Optimizing Symmetric Dense Matrix-Vector Multiplication on GPUs

Rajib Nath[*]
Computer Science and Engineering
University of California, San Diego
rknath@ucsd.edu

Stanimire Tomov
Electrical Engineering and Computer Sciences
University of Tennessee, Knoxville
tomov@eecs.utk.edu

Tingxing "Tim" Dong
Electrical Engineering and Computer Sciences
University of Tennessee, Knoxville
tdong@eecs.utk.edu

Jack Dongarra
Electrical Engineering and Computer Sciences
University of Tennessee, Knoxville
dongarra@eecs.utk.edu

ABSTRACT

GPUs are excellent accelerators for data parallel applications with regular data access patterns. It is challenging, however, to optimize computations with irregular data access patterns on GPUs. One such computation is the Symmetric Matrix Vector product (SYMV) for dense linear algebra. Optimizing the SYMV kernel is important because it forms the basis of fundamental algorithms such as linear solvers and eigenvalue solvers on symmetric matrices. In this work, we present a new algorithm for optimizing the SYMV kernel on GPUs. Our optimized SYMV in single precision brings up to a 7× speed up compared to the (latest) CUBLAS 4.0 NVIDIA library on the GTX 280 GPU. Our SYMV kernel tuned for Fermi C2050 is 4.5× faster than CUBLAS 4.0 in single precision and 2× faster than CUBLAS 4.0 in double precision. Moreover, the techniques used and described in the paper are general enough to be of interest for developing high-performance GPU kernels beyond the particular case of SYMV.

General Terms

Performance Optimization, Dense Linear Algebra

Keywords

GPU, Matrix-Vector Multiplication, Symmetric Matrix, Recursive Blocking, Pointer Redirecting, Autotuning

1. INTRODUCTION

[*]Part of this work is included in the author's master thesis. During the period of this work, the author was affiliated with University of Tennessee, Knoxville.

Implementations of the Basic Linear Algebra Subprograms (BLAS) interface are a major building block of dense linear algebra (DLA) 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 enables fast Level 3 BLAS implementations for GPUs [1, 2, 4, 5, 6]. Earlier attempts (before the introductions of shared memory) could not rely on memory reuse but on the GPU's high bandwidth; as a result Level 3 BLAS implementation on GPUs were slower than the corresponding CPU implementations.

Despite the current success in developing highly optimized BLAS for GPUs, the area is still new and presents numerous cases and opportunities for improvement. Many of the important BLAS kernels for DLA can be further optimized even for the old GPUs, e.g., GTX 280. Moreover, the introduction of the Fermi GPU architecture creates further opportunities to design algorithms that will exploit the new hardware features more efficiently [7].

This paper addresses one of the most important kernels – the symmetric matrix-vector multiplication (SYMV) – which is crucial for the performance of linear as well as eigenproblem solvers on symmetric matrices. Implementing a generic matrix-vector multiplication kernel is very straightforward on GPUs because of the data parallel nature of the computation. Irregular data access patterns in SYMV bring challenges in optimization however. In the GTX 280, SYMV provided by NVIDIA's CUBLAS 2.3 achieves up to 3 GFlops/s and 5 GFlops/s in single and double precision respectively. Even though NVIDIA optimized SYMV in their recent release of CUBLAS, the performance of the new SYMV is not that attractive. CUBLAS 4.0's SYMV achieves up to 15 GFlops/s in single precision and 6 GFlops/s in double precision on a GTX 280.

In this paper, we have provided two algorithms for SYMV. The first one (*algorithm 1*) achieves up to 50 GFlops/s in single precision and 18 GFlops/s in double precision on a GTX280. This version of SYMV was included in the *Matrix Algebra for GPU and Multicore Architectures* (MAGMA) version 0.2 Library [5], which was released at SC, 2009. This particular SYMV in double precision was used to speed up one of the mixed-precision iterative refinement linear solvers in MAGMA 0.2.

Although *algorithm 1* brought an excellent improvement over the contemporary CUBLAS at the time (16 $\times$ faster than CUBLAS-2.3), an optimality analysis showed that theoretically the SYMV kernel could be further accelerated. This motivated us to look for an alternative solution and we developed our second SYMV (*algorithm 2*). The SYMV kernel in *algorithm 2* achieves up to 105 GFlops/s in single precision on a GTX 280, but incurs memory overhead of about 0.78% of the matrix size. We implemented this kernel in March 2010 [3], which at the time demonstrated a 35 $\times$ speedup comparing to the contemporary CUBLAS-2.3 in single precision arithmetic. Moreover, based on *algorithm 2*, we developed a SYMV for the Fermi architecture. The SYMV kernel for Fermi C2050 GPU achieves up to 84 GFlops/s in single precision and 32 GFlops/s in double precision, whereas CUBLAS 4.0's SYMV gets up to 20 GFlops/s in single precision and 14 GFlops/s in double precision. The results of this work are included in the recently released and freely available MAGMA 1.0 Library [8].

The rest of the paper is organized as follows. Section 2 describes some of the basic principles of writing high performance GPU kernels. Section 3 describes the state-of-the-art in optimizing a Level 2 BLAS kernel for GPUs, in particular this is the matrix-vector multiplication (GEMV) kernel. Section 4 contains the main contribution of this paper - *algorithm 1* and *algorithm 2* for the SYMV kernel. Section 5 analyzes various overheads in *algorithm 2*. Section 6 explains the different optimization techniques that we have used to optimize *algorithm 2*. We introduce recursive blocking for GPU kernels in this section, which is an important contribution in this paper. Two other optimization technique that we used are *autotuning* and *pointer redirecting*. Section 7 shows the performance of our highly optimized SYMV kernel on GTX 280 and Fermi C2050 GPUs. This kernel was included in MAGMA 0.3 released in 2010. We also show the impact of this kernel on higher level DLA algorithms. Finally, we conclude in Section 8.

2. GPU KERNEL DEVELOPMENT

To achieve high performance, GPU kernels must be written according to some basic principles/techniques, stemming from the specifics of the GPU architecture. Some of these principles are well recognized and established by now. In particular, we stress on the following two:

Blocking Blocking is a DLA optimization technique where a computation is organized to operate on blocks or submatrices of the original matrix. The idea is that blocks are of small enough size to fit into a particular level of the CPU's memory hierarchy, so that once loaded, one can reuse the blocks' data to perform the arithmetic operations that they are involved in. This idea can be applied for GPUs using the GPUs' shared memory. The application of blocking is crucial for the performance of numerous GPU kernels, including the SYMV, as demonstrated in this paper.

Coalesced Memory Access GPU global memory accesses are costly, making it crucial to have the right access pattern in order to get maximum memory bandwidth. There are two access requirements [9]. The first is to organize global memory accesses in terms of parallel consecutive memory accesses – 16 consecutive elements at a time by the threads of a half-warp (16 threads) – so that memory accesses (to 16 elements at a time) are **coalesced** into a single memory access. This is demonstrated in the kernels' design throughout the paper. Second, the data should be properly aligned. In particular, the data to be accessed by half-warp should be aligned at $16 *$ `sizeof(element)`, e.g., 64 for single precision elements.

Clearly, fulfilling the above requirements will involve partitioning the computation into blocks of fixed sizes (e.g., multiple of 16) and designing memory accesses that are coalescent (properly aligned and multiple of 16 consecutive elements). The problem of selecting best performing partitioning sizes/parameters for the various algorithms, as well as the cases where (1) the input data is not aligned to fulfill coalescent memory accesses and (2) the problem sizes are not divisible by the partitioning sizes required for achieving high performance, need special treatment [10].

Further down a *thread block* will be denoted by TB, the size of a computation block in a kernel will be denoted by N_{TB} (or $N_{TBX} \times N_{TBY}$ in 2D), the number of threads in a TB by N_T (or $N_{TX} \times N_{TY}$ in 2D), and the size associated with second level blocking inside a TB by nb. In addition, the matrices considered in this paper are stored in column major data format.

3. LEVEL 2 BLAS ON GPUS

Level 2 BLAS routines are of low computational intensity and therefore DLA algorithms must be designed (ideally) to avoid them. This can be achieved for example using the *delayed update* approach where the application of a sequence of Level 2 BLAS is delayed, and accumulated in order to be applied at once as a more efficient single matrix-matrix multiplication [11]. In many cases, like MAGMA's mixed-precision iterative refinement solvers [12] or two-sided matrix factorizations [13], this approach only reduces the number of Level 2 BLAS, and therefore the availability of efficient implementations are still crucial for the performance. This section considers the GPU implementation of one fundamental Level 2 BLAS operation, namely the matrix-vector multiplication routine for general dense matrices (GEMV).

The GEMV multiplication routine performs one of:

$$y := \alpha A x + \beta y \quad \textit{or} \quad y := \alpha A^T x + \beta y,$$

where A is an M by N matrix, x and y are vectors, and α and β are scalars. The two cases are considered separately in the following two subsections.

3.1 Non-Transposed Matrix

The computation in this case can be organized in one dimensional grid of TBs of size N_{TB} where each block has $N_T = N_{TB}$ threads, as shown in Figure 1. Thus, each thread computes one element of the resulting vector y.

GEMV is the first of the kernels considered to which *blocking* can be applied. Although matrix A cannot be reused in any blocking, vector x can be reused by the threads in a TB. Specifically, the computation is blocked by loading nb consecutive elements of x at a time into the shared memory (using all N_T threads). This part of x is then used by all N_T threads in a TB to multiply it by the corresponding $N_{TB} \times nb$ submatrix of A. The process is repeated $\frac{N}{nb}$ times.

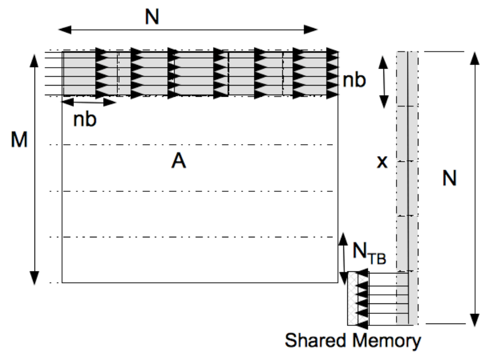

Figure 1: Algorithmic view of matrix-vector multiplication on GPUs where matrix A is non-transposed. A TB accesses a submatrix ($N_{TB} \times nb$) of matrix A. Each arrow in that submatrix denotes the activity of a single thread in a TB.

Note that the algorithm as described depends on two parameters – N_{TB} and nb. The starting addresses of A, x, and y are taken to be divisible by $16 * \texttt{sizeof(element)}$ and the leading dimension of A is divisible by 16. This guarantees that all memory accesses in the algorithm are coalescent.

3.2 Transposed Matrix

Following the non-transposed approach will lead to poor performance because the memory accesses are not going to be coalesced (see Figure 2). To improve the speed on accessing the data, blocks of the matrix A are loaded into the shared memory using coalesced memory accesses and then the data from the shared memory is used to do all the necessary computations (see Figure 3).

Although the new version significantly improves the performance, experiments that increase the design space of the algorithm show that further improvements are possible. In particular, one exploration direction is the use of higher number of threads in a TB (e.g., 64) as high performance DLA kernels are associated with the use of 64 threads (and occasionally more) for Tesla GPUs preceding the Fermi and 256 threads for Fermi [2, 7]. The use of 64 threads (for GPUs prior to Fermi) does not improve performance directly because the amount of shared memory used (a 64×64 matrix) gets to be excessive, prohibiting the effective scheduling of that amount of threads [9]. It is possible to decrease the use of shared memory while having higher thread level parallelism (e.g., 32×32 matrix in shared memory with a thread block size of 32×2) in the following way: (1) two groups of 32×1 threads each, e.g., denoted by 32_j where $j = 0/1$, load correspondingly the two 32×16 submatrices of the shared memory matrix using coalesced memory accesses, (2) each group performs the computation from the second GEMV version, but is constrained to the 16×32 submatrix of the shared memory matrix, accumulating their independent y_j results. The final result $y := y_0 + y_1$ can be accumulated by one of the $j = 0/1$ threads. The same idea can be used with more threads, e.g., 32×4, while using the same amount of shared memory.

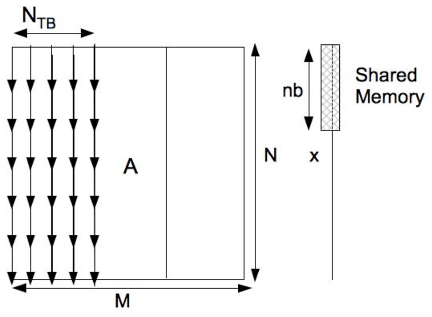

Figure 2: Basic implementation of matrix-vector multiplication on GPUs where matrix A is transposed.

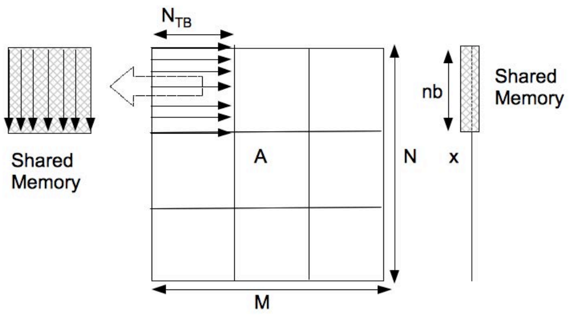

Figure 3: Optimized implementation of matrix-vector multiplication on GPUs where matrix A is transposed.

4. SYMMETRIC DENSE MATRIX VECTOR MULTIPLICATION ON GPUS

The SYMV multiplication routine performs:

$$y := \alpha A x + \beta y,$$

where α and β are scalars, x and y are vectors of size N, and A is an N by N symmetric matrix.

4.1 Symmetric Storage

As the matrix is symmetric, it is sufficient to store half of the matrix. The matrix is stored in the upper or lower triangular part of a two-dimensional array of size $N \times N$, as shown in Figure 4. The difficulty of designing a high performance SYMV kernel stems from the triangular data storage, for which it is challenging to organize a data parallel computation with coalescent memory accesses (both load and store). Indeed, if A is given as an $N \times N$ array storing both the upper and lower triangular parts of the symmetric ma-

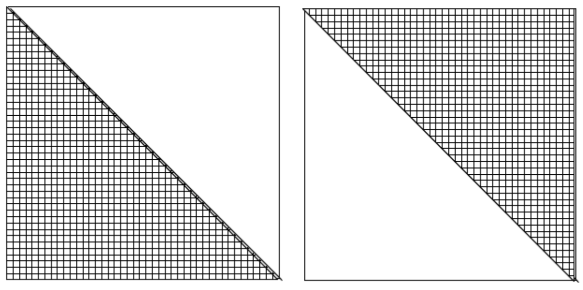

Figure 4: Symmetric matrix storage format (*left* figure shows the matrix stored in the lower triangle, *right* figure shows the matrix is stored in the upper triangle).

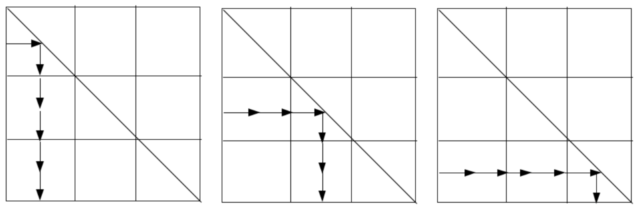

Figure 5: Three cases of TB computations in SYMV with *algorithm 1*: *left* is type A, *middle* is type B, and *right* is type C.

trix A, the SYMV kernel can be implemented using GEMV. We describe our two SYMV algorithms in the following two subsections.

4.2 Algorithm 1: Storage Oblivious

Similar to GEMV, the computation is organized in one dimensional grid of computational blocks of size N_{TB}. One computational grid is assigned to a single TB where each TB has $N_T = N_{TX} \times N_{TY}$ threads. For this algorithm $N_{TB} = nb$. This choice will divide the matrix into a 2D grid of $|TB| \times |TB|$ blocks where $|TB| = \frac{N}{nb}$. Thread block TB_i will access blocks $\{A_{i,j} : 1 \leq j \leq i\} \cup \{A_{j,i} : i \leq j \leq |TB|\}$ from matrix A. A TB computation can be classified as one of three cases (see the illustration in Figure 5):

- Type A – TB threads do a SYMV followed by a GEMV (transpose);
- Type B – TB threads do a GEMV (non-transpose) followed by a SYMV and a GEMV (transpose);
- Type C – TB threads do a GEMV (non-transpose) followed by a SYMV.

This way the computation within a TB is converted into one/two GEMVs (to reuse the GEMV kernels) and a SYMV involving a diagonal submatrix of size $N_{TB} \times N_{TB}$. The remaining SYMV is also converted into a GEMV by loading the $N_{TB} \times N_{TB}$ matrix into the GPU's shared memory and generating the missing symmetric part in the shared memory (a process defined as *mirroring*). Figures 6 and 7 compare the performance for a kernel with parameters $N_{TB} = nb = 32$, $N_T = 32 \times 4$ with that of CUBLAS.

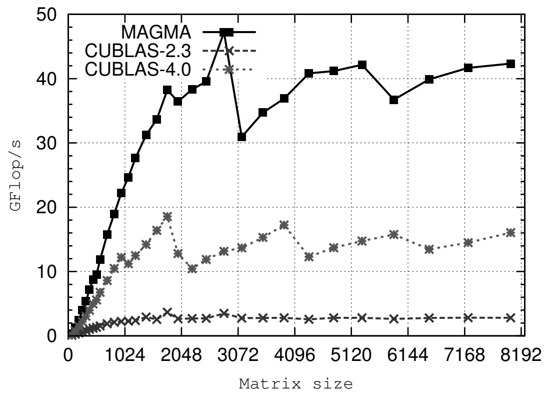

Figure 6: Performance of MAGMA's SYMV in single precision with *algorithm 1* on a GTX 280.

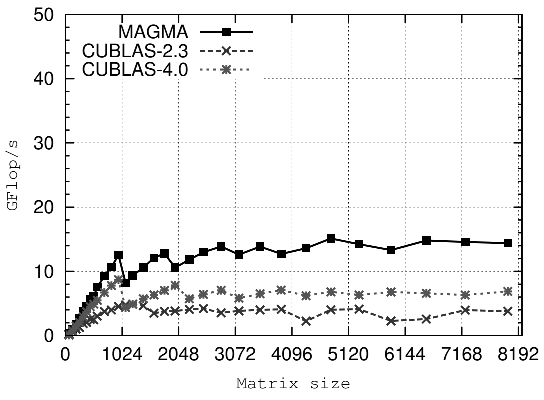

Figure 7: Performance of MAGMA's SYMV in double precision with *algorithm 1* on a GTX 280.

This SYMV kernel was included in the release of MAGMA 0.2. Although the algorithm described above yields better performance compared to CUBLAS on a GTX 280, the observed performance is far from the theoretical peak performance that relates to the bandwidth of the GPU. SGEMV gets up to 66 GFlops/s in a GTX 280 where the bandwidth is 140 GBytes/s. One might expect that the performance of SSYMV will be in the vicinity of 99 GFlops/s (see Section 7.2). *Algorithm 1* does not fully take the structure of the symmetric matrix into consideration. It loads the full A matrix whereas loading half of the symmetric matrix would have been sufficient. This insight provides the motivation for finding a better algorithm for SYMV that runs efficiently on GPUs by taking advantage of the data storage format of a symmetric matrix. This will reduce the loads from the GPU's global memory by half.

4.3 Algorithm 2: Storage Aware

The computation in this SYMV algorithm is organized in one dimensional grid of computational blocks of size N_{TB}, as it was done for the previous algorithm. One computational grid is assigned to a single TB where each TB has $N_T = N_{TX} \times N_{TY}$ threads. The layout of the thread block is irrelevant as inside a single kernel one can rearrange the threads configuration on the fly to match the required computation or memory access pattern. For this algorithm $N_{TB} = nb$. This choice will divide the matrix into a 2D grid of $|TB| \times |TB|$ blocks where $|TB| = \frac{N}{nb}$. Thread block TB_i will access blocks $\{A_{i,j} : 1 \leq j \leq i\}$ from matrix A as shown in Figure 8. Some blocks $\{A_{i,j} : i \neq j\}$ will be used twice

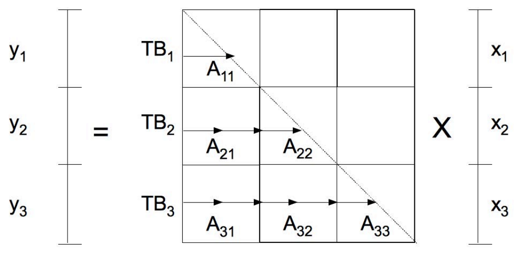

Figure 8: Data access pattern in SYMV with *algorithm 2*.

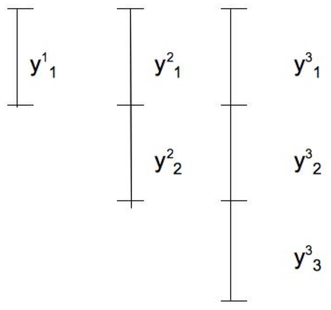

Figure 9: Results produced by each TB in SYMV with *algorithm 2*.

to compute partial results of resultant vectors y_i and y_j. Therefore, instead of computing a single final vector y_i, TB_i will be computing partial results of vectors $\{y_j : 1 \leq j \leq i\}$. These partial result vectors produced by TB_i are named as $\{y^i_j : 1 \leq j \leq i\}$ as shown in Figure 9. The computation by TB_i can be summarized as follows:

$$y^i_j := A^T_{i,j} x_i \ for\ j = \ 1\ to\ i - 1$$

$$y^i_i := \sum_{j=1}^{j=i} A_{i,j} x_j .$$

As described in the first algorithm, the missing symmetric parts in the diagonal blocks $(A_{i,i})$ are produced using mirroring. This completes the first phase of the new SYMV algorithm. In the second phase another kernel of the same one dimensional grid format is launched to compute the final y_i's as follows:

$$y_i := \sum_{j=i}^{j=|TB|} y^j_i .$$

Here $|TB|$ is the number of TBs required for a matrix of size N, i.e., $|TB| = \lceil \frac{N}{N_{TB}} \rceil$.

5. OVERHEAD ANALYSIS

Algorithm 1 is aware of the symmetric data storage format but it does not exploit it. In contrast, *Algorithm 2* is very promising as it exploits the symmetric data storage. However it has some overhead in terms of execution time and memory space as it launches an additional kernel to add up the partial results (y^i_j), and requires some extra memory to store the partial results. In addition, TBs are doing different amounts of work, which may lead to load imbalance. Hence, there are four kinds of overheads: (a) space overhead, (b) memory allocation overhead, (c) reduction overhead, and (d) load imbalance overhead. We discuss these overheads in this section and point out circumstances where these overheads are insignificant.

5.1 Space Overhead

The extra memory requirement for *algorithm 2* is:

$$\frac{N_{TB} \times |TB| \times (|TB| + 1)}{2} * sizeof(element)\ \texttt{Bytes}.$$

Figure 10 shows the memory overhead for different values of

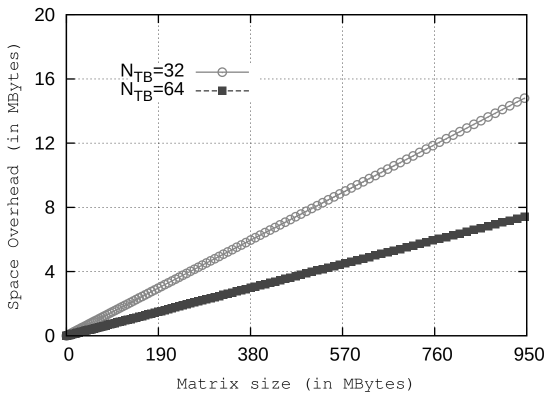

Figure 10: Memory overhead of MAGMA's SYMV in single precision with *algorithm 2* on a GTX 280.

N_{TB} (32 and 64). The space overhead is 1.56% and 0.78% of the matrix size with $N_{TB} = 32$ and $N_{TB} = 64$ respectively. Hence using a big N_{TB} is beneficial.

5.2 Memory Allocation Overhead

We have to allocate the additional memory to run *algorithm 2*. Figure 11 shows that the allocation overhead increases linearly with the space overhead. This also motivates us to use bigger value for N_{TB}.

Usually in high level DLA algorithm, we allocate the required workspace once and reuse it several times. Mixed-precision Iterative refinement solvers for example call the matrix-vector product several times. Tridiagonalization also calls the matrix-vector product multiple times. For these two algorithms, allocation overhead will not be an issue.

Figure 12 shows the percentage of time the kernel spends in allocating workspace and computing the matrix-vector product. As the matrix size gets larger, the percentage of time spent in workspace allocation decreases. So, for a larger problem size, the allocation overhead will be amortized if *algorithm 2* brings very good performance for SYMV.

5.3 Reduction Overhead

As mentioned before, *algorithm 2* requires some extra memory to store the partial results. It launches an extra kernel

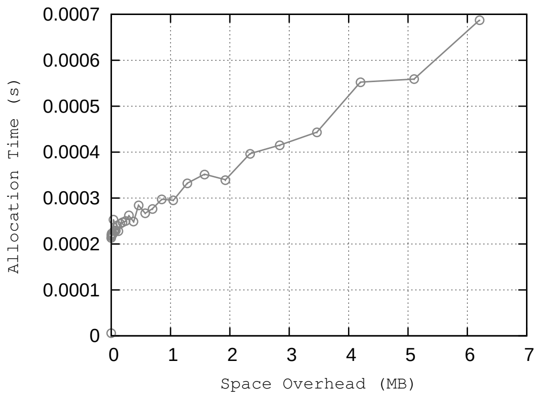

Figure 11: Allocation overhead of MAGMA's SYMV in single precision with *algorithm 2* on a GTX 280.

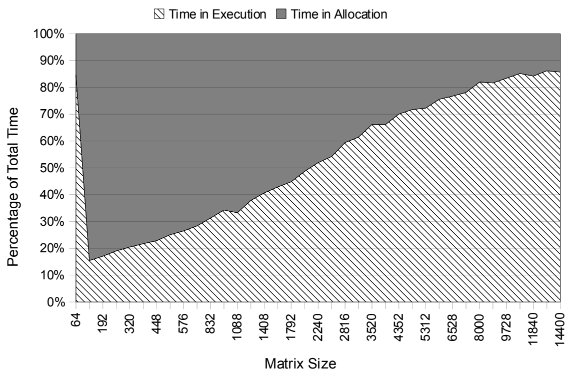

Figure 12: Distribution of kernel execution time and additional memory allocation time in the SYMV kernel in single precision with *algorithm 2* on a GTX 280.

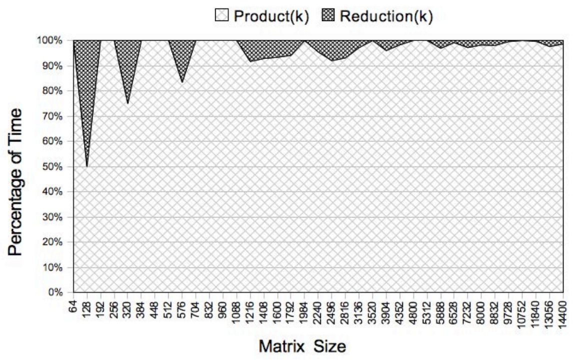

Figure 13: Distribution of kernel time spent in computing product and doing reduction in SYMV kernel in single precision with *algorithm 2* on a GTX 280.

to add up the partial results y_j^i. The performance benefit comes by paying this small extra reduction cost. Figure 13 shows the relative portion of kernel time spent in computing products (product kernel) and doing reductions (reduction kernel). Later in the performance section, we will show that *algorithm 2* brings good performance despite all the overheads.

5.4 Load Imbalance Overhead

TBs are doing different amounts of work. During the product computation phase, the amount of work done by TB_i is $\Delta + (i-1) \times \nabla$, where ∇ is work required in an off diagonal block and Δ is the work required in a diagonal block. The difference between amount of work assigned to two adjacent TBs (adjacent in terms of TB IDs, e.g. TB_i and TB_{i+1}) is ∇, which is a very small constant. If the TBs are scheduled in a monotonically increasing order according to their IDs, *algorithm 2* will not experience load imbalance problems. The performance section validates this assertion.

6. FURTHER OPTIMIZATION

In this section, we highlight all the tunable parameters in *algorithm 2* and describe an autotuning approach to tune those parameters. We also introduce a *recursive blocking* optimization technique to minimize the overhead as discussed in the previous section. Later on we describe our solution of generalizing the SYMV kernel to remove performance oscillations.

Parameter	Description
N_{TB}	Blocking Size
N_{TX}	Number of threads in X dimension
N_{TY}	Number of threads in Y dimension

Table 1: Tunable parameters in *algorithm 2*.

6.1 Autotuning

Automatic performance tuning (optimization), or autotuning in short, is a technique that has been used intensively on CPUs to automatically generate near-optimal numerical libraries. For example, ATLAS [15, 16] and PHiPAC [17] are used to generate highly optimized BLAS. In addition, FFTW [18] is successfully used to generate optimized libraries for FFT, which is one of the most important techniques for digital signal processing. Empirical auto-tuning [19] has also been used to optimize high level DLA algorithms [20].

With the success of auto-tuning techniques on generating highly optimized DLA kernels on CPUs, it is interesting to see how the idea can be used to generate near-optimal DLA kernels on modern high-performance GPUs. Indeed, work in this area [4] has already shown that auto-tuning for GPUs is a very practical solution to easily port existing algorithmic solutions on quickly evolving GPU architectures, and to substantially speed up even highly hand-tuned kernels.

There are two core components in our auto-tuning system:

Code generator The code generator produces code variants according to a set of predefined, parameterized templates/algorithms. The code generator also applies certain state of the art optimization techniques.

Heuristic search engine The heuristic search engine runs the variants produced by the code generator, and finds out the best one using a feedback loop, e.g., the performance results of previously evaluated variants are used as a guidance for the search on currently unevaluated variants.

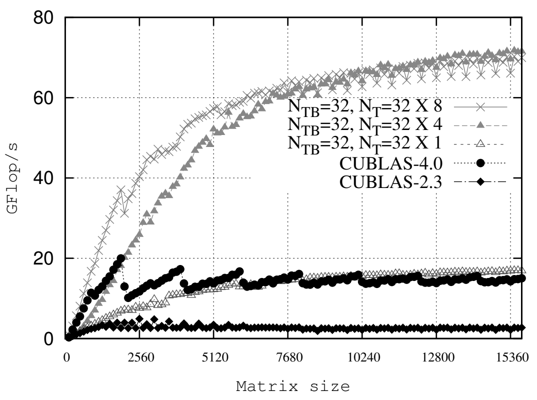

Figure 14: Performance of different automatically generated SYMV kernels in single precision with *algorithm 2* on a GTX 280.

Our SYMV algorithm has three tuning parameters as shown in Table 1. Figure 14 shows the performance of automatically generated kernels with different values for these parameters. Usually bigger values of N_{TB} bring greater performance. This strengthens the results from the overhead analysis section.

However, the autotuner failed to get beyond $N_{TB} = 32$ because there is not enough shared memory on the GTX 280. With $N_{TB} = 64$, we will need 64×64 dimension of shared memory for the on the fly mirroring operation in the diagonal computations, $A_{i,i} \times x_i$.

6.2 Recursive Blocking

Due to the limited amount of shared memory in GPUs (GTX 280), *algorithm 2* fails to work with $N_{TB} = 64$. This limitation can be overcome by using recursive blocking as shown in Figure 15. Recursive blocking is one of the important contributions in this work.

With $N_{TB} = 64$ and $N_T = 256$, a 64×16 dimension matrix is allocated in shared memory. In the off-diagonal computations, $A^T_{i,j} \times x_i$ where $i \neq j$ or $A_{i,j} \times x_j$ where $i \neq j$, the layout of the thread block is $N_T = 256 = 64 \times 4$. The mechanisms for these off-diagonal computations are straightforward. The diagonal computations, $A_{i,i}x_i$, are performed in a recursive way, using the same kernel with block size $N_{TB} = 32$.

As we can see from Figure 15, we will get two diagonal subblocks and one off diagonal subblock after applying recursive blocking in a diagonal block. These three subblocks are processed sequentially by the same 256 threads. During the recursive part of the kernel, 256 threads inside a thread block rearrange themselves as 32×8 threads to meet the computation and data access pattern. All of the intermediate results are stored in registers instead of in global memory.

6.3 Optimizing for Generic Size

Most of the available BLAS for GPUs achieve very high performance when the matrix size is divisible by the internal blocking size of the underlying algorithm or specific implementation. But there are performance dips when the user gives an irregular problem size (size that is not multiple of internal blocking size). BLAS with performance oscillation restricts us to use it in a generic way.

A few possibilities of dealing with matrix dimensions not divisible by the blocking size have been explored. We discuss three of them here:

Conditional Statement One approach is to have some "boundary" TBs doing selective computation. This will introduce several if-else statements in the kernel

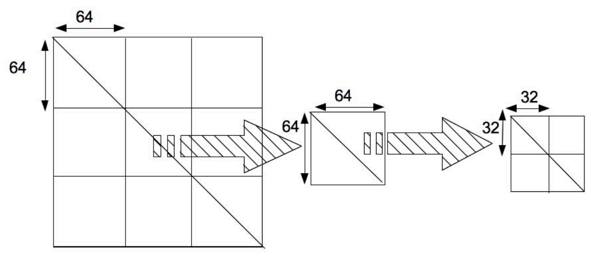

Figure 15: Recursive blocking in SYMV with *algorithm 2*.

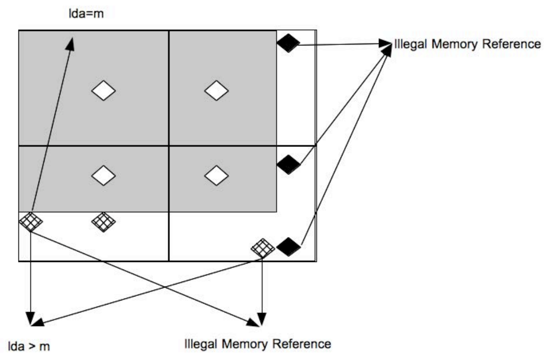

Figure 16: Possible illegal memory reference after blocking a matrix whose size is not a multiple of the blocking size. This happens if we let the threads in boundary blocks reference memory without readjusting the index.

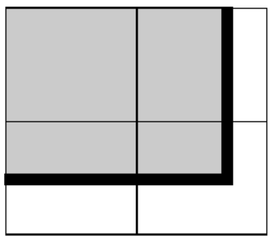

Figure 17: Last valid access (The gray box shows the original matrix. The Outer white box shows the matrix of a higher dimension to make the dimension multiple of the blocking size. The dark black row and column show the last valid row and column in the original matrix respectively).

which will prevent the threads inside a TB to run in parallel.

Padding Another approach is to use padding. Padding is the technique where a matrix of higher dimension (to make the new size divisible by N_{TB}) is allocated on the GPU memory, and the extra elements initialized by zero. Now the computation occurs in a natural way as elements with zero value will not change the final result.

Pointer Redirecting We adopt a different approach, in particular *pointer redirecting*. Our approach is to, instead of preventing certain threads from computing (with if-else statements), to let them do similar work as the other threads in a TB, and discard saving their results at the end. This can lead to some illegal memory references as illustrated in Figure 16. The *pointer redirecting* techniques redirect the illegal memory references to valid ones, within the matrix of interest, in a way that would allow the memory accesses to be coalescent. The specifics of the redirecting depend on the kernel, but in general, if a thread is to access invalid rows/columns of a matrix (beyond row/column M/N), the access is redirected towards the last row/column. This is shown in Figures 17, 18, and 19. In summary, each illegal memory address is converted into a memory address (inside the matrix of interest) which is the

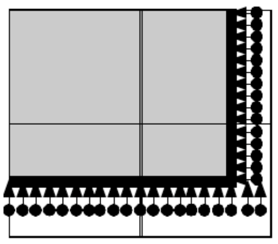

Figure 18: Pointer redirecting (Arrows indicate redirection of accesses. A thread destined to access an element at the start of the arrow will access the element pointed to by the end of the arrow).

closest to the illegal address in terms of cartesian distance in two dimension.

The case where the starting address of a data stream is not a multiple of $16 * sizeof(element)$ (but the leading dimension is multiple of 16) is handled similarly – threads that must access rows "before" the first row are redirected to access the first row.

Pointer redirecting has no memory overhead as we see in padding. If the input matrices have irregular dimensions and are given on the GPU memory, then padding cannot be as efficient as there would be memory and data copying overhead.

We adopted the *pointer redirecting* approach to make the SYMV kernel generic. We extended the previously proposed *pointer redirecting* to work in the upper and lower part of the matrix. This was initially done to accommodate the tridiagonalization routine as most of the SYMV in it start from an address that is not divisible by 16. As a result we would not have been able to keep the fully coalesced memory accesses to the GPU global memory. More details about *pointer redirecting* approach can be found here [10].

7. EXPERIMENTAL RESULTS

In this section, we give performance results of the final kernel and its impact on a high level DLA algorithm.

7.1 Kernel Performance

Figure 20 compares the performance of the new SYMV kernel optimized with recursive blocking and CUBLAS on a GTX 280 (performance for double precision is shown in Figure 21). The parameters of this kernel are : $N_{TB} = 64$ and $N_T = 256 = 32{\times}8 = 64{\times}4$. With $N_{TB} = 32$, the space overhead is 1.56% of the matrix size, and with $N_{TB} = 64$ the space overhead is 0.78% of the matrix size. Not only does $N_{TB} = 64$ with recursive blocking offer better performance (105 GFlops/s vs 70 GFlops/s), it also reduces the space overhead by a factor of two compared to the kernels with $N_{TB} = 32$. This performance number is valid when we ignore the space allocation time overhead. If we consider the space allocation time overhead, the maximum achievable performance drops from 105 GFlops/s to 90 GFlops/s. The performance of the kernel with and without space allocation overhead is also shown in Figure 20. However, by changing the SYMV interface we can incur this allocation overhead only once for many kernel calls. Our kernel performs worse than CUBLAS 4.0 when matrix size is less than 1600 when we consider the allocation overhead. When the matrix size is bigger than 1600, the allocation overhead is amortized and our kernel performs significantly better than CUBLAS. The only problem with this algorithm is that if there is not enough memory available on the GPU, the code will not be able to execute.

Figure 22, shows the performance of a generic SYMV ker-

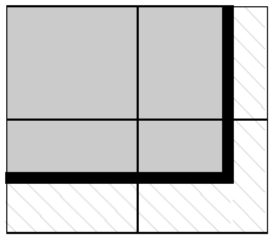

Figure 19: Mirroring (Hatched portion indicates that the elements here are replications of some valid element inside the original matrix. Replication is done in terms of cartesian distance from the source to any element inside the matrix.)

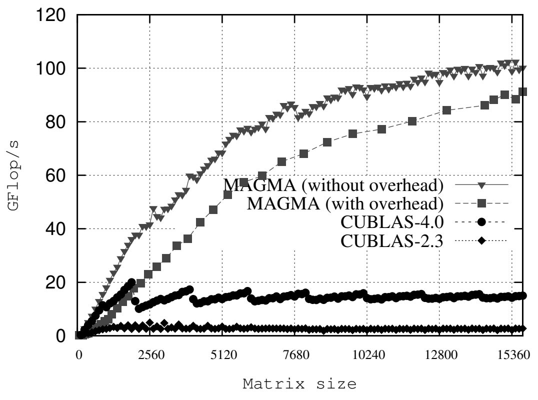

Figure 20: Performance of the new SYMV kernel in single precision with *algorithm 2* on a GTX 280. Recursive blocking was used in this kernel.

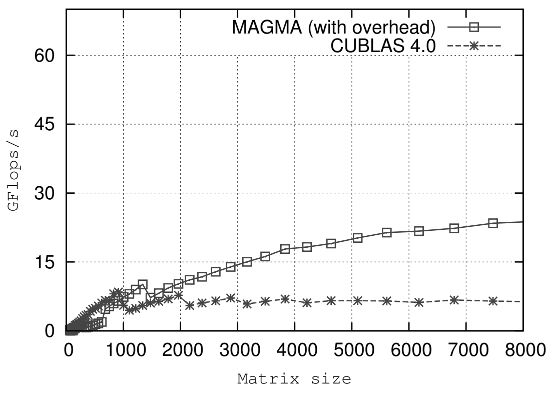

Figure 21: Performance of the new SYMV kernel in double precision with *algorithm 2* on a GTX 280.

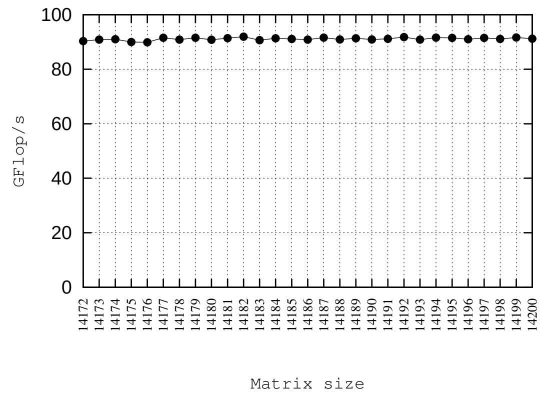

Figure 22: Performance of a generic SYMV kernels in single precision with *algorithm 2* on a GTX 280. Recursive blocking and pointer redirecting was used in this kernel.

nel for matrix sizes from 14172 to 14200. It is evident that the pointer redirecting approach works really well by removing the performance oscillation.

We tuned *algorithm 2* for getting an optimized SYMV kernel in Fermi C2050. The SYMV kernel for the Fermi C2050 GPU gets up to 83 GFlops/s in single precision and 32 GFlops/s in double precision, whereas CUBLAS 4.0's SYMV gets up to 20 GFlops/s in single precision and 14 GFlops/s in double precision. The performance is shown in Figures 23 and 24. Although the global memory bandwidths of GTX 280 and Fermi C2050 are close, SYMV in Fermi is slow for two reasons. First, our kernel depends highly upon reusing data from shared memory. Second, the shared memory access latency of Fermi is higher than that of GTX 280. The results of this work are included in the recently released and freely available MAGMA 1.0 [8].

7.2 Optimality Analysis

The optimal performance for memory bound kernels can be easily computed taking into account the machine's bandwidth. For example, the SYMV kernel needs to access $N^2 * sizeof(element)/2$ Bytes in order to make $2N^2$ Flops. In other words, for each Byte transferred one can do one single precision Flop (or 0.5 Flops in double precision). Thus, the theoretical optimal performance for the SYMV on a GPU like GTX280 with a memory bandwidth of 141.7 GB/s, is 141.7 GFlops/s in single and 70.85 GFlops/s in double precision arithmetic.

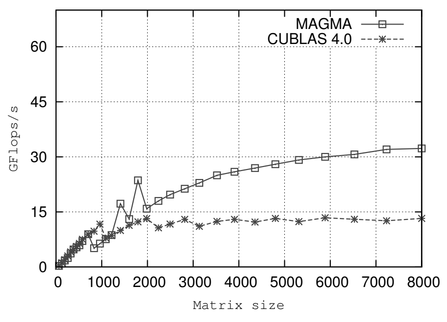

Figure 24: Performance of SYMV in double precision with *algorithm 2* on a Fermi C2050.

7.3 Impact on High Level DLA algorithms

The end goal of optimizing the SYMV kernel was to optimize the tridiagonal factorization routine. Previously in [14], we restricted ourselves from using the SYMV kernel implemented with *algorithm 1*, as 50% of the operations in the tridiagonal factorization were done in the SYMV kernel, and GEMV was faster (66 GFlops/s in single precision) than SYMV (45 GFlops/s in single precision) on a GTX 280. Therefore we used GEMV and adopted another optimization, namely a GPU implementation of symmetric matrix rank-2k update (SYR2K) that explicitly generates the entire symmetric matrix resulting from the operation so that we could use GEMV in the panels instead of the slower SYMV. That approach did not need extra memory. That particular implementation of the tridiagonalization scaled up to 80 GFlops/s in single precision on a GTX 280. With the new SYMV kernel, achieving up to 105 GFlops/s in single precision on a GTX 280, the tridiagonalization in single precision achieves up to 120 GFlops/s on a GTX 280 and thus brings up of 50% performance improvement over the old version [14]. The performance is shown in Figure 25.

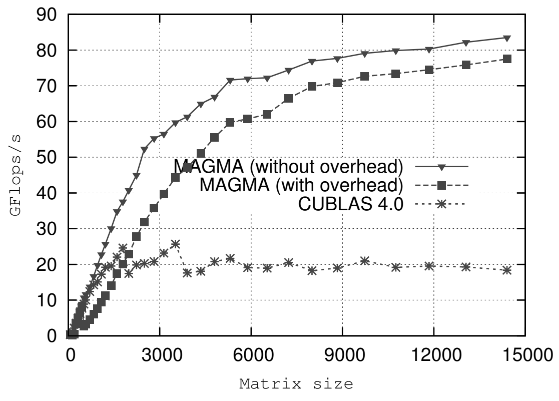

Figure 23: Performance of SYMV in single precision with *algorithm 2* on a Fermi C2050.

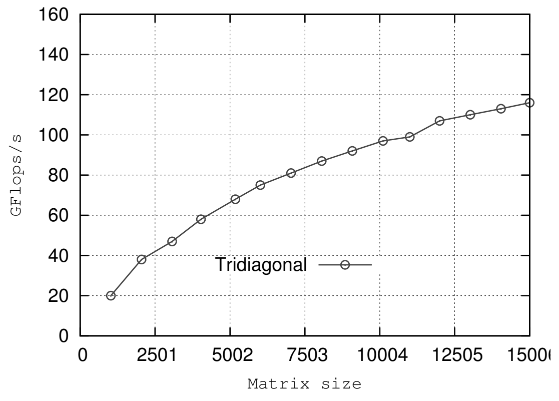

Figure 25: Performance of MAGMA's Tridiagonalization in single precision on a GTX 280.

8. CONCLUSION

We presented a highly optimized SYMV kernel for modern GPUs. The first version of the SYMV kernel in double precision, *algorithm 1*, was developed for MAGMA's iterative refinement procedure in 2009 [12]. The single precision version outperformed a contemporary CUBLAS 2.3's SYMV kernel by 40 GFlops/s on a GTX 280. Even though the kernel was 25x faster than CUBLAS 2.3's SSYMV, we could not use it in MAGMA's tridiagonalization [14]. Later on SYMV was optimized more to improve tridiagonalization and we improvised *algorithm 2*. Our contribution shows how GPU kernels should be written for irregular data access patterns. With the introduction of recursive blocking for shared memory in *algorithm 2*, we showed how GPU kernels can work with higher block size with limited amount of shared memory. This particular version of SYMV, developed in March 2010[3], was 50× faster than contemporary CUBLAS 2.3's SYMV kernel with a little memory overhead (0.78% of matrix size). It is also 7× faster than recently released CUBLAS 4.0. With this kernel, tridiagonalization was 50% faster than MAGMA's previous implementation. Later on we tuned *algorithm 2* for getting an optimized SYMV kernel on Fermi C2050. The SYMV kernel for Fermi C2050 GPU gets up to 84 GFlops/s in single precision and 32 GFlops/s in double precision, whereas CUBLAS 4.0's SYMV gets up to 20 GFlops/s in single precision and 14 GFlops/s in double precision. We are currently investigating how SYMV can be optimized further in Fermi. The results of this work are included in the recently released and freely available MAGMA version 1.0RC [8].

9. REFERENCES

[1] CUDA CUBLAS Library. http://developer.download.nvidia.com.

[2] V. Volkov and J. Demmel. Benchmarking gpus to tune dense linear algebra. In Proc. of *SC '08*, pages 1–11, Piscataway, NJ, USA, 2008.

[3] R. Nath Accelerating Dense Linear Algebra for GPUs, Multicores and Hybrid Architectures: an. Autotuned and Algorithmic Approach. Master Thesis, University of Tennessee, Knoxville, USA.

[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. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA version 0.2 Users' Guide. http://icl.cs.utk.edu/magma, 11/2009.

[6] R. Nath, S. Tomov, and J. Dongarra. BLAS for GPUs. Chapter 4, In Scientific Computing with Multicore and Accelerators, Computational Science Series, Chapman and Hall/CRC, 2010.

[7] R. Nath, S. Tomov., and J. Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. The International Journal of High Performance Computing Applications, 2010.

[8] MAGMA version 1.0RC2. http://icl.cs.utk.edu/magma, 2010.

[9] NVIDIA CUDA Compute Unified Device Architecture - Programming Guide. http://developer.download.nvidia.com, 2007.

[10] R. Nath, S. Tomov., and J. Dongarra. Accelerating GPU Kernels for Dense Linear Algebra. Vecpar 2010, Berkeley, CA, USA

[11] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, 1992.

[12] S. Tomov, R. Nath, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. UTK EECS Technical Report ut-cs-09-649, December 2009.

[13] S. Tomov and J. Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report 219, LAPACK Working Note, May 2009.

[14] S. Tomov, R. Nath, and J. Dongarra Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing PARCO 2010.

[15] R. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. Parallel Computing **27** (2001), no. 1-2, 3–35.

[16] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. Proceedings of the IEEE 93 (2005), no. 2, special issue on "Program Generation, Optimization, and Adaptation".

[17] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. International Conference on Supercomputing, 1997, pp. 340–347.

[18] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, IEEE, 1998, pp. 1381–1384.

[19] E. Agullo, J. Dongarra, R. Nath, S. Tomov Fully Empirical Autotuned Dense QR Factorization For Multicore Architectures. Research Report INRIA, to appear in europar 2011.

[20] Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) http://icl.cs.utk.edu/plasma/

10. ACKNOWLEDGMENTS

We would like to thank NVIDIA and NSF for their support. We would like to thank Julien Langou for his constructive discussion on symmetric matrix vector product on GPUs. We also want to acknowledge the support of the anonymous reviewers who gave us their valuable feedback. We are grateful to Tajana Rosing and Ferdinand Escudero. Ms. Rosing has aided us with beneficial comments on the final presentation. Six hours before the submission deadline, the laptop containing all the data and text crashed. Mr. Escuardo helped us recover the data, which made it possible to submit this paper.

Julien Langou is an Assistant Professor in Department of Mathematical and Statistical Sciences at University of Colorado Denver. Tajana Rosing is a Professor in Department of Computer Science and Engineering at University of California, San Diego (UCSD). Ferdinand Escudero is a system administrator in CSE Department at UCSD.