Systematic Approach in Optimizing Numerical Memory-Bound Kernels on GPU

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

[1] Division of Mathematical and Computer Sciences and Engineering
[2] Supercomputing Laboratory
King Abdullah University of Science and Technology
Thuwal, Saudi Arabia

Abstract. The use of GPUs has been very beneficial in accelerating dense linear algebra computational kernels (DLA). Many high performance numerical libraries like CUBLAS, MAGMA, and CULA provide BLAS and LAPACK implementations on GPUs as well as hybrid computations involving both, CPUs and GPUs. GPUs usually score better performance than CPUs for compute-bound operations, especially those characterized by a regular data access pattern. This paper highlights a systematic approach for efficiently implementing memory-bound DLA kernels on GPUs, by taking advantage of the underlying device's architecture (e.g., high throughput). This methodology proved to outperform existing state-of-the-art GPU implementations for the symmetric matrix-vector multiplication (SYMV), characterized by an irregular data access pattern, in a recent work (Abdelfattah et. al, VECPAR 2012). We propose to extend this methodology to the general matrix-vector multiplication (GEMV) kernel. The performance results show that our GEMV implementation achieves better performance for relatively small to medium matrix sizes, making it very influential in calculating the Hessenberg and bidiagonal reductions of general matrices (radar applications), which are the first step toward computing eigenvalues and singular values, respectively. Considering small and medium size matrices ($\leq$4500), our GEMV kernel achieves an average 60% improvement in single precision (SP) and an average 25% in double precision (DP) over existing open-source and commercial software solutions. These results improve reduction algorithms for both small and large matrices. The improved GEMV performances engender an averge 30% (SP) and 15% (DP) in Hessenberg reduction and up to 25% (SP) and 14% (DP) improvement for the bidiagonal reduction over the implementation provided by CUBLAS 5.0.

Keywords: Matrix-Vector Multiplication, GPU Optimizations, Memory-Bound Operations, Hessenberg Reduction, Bidiagonal Reduction.

1 Introduction

The high level of parallelism found on modern GPUs has attracted the scientific community to think about porting their legacy applications on GPUs. Since then, GPUs have been one of the favorite choices for accelerating many computational kernels, especially

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 207–216, 2013.
© Springer-Verlag Berlin Heidelberg 2013

when it comes to kernels with regular data access patterns. Considering only NVIDIA GPUs, they are usually equipped with hundreds of lightweight floating point (CUDA) cores, grouped into streaming multiprocessors (SMs). Starting from Fermi, each SM has a private parallel L1-cache/shared memory. All SMs share L2-cache and the global DRAM. Although the peak performance of modern GPUs has already reached the terascale, developers have to pay attention to several paramount parameters in order to achieve decent performance. For example, ensuring coalesced memory accesses, reducing shared bank conflicts, avoiding register spilling, and removing synchronization points are all performance optimizations that have direct impact on the overall GPU kernel quality. However, kernels that are memory-bound by nature rarely achieve performance close to the theoretical peak. In such kernels, the maximum achievable performance is usually limited by the global memory bandwidth.

This paper highlights a methodology for accessing dense matrices in numerical kernels, applied to the general matrix-vector multiplication (GEMV) kernel as a case study. This methodology was part of a previous work in accelerating the symmetric matrix-vector multiplication (SYMV) kernel, where significant improvement was achieved over state-of-the-art designs. Since both SYMV and GEMV are Level 2 BLAS operations, hence bounded by the bus memory throughput, we extended the same methodology to the GEMV kernel. The proposed way of accessing the matrix assumes that it is divided into square blocks. Each block is read from global memory using a simple double buffering technique combined with large occupancy to hide the latency of the stalled warps due to memory loads. Our experimental results against the latest CUBLAS 5.0 [6], CULA dense library [1] and MAGMA 1.2 [2] show improvements for small to medium matrix sizes, especially for the non-transposed case. The new design still maintains roughly the same performance for large matrix sizes. The GEMV enhancements obtained for these matrix sizes engender a significant speedup when plugged into the Hessenberg and bidiagonal reduction drivers, in which the GEMV variants (transpose and non-transpose) represent the fundamental operations. Both reductions correspond to the first step toward calculating the eigenvalues and singular values. Noteworthy to mention that radar detection applications [11,15] require the computation of several independent small eigenvalue and singular value decomposition problems.

The reminder of the paper is organized as follows: Section 2 presents some related work in optimizing numerical linear algebra for GPUs. In Section 3, we give some information about the work that has been done for the SYMV kernel [7]. Its potential extension to the GEMV kernel as well as the GEMV kernel implementation details are described in Section 4, while the results are shown in Section 5. We give our conclusion and provide suggestions for future work in Section 6.

2 Related Work

There is a lot of work done for accelerating dense linear algebra kernels on GPUs using CUDA. This is very critical as many applications rely on these basic block kernels to extract their actual performance. The latest CUBLAS 5.0 [6] is often the reference implementation for NVIDIA GPU BLAS. MAGMA [2] provides a very optimized subset of BLAS functions (MAGMABLAS) for GPUs and a hybrid implementation

(using both, the host and the device) of the major LAPACK routines. Indeed, it offers some improved implementations over CUBLAS, such as the SYMV [12] kernel and the GEMM kernel [13]. CUBLAS and MAGMA are freely available for public downloads. CULA Dense [1] is a commercial package and represents a set of dense linear algebra libraries developed for GPUs. It is also written in CUDA, and further improves CUBLAS implementations [8]. In this paper, we compare our results against the three aforementioned libraries i.e., CUBLAS, MAGMA, and CULA.

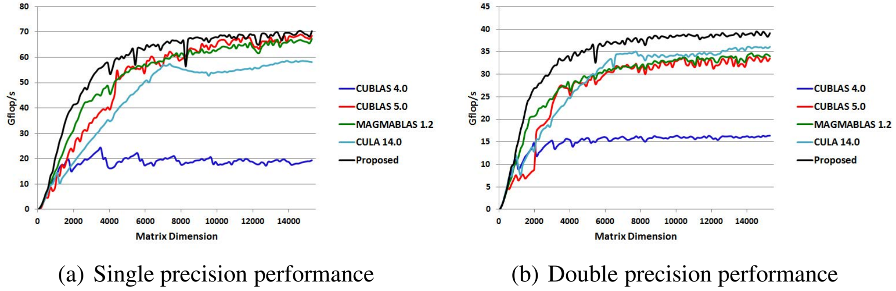
On the other hand, the evolving architecture of modern GPUs pushed the need for exposing tunable parameters or hooks in the kernel implementations so that productivity is maintained. Volkov and Demmel [14] proposed an early benchmarking scheme to tune DLA routines, such as matrix-matrix multiplication. Kurzak et. al [9] developed an autotuning framework called ASTRA, originally designed over Fermi GPUs for tuning also GEMM kernels and its variants. Autotuning using ASTRA has been recently extended to Kepler [10], the newest NVIDIA GPU architecture to date. Such frameworks can be applied to our SYMV and GEMV kernels, in order to benchmark our kernel design on future architectures, without starting from scratch.

3 SYMV Background

Our methodology of processing matrix blocks has been first introduced in [7], where a new SYMV kernel was proposed. The new SYMV kernel is 2.5x faster than CUBLAS 4.0 and 1.3x better than MAGMABLAS. The kernel design was described following two strategies: the *block-level strategy*, governing movements of thread blocks within the matrix, and the *thread-level strategy*, governing thread mapping within a single matrix block. The idea here is to apply both strategies to the GEMV kernel. However, the block-level strategy of the SYMV kernel necessitates an adjustment, since only half of the matrix needs to be processed for the symmetric case. In GEMV, the whole non-symmetric matrix requires to be scanned. Both block-level and thread-level strategies for GEMV are explained in Section 4. Our results [7] showed a direct impact on tridiagonalisation and symmetric eigenvalue computations.

In general, how a better strategy and optimizations can be found? The answer is to have insight through a profiling or tracing tool. From our experience in developing the SYMV kernel [7], the NVIDIA visual profiler [4] along with PAPI [5] gave us rich information about register usage, L1-cache, shared memory, and the DRAM. For example, during development, we detected register spilling through the number of local memory accesses along with the number of 32-bit registers used per thread. Shared memory bank conflicts can help the developer changing the strategy of accessing shared memory. In memory bounds kernels similar to SYMV and GEMV, the matrix should be loaded only once due to the absence of data reuse of matrix elements. The number of global memory loads help judge if coalesced access is satisfied. Generally, profiling CUDA kernels helps the programmer focus on specific parts of the kernel, identified as potential bottlenecks, in order to enhance its behavior.

We show an updated performance chart of our SYMV kernel, for single and double precisions in Figure 3. There are no changes in the design of our original kernel except a slight adjustment to better handle irregular matrix dimensions (dimensions that are



(a) Single precision performance (b) Double precision performance

Fig. 1. Performance of SYMV kernel using five different implementations

not multiple of the internal blocking size i.e., 64). However, what is new about the Figure 3 is that it includes performance curves for two other kernel implementations. The first one is the CUBLAS 5.0 kernel, which comes with the newly released CUDA 5.0. We noticed a significant improvement over the old SYMV from CUDA 4.0. However, the improved kernel can be only used by enabling the atomics mode [3]. The second implementation with which we compare ours is the SYMV kernel from CULA [8], a commercial dense linear algebra library for NVIDIA GPUs. Results show that our proposed kernel is still better than all existing implementations.

4 Extending Methodology to GEMV

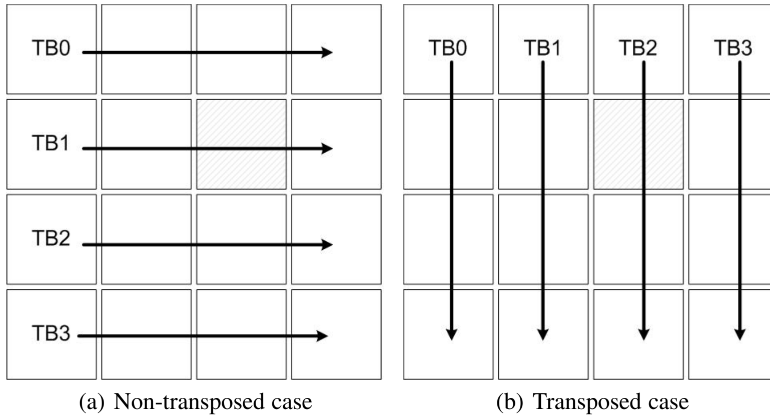
In this Section, we present the details of the kernel design in the same manner as we did in [7]. First, we begin by the block-level strategy, then we illustrate the thread-level strategy, which we recommend for similar memory-bound kernels.

4.1 Block-Level Strategy

The block-level strategy is simple and straightforward. The input matrix is divided into square blocks. The dimension of the block is a design parameter called `gemv_block_size` and is currently set to 64. This value proved to give the best performance for SYMV [7]. Let us assume for now, that the matrix can be completely divided into 64×64 square blocks. We will show how to manage arbitrary dimensions later on. The kernel is configured to run with `n` thread blocks (TBs), where:

$$\mathtt{n} = \mathtt{number_of_rows}/\mathtt{gemv_block_size} \qquad \text{for the non-transposed case, or}$$
$$\mathtt{n} = \mathtt{number_of_columns}/\mathtt{gemv_block_size} \qquad \text{for the transposed case.}$$

As shown in Figure 2, thread blocks originate at the left most blocks in the non-transposed case (Figure 2(a)) and move horizontally from left to right. In the transposed version (Figure 2(b)), they originate at the top blocks and move downwards. The kernel terminates when all TBs finish their horizontal/vertical pass over the matrix.



(a) Non-transposed case (b) Transposed case

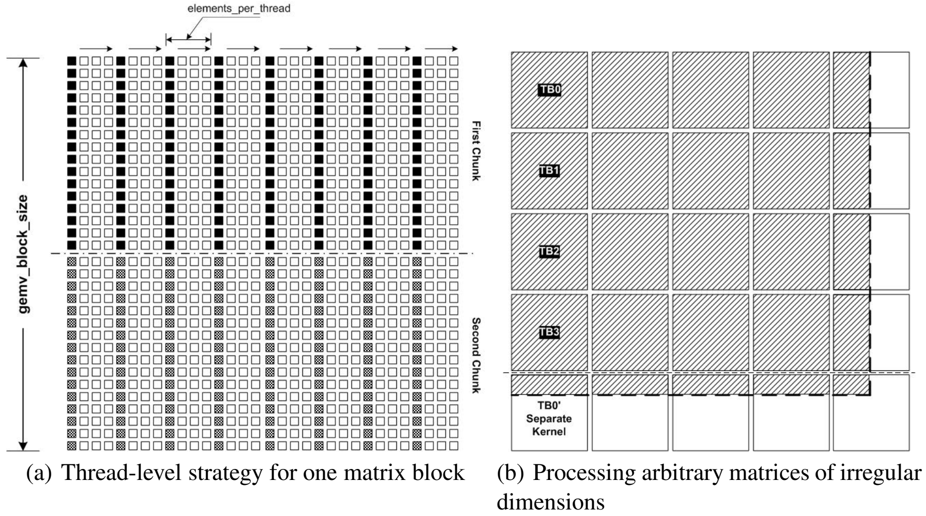
Fig. 2. Movements of thread blocks through the matrix

4.2 Thread-Level Strategy

The thread-level strategy is our main concern. To ease the presentation, we summarize the following design parameters:

- `thread_x` is the x-dimension of each thread block. `thread_x` is always set to `gemv_block_size`.
- `thread_y` is the y-dimension of each thread block.
- `elements_per_thread` is the number of matrix elements a single thread prefetches at one time. `elements_per_thread` is not really a design parameter, and is equal to `gemv_block_size`/(2×`thread_y`). However, it puts a restriction of the choice of `thread_y` since it has to be an integer.

Now, let us consider Figure 3(a) as an example. A 32×32 matrix block is processed using 128 threads. Originally, thread blocks are configured as 32×4 threads, meaning that `thread_x`=32, `thread_y`=4, and `elements_per_thread`=4. However, during actual processing, the whole matrix block is subdivided into two chunks, and all threads cooperate in processing each chunk. The reason behind this concept is to introduce more latency hiding for the stalled warps by applying a double buffering scheme. Thread blocks reorganize themselves as 16×8 threads while loading and processing each chunk from memory. Each thread is responsible for processing a number of elements equal to (2×`elements_per_threads`). For the example shown in Figure 3(a), each thread loads four elements from the first chunk. Before processing the first chunk, double buffering comes into play to prefetch the second chunk. Since SMs interleave execution of thread warps to hide the latency of stalled threads, increasing parallelism would lead to more latency hiding. However, we are always constrained by the SM resources of threads and registers (since all loaded elements are prefetched into registers). Similarly, before processing the second chunk, the first chunk of the next matrix block is prefetched. For the non-transposed case, each thread is responsible for the result of two elements in the final resulting vector. In the transposed case, the matrix block is processed in the same manner, but each thread is responsible for the result of `elements_per_thread` elements.



(a) Thread-level strategy for one matrix block (b) Processing arbitrary matrices of irregular dimensions

Fig. 3. Optimization Techniques

The drawback of this methodology is its intensive register usage, which may lead to low occupancy. Therefore, the tunable parameters introduced above should be carefully chosen depending on the underlying architecture. In particular, on Fermi C2070, thread blocks are configured as 64×8 threads.

One final comment on the proposed methodology is that the final result of an element in the final vector is not owned by one threads. For the non-transposed case, threads having the same `threadIdx.x` share partial results of the same element. For the transposed version, it is `threadIdx.y` instead. In both cases, threads use shared memory to compute the final result through a simple reduction operation.

4.3 Handling Arbitrary Dimension

Consider an arbitrary matrix where rows and columns are not necessarily multiple of `gemv_block_size`. Moreover, the matrix can be rectangular, i.e. rows are not equal to columns. In the presence of clean-up regions, the design is not so simple as in the trivial case shown in Figure 2.

We will consider a non-transposed case shown in Figure 3(b). The other case can be easily understood in a similar way. For the block-level strategy, we separate the processing of the bottom row-of-blocks in an another kernel that is always invoked with one thread block (TB0' in Figure 3(b)). This is because the TB0' will have some branches that are not necessary for TB0 through TB3. And as we recommended in [7], separating different computation strategy may lead to better occupancy. So, we preferred to separate TB0' as it surely consumes more SM resources due to the additional conditionals. To avoid executing the two kernels successively, we benefit from CUDA streams [3] so that the two kernels are concurrently executed. TB0 through TB3 march on from left to right processing blocks without any special treatment until they hit rightmost block,

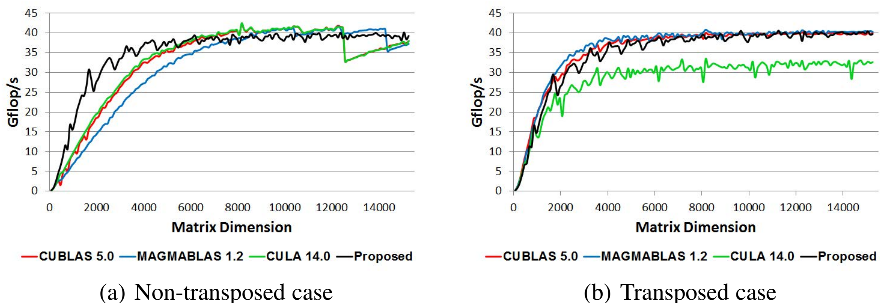
where some threads positioned after the matrix columns will be inactive. For TB0', it marches on similarly, but takes into account that threads with global row index more than the matrix rows do nothing. When it hits the rightmost block, it encounters the same situation of TB0 through TB3. Inactive threads in an irregular block are programmed to load zero instead of a matrix element. We can say that the matrix is virtually padded with zeros inside registers. The computation takes place similar to a regular block. The engendered extra flops are negligible compared to the overall algorithm complexity.

5 Experimental Results

The results shown in this section were obtained on a single Fermi C2070 GPU card, featuring 448 cores and 6 GB of DRAM. The host machine has a dual-socket quad-core Intel Xeon processor, running at 2.67 GHz, and is equipped with 24 GB of main memory. All kernel implementations were compiled using CUDA 5.0 compiler, except CULA Dense R14, which only supports CUDA driver/runtime version 4.1. The results, for both single precision (SP) and double precision (DP), will be shown in two parts. The first parts presents results for the Level 2 BLAS routine GEMV. Both transposed and non-transposed cases are covered. The second part shows the results of two LAPACK routines: The bidiagonal reduction (BRD) and the Hessenberg reduction (HRD). Because it is called at each column/row reduction of the matrix, the GEMV operation represent the main bottleneck. The presented graphs shows the impact of improving GEMV on these routines.

5.1 GEMV Performance

Figures 4 and 5 show the performance in Gflop/s for SGEMV and DGEMV, respectively. The improvement achieved in our design comes for relatively small to medium matrix sizes in the non-transposed case, especially for SP. For the non-transposed SP case, the kernels from CUDA 5.0 and CULA Dense R14 are almost identical. Our kernel has an average 80% improvement for matrix sizes below 1700. From 1700 and up to 4300, it drops to 30%. Between 6000 and 12000, our design drops to about 94%

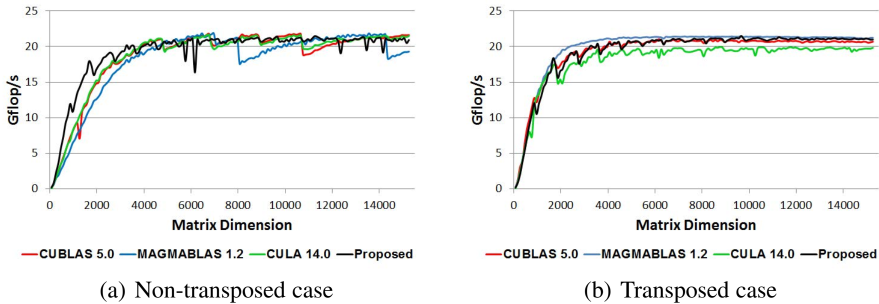


(a) Non-transposed case (b) Transposed case

Fig. 4. Performance of GEMV (SP)

of the other designs. A sharp degradation in the performance of the other implementations, starting from 12500, give our kernel an average 10% improvement (from 12500 to 15200).

Doing a similar analysis for double precision, we notice an average 47% improvement up to dimension 1700, then an average 10% improvement between 1700 and 3500. Afterwards, our design has nearly the same performance as the other kernels. As will be shown in Section 5.2, the improvement in the performance for relatively small matrices improves the LAPACK routines for both small and large matrices (again in a relative sense). Our intuition is that the thread-level strategy is able to increase the amount of parallel work even for small matrices. For example, a quick look at the MAGMABLAS GEMV source code informed us that each thread block uses either one 64 or 128 thread vector, compared to 4×64 thread vectors in our case. For large matrices, we probably lose little bit of performance because of the parallel reduction through shared memory.



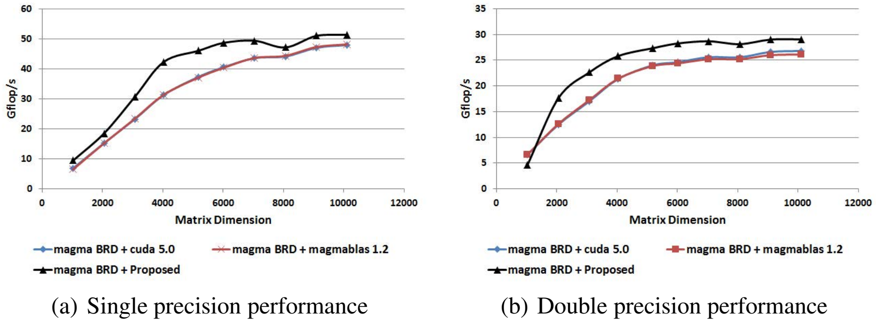
(a) Non-transposed case (b) Transposed case

Fig. 5. Performance of GEMV (DP)

5.2 BRD and HRD Performance

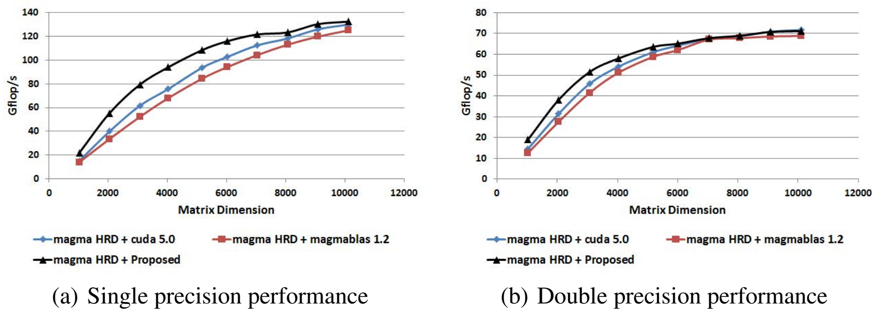
In this part, we show the performance for BRD and HRD algorithms, when using the GEMV kernels from CUBLAS 5.0, MAGMABLAS 1.2, and our proposed kernel. The reduction drivers for BRD and HRD are taken from MAGMA. Although CULA has implemented such algorithms, they are not available in the CULA-DENSE free edition.

We see in Figures 6 and 7 that the reduction algorithms perform better when using our GEMV kernel. It beats all other implementations, thanks to our improvements in performance for small matrices. For BRD (SP), the reduction performs 25% better (on average) for dimensions $\leq$ 8000, then drops to about 7% improvement. In DP, the average improvement is about 20% for dimensions $\leq$ 8000, then drops to 12%. Switching to the HRD algorithm, we achieve an average 35% improvement in SP for dimensions $\leq$ 8000, that drops to 7% afterwards. In DP, the improvement is 17% till 6000 then drops to 2%. As we can see, although our GEMV kernel is not competitive for every matrix size, it could lead to a high-level LAPACK driver that is better for every matrix size. The obvious reason is that reduction operations do successive matrix-vector multiplications of decreasing sizes. The conceptual lesson that we learned is this: slight improvements to low-level kernels can not be ignored. There are plenty of higher-level routines where



(a) Single precision performance (b) Double precision performance

Fig. 6. Performance of bidiagonal reduction using three different GEMV kernels



(a) Single precision performance (b) Double precision performance

Fig. 7. Performance of Hessenberg reduction using three different GEMV kernels

these kernels lie at their hearts. Such routines can move to another level of performance by slightly enhancing its core kernel.

6 Conclusions and Future Work

This paper introduced an improved implementation of the GEMV kernel on NVIDIA GPU accelerators. The design idea was utilized before in improving the SYMV kernel, which supports our claim that the computation strategy can be regarded as a general approach for processing memory-bound kernels. Although the results shows improvements only for relatively small to medium matrix sizes, the impact on higher-level LAPACK reduction algorithms was significant, even on large matrix sizes.

Possible future plans for this work is to treat the periodic dips in the performance of our GEMV kernel. These dips refer to worst case mapping of TBs on SMs. Since the work load is balanced among all thread blocks, a dip occurs when (`number_of_TBs` mod `number_of_SMs`) is relatively small (worst case will be 1). The apparent solution is to involve more workload distribution to maximize the number of active SMs. However, dividing the work of one TB among several SMs requires either reduction operations on the DRAM level. Such trade-off between workload distribution and DRAM access penalty requires further investigation.

References

1. CULA Dense Free Edition, http://www.culatools.com/
2. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee, http://icl.cs.utk.edu/magma/
3. NVIDIA CUDA Toolkit, http://developer.nvidia.com/cuda-toolkit
4. Nvidia visual profiler, http://developer.nvidia.com/nvidia-visual-profiler
5. Performance Application Programming Interface (PAPI). Innovative Computing Laboratory, University of Tennessee, http://icl.cs.utk.edu/papi/
6. The NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS), http://developer.nvidia.com/cublas
7. Abdelfattah, A., Dongarra, J., Keyes, D., Ltaief, H.: Optimizing Memory-Bound SYMV Kernel on GPU Hardware Accelerators. In: The 10th International Meeting on High Performance Computing for Computational Science, VECPAR 2012 (accepted, 2012)
8. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J.: CULA: Hybrid GPU Accelerated Linear Algebra Routines. In: Proceedings of SPIE Defense and Security Symposium, DSS (April 2010)
9. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM Kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems PP(99), 1 (2012)
10. Kurzak, J., Luszczek, P., Tomov, S., Dongarra, J.: Preliminary Results of Autotuning GEMM Kernels for the NVIDIA Kepler Architecture - GeForce GTX 680. LAPACK Working Note 267
11. Kwon, Y., Narayanan, R.M., Rangaswamy, M.: A multi-target detector using mutual information for noise radar systems in low snr regimes. In: 2010 International Waveform Diversity and Design Conference, WDD, pp. 000105–000109 (August 2010)
12. Nath, R., Tomov, S., Dong, T., Dongarra, J.: Optimizing symmetric dense matrix-vector multiplication on GPUs. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 6:1–6:10. ACM, New York (2011)
13. Nath, R., Tomov, S., Dongarra, J.: An Improved Magma Gemm for Fermi Graphics Processing Units. Int. J. High Perform. Comput. Appl. 24(4), 511–515 (2010)
14. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 31:1–31:11. IEEE Press, Piscataway (2008)
15. Yu, W.C., Quan, W.D.: On the signal processing in the life-detection radar using an fmcw waveform. In: 2010 Third International Symposium on Information Processing, ISIP, pp. 213–216 (October 2010)