 2, designed for GPUs of compute capability 1.x, can be automatically adjusted to account for those changes in the Fermi GPUs. In addition, there are other changes that must be exploited (for performance) which unfortunately, is impossible to accomplish by simply auto-tuning the old algorithm. For example, these are the changes that are related to added cache memories, and most importantly, that the latency to access register and shared memory were comparable in 1.x GPUs, but not in the Fermi (where accessing data from registers is much faster). This motivates to add one more level of blocking in the algorithm, namely register blocking, to account for the added memory hierarchy. A way to do it is to have blocks of both matrices A and B first loaded into shared memory, and second, additionally block the computation with the matrices in shared memory by loading parts of them in registers to get reuse of the data in registers (*vs* to reuse only data in the shared memory). Details on this new algorithm are given as follows.

The algorithmic view of the improved GEMM for Fermi is shown in Figure 2. Similarly to the old GEMM, the computation is divided into two-dimensional grid of TBs of size $N_{TBX} \times N_{TBY}$. Each TB is assigned to $N_T = N_{TX} \times N_{TY}$ threads. As mentioned above, sub-matrices of both A and B are loaded in shared memory. We take $N_{TBX} = N_{TBY} = 64$ and $N_{TX} = N_{TY} = 16$. With

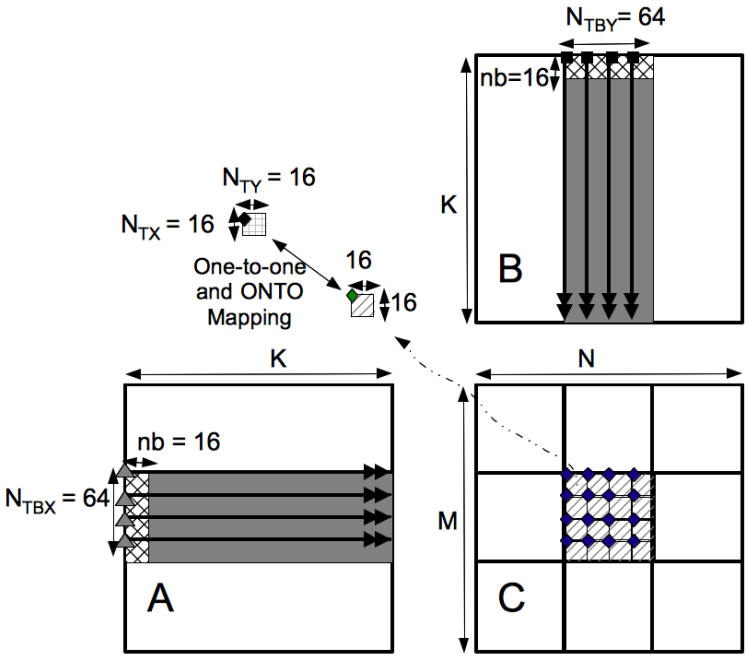
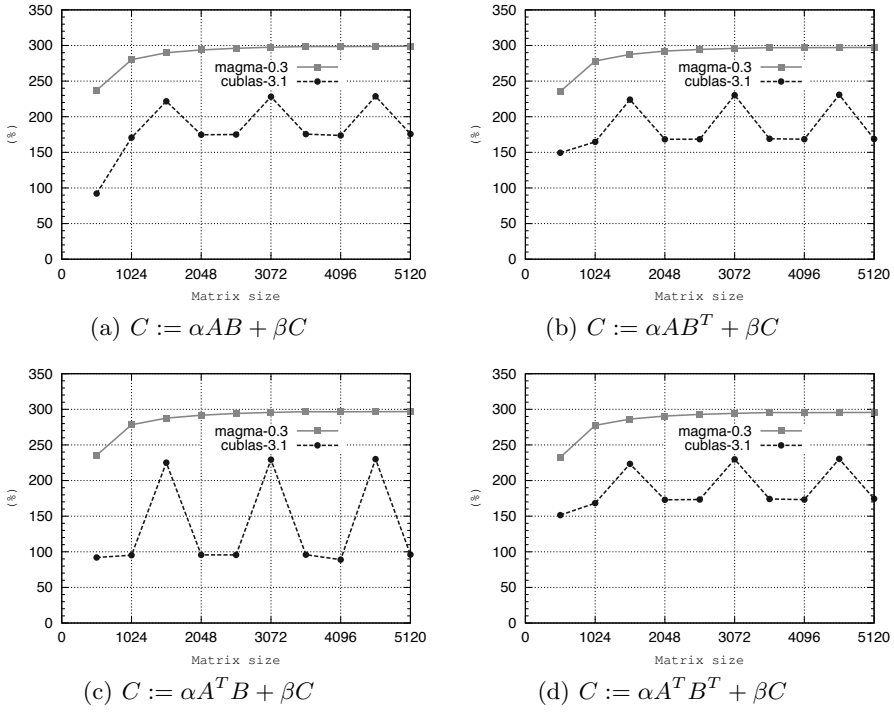


Fig. 2. The GPU GEMM ($C := \alpha AB + \beta C$) of a single TB for Fermi.

these parameter values, 16 × 16 threads will be computing 64×64 elements of matrix C. Hence each thread will be computing 16 elements. The 64×64 block of matrix C is divided into 16 sub-blocks of dimension 16×16 as illustrated in Figure 2. Each of the 16×16 sub-blocks is computed by 16×16 threads, i.e., one element is computed by one thread. More precisely, element (x, y) which is represented by a green diamond is computed by thread (x, y), represented by a black diamond, where $0 \leq x, y \leq 15$. All of the 16 elements computed by thread $(0, 0)$ are shown by blue diamonds in the figure. In summary, each thread will be computing a 4×4 matrix with stride 16. This distribution allows us to do coalesced writes of the final results from registers to the matrix C in global memory. Other distributions may not facilitate coalescent writes.

At each iteration of the shared memory blocking, all threads inside a TB load 64×16 elements of A and 16×64 elements of B to shared memory in a coalesced way. Depending upon $Op(A)$ and $Op(B)$, the 256 threads in the TB take one of the following shapes: 16×16 or 64×4. This reshaping helps coalesced memory access from global memory. The elements from matrices A and B, needed by thread $(0, 0)$, are shown by arrows. First, four elements from A (taken from the shared memory, shown by gray triangle) and four elements from B (again from the shared memory, shown by black rectangle) are loaded into registers. Then these 8 elements are used in 16 FMAD operations. To get a further small performance increase, all the accesses for matrices A and B are done through



(a) $C := \alpha AB + \beta C$ (b) $C := \alpha AB^T + \beta C$

(c) $C := \alpha A^T B + \beta C$ (d) $C := \alpha A^T B^T + \beta C$

Fig. 3. MAGMA BLAS DGEMM performance on Fermi (C2050).

texture memory. The performance of DGEMM in Fermi using this algorithm is shown in Figure 3, along with the DGEMM performance from CUBLAS 3.1. Note that the theoretical peak of the Fermi, in this case a C2050, is 515 GFlop/s in double precision (448 cores $\times$ 1.15 GHz $\times$ 1 instruction per cycle). The kernel described achieves up to 58% of that peak.

4 One-sided factorizations on Fermi

In this section we show the performance of the one-sided matrix factorizations using the new kernels. Figure 4 gives the performance of the LU, QR, and Cholesky factorizations from the MAGMA library in double precision. The computation is on a Fermi GPU (C2050) and uses the new DGEMM. The performance of these algorithms is derived from the performance of DGEMM and asymptotically should go to 300 GFlop/s. To put these results in the context of other efforts, we compare our LU with that of some commercial libraries, running correspondingly on GPUs and on x86 multicore systems.

The CULA library provides similar functionality to MAGMA. Asymptotically, e.g., for matrices of size $\approx 8,000$, the LU factorization performance, as

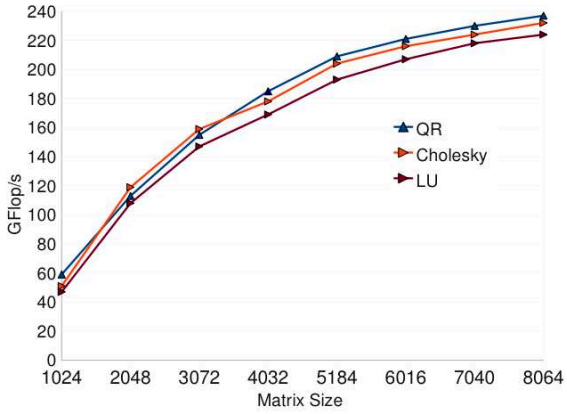


Fig. 4. Performance of the one-sided factorizations from the MAGMA library in double precision arithmetic on Fermi (C2050).

advertised on the CULA web-page [4], is about 137 GFlop/s in double precision. Note that MAGMA significantly outperforms CULA – on the same GPU (C2050) and matrix size MAGMA's performance is 224 GFlop/s. In other words, for this case MAGMA's LU is 65% faster on the Fermi architecture.

Figure 5 compares the performance of LU factorization in double precision arithmetic from MAGMA on Fermi (C2050) with that of MKL 11.0, PLASMA [2], and LAPACK on a 48 core system. The exact specifications are given in the figure. Note that the Fermi and the multicore system have the same theoretical peaks. The implementations of the LU factorizations in MAGMA, MKL and LAPACK use the same data layout and algorithm – LU with partial pivoting. The algorithm in PLASMA is different – LU with pairwise-pivoting on tile data-layout. It is interesting to note that MAGMA achieves significantly higher percentage of the GPU's peak than the percentage of the peak that the other libraries achieve on the multicore system. That is, we showed a case where LU runs more efficiently on GPUs than on current, high-end homogeneous x86-based multicore systems. Moreover, GPUs have better power efficiency and better system cost. As an example, the cost of the 48-core system is approximately $\$30,000$ and the cost of the Fermi GPU and its host is about $\$3,000$. These facts further motivate the need for developing fundamental linear algebra algorithms for GPUs.

5 Conclusions and future directions

The development of fast BLAS, and in particular GEMM, is crucial for the performance of many algorithms, and therefore is of extreme interest. We presented

[4] http://www.culatools.com/features/performance/

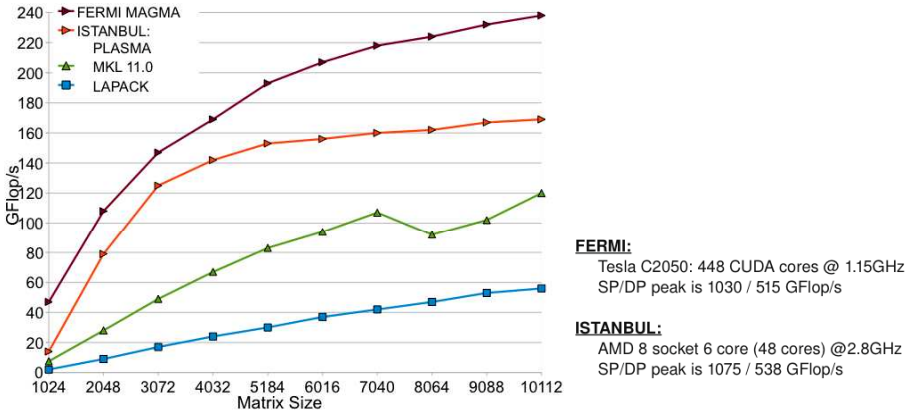


Fig. 5. Performance of LU factorization in double precision – compared is MAGMA on Fermi (C2050) *vs* MKL 11.0, PLASMA, and LAPACK on a 48 core system, having the same peak as a single Fermi GPU (C2050).

an improved GEMM algorithm for the Fermi GPUs, that significantly outperforms the currently available. Moreover, this new kernel opens the possibility for further improvements, e.g., based on auto-tuning. Also, this GEMM can be used directly or auto-tuned for developing other GEMM-based Level 3 BLAS.

In addition, we showed the effect of using the improved kernel on the performance of higher level algorithms, e.g., the one-sided factorizations. The old algorithms directly benefited from it, as evident from the performance results. For example, we showed MAGMA's LU running 63% faster than the CULA implementation. Compared to vendor libraries for multicore x86-based systems, the results are similar – MAGMA's LU on single Fermi can significantly outperform vendors' LU on high-end systems, such as the 48 core system in Figure 5. Further optimizations are possible, with one directions being tuning. For example, it is interesting to show that the performance can get up to 300 GFlop/s for smaller matrices.

A general conclusion is that DLA has become a better fit for the evolving GPU architectures, to the point where DLA can run more efficiently on GPUs than on current, high-end homogeneous multicore-based systems. This progress has been partially enabled by the added memory hierarchy in the GPUs, which in effect enabled the development of fast GEMM. The current implementation gets to achieve higher fraction of the peak, namely 58%, compared to the 40% on the previous generation of GPUs.

Acknowledgments. This work was supported by NVIDIA, Microsoft, the U.S. National Science Foundation, and the U.S. Department of Energy. We thank Everett Phillips and Massimiliano Fatica from NVIDIA for the useful discussions and optimization suggestions regarding the Fermi architecture.

References

1. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, http://developer.download.nvidia.com, 2007.
2. E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, *PLASMA users' guide*, http://icl.cs.utk.edu/plasma/, 11/2009.
3. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK user's guide*, SIAM, 1999, Third edition.
4. K. Fatahalian, J. Sugerman, and P. Hanrahan, *Understanding the efficiency of GPU algorithms for matrix-matrix multiplication*, HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (New York, NY, USA), ACM, 2004, pp. 133–137.
5. Y. Li, J. Dongarra, and S. Tomov, *A Note on Auto-tuning GEMM for GPUs*, ICCS '09 (Berlin, Heidelberg), Springer-Verlag, 2009, pp. 884–892.
6. R. Nath, S. Tomov, and J. Dongarra, *Accelerating GPU kernels for dense linear algebra*, Proc. of VECPAR'10, Berkeley, CA, June 22-25, 2010.
7. NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, http://www.nvidia.com/object/fermi_architecture.html, 2009.
8. ———, *NVIDIA CUDA C Programming Guide, version 3.1.1*, http://developer.nvidia.com/object/cuda_3_1_downloads.html, 7/21/2010.
9. S. Tomov, R. Nath, P. Du, and J. Dongarra, *MAGMA version 0.2 Users' Guide*, http://icl.cs.utk.edu/magma, 11/2009.
10. Vasily Volkov and James Demmel, *Benchmarking GPUs to tune dense linear algebra*, SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (Piscataway, NJ, USA), IEEE Press, 2008, pp. 1–11.
11. *AMD Core Math Library (ACML)*, www.amd.com/acml, ACML.
12. *CUDA CUBLAS Library*, http://developer.download.nvidia.com.
13. *GoToBLAS, Texas Advanced Computing Center*, http://www.tacc.utexas.edu/, GotoBLAS.
14. *IBM, Engineering and Scientific Subroutine Library (ESSL) and parallel ESSL.*, http://www-03.ibm.com/systems/p/software/essl/, ESSL.
15. *Math Kernel Library (MKL), Intel(R)*, http://www.intel.com/cd/software/products/asmo-na/eng.347757.htm, MKL.
16. R. Clinton Whaley, Antoine Petitet, and Jack Dongarra, *Automated Empirical Optimizations of Software and the ATLAS Project.*, Parallel Computing **27** (2001), no. 1-2, 3–35.
17. M. Wolfe, *Special-purpose hardware and algorithms for accelerating dense linear algebra*, http://www.hpcwire.com/features/33607434.html, HPC Wire, 10/2008.