Optimizing Memory-Bound Numerical Kernels on GPU Hardware Accelerators

Ahmad Abdelfattah[1], Jack Dongarra[2], David Keyes[1] and Hatem Ltaief[3]

[1] KAUST Division of Mathematical and Computer Sciences and Engineering, Thuwal, Saudi Arabia
[2] Innovative Computing Laboratory, University of Tennessee, Knoxville TN USA
[3] KAUST Supercomputing Laboratory, Thuwal, Saudi Arabia

Abstract. Hardware accelerators are becoming ubiquitous high performance scientific computing. They are capable of delivering an unprecedented level of concurrent execution contexts. High-level programming languages (e.g., CUDA), profiling tools (e.g., PAPI-CUDA, CUDA Profiler) are paramount to improve productivity, while effectively exploiting the underlying hardware. We present an optimized numerical kernel for computing the symmetric matrix-vector product on nVidia Fermi GPUs. Due to its inherent memory-bound nature, this kernel is very critical in the tridiagonalization of a symmetric dense matrix, which is a preprocessing step to calculate the eigenpairs. Using a novel design to address the irregular memory accesses by hiding latency and increasing bandwidth, our preliminary asymptotic results show 3.5x and 2.5x fold speedups over the similar CUBLAS 4.0 kernel, and 7-8% and 30% fold improvement over the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library in single and double precision arithmetics, respectively.

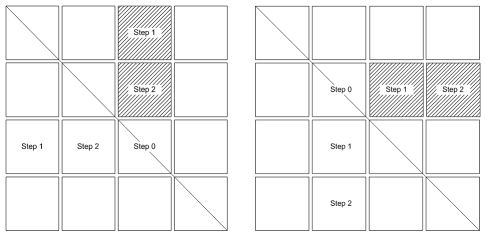
1 Introduction

GPUs have been, for a long time, dedicated for graphics processing. However, their increasing level of parallelism and computing capability have drawn attention in the HPC community, as low cost, low power, and high Gflop/s processing units. The latest architecture released by nVidia, codenamed Fermi, has a theoretical peak of 1 Tflop/s for single precision (SP), and about 500 Gflop/s for double precision (DP). Fermi has been highlighted as the first complete GPU computing architecture [5], with a complete memory hierarchy, ECC support, IEEE 754-2008 compliant floating point performance, and many novel features. Due to the drastic change from the previous GPU architecture, further tuning of existing numerical kernels is required to efficiently exploit new features in Fermi, in order to boost the performance.

One of the critical numerical kernels in dense linear algebra is the symmetric matrix-vector multiplication (SYMV). The kernel is, by nature, memory-bandwidth (BW) bound. It is a core step in computing the eigenpairs of a dense symmetric matrix. Having irregular memory access pattern due to the symmetric property of the matrix, the kernel design on GPUs is challenging. We present a novel design of the SYMV kernel. We try to exploit the new features introduced in Fermi. Most of the techniques used in this design target hiding memory latency and increasing memory bandwidth. When it comes to GPU programming for high performance, there are a lot of knobs to tune a kernel design. However, investigating all these knobs is daunting and time consuming. Therefore, we rely on performance counters to profile existing SYMV kernels in order to detect and identify weak points, where possible improvements can be made. PAPI CUDA Component [3] and the nVidia Compute Profiler [2] were the main performance counter tools used during the design process. The new kernel design is tested against two open-source SYMV kernels: the nVidia's CUBLAS 4.0 implementation and the Matrix Algebra on GPU and Multicore Architectures (MAGMA) 1.0.0-rc5 [1] implementation. MAGMA SYMV kernel [9] was tuned for Fermi. Our preliminary design is 3.5x better than CUBLAS 4.0 and 7-8% better than MAGMA in SP, while the speedup is about 2.5x over CUBLAS 4.0 and 1.3x over MAGMA in DP.

The rest of the paper is organized as follows. Section 2 discusses some previous work. Section 3 describes our proposed design in the SYMV kernel. Sections 4 and 5 present experimental and profiler results, respectively. Section 6 shows the impact of the new design on the overall symmetric eigenvalue problem. We summarize and propose some future work in Section 7.

2 Related Work



(a) MAGMA strategy (b) Proposed strategy

Fig. 1. Proposed computation strategy against MAGMA strategy. The vertical movement of thread blocks in (b) is more suitable for column major formats.

Accelerator-based hardware are nowadays employed in many HPC software libraries and applications, where they often outperform homogeneous x86 architecture in performance, power consumption, and cost-effectiveness. The STI Cell processor and GPUs have already been used in accelerating dense linear algebra ([7], [12] and [10]) as well as stencil computations [4].

An up-to-date highly tuned SYMV kernel was recently presented in [9]. The basic idea is to divide the matrix A into square blocks. Each Streaming Multiprocessor (SM) is responsible for one or more blocks. The kernel launches as many thread blocks as the number of diagonal matrix blocks. Each thread block is responsible for exactly one block-row. Figure 1(a) shows an example thread block movement. Each non-diagonal block is computed in two fashions: transposed and non-transposed. Partial results from transposed computations are written to global memory so that the correct thread blocks can consume them. The MAGMA implementation is, therefore, divided into two kernel calls. The first one does the computation. The second kernel is a final reduction step through global memory. Recursive blocking [9] was used to save shared memory usage in GPUs. In addition, pointer redirecting was adopted to handle matrix dimensions that are not multiples of the block dimension. The next section describes the design outlines of our proposed kernel and how it differs from the MAGMA kernel strategy.

3 Kernel Description

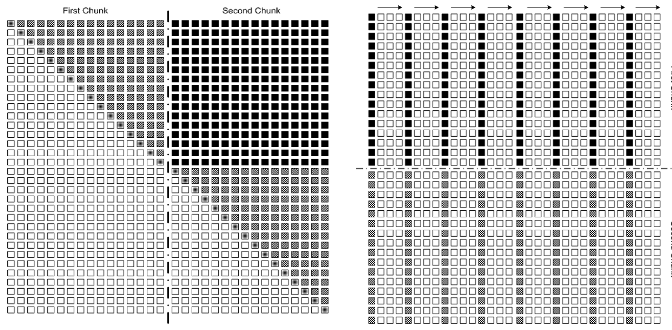
GPU kernels are conceptually designed following two main strategies. The first one (the *per-block* strategy) is how blocks are distributed over SMs, or in other words, how thread blocks travel throughout the matrix blocks. The second one (the *per-element* strategy) is how a single block is processed within the SM. The first strategy has to optimize memory accesses through global memory and L2 cache, while the second strategy should go deeper into the memory hierarchy i.e., registers and L1 cache/shared memory, to optimize the processing block elements through efficient use of the single SM's limited resources.

The new design has similar per-block strategy to the MAGMA kernel, with the exception it organizes memory accesses more efficiently. Moreover, as opposed to MAGMA, there are three successive kernel calls in the proposed design. The first kernel is a computation kernel for diagonal blocks only. The second one is a computation kernel for the non-diagonal blocks. The third kernel is a final reduction step done through global memory, which is very similar to the MAGMA kernel. The reason for separating the computation into three kernels will be shortly apparent. The proposed design divides the matrix into 64$\times$64 blocks. This is an auto-tuning result obtained from MAGMA's internal parameters. In the first kernel, we launch as many thread blocks as the number of the diagonal blocks. When a thread block finishes computation, the partial result (64 element-vector representing the block row of Y) is written into global memory. The second kernel has the same number of threads as the first kernel. Each thread block travels vertically through the matrix (Figure 1(b)). This is a more memory-friendly scheme compared to MAGMA, since blocks are fetched in compliance with the data layout (column-major format). This scheme achieves thus better profiling in terms of number of load instructions from global memory and L2 cache than MAGMA (see Section 5). Going at a lower level in the kernel design (the *per-element* strategy), each diagonal block computation produces a partial result, a 64-element vector. A non-diagonal block computation produces two 64-element vectors. We enumerate the new contributions in this strategy.

Separating Different Computation Patterns. Diagonal blocks have different per-element computation strategy than non-diagonal blocks. Therefore, they require different resources in terms of registers and shared memory. Separating different computation strategies into different kernels can achieve better occupancy for kernels that do not consume a lot of resources. This is the main reason why the diagonal block computation has been separated from non-diagonal block computation.

Data Prefetching. Data prefetching [6] arises almost everywhere in our design. Each block is divided into smaller pieces, which we refer to as *chunks*. A software pipeline is implemented to hide the memory latency by prefetching the next chunk of data, while a current chunk is being processed. This is a burden on the GPU memory resources, so organizing the work between threads has to be within the physical resource limit allowed per thread as well as per SM. Figures 2(a) and 2(b) describe how data prefetching is applied to diagonal and non-diagonal blocks, respectively. In the non-diagonal case, prefetching spans blocks; while processing the second chunk of a given block, the first chunk of the next matrix block is being prefetched.

Using More Registers. A very important feature of our kernel is that it completely avoids computing partial products in shared memory. Shared memory is used only in a final reduction step before a partial result of an entire block is written into global memory. This feature avoids paying penalty in terms of shared memory latency. It also reduces the occurrences of synchronization points. Using registers pays off very well, especially when register spilling to local memory does not happen. This is guaranteed on Fermi as long as each thread uses 63 registers or less.



(a) Diagonal computation. (b) Non-diagonal computation.

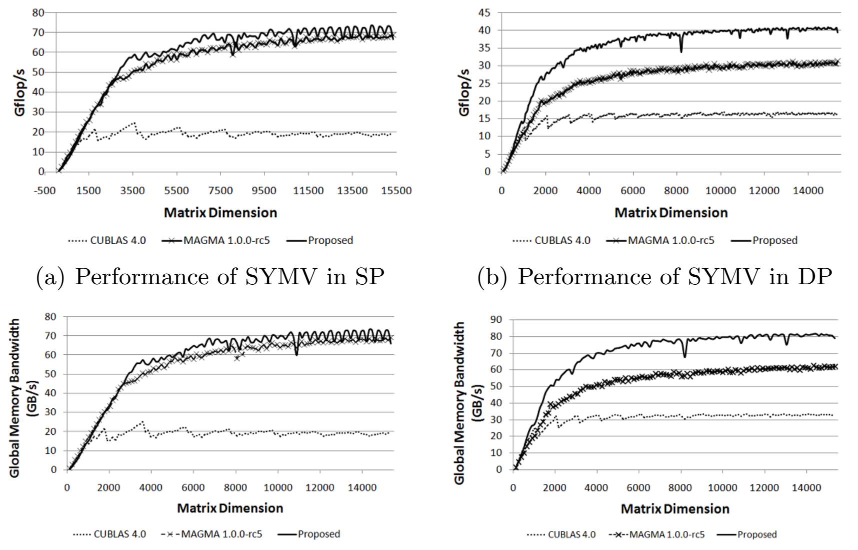
Fig. 2. Computation strategy inside a block. In (a), diagonal blocks are processed as two chunks. Hashed elements are loaded from DRAM then overwritten in a mirroring step. Black elements are not loaded at all from memory. Their values are loaded from shared memory during the mirroring step. In (b), non-diagonal blocks are also divided into two chunks. Threads are originated at the black elements. As threads move from left to right in the upper chunk, they prefetch hashed elements from the lower chunk in their registers.

4 Experimental Results

All experiments were executed on a single Fermi C2070 GPU, with 448 cores and 6 GB of DRAM. The kernel is implemented using CUDA C v4.0. The kernel is originally designed for matrices of dimensions that are multiples of 64. For other irregular dimensions, the matrix is padded with zeros inside the SM shared memory and registers. No padding is done in global memory. Figures 3(a) and 3(b) show the performance results (in Gflop/s) for SP and DP, respectively. The proposed design is far better than the CUBLAS 4.0 kernel. There are some dips in the SP performance, which we are trying to resolve. Ignoring such dips, there is a 7-8% improvement over MAGMA in SP. The performance gap widens in DP and reaches more than 30%. Although our kernel is mainly tuned for DP, the smaller improvement seen for SP against MAGMA is explained below along with the memory performance analysis. Since the kernel is memory bound, the reported performance numbers are far below the theoretical floating point peak performance. However, we can get intuition about the quality of the kernel design by translating Gflop/s into GB/s to see how close we are from the Fermi peak memory bandwidth. Fermi C2070 GPUs have theoretical peak memory bandwidth of 144 GB/s (with ECC turned on). However, the actual (sustainable) peak memory bandwidth is about 103 GB/s (when ECC is on). This information is obtained by running a CUDA implementation of the STREAM benchmark [8]. The memory bandwidth is calculated by dividing the amount of useful data loaded/stored from/into global memory by the total runtime of the kernel. For the SYMV kernel, and a matrix of dimension N, the total amount of useful data is from A, X, and Y, that is, $\frac{1}{2}N(N+1) + 2N$ elements, where each element consumes 4 bytes in SP and 8 bytes in DP. Figures 3(c) and 3(d) show the memory bandwidths of the SP and DP kernel versions. Our kernel scores about 70% (SP) and 80% (DP) of the actual peak memory bandwidth. This is 7-8% (SP) and 30% (DP) better than MAGMA, and 250% (SP) and 140% (DP) better than CUBLAS 4.0. It is interesting to see how the improvements in memory bandwidth matches those of performance. As previously mentioned, memory bandwidth improvement in SP is less than in DP. Running the same DP kernel for SP means saving more registers per thread, and loading less data each time. We thought that doubling the block size as well as the number of threads would result in memory bandwidth similar to the DP case. However, we were not able to double the number of threads in an SM because we are already using the maximum possible number on Fermi. A possible work around is to use 64-bit load instructions in SP instead of 32-bit instructions as in [11], which is, unfortunately, an assembly-level technique that is unavailable through the standard CUDA library.

5 Performance Analysis

In this section, we try to analyze the performance of the new kernel, by studying the performance counters obtained from the nVidia and PAPI-CUDA [3]

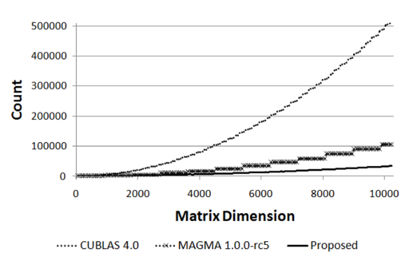


(a) Performance of SYMV in SP (b) Performance of SYMV in DP

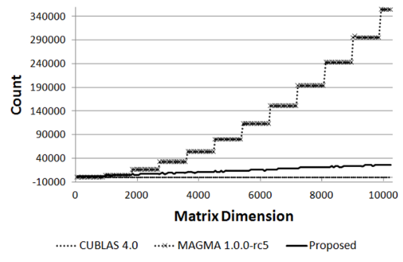
(c) Memory BW of SYMV in SP (d) Memory BW of SYMV in DP

Fig. 3. Performance of the SYMV kernel in SP and DP on Fermi C2070.

profilers. All three kernels were tested for matrix dimensions up to 10000. We selected the most relevant performance counters to the proposed kernel study. All results in this section are for the DP kernel. The first performance counter is the number of 64-bit load instructions made to the global memory. In general, going to global memory is a penalty, so the less we refer to global memory the better. Our experiments shows that the proposed design achieves 17% less instructions than CUBLAS, and 13% less instructions than MAGMA. Although the improvement is not significant, it could potentially have strong impact on performance, due to the huge penalty of going to global memory. In addition, shared memory has higher latency than registers. Since we minimize the usage on shared memory, Figures 4(a) and 4(b) show that we refer less to shared memory and thus, pay much less penalty in terms of bank conflicts. The burden is rather put on registers, which are faster to read and compute, and do not have restrictions of the load pattern. It is noteworthy to mention that CUBLAS does not encounter any bank conflicts, though being the slowest kernel. Two final performance counters are SM activity and registers-per-thread usage. Surprisingly, CUBLAS 4.0 took the lead for occupancy at 98.36%, followed by our design at 94.54%, and MAGMA at 80.90%. This result again shows it is indeed critical to consider all performance counters, when judging the kernel quality. A single performance metric can not reflect a comprehensive performance view. Regarding the registers-per-thread usage, CUBLAS 4.0 uses the least number of registers/thread i.e., 29, while MAGMA uses 51 and our kernel uses 63.



(a) Shared memory loads

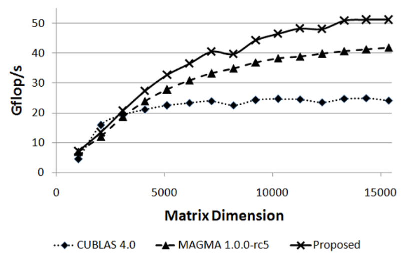


(b) L1 shared bank conflicts

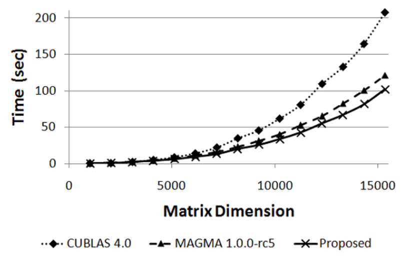
Fig. 4. Performance counters for shared memory on Fermi C2070.

6 Case Study: The Symmetric Eigenvalue Solver

The proposed DSYMV (in DP) was integrated into MAGMA, and a test was made for the tridiagonalization routine (DSYTRD) and the overall symmetric eigensolver (DSYEVD). We repeated the tests for MAGMA, and for CUBLAS. Results are shown in Figures 5(a) and 5(b). The new DSYTRD improves asymptotically by 88% with CUBLAS SYMV and by 20% with MAGMA SYMV. Looking at the overall symmetric eigensolver, the new DSYEVD is about 66% better with CUBLAS SYMV and about 17% better with MAGMA SYMV.



(a) DSYTRD Performance



(b) DSYEVD Runtime

Fig. 5. Impact of tuned SYMV kernels on DSYTRD and DSYEVD.

7 Summary and Future Work

This paper introduces an optimized kernel for computing the symmetric matrix-vector product on Fermi GPUs. The kernel achieves 3.5x (SP) and 2.5x (DP) fold speedups over CUBLAS 4.0, and 7-8% (SP) and 30% (DP) improvement over MAGMA, similarly to the memory bandwidth. One possible extension to the work presented in this paper is to consider the load imbalance in the per-block strategy. The vertical movement encounters different loads for thread blocks. We intend to apply a 1D block cyclic distribution of non-diagonal blocks. Non-diagonal blocks are to be mapped in a periodic manner over the available number of SMs (14 on Fermi C2070), as done in [7]. Although this scheme might not be friendly with respect to the column-major data layout, we expect that the load balance can compensate for this penalty, especially if a tile data layout within each block is considered.

Acknowledgements

We would like to thank Timothy Lanfear (nVidia), for providing the STREAM benchmark for CUDA, and Rajib Nath (UCSD) for his help in understanding MAGMA design outlines of the SYMV kernel.

References

1. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/magma/.
2. Nvidia visual profiler. http://developer.nvidia.com/nvidia-visual-profiler.
3. Performance Application Programming Interface (PAPI). Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/papi/.
4. K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-Point Stencil for Multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.
5. P. N. Glaskowsky. nVidia's Fermi: The first complete gpu computing architecture. Technical report, 2009.
6. D. Kirk and W. Mei Hwu. *Programming Massively Parallel Processors, A Hands-on Approach*. Morgan Kaufmann, 2010.
7. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, Sept. 2008.
8. J. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/.
9. R. Nath, S. Tomov, T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 6:1–6:10, New York, NY, USA, 2011. ACM.
10. R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for Dense Linear Algebra. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 83–92, Berlin, Heidelberg, 2011. Springer-Verlag.
11. G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, page 35, New York, NY, USA, 2011. ACM.
12. 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.