Procedia Computer Science

Volume 80, 2016, Pages 119–130

ICCS 2016. The International Conference on Computational Science

Performance Tuning and Optimization Techniques of Fixed and Variable Size Batched Cholesky Factorization on GPUs

Ahmad Abdelfattah[1], Azzam Haidar[1], Stanimire Tomov[1], and Jack Dongarra[123]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, USA
[2] Oak Ridge National Laboratory, Oak Ridge, USA
[3] University of Manchester, Manchester, U.K.
{ahmad,haidar,tomov,dongarra}@icl.utk.edu

Abstract

Solving a large number of relatively small linear systems has recently drawn more attention in the HPC community, due to the importance of such computational workloads in many scientific applications, including sparse multifrontal solvers. Modern hardware accelerators and their architecture require a set of optimization techniques that are very different from the ones used in solving one relatively large matrix. In order to impose concurrency on such throughput-oriented architectures, a common practice is to batch the solution of these matrices as one task offloaded to the underlying hardware, rather than solving them individually.

This paper presents a high performance batched Cholesky factorization on large sets of relatively small matrices using Graphics Processing Units (GPUs), and addresses both fixed and variable size batched problems. We investigate various algorithm designs and optimization techniques, and show that it is essential to combine kernel design with performance tuning in order to achieve the best possible performance. We compare our approaches against state-of-the-art CPU solutions as well as GPU-based solutions using existing libraries, and show that, on a K40c GPU for example, our kernels are more than 2× faster.

Keywords: Batched Computation, Cholesky Factorization, Tuning, GPUs

1 Introduction

Many scientific applications require the solution of a large number of relatively small independent linear systems in parallel. Examples are astrophysics, quantum chemistry, metabolic networks, CFD and resulting PDEs through direct and multifrontal solvers, high-order FEM schemes for hydrodynamics, direct-iterative preconditioned solvers, and image and signal processing. If we consider modern multi/many-core architectures, the amount of work associated with each individual problem does not provide sufficient parallelism to achieve good performance. It is advantageous, therefore, to use the independence among these problems to develop specialized software grouping the computation into a single *batched* routine.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.303

The development of a batched numerical kernel for multicore CPUs is relatively simple, and can be realized using existing numerical software. In most cases, individual matrices can fit into the large CPU caches, where the computation can be performed very quickly using optimized vendor-supplied libraries, such as as MKL or ACML. However, many-core architectures, such as GPUs, lack the ability to store even small matrices. In addition, a key factor for achieving high performance on a GPU is the ability to saturate all of the GPU's Streaming Multiprocessors (SMs) with work. Therefore, numerical GPU software originally optimized for large problems, cannot be used to efficiently solve batched small problems. A different design approach is needed in order to produce a high performance software on such workloads.

This work presents a set of design and optimization techniques for the Cholesky factorization on batches of relatively small matrices of fixed and variable sizes. We propose several design approaches and optimizations. Our methodology is to start by progressive optimization and tuning for fixed size problems, until a best configuration is reached. We then proceed with the best fixed size configuration by extending its design to support variable sizes. We show performance results against other GPU solutions that use existing numerical libraries. We also compare against a 16-core Intel Sandy Bridge CPU running the Intel MKL library. In either case we demonstrate that the new approaches significantly accelerate performance.

2 Related Work

Sufficiently large matrices can be factorized individually based on hybrid algorithms that use both the CPU and the GPU [12]. The motivation behind hybrid algorithms is that GPUs cannot be used on panel factorizations as efficiently as on trailing matrix updates [13]. Since such updates are mostly GEMMs [1, 4], 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 CPU-GPU communication.

There are some efforts that proposed batched computations on the GPU. Villa et al. [10], [11] proposed batched LU factorization for matrices up to size 128. Wainwright [14] 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. [3] proposed three different implementations for batched Cholesky factorization, where the performance was compared against a multicore CPU and the hybrid MAGMA [2] algorithm. Kurzak et al. [8] 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 [6], where several-fold speedup is obtained against a competitive design by CUBLAS. 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 [5] [7]. All the aforementioned efforts focus on batches with fixed sizes. This work improves the performance of the fixed size batched Cholesky factorization proposed in [5], and also addresses variable size batched problems.

3 Algorithmic Designs for Fixed Sizes

Recursive Multi-level Blocking. In a previous study [5], the fixed-size batched routine for a batch of $m \times m$ matrices was designed with two levels of blocking – the classical blocking of nb elementary factorization steps that allows one to take advantage of the Level-3 BLAS

(as in LAPACK), and a recursive blocking for the *nb* elementary factorization steps (known as panel factorization) in order to minimize its cost. To increase the parallelism and to have more compute intensive tasks, the algorithm is a *right looking*, which means a panel factorization is followed by a large trailing matrix update to the right of the panel. This design, presented in Algorithm 1, has been proved efficient as it takes advantage of large BLAS-3 operations like the TRSM and SYRK.

Recently, we performed a detailed performance study based on the collection and analysis of machine counters. Counter readings were taken using performance tools (NVIDIA's CUPTI and PAPI CUDA component [9]). While previously the algorithm was blocked with outer loops running on the CPU, calling the computational kernels on the GPU, we discovered that for small matrices a layer of fusing and blocking operations has to be added (or moved from the CPU) to the GPU kernels. The purpose of this optimization is to minimize the load/store to the main memory, increase the data reuse at the thread-block level, and decrease the number of register/shared memory required per thread-block to allow more thread-blocks to be executed by the same SM. To accomplish this for the panel factorization, we discovered that blocking at the kernel level should follow a left-looking Cholesky factorization, with a blocking size *ib*, as shown in Algorithm 2, which is known to minimize data writes (in this case from GPU shared memory to GPU main memory).

Algorithm 1: The right looking fashion.

for $i \leftarrow 0$ **to** m **Step** *nb* **do**
  `// Panel factorize` $\mathbf{A_{i:m,i:i+nb}}$
  POTF2 $A_{i:i+nb,i:i+nb}$;
  TRSM $A_{i+nb:m,i:i+nb} = A_{i+nb:m,i:i+nb} \times A^{-1}_{i:i+nb,i:i+nb}$;
  `// Update trailing matrix` $\mathbf{A_{i+nb:m,i+nb:m}}$
  SYRK $A_{i+nb:m,i+nb:m} = A_{i+nb:m,i+nb:m} - A_{i+nb:m,i:i+nb} \times A^{T}_{i+nb:m,i:i+nb}$;
end

Algorithm 2: The left looking fashion.

for $i \leftarrow 0$ **to** m **Step** *ib* **do**
  if *(i > 0)* **then**
    `// Update current panel` $\mathbf{A_{i:m,i:i+ib}}$
    SYRK $A_{i:i+ib,i:i+ib} = A_{i:i+ib,i:i+ib} - A_{i:i+ib,0:i} \times A^{T}_{i:i+ib,0:i}$;
    GEMM $A_{i+ib:m,i:i+ib} = A_{i+ib:m,i:i+ib} - A_{i+ib:m,0:i} \times A^{T}_{i:i+ib,0:i}$;
  end
  `// Panel factorize` $\mathbf{A_{i:m,i:i+ib}}$
  POTF2 $A_{i:i+ib,i:i+ib}$;
  TRSM $A_{i+ib:m,i:i+ib} = A_{i+ib:m,i:i+ib} \times A^{-1}_{i:i+ib,i:i+ib}$;
end

Kernel Fusion and Optimization. When the kernel's working data is small, the computation associated with it becomes memory bound. Thus, fusing the four kernels of one iteration of Algorithm 2 (into one GPU kernel), will minimize the memory traffic, increase the data reuse from shared memory, and reduce the overhead of launching multiple kernels. Using a left-looking Cholesky algorithm, the update writes a panel of size m×ib in the fast shared memory instead in the main memory, and so the merged POTF2 routines can reuse the panel from the

shared memory. Note that m and ib control the amount of the required shared memory; they are critical for the overall performance, and thus can be used to (auto)tune the implementation. The value of m is controlled by the level of recursion, while ib is defined at the kernel level.

We developed an optimized and customized *fused kernel* that first performs the update (SYRK and GEMM operations), and keeps the updated panel in shared memory to be used by the factorization step. The cost of the left looking algorithm is dominated by the update step (SYRK and GEMM). The panel C, illustrated in Figure 1, is updated as $C = C - A \times B^T$. In order to decrease its cost, we implemented a double buffering scheme that performs the update in steps of lb, as described in Algorithm 3. We mention that we prefix the data array by "r" and "s" to specify register and shared memory, respectively. We prefetch data from A into register array rAk while a multiplication is being performed between register array rAkk and the array sB stored in shared memory. Since the matrix B is the shaded portion of A, our kernel avoids reading it from the global memory and transposes it in place to the shared memory sB. Once the update is finished, the factorization (POTF2 and TRSM) is performed as one operation on the panel C, held in shared memory.

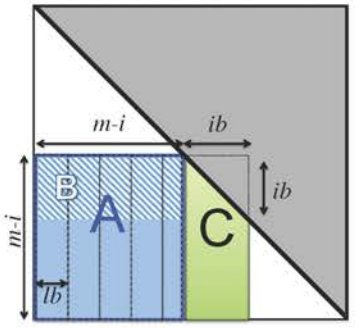


Figure 1: left-looking Cholesky factorization

Algorithm 3: The fused kernel correspond to one iteration of Algorithm 2.

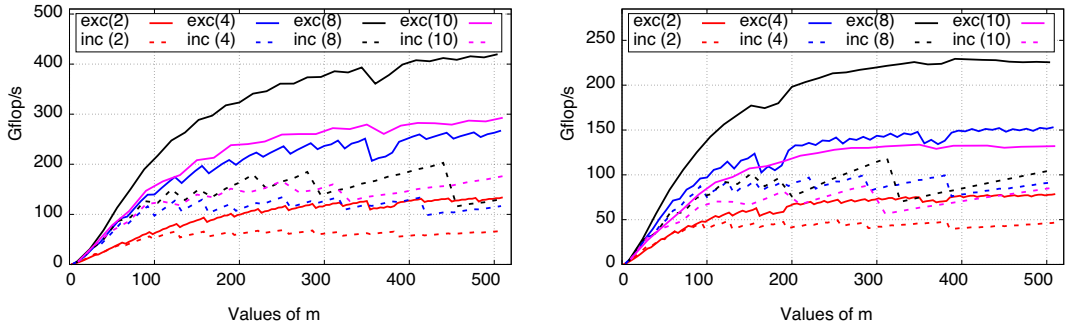
```
rAk ← A_(i:m,0:lb); rC ← 0;
for k ← 0 to m-i Step lb do
    rAkk ← rAk;
    sB ← rAk_(i:lb,k:k+lb) inplace transpose;
    barrier();
    rA1 ← A_(i:m,k+lb:k+2lb) prefetching;
    rC ← rC + rAkk×sB multiplying;
    barrier();
end
sC ← rA1 - rC;
factorize sC;
```

Loop-inclusive *vs.* **Loop-exclusive Kernels.** In order to develop Algorithm 2, a first step is to decide whether the main loop is on the CPU or on the GPU (inside the kernel). In this context, we developed *loop-inclusive* and *loop-exclusive* kernels. The *loop-inclusive* kernel is launched once from the CPU side, meaning that the loop iteration over ib of Algorithm 2 are unrolled inside the kernel. The motivation behind the *loop-inclusive* approach is to maximize the reuse of data, not only in the computation of a single iteration, but also among iterations. For example, the *loop-inclusive* technique can reuse the factorized panel of iteration $i-1$ (which is in shared memory) to update the panel of iteration i, which means replacing the load from slow memory of the last blue block of A (illustrated in Figure 1) by directly accessing it from fast shared memory. It is beneficial to minimize the loads from slow memory, even if they are for small amounts of data (the economy here is to avoid loading the previous panel, which is of size $m - i \times ib$). However, the disadvantage of such a design is that when the kernel is launched from the CPU, it should be configured based on the tallest sub-panel (i.e., based on the size m). This means that the amount of shared memory required will be $m \times ib$ and the threads used will be fixed for all iterations of Algorithm 2. The analysis of the occupancy and the throughput of the computation of such design encourages us to develop the *loop-exclusive* version. The *loop-exclusive* kernel executes one iteration of Algorithm 2 at each launch. This will signify that we will have to re-load the previous panel from main memory, but our goal is to optimize the resources for each factorization step in terms of required shared memory and registers (e.g.,

the thread/shared memory configurations will be based on $m-i$). The prefetching technique of our fused kernel overlaps the cost of re-loading the previous panel. The only extra cost is that the CPU launches kernels as many times as the number of factorization steps. Nevertheless, our experiments show that this cost is marginal. The results, summarized in Figure 2, prove that the *loop-exclusive* approach tends to help the CUDA runtime increase the throughput of the factorized matrices during execution by increasing the occupancy at the SMs' level.

Performance Autotuning. The autotuning process of the developed kernels has one tuning parameter to consider (ib). Since the range of values for ib is intended to be small, we conducted a brute-force sweep of all possible values of ib up to 32. The fact that we have one tuning parameter makes it feasible, in terms of autotuning overhead, to conduct a brute-force experiment, after which the ib value corresponding to the best performance is used. In general, we can define different best-performing values of ib with different GPUs. The autotuning experiment is offline and needs to be conducted once per GPU model/architecture.

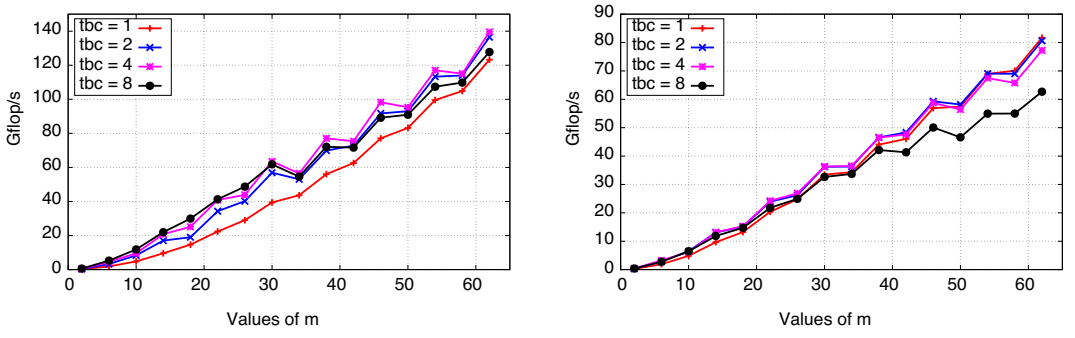
Figure 2 shows the tuning results for both the *loop-inclusive* and the *loop-exclusive* kernels for different values of ib and for different values of m. We plot the performance for ib of values up to 10, since we observe consistent drop in performance after this value. As expected, we observe a relatively low performance for the *loop-inclusive* kernels except for matrices of size below 100. The main reasons behind this behavior are as described above: (1) Thread divergence, as the number of threads in a thread block (TB) is configured based on the size m, but when the factorization progresses the working area is $m-i$, meaning more threads become idle; (2) Similarly, the size of the allocated shared memory and registers used is defined based on m, and thus becomes unused as the factorization progress; and (3) Low SM occupancy, as the used resources limit the number of TBs that can be scheduled on the same SM. On the other hand, the *loop-exclusive* approach launches the kernels with the exact number of threads required at each iteration of the factorization, and thus optimizes allocation of shared memory and registers, allowing more TBs to be scheduled on the same SM. Figure 2 shows that for the same ib the *loop-exclusive* technique is always better than the *loop-inclusive*. The best configuration was obtained with the *loop-exclusive* approach for ib = 8 and for any value of m.



(a) Single precision (b) Double precision

Figure 2: Performance tuning of *loop-inclusive*(inc) and *loop-exclusive*(exc) kernels on a K40c GPU, batchCount = 3000. The value of ib is shown between brackets.

TB-level Concurrency. The aforementioned batched design associates one matrix to a TB. When matrices are very small (e.g., less than 32), and for single precision in particular, we observe that the launching cost (which is about 5-6 μs) is comparable to the computational



(a) Single precision (b) Double precision

Figure 3: Impact of tbc on performance, batchCount = 10,000, on a K40c GPU.

cost of the kernel. Thereby, for very small matrices, we propose an additional optimization, which is to schedule multiple matrices per TB. We define tbc to be the number of matrices scheduled per TB. Figure 3 shows the impact of this technique. In single precision, speedups are about 1.5× to 2×. For example, for 10×10 matrices using tbc=2,4,8, we achieve speedups of 1.74×, 2.00×, and 2.46×, respectively. However, in double precision, speedups of 10% to 30% are observed for very small sizes (e.g., less than 32). Considering the same example for 10×10 matrices, increasing tbc can achieve only 34% speedup at its best. Note that this optimization is applied only in the case of matrices with fixed sizes, and cannot be used for variable size batched computations discussed below.

4 Algorithmic Designs for Variable Sizes

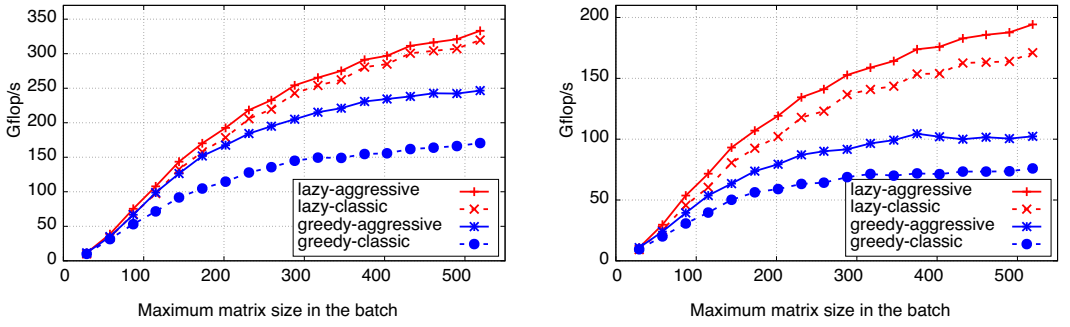
The batched designs for matrices with fixed sizes, as proposed above, are characterized by two levels of parallelism. The first level is the **task-level parallelism** among the independent matrices that are associated with a TB and simultaneously processed. The second level – **fine-grained data parallelism** – is per each matrix to exploit the SIMT architecture through the customized and optimized device kernels that we developed. Thus, at the TB level, every kernel has only a view of one matrix. From this point of view, the design of batched computation for matrices of variable sizes can be handled at the kernel level. The goal is to reuse the same optimized and autotuned kernel infrastructure for the fixed size matrices. To accomplish this at the kernel level, each matrix has a unique ID, its own sizes, and is associated with one TB independently from the other matrices. This design allows us to easily handle batches with matrices of variable sizes. The implementation and the optimizations are not straightforward, as many other difficulties must be resolved. First, the main loop is unrolled outside the kernel and thus it should be based on the maximum m among the matrices in the batch (denoted max_m), meaning that at some iteration, many matrices are already factorized and no computation is needed. Second, the thread configuration and the shared memory allocation are specified outside the kernel at the higher level, and they are related to the size max_m $-i$ of the main loop iteration, thus many TB will have extra threads and extra shared memory to deal with. This requires a sophisticated mechanism to minimize the unused resources and also to control whether a TB will have data to work on it or not, as well as to avoid read/write data out of the matrix bounds. We propose two design conceps in order to support batches of variable sizes: *Early Termination Mechanisms (ETMs)*, and a *Greedy/Lazy Scheduling*. From now on,

variable size batched kernels are abbreviated as *vbatched* kernels.

Early Termination Mechanisms (ETMs). In a vbatched kernel, we have many different sizes that are known only at the kernel level. As mentioned above, the CPU uses the maximal size `max_m` $-i$ of iteration "i" to configure and launch the kernel. As a result, some threads, or even full TBs, may have no computation to perform. The ETM is a technique that we developed to identify at the kernel level the threads and the TBs with no computational work in order to be terminated. In the ETM we compute the local sizes corresponding to the current iteration, the local parameters of a TB, and decide whether the TB is idle, as well as adjust the TB's sizes to avoid out of matrix bound accesses. In general, there are two types of ETMs that we present. The first one is called ETM-classic. This mechanism terminates only TBs with no work. For example, assume two matrices of sizes 16 and 32 are being factorized with ib=8. Then, the main loop consists of four iterations, where at each iteration the CPU launches a kernel. The thread configurations of the four launches are $(32, 1, 1)$, $(24, 1, 1)$, $(16, 1, 1)$, and $(8, 1, 1)$, respectively. The 16×16 matrix is factorized during the first two launches. The TBs assigned to this matrix in the remaining launches are idle and so terminated using the ETM-classic. Note also that the extra threads in the first two launches are not used for this matrix. The second type of ETMs is called ETM-aggressive. This mechanism terminates not only idle TBs, but also idle threads in live TBs, in an effort to maximize the amount of released resources during kernel execution. For the same example, the ETM-aggressive still terminates the TBs assigned to the smaller matrix in the third and fourth launches, but it will also terminate threads 16 through 31 in the first launch, as well as threads 8 through 23 in the second launch.

Greedy *vs.* **Lazy Scheduling.** Since the CPU launches kernels as many times as it is required by the largest matrix, there is flexibility in determining when to start the factorization of the smaller matrices. We present two different techniques for scheduling the factorization. These techniques control when a factorization should start for every matrix in the batch. The first one is called *greedy scheduling*, where the factorization begins on all the matrices at the first iteration. Once a matrix is fully factorized, the TB assigned to it in the following iterations becomes idle and is terminated using the ETM techniques. With greedy scheduling, completion of factorization on individual matrices occurs at different iterations. There are two drawbacks of such a technique, as shown in Figure 4. First, the kernel is configured based on the largest size, meaning a working TB on a matrix with size less than the maximal will have idle threads at every launch of its life. The ETM-aggressive overcomes this problem and guarantees to terminate these threads as illustrated by the performance difference in Figure 4. The second issue is that the shared memory allocated is always based on the maximal iteration size, and thus many TBs will have unused shared memory. There is no dynamic mechanism to free it, and thus such scheduling may result in low occupancy. Even though we can resolve the first issue using the ETM-aggressive, there is still a need to fix the second issue. For that we propose the so called *lazy scheduling*. Here individual factorizations start at different iterations, such that they all finish at the last iteration. At each iteration, lazy scheduling allows matrices with local sizes within the range `max_m` $-i$ to `max_m` $-i + nb$ to be computed. This technique can be viewed as sorting the matrices from the largest to the smallest one, where the smallest are at the bottom right corner of the working area. The benefit here is that a TB is considered alive only when the iteration size reaches its local size. As result, the shared memory allocated will be closest to the optimal for all the matrices across the batch, and the configuration of the number of threads within a TB will be the minimal amount possible $\pm nb$. This is the reason why the *lazy scheduling* has proved efficient and better designed than the *greedy scheduling*, as shown in Figure 4 for the two ETM techniques. Moreover, the difference between ETM-classic and ETM-aggressive is minimal in this case because the number of extra threads is always less

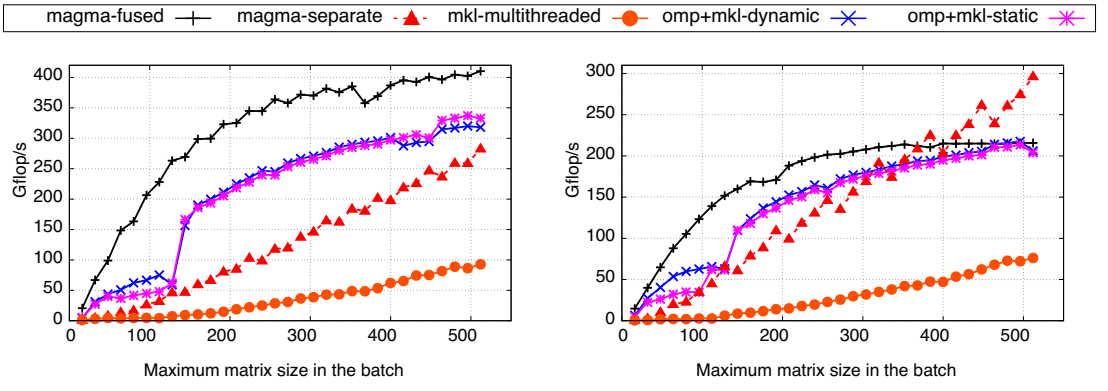
than *nb*. Figure 4 illustrates the four proposed versions of the vbatched kernel. The batches



(a) Single precision (b) Double precision

Figure 4: Optimizing vbatched kernel performance, batchCount = 3,000, on a K40c GPU.

used in the test cases have matrix sizes that are randomly generated, where the `x` axis shows the maximal size of all of them. Considering the *greedy scheduling*, an aggressive ETM is up to 40-50% better than a classic ETM. With *lazy scheduling*, we observe up to 87% speedup in single precision (and 125% in double) over the *greedy scheduling*-classic ETM, and up to 40% in single (89% in double) over the *greedy scheduling*-aggressive ETM.



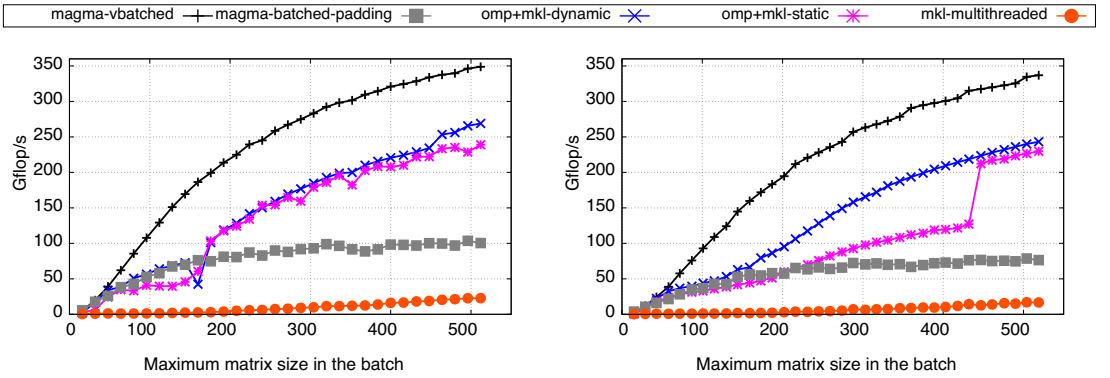
(a) Single precision (b) Double precision

Figure 5: Performance on batches of fixed size, batchCount = 3,000.

5 Performance Results

Our system setup is two 8-core Intel Sandy Bridge CPUs (Xeon E5-2670 - 2.6 GHz) using MKL 11.3.0, and a Kepler K40c GPU (ECC on) using CUDA 7.0. For performance comparisons against the CPU, we use: (1) A loop over matrices where each matrix is factorized using the multi-threaded MKL xpotrf routine executing on 16 cores, (2) An OpenMP loop statically unrolled among the 16 cores, where each core factorizes a single matrix at a time using the sequential MKL xpotrf routine, and (3) Similarly to (2), but the openMP loop is dynamically unrolled.

Figure 5 shows the performance of our fixed size batched routine, with the best configuration. We compare the performance against different CPU techniques, as well as our MAGMA



(a) Single precision - Uniform

(b) Single precision - Gaussian

(c) Double precision - Uniform

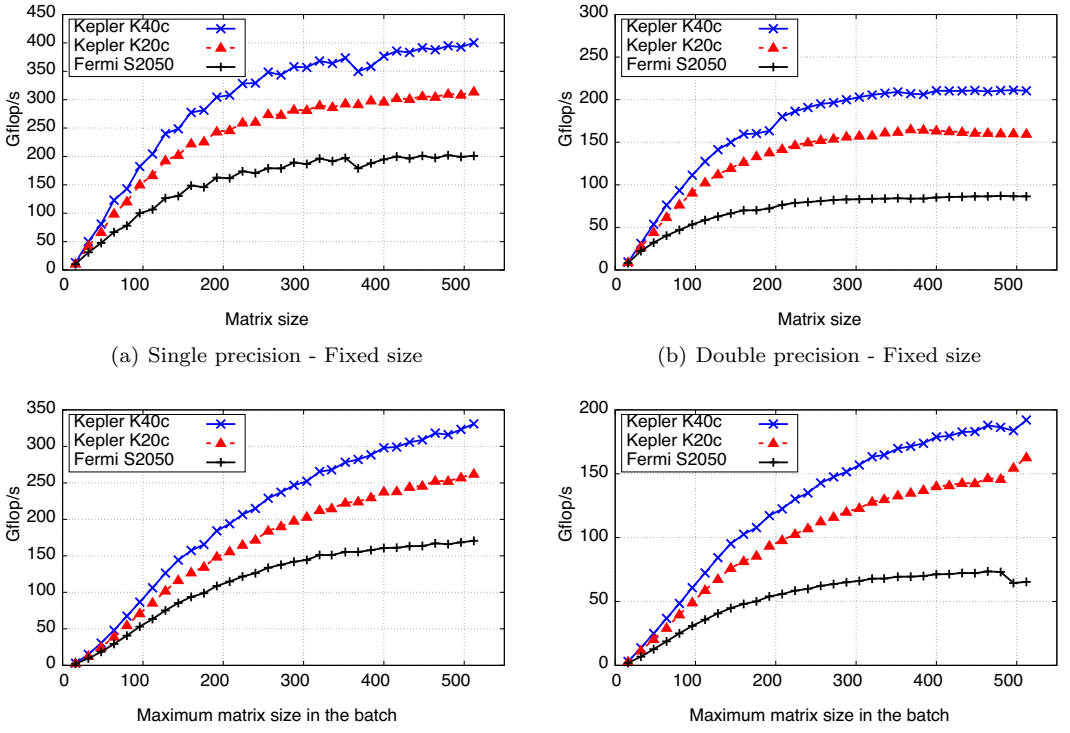
(d) Double precision - Gaussian

Figure 6: Performance on batches of variable sizes, batchCount = 3,000.

routine proposed in [5], which uses a separate BLAS kernels approach. It is clear that a multi-threaded CPU configuration is not a suitable solution for batched computation. If we compare the performance of the proposed magma-fused routine with magma-separate, we observe an interesting speedups in both single and double precisions. However, we observe that magma-separate starts to achieve better performance with matrices of size > 400 in double precision. This is due to the fact that, at this size, the computation is dominated by the level 3 BLAS routine, and thus the improvments seen at very small size became marginal here. This eventually leads to a final design that combines both approaches with a crossover point that switches which routine must be used. A similar crossover point is observed in single precision around size 700 (not shown). In most cases, the best competitor is omp-mkl-dynamic, against which the proposed approach achieves speedups that exceed 3× in single and 2× in double precision for small matrix sizes.

Figure 6 shows the overall performance of the vbatched routine against different configurations of MAGMA and MKL. The matrix sizes are generated using either uniform or Gaussian distribution, and we show the maximal size allowed on the `X` axis. Our vbatched routine is at least 3× faster than using the MAGMA fixed size routine with padding. The best competitor remains the OpenMP dynamic with MKL. The MAGMA vbatched routine is up to 2.3× faster in single (1.88× in double) for uniform distribution of matrix sizes, and up to 2.4× faster in single (1.83× in double) for a Gaussian distribution. Such impressive speedups occur if the range of sizes is up to 200. Speedups then decay asymptotically to reach 10%-30%. As we pointed out earlier for fixed size, there is a different technique [5] that can be used as the range

of sizes starts to exceed the crossover point of 500.



(a) Single precision - Fixed size

(b) Double precision - Fixed size

(c) Single precision - Variable size

(d) Double precision - Variable size

Figure 7: Performance on different GPUs, batchCount = 1,000.

Figure 7 shows the performance of the proposed kernel on different GPUs. The figure shows that our design takes advantage of the increasing compute power on the GPU. Variable size performance results (Figures 7(c) and 7(d)) use uniform distribution to generate the matrix sizes in each batch. In single precision, a K20c GPU can be up to 64% faster than a Fermi S2050 GPU, for both fixed and variable size test cases. A smaller performance gain, up to 30%, is observed moving from the K20c to the K40c GPU, since both GPUs have the same architecture but differ in the amount of resources available (number of multiprocessors and memory bandwidth). In double precision, we observe up to 1.97×/2.49× speedups by the K20c GPU over the Fermi GPU for fixed/variable size test cases. Similarly, a the performance is improved by just up to 30% by using the K40c GPU.

6 Conclusion and Future Work

This work presented a set of design ideas, tuning, and optimization techniques to address linear algebra operations on batches of fixed and variable size matrices on GPUs. The focus was on the Cholesky factorization. The paper shows that it is necessary to consider various design ideas in several combinations with tunable parameters in order to achieve high performance for such workloads. The proposed design proved to outperform all state-of-the-art techniques on both the CPU and the GPU. Future directions include benchmarking vbatched kernels against

size distributions that arise in real applications, and leveraging the same design concepts for the LU and QR algorithms.

Acknowledgments

This material is based on work supported by NSF under Grants No. CSR 1514286 and ACI-1339822, NVIDIA, and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Proc. of ICPP'11*, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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