On the Development of Variable Size Batched Computation for Heterogeneous Parallel Architectures

Ahmad Abdelfattah*, Azzam Haidar*, Stanimire Tomov*, Jack Dongarra*†‡
{ahmad,haidar,tomov,dongarra}@icl.utk.edu
**Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA*
†*Oak Ridge National Laboratory, Oak Ridge, TN, USA*
‡*University of Manchester, Manchester, UK*

***Abstract*—Many scientific applications, ranging from national security to medical advances, require solving a number of relatively small-size independent problems. As the size of each individual problem does not provide sufficient parallelism for the underlying hardware, especially accelerators, these problems must be solved concurrently as a batch in order to saturate the hardware with enough work, hence the name *batched computation*. A possible simplification is to assume a uniform size for all problems. However, real applications do not necessarily satisfy such assumption. Consequently, an efficient solution for variable-size batched computations is required.**

This paper proposes a foundation for high performance variable-size batched matrix computation based on Graphics Processing Units (GPUs). Being throughput-oriented processors, GPUs favor regular computation and less divergence among threads, in order to achieve high performance. Therefore, the development of high performance numerical software for this kind of problems is challenging. As a case study, we developed efficient batched Cholesky factorization algorithms for relatively small matrices of different sizes. However, most of the strategies and the software developed, and in particular a set of variable size batched BLAS kernels, can be used in many other dense matrix factorizations, large scale sparse direct multifrontal solvers, and applications. We propose new interfaces and mechanisms to handle the irregular computation pattern on the GPU. According to the authors' knowledge, this is the first attempt to develop high performance software for this class of problems. Using a K40c GPU, our performance tests show speedups of up to $2.5\times$ against two Sandy Bridge CPUs (8-core each) running Intel MKL library.

***Keywords*-batched computation, GPUs, variable small sizes**

I. Introduction

The purpose of batched routines is to solve a set of independent problems in parallel. Such configuration arises in many real applications, including astrophysics [21], quantum chemistry [9], metabolic networks [19], CFD and resulting PDEs through direct and multifrontal solvers [31], high-order FEM schemes for hydrodynamics [10], direct-iterative preconditioned solvers [16], image [22] and signal processing [8]. If the matrix size is large enough to allow efficient use of the entire device, there is no benefit of using batched computation; it is preferred to solve the set of independent problems in serial fashion as a sequence of problems, to better enforce locality of data and increase the cache reuse. However, when matrices are small, the amount of work needed to perform the computation cannot saturate the device, either CPU or GPU, and thus there is a need for batched routines. In general, matrices do not necessarily have the same size.

The development of an efficient framework for batched matrix computation depends on the nature of the underlying hardware (latency oriented *vs.* throughput oriented). For multicore CPUs, the framework can be developed both easily and efficiently using the existing software stack. Since matrix sizes are relatively small, they can fit into the fast CPU cache, and vendor supplied libraries such as MKL [18] or ACML [6] can be used to achieve high performance. In addition, vectorization can be used to achieve even higher performance, either explicitly (the Intel Small Matrix Library [17]) or implicitly (intrinsic instructions and vectorized BLAS operations). If each individual matrix can fit into the cache, one CPU core can be assigned to solve one problem at a time. As we show in Section IV, this is a better practice than using all cores to work on a single matrix at a time. More details about batched computation using multicore CPUs can be found in [13].

By using properly designed and implemented batched operations, small problems can be solved two to three times faster on GPUs, and with four to five times better energy efficiency than on multicore CPUs alone (subject to the same power draw). Given the fundamental importance of numerical libraries to science and engineering applications, the need for libraries that can perform batched operations on small matrices has clearly become acute. The same approach cannot be used for GPUs, due to their throughput-oriented architecture. GPUs have small caches/shared memories that can host only tiny matrices. For example, a modern Kepler GPU has about 48KB of shared memory, which can host one matrix whose size is up to 78×78 in double precision (assuming that no other shared variables/workspaces are needed). In addition, most of the existing software infrastructure for GPUs target relatively large matrices to achieve high performance. As a result, there is a need to develop a dedicated framework for batched computation on GPUs, supporting both fixed and variable size problems.

This paper is a first attempt to extend the MAGMA [27] framework proposed in [13] in order to support variable

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.190

size batched computation. The proposed work focuses on a Cholesky factorization problem of the configuration shown in Figure 1. However, we emphasize that many of the kernels proposed in this paper create a foundation for the class of variable-size batched factorization and solve routines.

$$\begin{array}{l}
\texttt{xPOTRF}(A^{(1)}_{LDA_1 \times N_1}) \rightarrow L_1 L_1^\intercal \\
\texttt{xPOTRF}(A^{(2)}_{LDA_2 \times N_2}) \rightarrow L_2 L_2^\intercal \\
\ldots \\
\texttt{xPOTRF}(A^{(k)}_{LDA_k \times N_k}) \rightarrow L_k L_k^\intercal
\end{array}$$

Figure 1. Schematic view of a variable-size batched Cholesky factorization problem for a set of k dense matrices. Each matrix is assumed to have a different size and leading dimension.

The rest of the paper is organized as follows. Section II discusses some previous efforts in GPU-accelerated matrix factorizations, with a focus on batched routines. Section III presents a detailed description of the framework proposed to handle a variable-size batched matrix computation problem, taking Cholesky factorization as a case study. In Section IV, we show some performance results and the speedups achieved against other techniques. We also show how the performance of some routines is progressively improved. The paper ends with a conclusion in Section V.

II. Related Work

A problem size is usually a defining factor to decide which algorithm and hardware to use. In a batched problem, we have two factors, the size of each individual problem, and the batch size. For simplicity, we always assume that the batch size is large enough to fill up the resources of the hardware.

As mentioned earlier, sufficiently large matrices do not need a batched routine. The GPU can handle one matrix at a time based on prior work that proposes hybrid algorithms that use both the CPU and the GPU [28]. The motivation behind hybrid algorithms arises from the fact that GPU can not be used on panel factorizations as efficiently as on trailing matrix updates [29]. Since such updates are mostly gemms [4], [12], many hybrid algorithms perform the panel factorization on the CPU, while the updates are performed on the GPU. For small problems, however, hybrid algorithms lose efficiency due to lack of parallelism, especially in the trailing matrix updates which fail to hide the latency of both the panel factorization and the data movement between the CPU and the GPU.

There are some efforts that proposed batched computations on the GPU. For example, the work done by Villa et al. [25], [26] proposed batched LU factorization for matrices up to 128, where a single thread block solves one system at a time. Similarly, Wainwright [30] proposed a design based on using a single CUDA warp to perform LU with full pivoting on matrices of size up to 32. Dong et al. [11] proposed three different implementations for batched Cholesky factorization, where the performance was compared against a multicore CPU and the hybrid MAGMA [5] algorithm. Kurzak et al. [20] proposed an implementation and a tuning framework for batched Cholesky factorization for small matrices ($\leq$100) in single precision. Batched QR factorization has also been accelerated using GPUs [14], where several-fold speedup is obtained against a competitive design by CUBLAS [24]. Haidar et al. proposed common optimization techniques, based on batched BLAS kernels, that can be used for all one-sided factorizations (LU, QR, and Cholesky) using NVIDIA GPUs [13] [15]. However, all the aforementioned efforts focus on the case where all the matrices have the same size. This paper addresses the generic case of having different sizes in one batch.

III. Algorithmic Advancment and Design

We have been working closely with affected application communities to define modular, language agnostic interfaces that can be implemented to work seamlessly with the compiler, and be optimizable using techniques such as *code replacement* and *inlining*. The goal is to provide the developers of applications, and runtime systems with the option of expressing interfaces to batch computations as a single call to a routine, that would also allow the entire linear algebra (LA) community to collectively develop a wide range of small matrix problems. Success in such an effort will require innovations in interface design, computational and numerical optimization, as well as packaging and deployment at the user site to trigger final stages of tuning at the moment of execution. In this section, we describe a framework for the development of efficient kernels for batched computation of small matrices with different sizes. From now on, variable-size batched routines will be abbreviated as *vbatched routines*.

A. Interface of a Vbatched Routine

Batched routines for fixed-size matrices have an interface similar to the classic LAPACK [7] routines except that:

- Input matrices are passed as an array of pointers instead of just a pointer to a single matrix;
- The routine needs information about the batch count, thus an extra parameter is added.

However, if matrices do not have the same size, both the matrix sizes and the leading dimensions need to be passed (as arrays of integers). A vbatched routine assumes that each matrix has an independent size and leading dimension. A consequence of this modification is that any pointer displacement or any simple arithmetic operation on the matrix size need to be performed on the whole array. Thus, all arrays need to reside on the GPU memory and specific GPU kernels required for these kind of operations (such as integer addition and min/max operations) must be developed.

Another implicit consequence from the nature of the vbatched routine is that any kernel has to accommodate the largest matrix in the batch. Therefore, any vbatched kernel takes as input the maximum size across all matrices. We propose to have two interfaces for the vbatched routines, one that requires the maximum dimension(s) across all matrices as an input parameter, and a simple one very close to the LAPACK interface where the maximum value is computed using a GPU kernel. The former is recommended when the user has such information so that computing the maximums is waived. The latter wraps the first interface and calls GPU kernels to compute these maximums. In most cases, the overhead of computing the maximum is negligible.

B. Methodology and Optimization Techniques

The experience of the research community over the last few years has shown that applications dominated by small matrices cannot be executed efficiently using standard LA libraries for heterogeneous systems with GPUs. We know, from collaborating with computational scientists who have problems involving many small matrices, that making a separate call to one of these existing "optimized" libraries for each small matrix will consistently result in the same low performance. We call this approach *naïve* because it does not recognize that the techniques used to optimize these libraries presuppose large matrices. Consequently, we try to discourage this approach by showing, that getting good performance for small matrices demands a different approach.

There are relatively straightforward explanations of why neither standard (i.e. naïve) implementations, which assume "fast large matrix-matrix multiply" as the base, nor classic autotuning techniques will solve the problems that small matrices encounter. In the case of the former, fast matrix-matrix multiply only achieves its close-to-peak speed at the asymptotic limit, whereas small matrix operations never attain this asymptotic behavior. It assumes that the surface-to-volume effect will dominate the time in terms of computation, so that other overheads (communication, memory hierarchy prefetch, kernel launch overhead, etc.) will not have serious impact because their influence is overshadowed by computation. Batch operations do not have this luxury. Classical optimization and autotuning can help in the large matrix-matrix-multiplication case because it provides a well defined target; it can, if done judiciously and with knowledge of the architecture, often hide problems like insufficient memory bandwidth and/or main memory latency; batched operations require much more elaborate autotuning and kernels fusions.

C. Fusion vs. Separation of BLAS Kernels

As proposed by Haidar et al. [13], a batched BLAS approach is superior to other methodologies where a single CUDA thread or thread block does the whole factorization [25], [26], [30]. However, such batched BLAS kernels are always called from the CPU, which might introduce a large overhead of kernel launches, especially if the matrix sizes are below a certain threshold. In this paper, we adopt the same batched BLAS approach, and investigate both fusion and separation of these kernels. Kernel fusion is expected to outperform kernel separation up to some crossover point at which the separation technique starts to take over.

D. Approach 1: Fused BLAS Kernels

The motivation behind fusing BLAS kernels is to reduce kernel launch overhead and avoid workspace allocations, which are unnecessary if certain assumptions can be made about the matrix size. It also enables data reuse across the fused kernels, thus minimizing global memory traffic. We describe our approach on the Cholesky factorization, described in Algorithm 1. The computation follows three steps. The panel is updated by the effect of the previous steps. This is a rank-k update that can be performed by a customized syrk. Then the tile is factorized (potf2) and the lower portion of the panel is factorized by a trsm.

Algorithm 1 The blocked Cholesky factorization.

```
for $i \in \{1, 2, 3, \ldots, n/nb\}$ do
    if (i $>$ 1) then
        Panel Update $C_{m\times nb} = C_{m\times nb} - A_{m\times n} \times (B^T)_{n\times nb}$
    end if
    Tile Factorize $C_1$ :=Cholesky($C_1$) (dpotf2)
    Panel Factorize $C_2 = C_2(C_1^T)^{-1}$ (dtrsm)
end for
```

We designed a high performance kernel for the left-looking Cholesky factorization that fuses these steps. For a matrix of size m, the kernel requires that a panel of size $m \times nb$ can be stored in shared memory, where *nb* is a tuning blocking size. This enables fast data reads and writes for the potf2 and trsm operations. One of the advantages of kernel fusion is that we do not need to implement all the possibilities that the batched BLAS provides. For example, the syrk operation inside the fused kernel is customized to compute $C_{m\times nb} = C_{m\times nb} - A_{m\times n} \times (B^T)_{n\times nb}$, as shown in Figure 2. Unlike generic Batched BLAS syrk routines, we implement one version (A is non-transposed and B is transposed). Moreover, we know that for Cholesky, B consists of a portion of A. Thus, we take advantage of it in the customized routine and avoid redundant loads from the GPU main memory. Using a standard syrk kernel cannot take advantage of such special cases, as it always assumes the generic standard rank-k update. We also employ a double buffering technique to perform the panel update using pipelined stages, where data movements from the global memory at one stage overlap computation from the previous stage. We autotuned this kernel for all the possible sizes. We defined a modular templated interface so that we call the kernel using the predefined template where the nb tuning parameter is predefined at compile time, which allows the compiler to provide more optimizations and to store data in the constant cache memory. For simplicity, fused kernels were initially developed for fixed-size batched operations, where we observed significant speedups for very

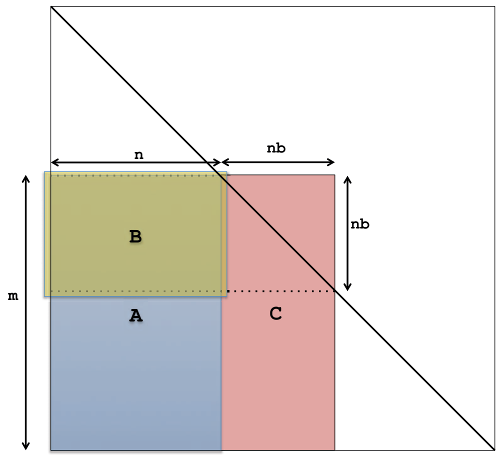


Figure 2. Panel update for a left-looking Cholesky factorization

small matrices against the separated kernel approach, as we point out in Section IV.

Having the fused kernels approach working for fixed-size test cases, we developed the following techniques to support variable-size batched computation.

1) Early Termination Mechanisms (ETM): Matrices of different sizes require different number of factorization steps. For example, given *nb*=8, a matrix of size 100 requires 13 factorization steps, while a matrix of size 25 requires only 4. As pointed out in Section III-A, any vbatched kernel should be configured to accomodate the largest matrix in the batch. This means that some thread blocks may not have work to do at some factorization steps, because the matrix assigned to them is fully factorized. An ETM enables such thread blocks to terminate immediately after launch. A decision is made at the kernel level, where each thread block independently determines whether it has work to do. This decision is based on thread block coordinates, the size of the assigned matrix, and the number of factorization steps carried out so far. We propose two ETMs:

- ETM-classic: This mechanisms terminates only full thread blocks, which means that all threads in a thread block must have no work to do in order to terminate it. If at least one thread has work to do, the corresponding thread block is kept alive. This mechanism can be safely used with any kernel regardless of the implementation details or the operation being performed.
- ETM-aggressive: This mechanism is more aggressive than ETM-classic. It implements ETM-classic, but also looks into threads in live thread blocks that might not have work to do. This mechanism is kernel-specific and is not valid to use for any implementation. The aforementioned fused kernel was developed to support ETM-aggressive. For example, assuming a 1D configuration of thread blocks with 64 threads, two matrices whose sizes are 24 and 63 require one thread block each. However, ETM-aggressive will terminate 40 threads for the first matrix, and 1 thread for the second. On the contrary, ETM-classic will keep all the 64 threads on both thread blocks alive.

2) Implicit Sorting: To achieve better occupancy and load balance, in addition to the ETMs techniques, we developed an *implicit sorting* scheme which emphasizes computation on matrices within the same range of sizes. The idea can be viewed as a scheduling techniques. At every step of the computation, a window of sizes is noted as "*active sizes*" meaning ready to be factorized. Matrices of size within this window move to a ready state queue. This approach allows the algorithm to go through the matrices by batch of "nearly similar sizes", improving occupancy and workload balance. The window size is determined by the block size nb.

E. Approach 2: Separated BLAS Kernels

The fused kernel approach cannot be used for medium matrix size, since the shared memory requirement can exceed the resources on the GPU if m is large. At this point, an approach based on separated batched BLAS kernels is adopted, which can handle from medium to very large sizes. This approach is based on building vbatched kernels which implement standard BLAS operations (potf2, trsm, gemm, and syrk). While the focus of this paper is on Cholesky factorization, we emphasize that these kernels are a foundation for other variable-size batched factorizations (LU and QR) as well as other higher level LAPACK algorithms.

1) Panel Factorization: This kernel performs the Cholesky factorization as described by the potf2 routine. In fact, we reuse the fused kernel described in Section III-D in order to factorize a square panel of size *NB*, where *NB*>*nb*. Tuning nb takes into consideration the shared memory constraint of the fused kernel.

2) Triangular Solve: We developed a vbatched trsm kernel that is based on the design proposed in [13] which starts by inverting the diagonal blocks of size typically 32×32 using a vbatched trtri routine, and then updates the solution matrix based on several calls to a vbatched gemm kernel [3], which was optimized and autotuned based on techniques from the classic MAGMA gemm routine [23]. The vbatched kernels for trtri and gemm operations use ETM-classic. They cannot use ETM-aggressive since the implementation of these kernels requires all threads in live thread blocks to be in sync.

3) Symmetric rank-k update: The trailing matrix update is done by the syrk kernel, which inherits its implementation from the vbatched gemm kernel. The syrk operation is realized as a gemm with an additional decision layer that identifies thread blocks required to update either the upper or the lower triangular part of the trailing submatrix, and thus terminating all other thread blocks. We also use another alternative for the trailing matrix updates, based on the high performance syrk from CUBLAS [24], where one kernel is launched per matrix and concurrent kernel execution is realized using CUDA streams. The decision to select either vbatched MAGMA kernel or the streamed syrk kernel is based on a performance tuning process that is beyond the scope of this paper.

F. Factorization Driver

In addition to the kernels mentioned above, there is a top layer that runs on the CPU side and controls the launch of the vbatched kernels. It consists of the main loop of the algorithm and for that we call it the factorization driver. It provides information to the kernels about step id and sizes. Such information prevents creating out-of-bound memory accesses or kernel launch failures. For example, if the batch contains some matrices that are less than nb in size, then these matrices will be fully factorized during the first panel factorization phase. For such matrices, any calls to the trsm or the syrk kernels are useless and should be terminated. The factorization driver uses auxiliary kernels to pass the necessary information to the trsm and the syrk kernels to ignore the factorized matrices onward as the computation progresses. As we will show in Section IV, the overhead of these auxiliary kernels is almost negligible.

IV. Experimental Results

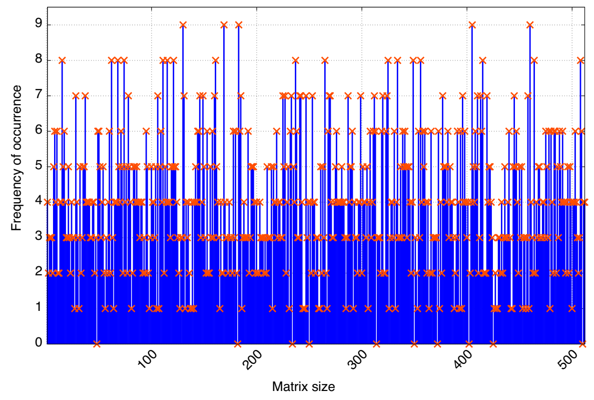
A. System Setup

All the experiments are conducted on a machine equipped with two 8-core Intel Sandy Bridge CPUs (Intel Xeon E5-2670, running at 2.6 GHz), and a Kepler generation GPU (Tesla K40c, running at 745 MHz, with ECC on). CPU performance tests use Intel MKL Library 11.3.0. GPU performance tests use CUDA Toolkit 7.0. While we show performance tests for single and double precisions only, the proposed framework supports complex precisions.

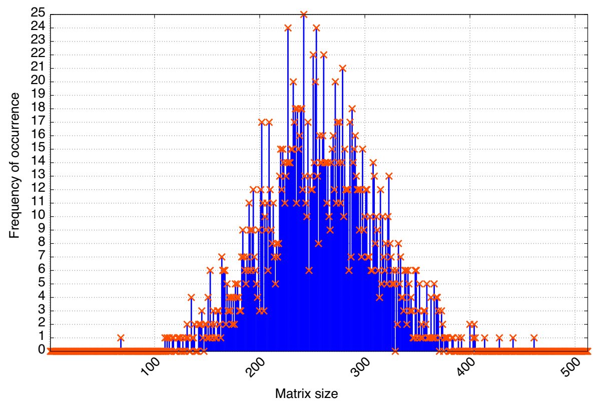
B. Matrix Size Distribution

Considering the test cases for the vbatched routines, the matrix sizes in any batch are randomly generated using two different pseudo random number generators. The first one uses a uniform distribution so that given a maximum size N_{max}, the batch contains sizes that range from 1 to N_{max} based on a uniform distribution. As an example, Figure 3a shows a histogram for the matrix sizes generated for a batch size of $2,000$. The figure shows that most sizes appear at least once, with the majority of sizes appearing between 1 and 5 times. The second generator is based on a Gaussian distribution, so that given a maximum size N_{max}, most of the matrix sizes in the batch are around the mean value $\lfloor \frac{N}{2} \rfloor$, with fewer sizes appearing near the boundaries of the interval $[1{:}N_{max}]$, as shown in Figure 3b.

The following sections show the performance in Gflop/s for different versions developed. The total number of flops is computed as the summation of the flops required to perform the factorization on each individual matrix, which reflects the elapsed time, e.g., a twice Gflop/s means twice faster.



(a) Uniform Distribution



(b) Gaussian Distribution

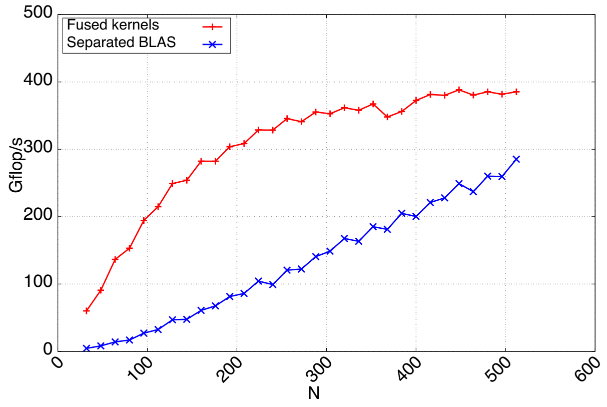
Figure 3. Histograms of the size distribution for a batch count equal to 2000 with maximum matrix size set to 512

C. Impact of kernel fusion on fixed-size batched routine

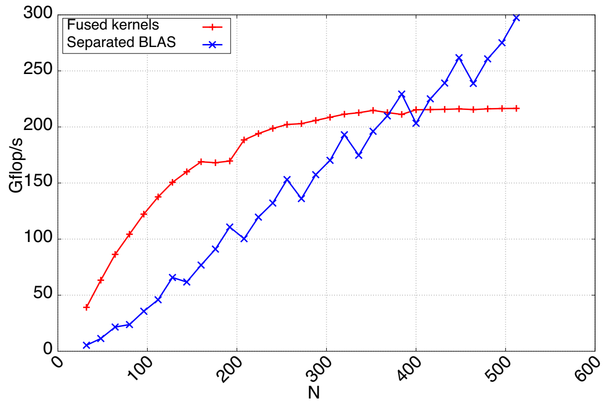
As discussed in Section III-D, our starting point to develop the vbatched framework is to enhance the performance of the fixed-size batched routines, especially when matrices are very small. Figure 4 justifies the motivation to use kernel fusion, where significant performance gains are scored against the traditional approach (e.g., separated building block BLAS kernels). The relative speedups are up to $13\times$ for single precision and $7\times$ for double precision. Both speedups decay as the matrix size gets larger. This is expected, since the kernel fusion approach will require larger shared memory, which affects the number of kernels that can be processed concurrently. Meanwhile, the overhead of separating BLAS kernels gets smaller with respect to the amount of computation associated with larger matrices. This is why Figure 4c shows a steady trend where the speedup is going below one. Our proposed framework is designed to select the best out of the two approaches. It defines a crossover point after which separated BLAS kernels are used.

D. Performance of The Fused Kernels Approach

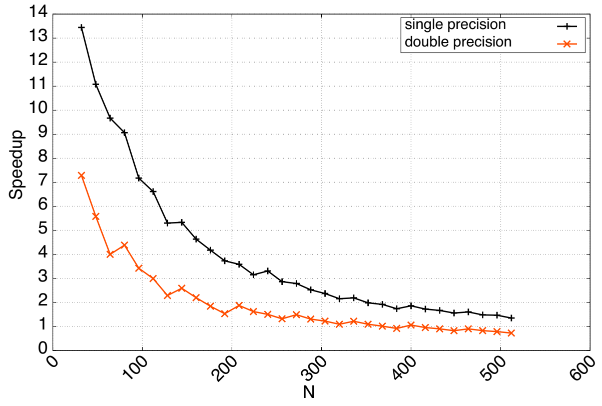
There are four different versions that we progressively developed for the kernel fusion approach. They are:



(a) Single Precision



(b) Double Precision



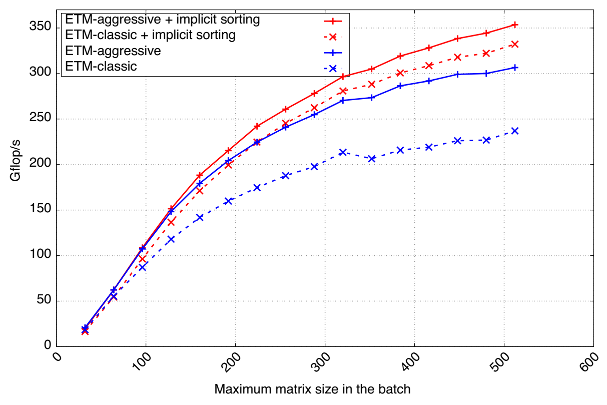
(c) Relative speedup

Figure 4. Fused kernels performance and speedup over separated BLAS approach (fixed-size matrices)

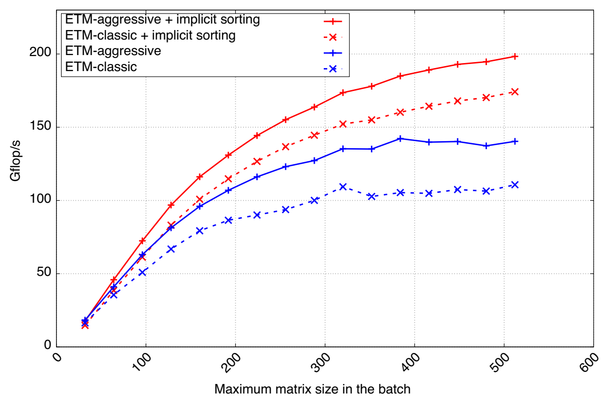
1) ETM-classic only;
2) ETM-aggressive only;
3) ETM-classic + implicit sorting; and
4) ETM-aggressive + implicit sorting.

The performances of these versions are shown in Figure 5 for a uniform, and in Figure 6 for a Gaussian distribution.

Considering the uniform distribution test case, with the absence of the implicit sorting technique, ETM-aggressive



(a) Single Precision

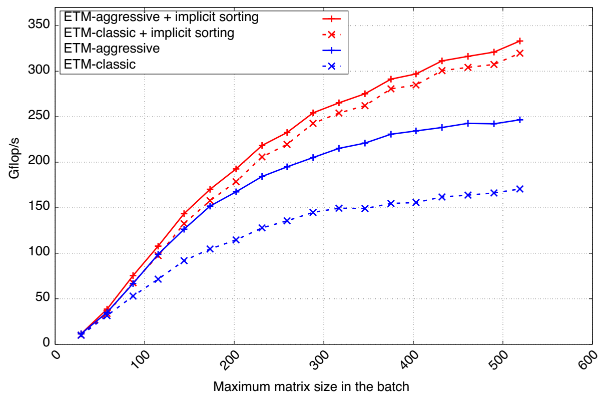


(b) Double Precision

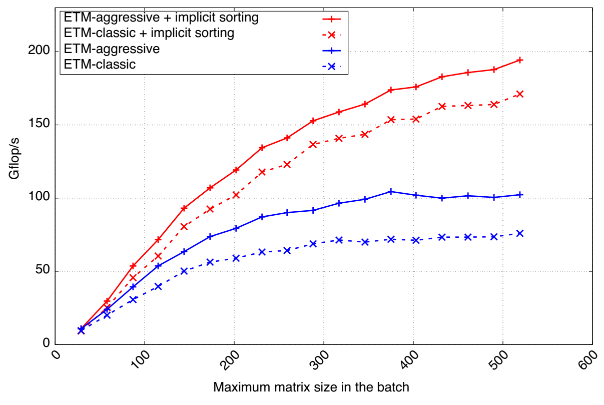
Figure 5. Performance of different versions of the vbatched xPOTRF based on the fused kernels approach, batch count equal to 3000 (uniform distribution)

has an advantage over ETM-classic with speedups ranging from 12% up to 33% in single precision, and from 11% up to 35% in double precision. These speedups are obtained because ETM-aggressive employs a more fine-grain technique to terminate both idle thread blocks as well as idle threads in live thread blocks. Eventually, ETM-aggressive allows more resources to be immediately released to be ready for useful computation. With the addition of the implicit sorting technique, we notice that the performance of ETM-classic is improved by up to in 42% single precision and 60% in double precision. Similarly for ETM-aggressive, implicit sorting improves the performance by up to 15% in single precision and 41% in double precision. Implicit sorting reorders the computation so that, at any factorization step, live thread blocks only consider matrices whose sizes are within a window of *nb*, and so it achieves more load balancing and reduces the overhead of early termination mechanisms.

Considering the Gaussian distribution test case, we observe a similar behavior to Figure 5, except that the impact of implicit sorting is much more significant than the case of uniform distribution. The performance improvements are up to 87.5% (single precision) and 125.26% (double precision)



(a) Single Precision



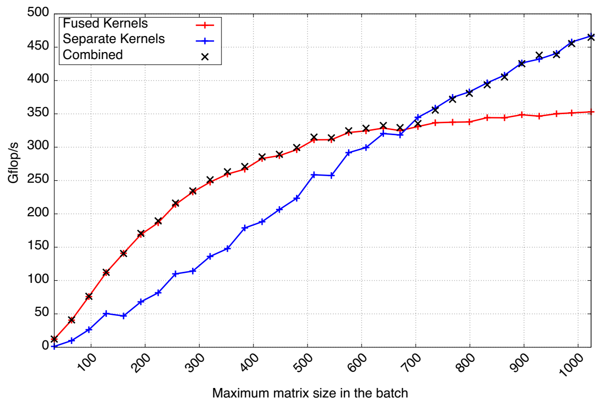
(b) Double Precision

Figure 6. Performance of different versions of the vbatched xPOTRF based on the fused kernels approach, batch count equal to 3000 (Gaussian distribution)

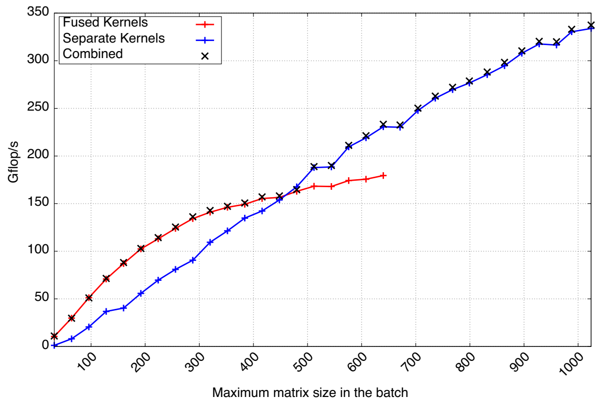
with ETM-classic, and up to 35.1% (single precision) and 89.9% (double precision) with ETM-aggressive. The reason behind such behavior is that the Gaussian distribution, by nature, introduces few matrices that are far from the mean value, especially matrices whose sizes are larger than the mean. The presence of such larger matrices causes even more load imbalance. The absence of implicit sorting causes factorizations to start on all matrices at the same time, leading to more load imbalance and large overhead of ETMs. With implicit sorting involved, the computation reordering achieves a more robust behavior, so that nearly the same performance is achieved, compared to the uniform distribution test case.

E. Combining kernel fusion and separation

Figure 7 shows the crossover points between the two approaches discussed in Sections III-D and III-E for a uniform distribution test case. The performance of the kernel fusion approach is based on the best configuration highlighted in Section IV-D (ETM-aggressive + implicit sorting). For the test cases generated here, the crossover point is marked by the maximum size in the batch. The reason behind choosing the maximum as the deciding criteria is that the kernel fusion approach cannot work for any matrix size, due to its shared memory requirements. Checking the maximum size decides whether it is safe to run such approach for the input batch. Otherwise, there is no choice but to run the separated kernel approach.



(a) Single Precision



(b) Double Precision

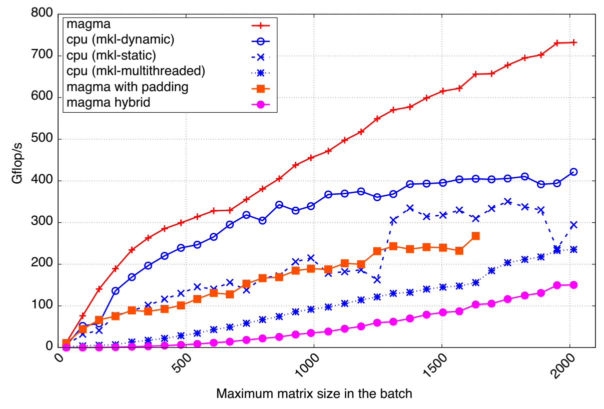
Figure 7. Crossover points for vbatched xPOTF, batch count equal to 800.

F. Overall Performance

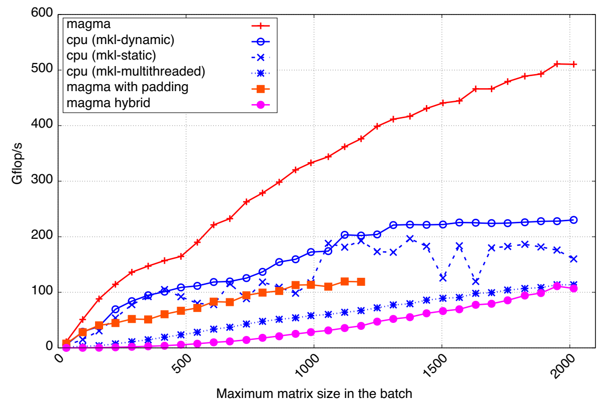
Figures 8 and 9 show the overall performance of the vbatched Cholesky factorization and compare it against the following alternatives:

- MAGMA hybrid routines, which use a hybrid CPU+GPU algorithm;
- MAGMA fixed-size batched routines with padding;
- A multithreaded CPU kernel using Intel MKL. This technique uses all cores to factorize one matrix at a time;
- A CPU implementation with one core assigned per matrix at a time. A scheduling schema is used to assign matrices to cores (16 CPU cores in our experiments). The scheduling can be either static or dynamic. Both variants are plotted.

Obviously, the MAGMA hybrid routines are not the correct choice for this type of workload. As these routines assume

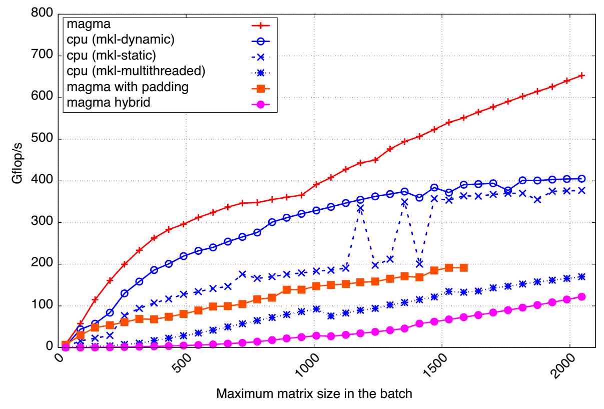


(a) Single Precision

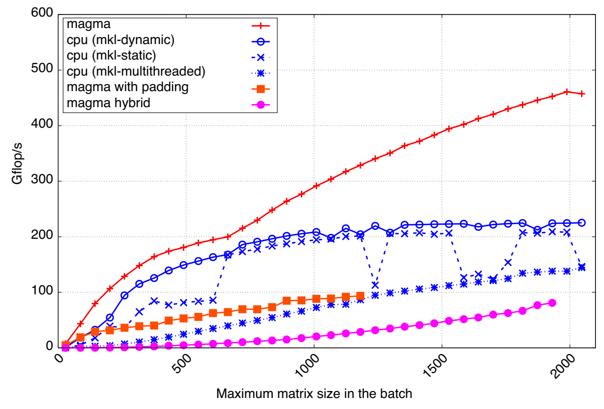


(b) Double Precision

Figure 8. Overall performance of the vbatched xPOTRF routine, batch count equal to 800 (uniform distribution)



(a) Single Precision



(b) Double Precision

Figure 9. Overall performance of the vbatched xPOTRF routine, batch count equal to 800 (Gaussian distribution).

a relatively large matrix, the performance for a batch of small matrices is low, as discussed in Section II. There exist libraries developed and optimized for batch computation but for fixed-size matrices only (MAGMA or CUBLAS). Thus, application that requires variable sizes computation will not have choice except to use the fixed-size batch routines. For that, the users need to pad the matrices with zeros in order to make them fixed-size. Although the fixed-size batched routines perform better than the hybrid ones, their performance is still unacceptable and it is even lower than some CPU performance graphs. The padding technique results in a lot of extra computation, and its influence on performance depends on the size distribution.

A multithreaded CPU scheme is not a wise option either for this kind of problems, since each individual matrix is too small to have multiple cores working on it. This is why it lags behind other schemes that use one core per matrix at a time. The best competitor to the proposed approach is dynamic assignment of one CPU core at a time for a given matrix. Most of the matrices in such workloads will fit in the fast cache levels, and the dynamic scheduling ensures balancing the load among cores. This is why the static scheduling results in some performance oscillations.

For a uniform distribution test case, speedups against the best competitor range from $1.11\times$ up to $2.42\times$ in single precision, and from $1.51\times$ up to $2.29\times$ in double precision. Respective speedups for a Gaussian distribution range from $1.31\times$ to $2.07\times$ in single precision, and from $1.21\times$ to $2.52\times$ in double precision. Without the introduction of the vbatched routines, the only way a user can use batch routine on GPU is to use padding, against which the proposed vbatched routines are up to $3\times$ faster. The performance graphs of the padding technique look truncated due to running out of the GPU memory, which is yet another motivation to have a vbatched routine.

G. *Energy Efficiency*

An interesting metric to study and measure is the energy to solution. All our benchmark runs show that energy efficiency significantly favors the GPU implementation. Figure 10 shows the total amount of energy consumed by both hardware CPU and GPU for a benchmark testbed for matrices with different range of sizes. The experiment is conducted for the proposed vbatched routine against the fastest CPU implementation, which calls the optimzed MKL Library within a dynamically unrolled parallel OpenMP loop, assigning one core per

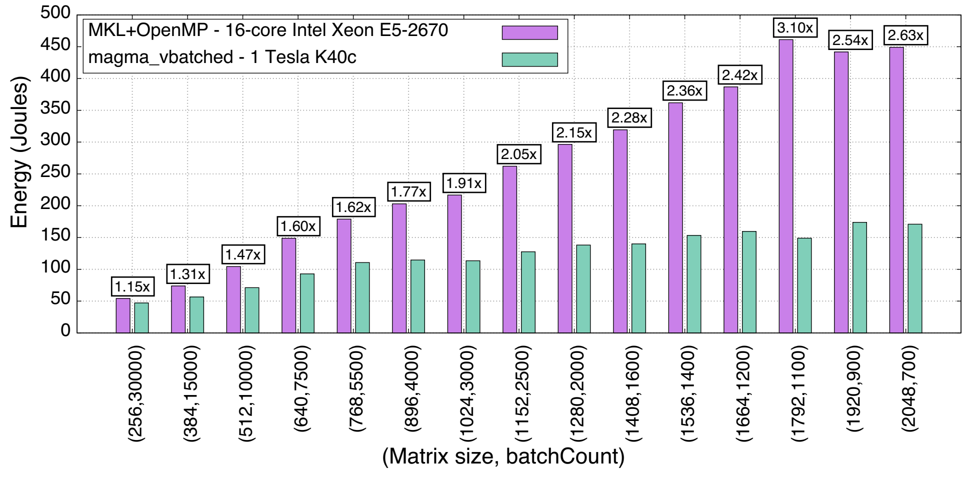


Figure 10. Total energy consumed by the CPU and the GPU for vbatched dpotrf

matrix at a time. The system setup is identical to the one described in Section IV-A. We use PAPI [2] (version 5.4.0) and NVIDIA Management Library (NVML) [1] in order to record the power consumed by either the CPU and the GPU implementation, respectively. The results in Figure 10 represent the integration of the power measurements over time. Our experiments proved that the GPU implementation is always more efficient than the CPU ones, in terms of both time and energy to solution. The results in Figure 10 shows that the GPU design can reach a factor up to 3 $\times$ more energy efficient.

V. Conclusion and Future Work

This paper developed a framework for variable size batched matrix computations using GPU accelerators. While Cholesky factorization has been chosen as a case study, the proposed ideas and methodology can be used for other factorizations and solver algorithms. In fact, the new interfaces and techniques proposed in this paper create a foundation for high performance variable-size batched computation. We showed that kernel fusion achieves better performance when matrix sizes are below some threshold, after which the design switches to another technique where BLAS kernels are separated from each other and used as a building block modular routines. Many fold speedups are achieved against alternative techniques using the CPU or the GPU.

Future directions include the extension of this work to the LU and QR factorizations, sparse direct multifrontal solvers, and applications, where many of the BLAS kernels proposed here can be reused out of the box. It is also important to test the impact of different size distributions on performance, and how the variation in sizes might affect the crossover points. Another open direction is to investigate LAPACK compliance of these routines, especially with respect to error checking, and to propose an alternate scheme to report possible errors to the user.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CSR 1514286, NVIDIA, the Department of Energy, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

[1] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[2] Performance Application Programming Interface (PAPI). Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/papi/.

[3] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance, Design, and Autotuning of Batched GEMM for GPUs. Technical Report UT-EECS-16-739, 02-2016 2016.

[4] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In W. mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

[5] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[6] ACML - AMD Core Math Library, 2014. Available at http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml.

[7] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, third edition, 1999.

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

[9] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Luc, M. Nooijene, R. Pitzerf, J. Ramanujamg, P. Sadayappan, and A. Sibiryakovc. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[10] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[11] T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched Cholesky factorization on a GPU. In *Proceedings of 2014 International Conference on Parallel Processing (ICPP-2014)*, September 2014.

[12] J. Dongarra, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and A. YarKhan. Model-driven one-sided factorizations on multicore accelerated systems. *International Journal on Supercomputing Frontiers and Innovations*, 1(1), June 2014.

[13] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, 2015.

[14] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. A Framework for Batched and GPU-Resident Factorization Algorithms Applied to Block Householder Transformations. In J. M. Kunkel and T. Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 31–47. Springer International Publishing, 2015.

[15] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra. 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, San Francisco, CA, 02/2015 2015. ACM, ACM.

[16] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization Framework for Sparse Matrix Kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, Feb. 2004.

[17] Intel Pentium III Processor - Small Matrix Library, 1999. Available at http://www.intel.com/design/pentiumiii/sml/.

[18] Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/.

[19] J. L. Khodayari A., A.R. Zomorrodi and C. Maranas. A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. *Metabolic engineering*, 25C:50–62, 2014.

[20] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[21] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. 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."*, 2012.

[22] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.

[23] R. Nath, S. Tomov, and J. Dongarra. An Improved Magma Gemm For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.

[24] CUBLAS, 2014. Available at http://docs.nvidia.com/cuda/cublas/.

[25] V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo. Power/performance trade-offs of small batched LU based solvers on GPUs. In *19th International Conference on Parallel Processing, Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 813–825, Aachen, Germany, August 26-30 2013.

[26] V. Oreste, N. A. Gawande, and A. Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2013)*, Indianapolis, Indiana, September, 23-27 2013.

[27] S. Tomov, J. Dongarra, and M. Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010.

[28] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[29] V. Volkov and J. W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, University of California, Berkeley, May 13 2008. Also available as LAPACK Working Note 202.

[30] I. Wainwright. Optimized LU-decomposition with full pivot for small batched matrices, April, 2013. GTC'13 – ID S3069.

[31] S. N. Yeralan, T. A. Davis, and S. Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.