Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Jack Dongarra
Innovative Computing Laboratory,
University of Tennessee
Knoxville, USA
{ahmad,haidar,tomov,dongarra}@icl.utk.edu

ABSTRACT

This paper presents a software framework for solving large numbers of relatively small matrix problems using GPUs. Our approach combines novel and existing HPC techniques to methodically apply performance analysis, kernel design, low-level optimizations, and autotuning to exceed in performance proprietary vendor libraries. As a case study, we discuss the fundamental matrix operations defined by the Basic Linear Algebra Subprograms (BLAS) standard. This case study is significantly important for wide range of applications, including astrophysics, tensor contractions, sparse direct solvers, and others. We provide a generic design that is capable of dealing with problems of different sizes, and handling the irregularity arising from size variations. The developed solution adopts a *batched computation* scheme, where the same operation is concurrently applied to all matrices within a single computational kernel. The paper discusses the data layout, kernel design, and optimization techniques. We also propose a design scheme that is centralized around matrix-matrix multiplication (GEMM) kernel, so that any improvement on this particular kernel propagates automatically to other routines. Our performance results show significant speedups using a Pascal generation GPU (Tesla P100) against state-of-the-art solutions using cuBLAS, as well as against two 10-core Haswell CPUs running the MKL library. This work is part of the MAGMA library.

CCS CONCEPTS

•**Computing methodologies** →*Massively parallel algorithms;*

KEYWORDS

Batched Computation, GPU Computing, Basic Linear ALgebra Subprograms

ACM Reference format:
Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Jack Dongarra. 2017. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079103

1 INTRODUCTION

The processing of many small independent problems has gained a lot of attention in the HPC community. Such workloads, which we call *batched workloads*, appear in many large-scale scientific computing applications, such as quantum chemistry [4], sparse direct solvers [26], astrophysics [16], and signal processing [2]. In such applications, the operations required on each individual problem often involve performing dense matrix operations. Most high performance solutions use Basic Linear Algebra Subprograms (BLAS) for the bulk of their computations, especially Level-3 BLAS routines, which are compute intensive and rich in data reuse. It has been shown that using non-batched BLAS kernels usually achieves poor performance on such small sizes [1]. It is crucial, therefore, to develop high performance batched BLAS in order to further accelerate the applications relying on them.

Our focus in this paper is on the general case where matrices differ in size, which is the typical case in the aforementioned applications. We also focus on developments for Graphics Processing Units (GPUs), which are not trivial compared to similar developments for multicore CPUs. Considering batched workloads, multicore CPUs have the advantage of possessing large caches, where most of the problem sizes of interest would fit. In addition, it is reasonable to run parallel threads (e.g., using OpenMP), where each thread processes one problem at a time. Such configuration is expected to deliver high performance as long as fast single-thread execution is achieved. The latter can be obtained by running optimized vendor libraries such as Intel's Math Kernel Library (MKL) [10]. On the other hand, the same approach cannot be used for throughput-oriented hardware, such as GPUs. On the architecture side, individual threads on GPUs run much slower than CPU threads, and the capacities provided by the L1 cache/shared memory on the GPUs are very small compared to modern multicore CPUs. On the software side, most of the available GPU software focuses only on big matrices, where enough parallelism is available for the GPU to achieve high performance. These are the reasons behind the need for dedicated GPU kernels for batched workloads.

This paper presents a complete set of Level-3 BLAS routines designed specifically for batched workloads on GPUs. The developed kernels share common design concepts that are capable of dealing with matrices of different sizes within the same computational kernel. The routines that encapsulate such kernels are called *vbatched* routines. We present discussions about the general design concepts for batched computation, data layout, and optimization techniques for every kernel. An important aspect of the proposed design is the reliance on the matrix-matrix multiplications (GEMM) kernel in almost all BLAS routines. This approach not only reduces the

ICS '17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5020-4/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079103

code base required for the development, but also achieves high performance for all routines, thanks to the GEMM kernel being a common target for continuing research, development, and tuning [21] [13] [3]. In fact, any performance improvement done to the GEMM kernel would automatically propagate to almost all other routines in Level-3 BLAS. Due to the lack of similar competitive software, we compare against an implementation that submits individual cuBLAS [18] kernels into concurrent execution queues (i.e. CUDA streams), as well as against a CPU implementation that is based on the MKL library [10]. Significant speedups are achieved against both implementations using a system with a Pascal generation GPU (Tesla P100) and two 10-core Intel Haswell CPUs. All the developed kernels have been released in the MAGMA library [9].

The rest of the paper is organized as follows. Section 2 highlights related work about development of batched routines. Section 3 discusses the abstract structure and organization of batched GPU kernels. Kernel drivers are also discussed in Section 4. Section 5 illustrates the design of our case study, which is the entire Level 3 BLAS routines. Performance results are presented in Section 6. The paper ends with a conclusion in Section 7.

2 RELATED WORK

GPUs have long been used in hybrid algorithms that solve dense matrix problems of relatively large sizes [22]. Compute intensive tasks (e.g. GEMMs in trailing matrix updates [24]) are offloaded to the GPU, while latency sensitive tasks (e.g. panel factorization) are performed using the CPU. This kind of algorithmic design does not work well for small problems, due to the lack of parallelism, which fails to overlap any CPU-GPU communication. Therefore, GPU-based solution for small matrix computation should not involve hybrid CPU-GPU solutions.

There have been early efforts dealing with many small problems on GPUs. Small LU factorizations were investigated by Villa et al. [19, 20] (for size up to 128), and Wainwright [25] (for sizes up to 32). Kurzak et al. also showed batched Cholesky factorization in single precision for sizes up to 100 [12]. The size limits in these contributions arise from a customized design that does the entire factorization of one matrix using one CUDA thread block (TB). While the idea of fusing all computational steps into one TB pays off for very small matrices, such design approach fails to work for the midrange sizes (e.g. up to 512) due to the hardware constraints on the resources available per TB (e.g. running out of shared memory or requiring too many threads beyond the TB maximum capacity). Such constraints motivated our more generic approach, which is to have the building blocks (i.e. the BLAS kernels) developed specifically for batched workloads. While some efforts focus on fixed size problems[8] [6] [7], we consider the generic case of having problems of different sizes, which is of particular importance to some applications such as sparse direct solvers[26]. In this paper, we pay attention to Level-3 BLAS routines, which are at the core of any matrix factorization/solve algorithm.

3 ABSTRACT DESIGN CONCEPTS

3.1 Unified Kernel Structure

The design of our solution uses advanced template techniques for better reusability, portability and adaptability of the code. All the developed routines have a unified kernel structure that is independent from the operation being performed, so that it can be used for any kind of batched operations other than the ones discussed in this paper later on.

In general, a batched kernel is organized as a 3D grid of TBs, so that the grid configuration is (G_x, G_y, G_z), and each thread block has an ID (B_x, B_y, B_z). The dimension G_z is always set to the number of problems (which we refer to as batchCount). The first two dimensions of the grid are called "*subgrid*" dimensions. These dimensions are calculated based on the input problem size and possibly some other tuning parameters of the kernel itself. For example, all of the kernels discussed in this paper use a 2D subdivision of the input and output matrices. The subgrid dimensions (G_x, G_y) are then calculated based on the problem size(s), and the subdivision (blocking) size. In other words, the grid is internally reorganized as an array of 2D subgrids, where each subgrid represents one problem configuration. While subgrids are independent from each other, they all share the same device code that is written to perform the computation of one problem. Eventually, each subgrid executes the same code, but a unique problem. Figure 1 shows the abstract structure of a batched GPU kernel.

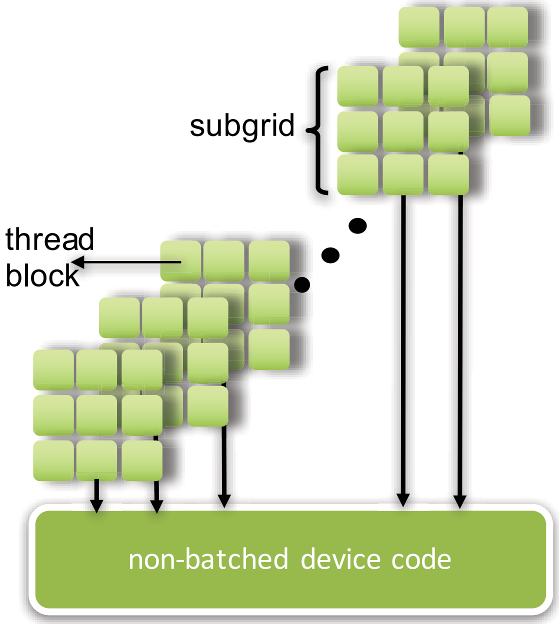


Figure 1: Abstract structure of a batched GPU kernel.

3.2 Adaptive Subgrid Truncation (ASGT)

The design shown in Figure 1 shows that all subgrids have the same configuration. In fact, the CUDA runtime does not allow launching subgrids of different sizes, which means that we cannot define as many values for (G_x, G_y) as the number of problems. In order to support operating on problems of different sizes, the proposed solution sets $G_x = \texttt{MAX}(G_{x\_1}, G_{x\_2}, \cdots, G_{x\_\texttt{batchCount}})$, and $G_y = \mathtt{MAX}(G_{y\_1}, G_{y\_2}, \cdots, G_{y\_\mathtt{batchCount}})$. This configuration ensures that each subgrid can accommodate the problem assigned to it. However, the size of the assigned problem may require a smaller subgrid. In order to avoid unnecessary resource allocation, we propose an **A**daptive **S**ub**G**rid **T**runcation (ASGT) technique. The ASGT technique is a lightweight preprocessing step that is executed by every CUDA thread in the subgrid. This layer reads the local size(s) of the assigned problem, and determines exactly how many TBs the problem needs. This is done by computing the corrected values $(\bar{G_x}, \bar{G_y})$ of the subgrid. Based on the corrected values, any thread block having $B_x \geq \bar{G_x}$ or $B_y \geq \bar{G_y}$ is terminated immediately before any computation takes place. Note that in any case, $\bar{G_x} \leq G_x$ and $\bar{G_y} \leq G_y$, and that the values of $(\bar{G_x}, \bar{G_y})$ are homogeneous for all threads belonging to the same subgrid.

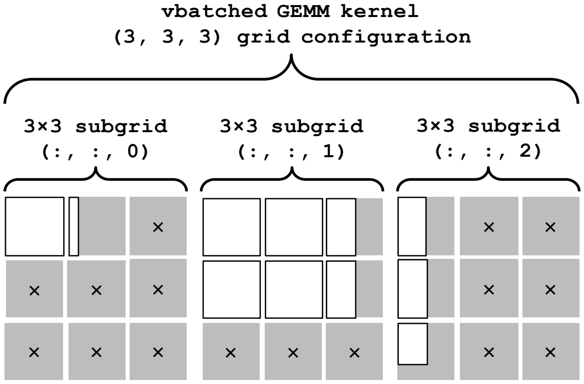


Figure 2: Example of vbatched GEMM using ASGT.

Figure 2 shows an example of the ASGT technique when executed in a variable size batched `GEMM` operation, where the value of `batchCount` is 3. The figure only shows the output matrices. According to the explanation above, this leads to $G_z = 3$. The `GEMM` kernel driver computes G_x and G_y based on the dimensions of the output matrices and the blocking size, leading to $G_x = G_y = 3$ in our example. Upon the kernel start, the ASGT layer is executed, terminating all thread blocks marked by '**×**' for smaller problems. The final $(\bar{G_x}, \bar{G_y})$ values for the three subgrids are $(1, 2)$, $(2, 3)$, and $(3, 1)$.

4 KERNEL LAUNCHER

The kernel launcher is a driver code that is executed on the host CPU. It performs any setup necessary for having safe launch on the GPU side. While we always assume that all the data exist in the GPU memory before the launch, our design is generic in the sense that it does not require individual problems to be stored consecutively in memory, neither does it assume that they should be equidistant from each other. Our design assumes that problems can be scattered totally randomly in the GPU memory space. This comes at the cost of providing pointer arrays to the GPU kernel to locate the data.

While the kernel launcher is executed on the CPU, there are two cases where a preprocessing GPU kernel might be launched prior to the launch of the computational kernel:

(1) Argument Checking: The purpose of argument checking is to avoid any run time errors during execution. For example, in our case study of BLAS routines, we can check the input arguments to make sure a valid operation is requested. Since problem sizes are stored in the GPU memory, it becomes so expensive to make the CPU check for arguments, since it will have to copy the array of sizes from the GPU, loop over and perform the check. Thus, it is the GPU which has to perform the checking. As a result, customized GPU kernels were developed to perform the necessary checks. Such kernels report the checking result back to the CPU, which either launches the computational kernel, or inform the user about the errors found in the input arguments. While the kernel is lightweight, the CPU-GPU communication can impose some overhead, especially if the batch is relatively small, but in any case it remains way less expensive than the CPU checking.

(2) Kernel Configuration: Since the CPU is responsible for the kernel launch, it needs to compute $(\boldsymbol{G_x}, \boldsymbol{G_y})$ as described in Section 3. Therefore, the CPU computes such values according to the maximal sizes across all problems. The preprocessing step of finding these maximum values can be done at the same time with the arguments checking operation inside the same preprocessing kernel.

We observe that the overhead of the preprocessing steps is minor in most cases. However, if the arguments are guaranteed to be correct, and/or the maximum values are predetermined, there is no need to launch the preprocessing GPU kernel.

5 DESIGN CONCEPTS AND DETAILS

This section provides a detailed illustration of the design of the vbatched Level-3 BLAS routines. Table 1 summarizes basic information about these routines. While all routines support different settings of the operation (e.g. transposed matrices in `GEMM`), the table shows one setting per kernel as an example. The developed kernels have the same behavior, and support all the different settings of BLAS compliant routines. We also assume that all the matrices are stored in the GPU memory using a column-major layout.

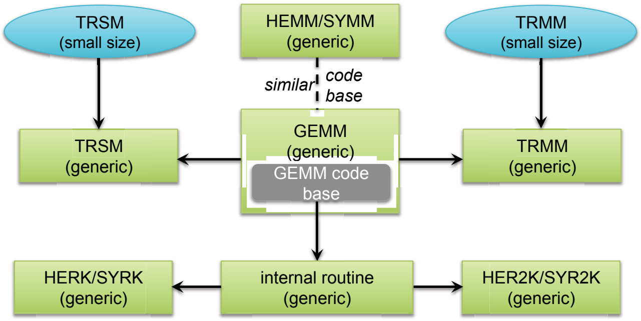


Figure 3: Overall design of BLAS-3 routines.

Abbreviation	Routine Description	Sample operation (double precision)	Notes
GEMM	General Matrix Matrix Multiplication	Compute: $(C = \alpha A \times B + \beta C)$	-
HERK	Hermitian rank-k update	Compute: $(C = \alpha A \times A^H + \beta C)$	C Hermitian
HER2K	Hermitian rank-2k update	Compute: $(C = \alpha A \times B^H + \bar{\alpha} B \times A^H + \beta C)$	C Hermitian
HEMM	Hermitian Matrix Matrix Multiplication	Compute: $(C = \alpha A \times B + \beta C)$	A Hermitian
TRMM	Triangular Matrix Matrix Multiplication	Compute: $(B = \alpha A \times B)$	A Triangular
TRSM	Triangular solve	Solve for X: $(A \times X = \alpha B)$	A Triangular

Table 1: Description of the Level-3 BLAS routines. All matrices are general unless those mentioned in the last column. The variables α and β are scalars, with $\bar{\alpha}$ being the conjugate of α.

5.1 Centralized Design Strategy

One goal of the proposed design is to reduce the code base as much as possible. Among all routines, the GEMM kernel is the most important kernel. In fact, it has been studied extensively for performance optimization and tuning across different architectures ([11, 13–15, 17, 21, 23]). This is because GEMM is the main performance key for LAPACK algorithms. In a similar manner, we designed Level-3 BLAS routines to extract their high performance from the GEMM kernel, so that any future performance improvement in this particular kernel will positively impact the performance of the other routines.

Figure 3 shows an overview of the BLAS-3 routine design, where every routine is dependent on the GEMM kernel. Following the abstract structure of Figure 1, the GEMM implementation consists of a code base, written using CUDA device routines, wrapped into a CUDA kernel. A device routine performs one GEMM operation, meaning that they do not have a *batched* interface. They are generic GEMM functions that can be invoked from within a CUDA kernel. The wrapping kernel takes care of assigning subgrids into different problems, where each subgrid is assigned a unique batch_id. The subgrid uses its ID to read the corresponding pointers, sizes, and leading dimensions of a specific problem from the batch. Once these arguments are read, they are passed into the device routines to start the computational part of the kernel. The idea of separating the GEMM code base from the wrapping kernel is very beneficial, since both the HERK and the HER2K kernels call the GEMM device routines rather than the kernels. The HEMM kernel is the only routine that does not share any code base with GEMM. However, its design and implementation is quite similar to GEMM. The TRMM and TRSM routines are developed using a recursive approach, which breaks down the operation into a TRMM/TRSM of a small size problem that fits into the GPU shared memory, plus a number of GEMM updates. Apart from the HEMM routines, all the operations either invoke a GEMM device routine from within its kernel, or call the vbatched GEMM routine as it is.

5.2 Matrix Multiplication (GEMM)

The main design idea of the GEMM kernel is to subdivide the output matrix C into square or rectangular blocks, where each block is computed by one TB. As shown in Figure 4, a TB processes a block row of A, in steps of ($\mathrm{BLK_M}$×$\mathrm{BLK_K}$) blocks, and a block column of B, in steps of ($\mathrm{BLK_K}$×$\mathrm{BLK_N}$) blocks, to compute a single ($\mathrm{BLK_M}$×$\mathrm{BLK_N}$) block of C. The kernel uses register blocking to hold three blocks of A, B, and C in the register file. As the TB moves across A and B, new blocks of A and B are read in the registers, while the C block is kept for result accumulation. The multiplication between blocks takes place in shared memory, which allows data prefetching of the next A and B blocks in registers. The kernel has at least five tuning parameters, which are $\mathrm{BLK_M}$, $\mathrm{BLK_N}$, $\mathrm{BLK_K}$, and the (x, y) configuration of TBs.

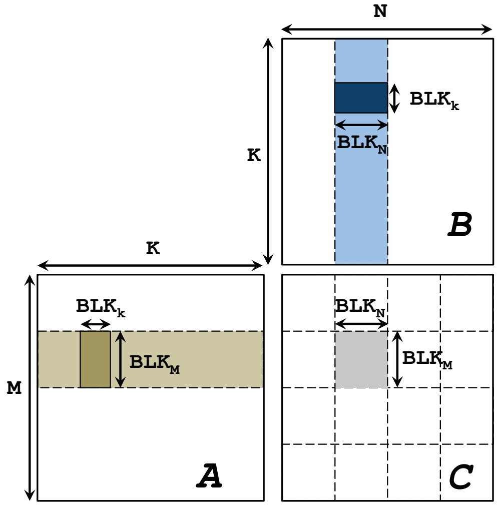


Figure 4: GEMM design.

5.3 GEMM Autotuning

We conducted a comprehensive autotuning experiment particularly for the GEMM kernel. The main goal of this experiment is to have the best performance on the GPU, not only for square matrices, but also for realistic scenarios where the GEMM kernel is called within a higher-level algorithm such as batched one-sided factorization.

Our autotuning framework starts by fully expanding the search space, enumerating all possible values of the tuning parameters. It then proceeds to prune the search space according to two sets of constraints. The first is the *hard constraints* defined by the target hardware. For example, the CUDA runtime does not allow more than 1024 threads per TB, or more than 2048 threads on a multiprocessor (on a modern post-Kepler GPU). The second set of constraints are *soft constraints* that are defined based on our experience with GPU computing. For example, we consider blocking sizes that are power of 2, or at least multiples of 16 and 32. We also consider tuning parameters that have relatively few threads per TB (e.g. less than 256). This decision allows scheduling more TBs on the same multiprocessor, which leads to better occupancy and execution throughput.

Finally, we defined a set of realistic test cases that are commonly found in higher-level algorithms, especially those found in LAPACK algorithms. For each test case, the autotuning framework automatically records the tuning parameters of the top five performing kernel instances. We then aggregate the results and try to find common winning kernel instances among different test cases. Following a C++ template-based design, the kernel launcher eventually has access to a database of winning kernel instances, so that it can choose the best kernel instance based on the problem configuration.

5.4 Hermitian Rank Updates (HERK and HER2K)

As shown in Figure 3, the Hermitian rank updates depend on a custom internal routine that inherits the GEMM code base. This routine does not exist in the standard BLAS, but it helps realize two of the standard routines. The internal routine computes the upper or the lower triangular part of a matrix C, where ($C = \alpha P \times Q^H + \beta C$) for a non-transposed operation or ($C = \alpha P^H \times Q + \beta C$) for a transposed operation. The kernel starts with the same 2D thread block configuration as the GEMM kernel, except that TBs that correspond to the unreferenced part of C are terminated immediately using the ASGT technique. To provide the HERK functionality (Table 1), the internal routine is invoked with $P = Q = A$. The HER2K operation requires two calls to the internal routine. Considering a non-transposed operation, the first call has $P = A$ and $Q = B$, so that $C_1 = \alpha A \times B^H + \beta C$. The second call has $P = B$, $Q = A$, and $C = C_1$. It also uses the conjugate of α and sets $\beta = 1$. The result is $C_2 = \bar{\alpha} B \times A^H + C_1$. By substituting C_1 into the latter equation, the final result is $\alpha A \times B^H + \bar{\alpha} B \times A^H + \beta C$, which is the standard HER2K operation. Eventually, three of the batched BLAS kernels share the exact same code base (GEMM, HERK, and HER2K).

5.5 Hermitian Matrix Multiplication (HEMM)

The HEMM routine has a similar design to the GEMM kernel, except for reading the Hermitian matrix A. Consider Figure 5, where the lower triangular part of the Hermitian matrix A is multiplied by B. Instead of reading a block row of A, a TB reads a partial block row and a partial block column. In general, there are three phases in the lifetime of every TB. The first is a non-transposed multiplication where the multiplication between A and B blocks occur exactly like a GEMM operation. This corresponds to the partial block row of A before the diagonal. The second and third phases involve some preprocessing performed on A blocks before multiplication takes place. The second phase reads the diagonal block of A, mirrors it along its diagonal in shared memory, and then proceeds normally with multiplication. The third phase switches the direction of the TB to move vertically through A. In this phase, blocks from A are conjugate-transposed in shared memory before doing the usual multiplication. Note that the processing of matrices B and C is the same as the GEMM kernel. We also point out that blocking of the matrix A is restricted to square blocks only, which is a typical design choice for Hermitian matrices.

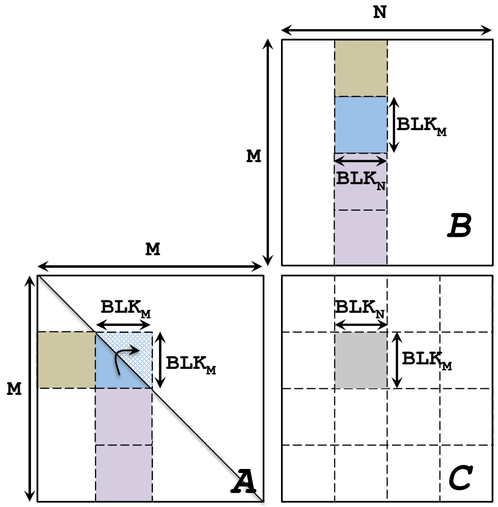


Figure 5: HEMM design.

5.6 Triangular Matrix Multiplication (TRMM)

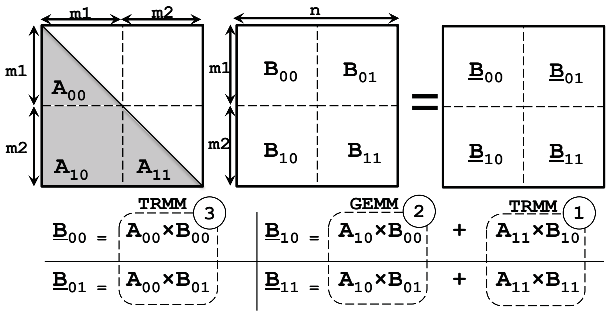
The TRMM routine cannot be designed similar to any of the previous kernels because the operation is done in-place, meaning that the result of the multiplication overwrites the matrix B (refer to Table 1). Therefore, we use a different design strategy that relies on a recursive implementation, similar to what has been done for non-batched kernels [5]. We developed a TRMM kernel that works only on small triangular matrices that fit in the GPU shared memory. Figure 6 shows a simplified version of the kernel for a lower unit-triangular matrix A that is multiplied by an NB×NB general matrix B. The kernel does the operation in-place because A can be stored entirely in shared memory. The size of B is not a concern, because it can be subdivided among independent TBs. This kernel is invoked only if the size of A is ≤ NB, which is a tuning parameter.

In order to support any matrix size, we adopt a recursive implementation that is shown in Figure 7. The recursive TRMM routine checks for the triangular matrix size. If it is less than or equal to NB, it reaches a stopping condition and invokes the TRMM kernel. Otherwise, the matrix A is subdivided as shown in Figure 7. We begin by calling the recursive TRMM routine with respect to A_{11} and update B_{1x} in-place (step 1). We then call the GEMM routine to compute the remaining portion for B_{1x} (step 2). The final step is to invoke the recursive TRMM routine with respect to A_{00} and update B_{0x} in-place. Note that these three steps have to be performed in the specified order. For example, exchanging steps 1 and 2 destroys the original data in B_{1x}, which is needed in step 1. Also performing step 3 before 2 overwrites B_{0x}, whose original values are needed to perform the GEMM operation in step 2. We point out that the subdivision of A does not have to be even. In fact, the subdivision follows a tuning experiment (not the scope of this paper), which

```
template<typename T, const int NB>
static __device__
void trmm_lNU_template_device(
        enum uplo, enum diag,
        T alpha, T* A, int ldda,
                 T* B, int lddb)
{
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;

    __shared__ T sA[NB * NB];
    __shared__ T sB[NB * NB];
    T rb = make_zero<T>();
    B += bx * NB * lddb;

    // load A and B
    sA[ty * NB + tx] = A[ty * ldda + tx];
    sB[ty * NB + tx] = B[ty * lddb + tx];

    // ignore diagonal elements
    if(ty == tx)
        sA[ty * NB + tx] = make_one<T>();

    // ignore upper triangle
    if(tx < ty)
        sA[ty * NB + tx] = make_zero<T>();
    __syncthreads();

    // multiply
    #pragma unroll
    for(int i = 0; i < NB; i++)
        rb += sA[i * NB + tx] * sB[ty * NB + i];
    rb *= alpha;
    // write B
    B[ty * lddb + tx] = rb;
}
```

Figure 6: A sample of the TRMM device routine in CUDA.

tends to make the execution time mainly dominated by the GEMM operation of step 2.


m1
m2
n
m1
m2
m1
m2
A_{00}
A_{10}
A_{11}
B_{00}
B_{01}
B_{10}
B_{11}
=
$\underline{B}_{00}$
$\underline{B}_{01}$
$\underline{B}_{10}$
$\underline{B}_{11}$
TRMM 3
GEMM 2
TRMM 1
$\underline{B}_{00} = A_{00} \times B_{00}$
$\underline{B}_{01} = A_{00} \times B_{01}$
$\underline{B}_{10} = A_{10} \times B_{00} + A_{11} \times B_{10}$
$\underline{B}_{11} = A_{10} \times B_{01} + A_{11} \times B_{11}$

(1) $[\underline{B}_{10}\ \underline{B}_{11}] = A_{11} \times [B_{10}\ B_{11}]$ (recursive TRMM)

(2) $[\underline{B}_{10}\ \underline{B}_{11}] \mathrel{+}= A_{10} \times [B_{00}\ B_{01}]$ (GEMM)

(3) $[\underline{B}_{00}\ \underline{B}_{01}] = A_{00} \times [B_{00}\ B_{01}]$ (recursive TRMM)

Figure 7: TRMM design.

5.7 Triangular Solve (TRSM)

Figure 11 shows an example for a triangular solve problem, where the solution $\boldsymbol{X}$ overwrites the matrix $\boldsymbol{B}$. Similar to TRMM, this is an in-place operation. We present two different approaches to address this problem. The first one is inspired by the non-batched TRSM kernel in the MAGMA library[9]. Such design is based on inverting the NB×NB triangular matrices that reside on the block diagonal of $\boldsymbol{A}$. The value of NB is a tuning parameter. The inverses are stored in a separate workspace. The routine loops over the matrix $\boldsymbol{A}$ in steps of NB, where at each iteration, it multiplies an inverted triangular block with the corresponding right hand side (RHS), which produces part of the solution matrix $\boldsymbol{X}$. The partial solution is then used to update the unsolved part of the matrix. Figures 8 and 9 show an example for realizing TRSM using matrix inversion. There are few sources of overheads that add up to the execution time of this approach with respect to variable size batched workloads. The first is that it needs allocation and initialization of internal workspaces, which are used to store the inverses. The second is that figuring out the total amount of workspace needed is not trivial. Since matrices are assumed to have different sizes, each matrix requires a different amount of workspace. In particular, the total amount required is $\sum_{i=1}^{\texttt{batchCount}} \left\lceil \frac{m_i}{\texttt{NB}} \right\rceil \times \texttt{NB}^2$, which requires a reduction operation to take place in order to compute the correct workspace. In addition, the allocation of the workspace is followed by assigning a sub-workspace pointer for each problem in the batch. Since the displacement between two adjacent sub-workspaces is no longer uniform, a dedicated preprocessing kernel was developed to correctly set the sub-workspace pointer for each problem. The third source of overhead is that all the multiplications in Figure 9 are performed through GEMM, which means that the computation is done out-of-place. An additional workspace is used to store the solution $\boldsymbol{X}$, which is then copied back (using device memory copies) to overwrite $\boldsymbol{B}$. Despite these overheads, this approach is very competitive in performance when the problem size increases, as the impact of the overheads get smaller.

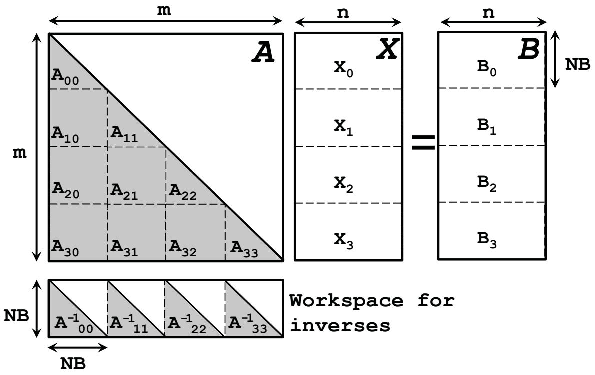


Figure 8: TRSM design.

We also propose a second approach that uses an exact triangular solve (no inversions), and performs the operation in-place, with no need to workspaces or memory copies. The kernel is based on a recursive approach similar to the TRMM routine. We developed a TRSM kernel which assumes that the triangular matrix $\boldsymbol{A}$ fits into shared memory. The kernel also uses register blocking to store the right hand sides. Independent TBs solve different right hand sides

```
for $i = 0$ to 3 do
    $X_i \leftarrow A_{ii}^{-1} \times B_i$;
    for $j = (i + 1)$ to 3 do
        $B_j \leftarrow B_j - A_{ji} \times X_i$;
    end for
end for
```

Figure 9: Pseudocode for computing X in the TRSM shown in Figure 8.

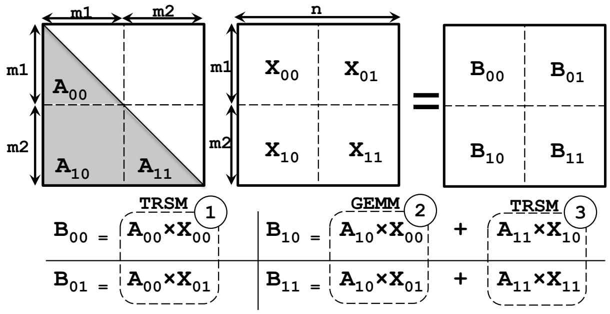
through in-register data exchanges that use shuffle operations. A simplified example for a unit lower triangular matrix of size NB is shown in Figure 10. Larger triangular matrices are handled using a recursive implementation that is shown in Figure 11. Similar to the TRMM routine, the order specified in the Figure has to be followed in order to produce the correct solution X. The advantage of the recursive approach is that it does not have any overheads. First, it does not require any workspace allocation/initialization. Second, the solution is done in-place, which means that no memory copies are required. Another advantage is that the recursive subdivision is more flexible than the blocked implementation shown in Figure 8. The values m_1 and m_2 follow a tuning experiment that aims to maximize the time spent performing GEMM. On the other side, the blocked implementation restricts the stepping to be equal to the size of the inverted diagonal blocks (NB).

6 PERFORMANCE RESULTS

Performance experiments are conducted on a machine equipped with two 10-core Intel Haswell CPUs (Intel Xeon E5-2650 v3, running at 2.30 GHz), and a Pascal generation GPU (Tesla P100, running at 1.189 GHz). Results are shown for real double precision on batches of 2000 matrices. Because of the lack of competitive vbatched routines on the GPU, we compare our proposed solution (MAGMA) against cuBLAS called within concurrent CUDA streams. GPU performance tests use CUDA Toolkit 8.0. CPU performance tests use Intel MKL Library 11.3.0, with one core assigned per matrix at a time. We use an OpenMP parallel for pragma with dynamic loop scheduling to balance the workload among cores. The sizes in every test batch are randomly sampled within the range $[1 : \bar{x}]$, where $\bar{x}$ represents an arbitrary point on the x-axis of the performance graphs. MAGMA has been timed with the overheads mentioned in Section 4 included. While we show performance results for double precision only, the developed solution runs on all the four precisions of the reference BLAS implementation.

```
template<typename T, const int NB>
static __device__
void trsm_lNL_template_device(
        enum diag, int m, int n,
        T alpha, T* A, int ldda,
                 T* B, int lddb)
{
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;
    B += bx * NB * lddb;

    __shared__ T sA[NB * NB];
    T rB = make_zero<T>();

    // load A
    sA[ty * NB + tx] = A[ty * ldda + tx];

    // ignore diagonal elements
    if(ty == tx){
        sA[tx * NB + tx] = make_one<T>();
    }
    __syncthreads();

    // load B
    rB = alpha * B[ty * lddb + tx];

    // solve
    #pragma unroll
    for(int i = 0; i < NB; i++){
        if(tx == i) rB *= sA[i * NB + i];
        T rT = xshfl__(rB, i, NB);
        if(tx > i) rB -= sA[i * NB + tx] * rT;
    }
    // write B
    B[ty * lddb + tx] = rB;
}
```

Figure 10: A sample of the TRSM device routine in CUDA. The problem is a simplified example for an NB×NB unit lower triangular matrix and NB right hand sides.

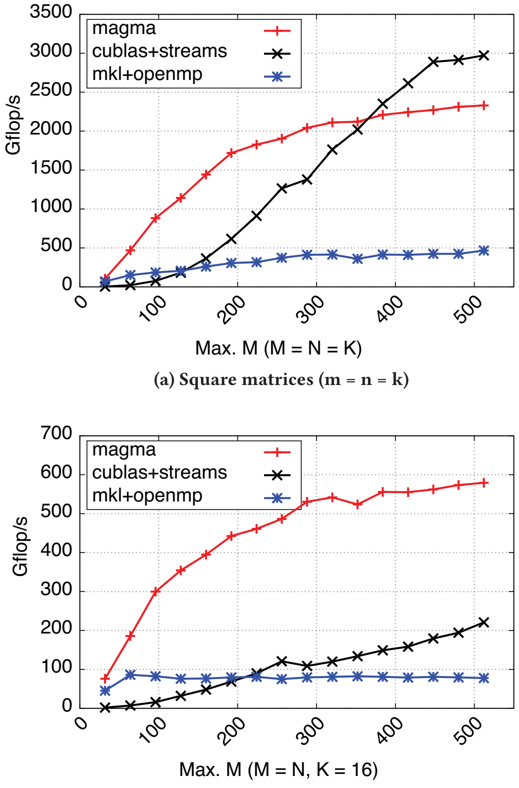


(1) $[B_{00}\ B_{01}] = A_{00} \times [X_{00}\ X_{01}]$ (recursive TRSM)
(2) $[B_{10}\ B_{11}] -= A_{10} \times [X_{00}\ X_{01}]$ (GEMM)
(3) $[B_{10}\ B_{11}] = A_{11} \times [X_{10}\ X_{11}]$ (recursive TRSM)

Figure 11: TRSM design.

Figure 12a shows the performance of the DGEMM kernel when launched on square problems, which can be regarded as an upper bound for the performance of all other routines. We observe that, at sizes around 350, concurrent cuBLAS launches into independent streams outperforms MAGMA. The reason is that cuBLAS has a highly optimized kernel that is based on an optimal algorithm written in the native machine language [21]. For example, at size 384, which is the approximate intersection point, the cuBLAS kernel achieves 1.25 Tflop/s on one problem of size 384×384, while the MAGMA scores 0.45 Tflop/s. However, MAGMA is still competitive at this point because it aggregates the work of all matrices into one computational kernel, maximizing occupancy and minimizing the overhead of launching concurrent kernels. On the left of the intersection point, the MAGMA DGEMM scores speedups against cuBLAS that range from 1.3× to more than 10× (on sizes less than

100). The MAGMA kernel also maintains an asymptotic speedup of more than 4.5× against the CPU implementation.



(a) Square matrices (m = n = k)

(b) Rectangular matrices (m = n, k = 16)

Figure 12: Performance of vbatched DGEMM.

Figure 12b shows a more realistic test case for batched workloads. Consider an LU factorization with a panel width equal to 16, during which trailing matrix updates involve calls to the GEMM kernel with k = 16 ($C_{m\times n} - = A_{m\times k} \times B_{k\times n}$), and $\boldsymbol{m} = \boldsymbol{n}$ if the factorized matrices are square. For such practical workloads, the MAGMA DGEMM is much faster than cuBLAS, achieving speedups from 2.6× up to more than 20× on sizes less than 100. A significant speedup of approximately 6× is also achieved against the CPU.

Similar to LU factorization, trailing matrix updates in the Cholesky factorization algorithm call the HERK routine with a relatively small k ($C_{n\times n} - = A_{n\times k} \times A^{H}{}_{k\times n}$). Figure 13 shows a test case with k=32. Thanks to the GEMM dependent design, the MAGMA kernel is 2.8× to 25× faster than cuBLAS. It is also up to 7.3× faster than MKL. We point out that since the HER2K routine shares the same internal routine with HERK, it has a nearly identical performance to HERK, and so is not shown to avoid duplication.

Figure 14 shows the performance of the DSYMM kernel on square matrices. We first observe that the asymptotic performance of the MAGMA implementation is about 85% of the DGEMM performance in Figure 12a, since both kernels have very similar design, except for the processing of the Hermitian matrix. The speedups scored against cuBLAS ranges from 1.7× up to more than 7× as the sizes get smaller than 100. In addition, our solution is asymptotically 4× faster than MKL.

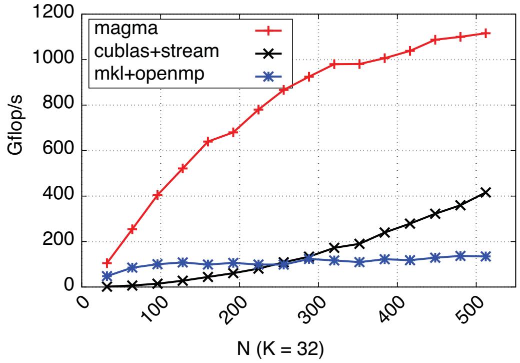


Figure 13: Performance of vbatched DSYRK (n is random, k = 32).

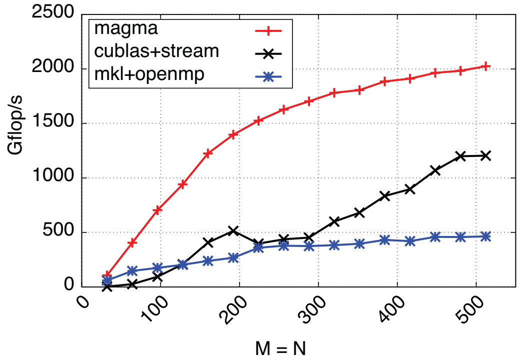


Figure 14: Performance of vbatched DSYMM.

In Figure 15, we observe that the performance of the DTRMM kernel is generally less than the performances of the DGEMM and the DSYMM kernels, in Figures 12a and 14, respectively. The reason behind this behavior is that the TRMM operation is done in-place (e.g., $\boldsymbol{B} = \boldsymbol{A} \times \boldsymbol{B}$), which imposes a certain order of execution (Figure 7) to respect the data dependency. This is unlike the embarrassingly parallel design of GEMM and HEMM, where each TB is totally independent from other TBs. The MAGMA kernel is 1.2× to 6.5× faster than cuBLAS. It also scores 3× asymptotic speedups against the CPU implementation.

We show two performance graphs for the DTRSM kernel. The first is shown in Figure 16a, which considers the square problem: ($A_{m\times m} \times X_{m\times n} = B_{m\times n}$, $\boldsymbol{m} = \boldsymbol{n}$). Such a problem can be considered as an upper bound test. In Figure 16a, we observe that the two proposed approaches are asymptotically close to each other, with the *MAGMA w/solve* routine being slightly better. This means that the overheads associated with the *MAGMA w/inv* routine are minor if the problem sizes are not very small. The *MAGMA w/solve* routine is 2× to 12× faster than cuBLAS. It is also 2.5× faster (asymptotically) than MKL. Figure 16a also shows that the performance graph of the *MAGMA w/solve* kernel has a similar shape, but less performance

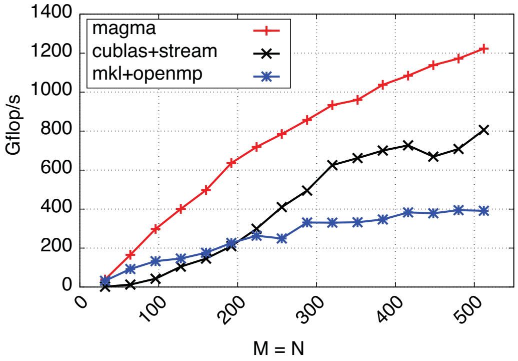


Figure 15: Performance of vbatched DTRMM.

than the DTRMM, although the two routines have a nearly identical design. The reason is that the TRSM kernel for small matrices (Figure 10) imposes a more serial execution than the corresponding TRMM code in Figure 6, which is basically a matrix multiplication in shared memory.

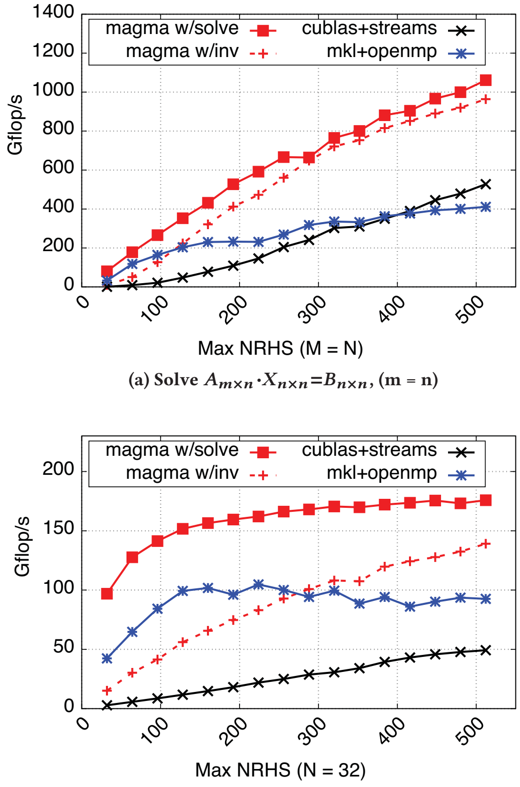
The second TRSM test (Figure 16b) is a more practical test case that is found in Cholesky factorization of the lower triangular part of a Hermitian matrix, which is of the form $(X_{m\times n}\times A^{H}{}_{n\times n}=B_{m\times n})$. In batched workloads, we use a small value of n. Figure 16b shows an example with $n = 32$. In Figure 16b, however, we observe a larger gap between the *MAGMA w/solve* and the *MAGMA w/inv* routines. Since the figure represents smaller problems, the overheads associated with the *MAGMA w/inv* routine become more significant. We also observe more competition from the CPU, which by nature achieve very good performance in serial executions of small problems. The *MAGMA w/solve* routine is 3.5× to more than 15× faster than cuBLAS. It also maintains a 1.8× asymptotic speedup against the CPU implementation using MKL.

7 CONCLUSION

This paper introduced an abstraction for batching execution of many small problems of different sizes using GPUs. As a case study, the paper applies the abstract design concepts to the complete set of Level-3 BLAS kernels. Thanks to a carefully designed and thoroughly tuned GEMM kernel, and a GEMM-concentric design, the proposed kernels achieve high performance on a modern GPU and are significantly faster than other state-of-the-art approaches. Future directions include applying the same design concepts for more routines, such as one-sided factorizations and singular value decomposition, and studying the impact of such batched routines on real applications.

ACKNOWLEDGEMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The work was also partially supported by Nvidia and NSF under grant No. **1514406**.



(a) Solve $A_{m\times n}\cdot X_{n\times n}=B_{n\times n}$, (m = n)

(b) Solve $X_{m\times n}\cdot A_{n\times n}=B_{m\times n}$, (n = 32)

Figure 16: Performance of vbatched DTRSM.

REFERENCES

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. 21–38. DOI: http://dx.doi.org/10.1007/978-3-319-41321-1_2

[2] M.J. Anderson, D. Sheffield, and K. Keutzer. 2012. A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*.

[3] Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. 2015. Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5096–5113. DOI: http://dx.doi.org/10.1002/cpe.3516

[4] Alexander A Auer, Gerald Baumgartner, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Luc, Marcel Nooijene, Russell Pitzerf, J Ramanujamg, P Sadayappanc, and Alexander Sibiryakovc. 2006. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics* 104, 2 (2006), 211–228.

[5] Ali Charara, Hatem Ltaief, and David E. Keyes. 2016. Redesigning Triangular Dense Matrix Computations on GPUs. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*. 477–489. DOI: http://dx.doi.org/10.1007/978-3-319-43659-3_35

[6] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA* 29, 2 (2015), 193–208. DOI: http://dx.doi.org/10.1177/1094342014567546

[7] Azzam Haidar, TingxingTim Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. 2015. A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Lecture Notes in Computer Science, Vol. 9137. Springer International Publishing, 31–47. DOI:

http://dx.doi.org/10.1007/978-3-319-20119-1_3

[8] Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. 2015. Towards Batched Linear Solvers on Accelerated Hardware Platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, ACM, San Francisco, CA.

[9] Innovative Computing Laboratory at the University of Tennessee 2014. Matrix Algebra on GPU and Multicore Architectures (MAGMA). (2014). Available at http://icl.cs.utk.edu/magma/.

[10] Intel Corporation 2016. Intel Math Kernel Library. (2016). Available at http://software.intel.com/intel-mkl/.

[11] Chetan Jhurani and Paul Mullowney. 2013. A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices. *CoRR* abs/1304.7053 (2013). http://arxiv.org/abs/1304.7053

[12] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. 2015. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *Parallel and Distributed Systems, IEEE Transactions on* PP, 99 (2015), 1–1. DOI:http://dx.doi.org/10.1109/TPDS.2015.2481890

[13] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (November 2012), 2045–2057.

[14] Junjie Lai and Andre Seznec. 2013. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–10. DOI:http://dx.doi.org/10.1109/CGO.2013.6494986

[15] Y. Li, J. Dongarra, and S. Tomov. 2009. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*. Springer, Baton Roube, LA.

[16] O.E.B. Messer, J.A. Harris, S. Parete-Koon, and M.A. Chertkow. 2012. Multicore and Accelerator Development for a Leadership-Class Stellar Astrophysics Code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*.

[17] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.* 24, 4 (Nov. 2010), 511–515. DOI:http://dx.doi.org/10.1177/1094342010385729

[18] NVIDIA Corporation 2016. NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS). (2016). Available at https://developer.nvidia.com/cublas.

[19] Villa Oreste, Massimiliano Fatica, Nitin A. Gawande, and Antonino Tumeo. 2013. Power/Performance Trade-offs of Small Batched LU Based Solvers on GPUs. In *19th International Conference on Parallel Processing, Euro-Par 2013 (Lecture Notes in Computer Science)*, Vol. 8097. Aachen, Germany, 813–825.

[20] Villa Oreste, Nitin A. Gawande, and Antonino Tumeo. 2013. Accelerating Subsurface Transport Simulation on Heterogeneous Clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2013)*. Indianapolis, Indiana.

[21] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 35, 11 pages. DOI:http://dx.doi.org/10.1145/2063384.2063431

[22] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. 2010. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proc. of the IEEE IPDPS'10*. IEEE Computer Society, Atlanta, GA, 1–8. DOI: 10.1109/IPDPSW.2010.5470941.

[23] Vasily Volkov and James Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, Piscataway, NJ, USA, 1–11.

[24] Vasily Volkov and James W. Demmel. 2008. *LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs*. Technical Report UCB/EECS-2008-49. University of California, Berkeley. Also available as LAPACK Working Note 202.

[25] Ian Wainwright. April, 2013. Optimized LU-decomposition with Full Pivot for Small Batched Matrices. (April, 2013). http://on-demand.gputechconf.com/gtc/2013/presentations/S3069-LU-Decomposition-Small-Batched-Matrices.pdf GTC'13 – ID S3069.

[26] SENCER NURI YERALAN, TIMOTHY A DAVIS, SID-LAKHDAR WISSAM M, and SANJAY RANKA. 2015. Algorithm 9xx: Sparse QR Factorization on the GPU. *ACM Trans. Math. Software* (2015). http://faculty.cse.tamu.edu/davis/publications_files/qrgpu_revised.pdf