Accelerating GPU kernels for dense linear algebra[⋆]

Rajib Nath, Stanimire Tomov, and Jack Dongarra [⋆⋆]

Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville
{rnath1, tomov, dongarra}@eecs.utk.edu

Abstract. Implementations of the Basic Linear Algebra Subprograms (BLAS) interface are major building block of dense linear algebra (DLA) libraries, and therefore have to be highly optimized. We present some techniques and implementations that significantly accelerate the corresponding routines from currently available libraries for GPUs. In particular, *Pointer Redirecting* – a set of GPU specific optimization techniques – allows us to easily remove performance oscillations associated with problem dimensions not divisible by fixed blocking sizes. For example, applied to the matrix-matrix multiplication routines, depending on the hardware configuration and routine parameters, this can lead to two times faster algorithms. Similarly, the matrix-vector multiplication can be accelerated more than two times in both single and double precision arithmetic. Additionally, GPU specific acceleration techniques are applied to develop new kernels (e.g. syrk, symv) that are up to 20× faster than the currently available kernels. We present these kernels and also show their acceleration effect to higher level dense linear algebra routines. The accelerated kernels are now freely available through the MAGMA BLAS library.
Keywords: BLAS, GEMM , GPUs.

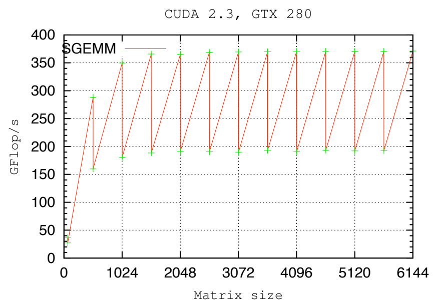
1 Introduction

Implementations of the BLAS interface are major building block of dense linear algebra libraries, and therefore have to be highly optimized. This is true for GPU computing as well, especially after the introduction of shared memory in modern GPUs. This is important because it enabled fast Level 3 BLAS implementations for GPUs [2, 1, 4], which in turn made possible the development of DLA for GPUs to be based on BLAS for GPUs [1, 3]. Earlier attempts (before the introductions of shared memory) could not rely on memory reuse, only on the GPU's high bandwidth, and as a result were slower than the corresponding CPU implementations.

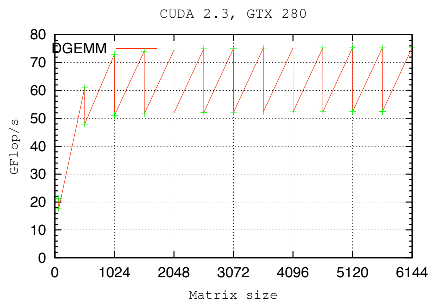
Despite the current success in developing highly optimized BLAS for GPUs [2, 1, 4], the area is still new and presents numerous cases/opportunities for improvements. This paper addresses several very important kernels, namely the

⋆ Candidate to the Best Student Paper Award

⋆⋆ Research reported here was partially supported by the National Science Foundation, NVIDIA, and Microsoft Research.



(a) Single Precision



(b) Double Precision

Fig. 1. GEMM Performance on Square Matrices.

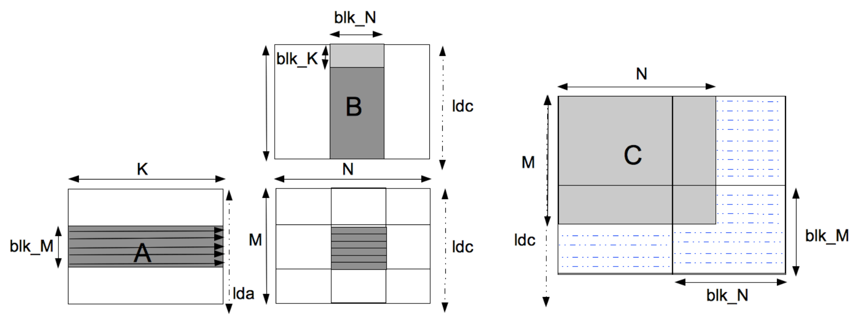
matrix-matrix multiplication that are crucial for the performance throughout DLA, and matrix-vector multiplication that are crucial for the performance of linear solvers and two-sided matrix factorizations (and hence eigen-solvers). The new implementations are included in the recently released and freely available *Matrix Algebra for GPU and Multicore Architectures* (MAGMA) version 0.2 BLAS Library [3].

The rest of the paper is organized as follows. Section 2 gives some performance results of current kernels and points out our optimization targets. Section 3 presents the *Pointer Redirecting* techniques and their use to accelerate the xAXPY, xGEMV, and xGEMM routines. Section 4 summarizes the results on accelerating selected MAGMA BLAS kernels. Next, in Section 5 we give the performance results for the new kernels. Finally, Section 6 summarizes this work and describes on-going efforts.

2 Performance of Current BLAS for GPUs

One current BLAS library for GPUs is NVIDIA's CUBLAS [2]. Figure 1(a) shows the performance of the single precision matrix-matrix multiplication routine (SGEMM) for a discrete set of matrix dimensions. Figure 1(b) shows similar data but for double precision arithmetic. Note that at some dimensions the performance is much higher than at other dimensions, e.g. taken at odd numbers like 65, 129, etc. These performance dips, that actually happen in the majority of matrix dimensions are one of our acceleration targets. The reason for these dips is very likely related to an implementation that has even inner-blocking size to match various hardware parameters and considerations to get high performance. The performance graphs illustrate a quite high performance loss for the cases when the matrix dimension is obviously not a multiple of the inner blocking size. In particular, the performance gap is more than 24 GFlops/s in double precision (around.34% of the peak performance), and is worse for single precision.

There are ways around to work with these BLAS routines and still get high performance in high level algorithms. One possible solution is to force the user



(a) GEMM for GPUs (b) Acceleration target

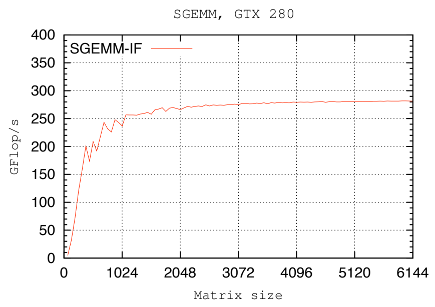
Fig. 2. The algorithmic view of GEMM for GPUs.

to allocate and work with matrices multiple of the blocking size. This though leads to memory waste. Sometimes it is a burden to the user if the application is already written, and in general is obviously not a good solution. Another solution is padding with 0s to fit the blocking factor, do the computation and keep this transparent to the user. This approach has the overhead of copying data back and forth, and possibly some extra computation. A third approach is to rewrite the kernels in such a way that there are no extra computations, no data movement or any other overheads. This rewriting though is difficult and time consuming, especially taken into account different GPU specifics as related to data coalescing, data parallel computation, computation symmetry, and memory bank layout.

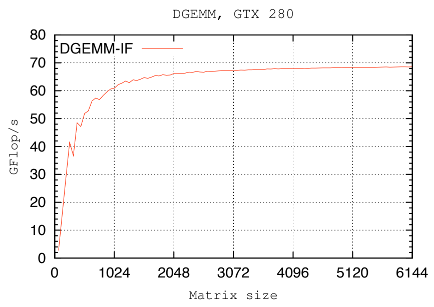
3 Pointer Redirecting

The matrix-matrix multiplication (xGEMM; e.g. $C = AB$) algorithm for GPUs is schematically represented in Figure 2(a). Matrix C is divided into blocks of size $blk_M \times blk_N$ and each block is assigned to a block of $nthd_x \times nthd_y$ threads. Each thread inside a thread block computes a row of sub matrix $blk_M \times blk_N$. Each thread accesses corresponding row of matrix A as shown by an arrow and uses the sub-matrix $K \times blk_N$ of matrix B for computing the final result. As the portion of matrix B needed by each thread inside a thread block is the same, they load a sub-matrix of matrix B of size $blk_N \times blk_K$ from global memory to shared memory in a coalesced way, synchronize themselves, do the computation and repeat until the computation is over. All these happen in a series of synchronized steps. With an optimal selection of $blk_M, blk_N, blk_K, nthd_X, nthd_Y$, we can get the best kernel for the matrix sizes that are divisible by blocking factors, i.e. $M\%blk_M = 0,\ N\%blk_N = 0,\ K\%blk_K = 0$.

The question is how to deal with matrix dimensions that are not divisible by the blocking factor. Whatever solution we choose, we have to keep it transparent to the user while maintaining highest flexibility. The goal is to allow reasonable



(a) Single Precision



(b) Double Precision

Fig. 3. GEMM Implementation with Conditional Statement in Inner Loop.

overhead (if needed) and to achieve high performance in general cases. We show in Figure 2(b) matrix C of a xGEMM operation ($C = \alpha C + \beta Op(A)Op(B)$) where dimensions M and N are not divisible by the blocking factor. The matrix has only one full block. We can do the computation for the full block and do the other partial blocks by loading data and doing computation selectively. This will introduce several if-else statements in the kernel which will prevent the threads inside a thread-block to run in parallel. Figure 3 shows the performance of one such implementation. Note that GPUs run all the threads inside a thread block in parallel as long as they execute the same instruction on different data. If the threads ever execute different instruction, their processing would become temporary sequential until they start executing the same instructions again.

Another approach is to let the unnecessary threads do similar work so that the whole thread block can run in data parallel mode. In Figure 2(b) the dashed blue lines correspond to unnecessary flops that are done by respective thread. It is not clear yet which data they will operate on, but it also does not matter because the computation will be discarded. Lets take a look at the scenario where all the threads assume that the matrix fits into the block and do the work in a natural way until updating matrix C. In Figure 4, the shaded region corresponds to original matrix and the outmost rectangle corresponds to the largest matrix that best fits in terms of blocking factor. We are going to make $\lceil \frac{M}{dim_M} \rceil \times \lceil \frac{N}{dim_N} \rceil$ number of grids and allow threads at the partial block to compute the same way as it is done in a full block. It is evident that memory accesses inside the shaded region in Figure 4, denoted by white diamond, are always valid. Memory accesses denoted by red diamonds are always invalid. Memory accesses represented by green diamond could be valid or illegal. As we can see in the Figure 4, the leftmost green diamond could be an element from the next column, e.g. when $lda \leqq blk_M \times \lceil \frac{M}{blk_M} \rceil$. It could be an element in the same column when $lda > blk_M \times \lceil \frac{M}{blk_M} \rceil$, or it could be invalid memory reference.

In Figure 5(Left), the blue lines in last row and last column are last valid memory reference irrespective of any values of lda, M, N, K, blk_M, blk_N, $nthd_X$, $nthd_Y$. If some thread needs to access some memory location beyond this last

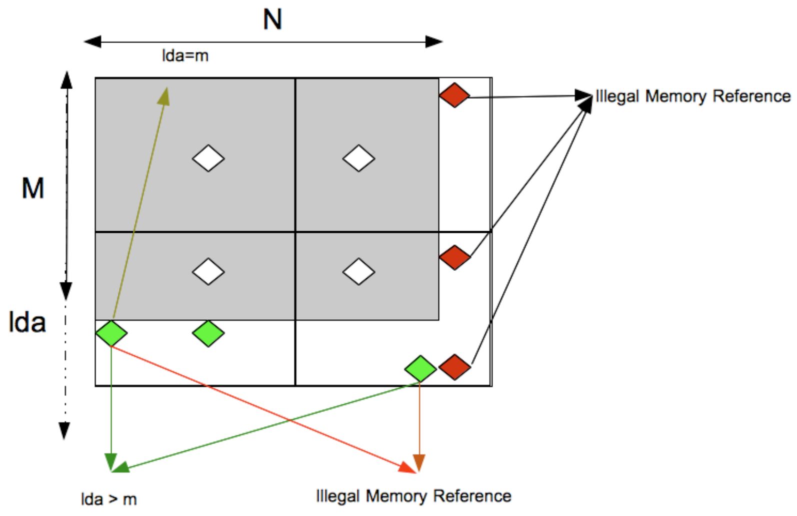


Fig. 4. Possible Illegal Memory Reference in Matrix Multiply.

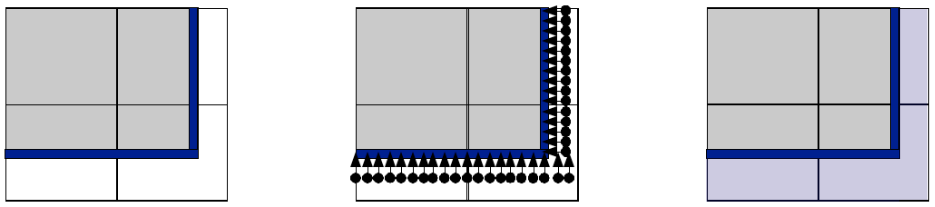
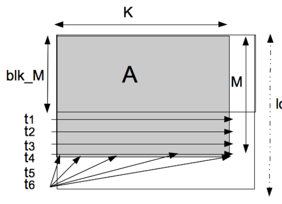


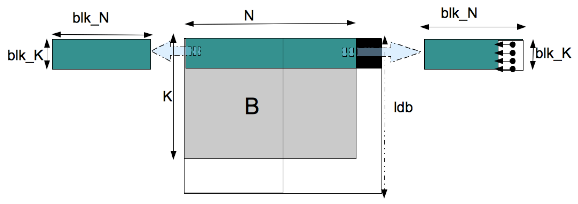
Fig. 5. (Left) Last Valid Access (Middle) Pointer Redirecting (Right) Mirroring

row/column, we are going to force him reference to this last row/column by adjusting the pointer. These threads will be doing unnecessary computation, we don't care from where this data is coming from. All we care is that together they make best use of memory bandwidth and layout, access data in a coalesced manner. Figure 5(Middle) depicts the complete scenario how the memory is referenced. As a result the matrix will have some virtual row where rows beyond the last row are replication of last row and columns beyond the last column are replication of last column. It is shown in Figure 5.

Let's see how it fits into xGEMM's(Op(A) = Op(B) =Non-Transposed) context in terms of accessing matrix A. As in Figure 6(a), thread t1, t2, t3, t4 will be accessing valid memory location. And all the threads beyond thread t4, e.g. thread t5, t6 will be accessing same memory thread t4 is accessing. As a result no separate memory read operation will be issued and no latency will be experienced for this extra load. If we look at Figure 6(b), $blk_K \times blk_N$ data of matrix



(a) Accessing Matrix A



(b) Accessing Matrix B

Fig. 6. Algorithmic view of GEMM for GPUs with Pointer Redirecting.

B are brought into shared memory by $nthd_X \times nthd_Y$ threads in a coalesced manner. The left $blk_K \times blk_N$ block is necessary as we can see. But the right $blk_K \times blk_N$ is partially needed. The black portions are unnecessary memory access. As discussed before, it will access the last row or column that is needed instead of accessing invalid memory. This will still be done in a coalesced way and it is accessing less memory now. Some memory are accessed more than once, which doesn't hamper performance. This a simple solution to the problem with little overhead that doesn't break the pattern of coalesced memory access. Note that we will not be doing any extra computation in K dimension, so we don't need to zeroing out values to keep the computation valid.

4 MAGMA BLAS kernels

MAGMA BLAS includes a subset of CUDA BLAS that are crucial for the performance of MAGMA routines. The pointer redirecting technique were applied to most of the kernels. Here we mention a few of the new kernels and their use in high level MAGMA routines.

xGEMM: Various kernels were developed as an extension to the approach previously presented in [4]. The extensions include more parameters to explore xGEMM's design space to find best performing versions in an auto-tuning approach. The new algorithms are of extreme importance for both one-sided and two-sided matrix factorizations as they are in general based on xGEMMs involving rectangular matrices, and these are the cases that we managed to accelerate most significantly.

xGEMV: Similarly to xGEMM, various implementations were developed and parametrized to prepare them for auto-tuning based acceleration. Different implementations are performing best in different settings. xGEMVs are currently used in MAGMA's mixed-precision iterative refinement solvers and the Hessenberg reduction algorithm.

xSYMV: Similarly to xGEMM and xGEMV, various implementations were developed. xSYMV is used similarly to when xGEMV is used with the difference when symmetric matrices are involved. This is again the mixed-precision iterative refinement solvers and the reduction to three diagonal form.

xTRSM: Algorithms that trade off parallelism and numerical stability, especially in algorithms related to triangular solvers, have been known and studied before, but now are getting extremely relevant with the emerging highly parallel architectures, like the GPUs. We use an approach where diagonal blocks of the matrix are explicitly inverted and used in a block algorithm. Multiple kernels, including kernels where the inverses are computed on the CPU or GPU, with various block sizes (e.g., recursively increasing it from 32), are developed.

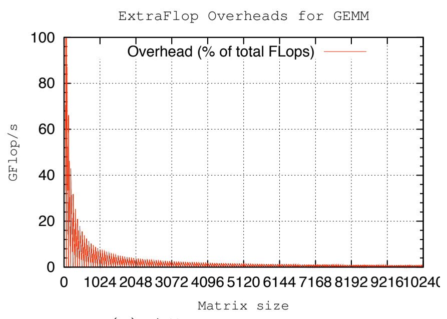
xSYRK: A block index reordering technique is used to initiate and limit the computation only to blocks that are on the diagonal or in the lower (correspondingly upper) triangular part of the matrix. In addition, all the threads in a diagonal block are responsible to compute redundantly half of the block in a data parallel fashion in order to avoid expensive conditional statements that would have been necessary otherwise. Some threads also load unnecessary data so that data is fetched from global memory in a coalesced manner. These routines are used in both some one-sided and two-sided matrix factorization algorithms.

5 Performance

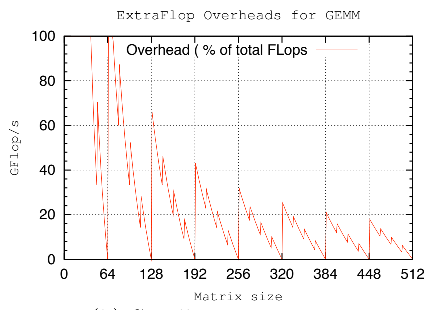
For the unnecessary computation there will be some overhead. Figure 7 shows the percentage of extra flops needed for different dimensions of matrix with parameters $blk_M = 64, blk_N = 16, blk_K = 16, nthd_X = 16, nthd_Y = 4$ for different matrix sizes. The overhead is scaled to 100 for visibility. Figure 9 and Figure 8 shows the performance results for GEMM in single and double precision respectively. In double precision we are seeing an improvement of 24 GFlops/s and in single precision it is like 170 GFlops/s. As we have discussed before other than small dimensions the improvement is significant The zig-zag patterns in performance graph resembles the blocking factor of the kernel.

As we have discussed before, if the matrices are in CPU memory one can use padding, e.g., as in [5]. We have to allocated a bigger dimension of matrix in GPU memory, put zeroes in the extra elements, then transfer the data from CPU to GPU and then call the Kernel. Figure 10 shows the performance comparison when data is in CPU memory. It is evident that for small matrix size our implementation is better and for higher dimension they are very identical. We note that the pointer redirecting approach does not use extra memory, does not require a memory copy if non padded matrix is given on the GPU memory, and finally does not require initialization of the padded elements.

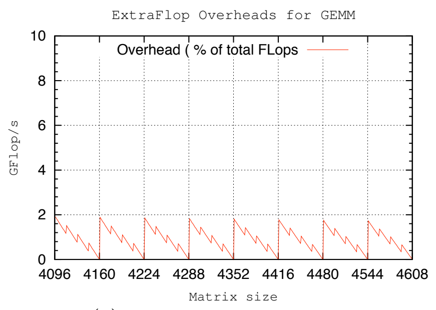
Finally Figure 11 gives an illustration on the effect of optimized BLAS on high level routines. We see similar results throughout MAGMA algorithms. Table 1 shows the performance of the one-sided QR factorization using CUBLAS and MAGMA BLAS for matrix sizes not divisible by the kernel's block size. The pointer redirecting approach brings 20% to 50% performance improvement over CUBLAS in this case.



(a) All Dimension

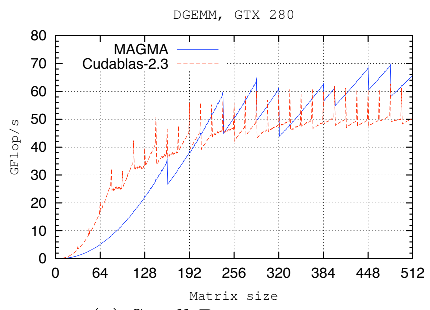


(b) Small Dimension

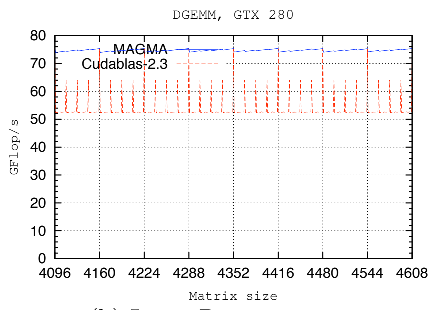


(c) Large Dimension

Fig. 7. Flops overhead in xGEMM

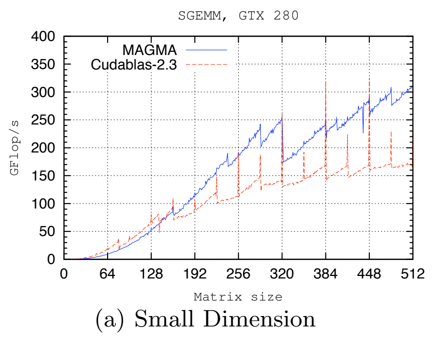


(a) Small Dimension

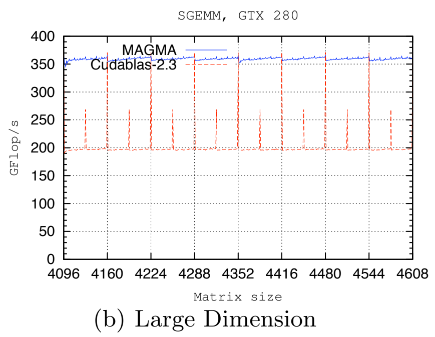


(b) Large Dimension

Fig. 8. Performance dGEMM

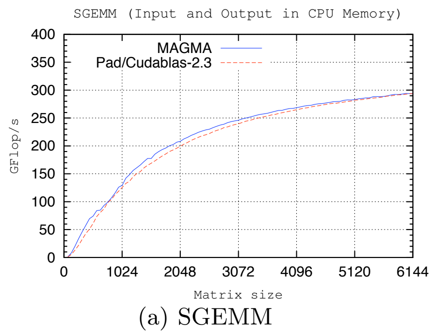


(a) Small Dimension

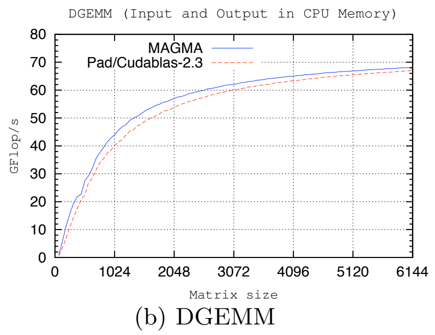


(b) Large Dimension

Fig. 9. Performance sGEMM



(a) SGEMM



(b) DGEMM

Fig. 10. Performance xGEMM with Padding (Data In/Out in CPU Memory).

Matrix Size	CUBLAS	MAGMA BLAS
1001	47.65	46.01
2001	109.69	110.11
3001	142.15	172.66
4001	154.88	206.34
5001	166.79	226.43
6001	169.03	224.23
7001	175.45	246.75
8001	177.13	251.73
9001	179.11	269.99
10001	180.45	262.90

Table 1. Performance comparison between MAGMA BLAS with pointer redirecting and CUBLAS for the QR factorization in single precision arithmetic

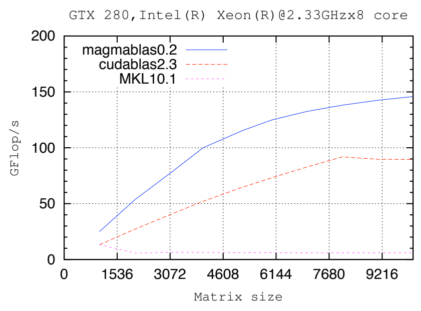


Fig. 11. Effect of optimized SGEMV on the Hessenberg reduction.

6 Conclusions and On-going Work

We presented techniques to accelerate GPU BLAS kernels that are crucial for the performance of DLA algorithms. Performance results, demonstrating significant kernels acceleration and the effect of this acceleration on high level DLA, were also presented. On-going work includes the extension of these techniques to more routines, and their inclusion in the MAGMA BLAS library.

References

1. V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In Proc. of *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008.
2. CUDA CUBLAS Library. http://developer.download.nvidia.com.
3. S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. http://icl.cs.utk.edu/magma, 11/2009.
4. Y. Li, J. Dongarra, and S. Tomov. A Note on Auto-tuning GEMM for GPUs. In Proc. of *ICCS '09*, pages 884–892, Baton Rouge, LA, 2009.
5. S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti Evaluation and Tuning of the Level 3 CUBLAS for Graphics Processors In PDSEC '08.