Fast Cholesky Factorization on GPUs for Batch and Native Modes in MAGMA

Ahmad Abdelfattah[a,*], Azzam Haidar[a], Stanimire Tomov[a], Jack Dongarra[a,b,c]

[a] *Innovative Computing Laboratory, University of Tennessee, Knoxville, USA*
[b] *Oak Ridge National Laboratory, Oak Ridge, USA*
[c] *University of Manchester, UK*

Abstract

This paper presents a GPU-accelerated Cholesky factorization for two different modes of operation. The first one is the *batch* mode, where many independent factorizations on small matrices can be performed concurrently. This mode supports fixed size and variable size problems, and is found in many scientific applications. The second mode is the *native* mode, where one factorization is performed on a large matrix without any CPU involvement, which allows the CPU do other useful work. We show that, despite the different workloads, both modes of operation share a common code-base that uses the GPU only. We also show that the developed routines achieve significant speedups against a multicore CPU using the MKL library. This work is part of the MAGMA library.

Keywords: GPU computing, Cholesky factorization, batched execution

1. Introduction

High performance solutions of many small independent problems are crucial to many scientific applications, including astrophysics [1], quantum chemistry [2], metabolic networks [3], CFD and resulting PDEs through direct and

*Corresponding author
Email addresses: ahmad@icl.utk.edu (Ahmad Abdelfattah), haidar@icl.utk.edu (Azzam Haidar), tomov@icl.utk.edu (Stanimire Tomov), dongarra@icl.utk.edu (Jack Dongarra)

multifrontal solvers [4], high-order FEM schemes for hydrodynamics [5], direct-iterative preconditioned solvers [6], image [7] and signal processing [8]. The lack of parallelism in each of the small problems drives researchers to take advantage of the mutual independence among these problems, and develop specialized software that groups the computation into a single *batched* routine. Such software is relatively easy to develop for multicore CPUs using the existing optimized sequntial vendor libraries as building blocks. For example, considering Intel CPUs, a combination of the MKL library and OpenMP (scheduling individual cores dynamically across the input problems) usually achieves a very high performance, since most of the computation can be performed through the fast CPU cache. However, the same technique cannot be used for GPUs, fundamentally due to the lack of large caches.

On the other hand, there is a need to develop factorizations and linear system solvers that work entirely on the GPU, with no computational work submitted to the CPU. We call this mode of operation the *native* mode. Native execution on the GPU would allow the CPU to do other useful work. It can also be used in power sensitive environments, or in embedded systems with relatively slow CPUs, such as the Jetson TK1. Since GPUs are inherantly more energy efficient than CPUs, it is expected that a native code, although slower than a hybrid code using both CPU and GPU, be more energy efficient than the hybrid one [9].

While MAGMA [10] provides high performance LAPACK functionality on GPUs, most of the MAGMA routines are hybrid. This means that both the CPU and the GPU are engaged in performing the computation. This technique is proved to achieve very high performance on large problems [11]. However, it cannot be used efficiently to solve a batch of small problems due to the prohibitive cost that CPU-GPU communications will have for small problems. It cannot be used either in systems with low-end CPUs, or when the CPU is required to do other work. In general, we need a different design approach that uses the GPU only.

This paper presents a high performance Cholesky factorization that can run entirely on the GPU. We discuss two modes of operations. The first is the

batch[1] mode, where many small independent problems, of the same size or different sizes, are factorized concurrently. We extend the work presented in this direction [12], by showing a design that works for any size, not only those sizes where the panel fits into the GPU shared memory. The second mode of operation is called the *native* mode, where one large matrix is factorized using the GPU only. We show that the developed software for both modes share a common code-base while achieving high performance. Eventually, with this work integrated into the MAGMA library, we provide various choices to perform the factorization efficiently according to the different situations summarized in Figure 1.

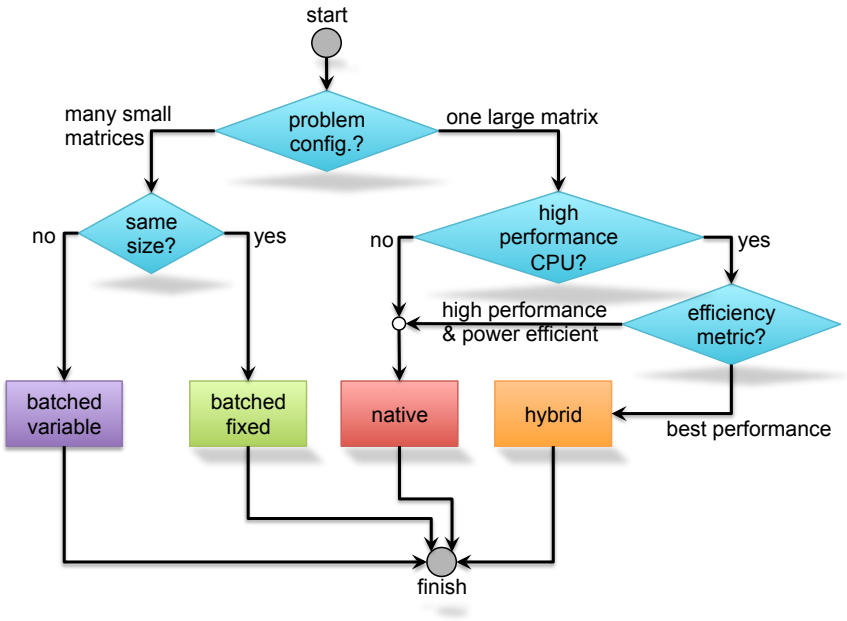


Figure 1: Decision flowchart for modes of operations in MAGMA

The paper starts by progressive optimization and tuning for the batched mode where all problems have the same size. We then proceed with the best configuration for fixed size problems and extend it to support variable size prob-

[1] We use the words *batch* and *batched* interchangeably.

lems. The native mode is realized by using the same code base with different tuning parameters to work on one large problem at a time. We show experimental results that demonstrate the performance of the proposed routines against state-of-the-art CPU and GPU solutions.

The rest of the paper is organized as follows. Section 2 discusses some previous efforts in GPU-accelerated matrix factorizations, with a focus on batched routines. Different modes of operation are discussed in Section 3. Section 4 presents a detailed description of the design approach, In Section 5, we discuss the obtained performance results. The paper ends with a conclusion in Section 6.

2. Related Work

Since the emergence of general purpose GPU computing (GPGPU), performance optimization of matrix factorization algorithms on GPUs has been a trending research topic. The hybrid algorithms in MAGMA represent the state-of-the-art in this area, where GPUs significantly accelerate the compute-intensive trailing updates [13, 14], and the CPU, in the meantime, prepares the next panel factorization [11]. It has been shown, however, that such an algorithmic design is not suitable for batched workloads [15], mainly due to the lack of parallelism in trailing matrix updates. This led to some research efforts that deal with small matrix computations on GPUs. Small LU factorizations were investigated by Villa et al. [16, 17] (for size up to 128), and Wainwright [18] (for sizes up to 32). Batch one-sided factorizations have been the focus of some research efforts, including Cholesky factorization ([19], [20]), and LU and QR factor factorizations ([21], [22], [23]). Some contributions focus on very small matrices, where all the computational stages are fused and performed by a single thread block (TB), as proposed in [20], [12], and [24].

The authors of this paper introduced variable size batched matrix multiplication (`GEMM`) [25] as a first step to develop LAPACK algorithms on variable size batched workloads. In addition, the work done by the authors [12] presented

optimized batched Cholesky factorization that had a limitation on the matrix sizes it could operate on. In fact, the kernel design proposed in [12] requires a dynamic shared memory allocation that is a function of the matrix size, meaning that it cannot work on any size. This paper extends such work and provides a design that can work on any matrix size, while supporting batches of fixed and variable sizes. It also uses the same code-base to develop native GPU factorization on very large matrices. We also show that the performance of the developed work is portable to three different GPU architectures, achieving high performance in all scenarios.

3. Modes of Operation

As mentioned earlier, we are designing a full GPU solution that can operate in two modes. We set a design goal to have a *unified code base* for both modes. As an example, Figure 2 shows the modes of operation for the `POTf2` algorithm, which is used to perform the Cholesky panel factorization. A code base, written using CUDA device routines, represents the core operation for one matrix. Such a code base is oblivious to any tuning parameters, which are defined later for each mode. The device routines are then wrapped into three CUDA kernels as shown in the figure. The native mode is the simplest, as it considers only one problem. The kernel passes the input arguments directly to the device routine, with no preprocessing required.

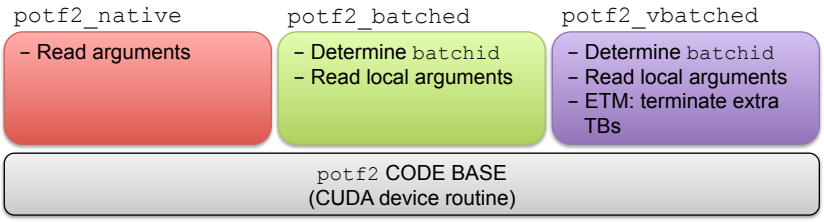


Figure 2: Modes of operations for the `POTF2` routine

The batched mode requires some preprocessing. The `potf2_batched` kernel is used for fixed size batched problems. It is internally organized into a number

of *subgrids*, each with a unique `batchid`. The `batchid` is used to map a certain matrix to a specific subgrid. The kernel reads the local input arguments of the assigned problem and passes them to the device routine. On the other hand, the variable size batched routine (`potf2_vbatched`) assumes that each matrix has a different size and leading dimension. The kernel is configured according to the largest matrix in the batch, which means that all subgrids can accomodate this matrix. An extra preprocessing step called *Early Termination Mechanism (ETM)* [25] [12] trims each subgrid according to the local size of the assigned problem. This step is necessary to avoid any runtime failures or memory access violations. After trimming subgrids, the kernel normally passes the local input arguments to the device routine to start the execution. We use the same approach in Figure 2 for all other building block routines discussed in this paper.

4. Algorithmic Design

This section describes the design details for Cholesky factorization in both batched and native modes. Our starting point is to have a high performance design and implementation for fixed size batched workloads. Such a design can then be ported easily to support variable size batched workloads, as well as the native mode for large matrices.

4.1. Overall Design

Figure 3 shows the overall design for the Cholesky factorization algorithm. The three main computational stages of the algorithm are the Cholesky panel factorization (`POTF2`), the triangular solve (`TRSM`), and the Hermitian rank-k update (`HERK`). The right side of the figure is the conventional way of performing the computation as three separate BLAS kernels, each of which is launched by the CPU. However, if the matrix size (`N`) is less than a threshold (`C`), then we use the blocked `POTF2` routine to perform the entire factorization. The blocked `POTF2` routine is recursively blocked to make use of level-3 BLAS operations, and

thus achieve high performance (left side of the figure). It consists of three stages (unblocked `POTF2`, `TRSM`, and `HERK`) that are fused together into a single kernel. The fusion of these routines helps save global memory traffic and reuse data in shared memory across the computational stages, which gives a big performance advantage for very small matrices. The blocked `POTF2` routine serves the panel factorization step on the right side of the figure if the matrix size is larger than `C`.

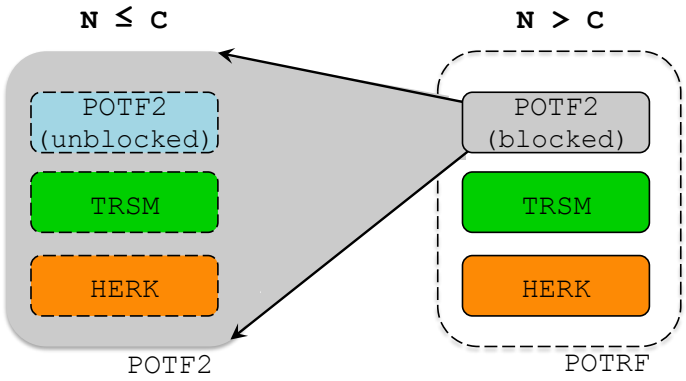


Figure 3: Overall design of the Cholesky factorization algorithm

The blocked `POTF2` routine is probably the most important routine in Figure 3. This is because it is used solely in the batched mode to perform the factorization on small matrices. In the native mode, it replaces the panel factorization done by the multicore CPU. Therefore, it has to be well optimized in order to deliver the best performance in the batched mode, and to introduce a minimum overhead to the execution time in the native mode. This is why we focus more on the design details of `POTF2` in the following subsections. The other routines (`TRSM` and `HERK`) are simpler to optimize due to their reliance on our batched `GEMM` kernel [25].

4.2. Cholesky Panel Factorization (`POTF2`)

Previous studies [22][12] showed that an efficient panel factorization of an $N \times N$ matrix should be recursively blocked, as shown in Figure 3, in order

to use the fused level-3 BLAS routines instead of the memory-bound level-2 BLAS operations. For example, thanks to the recursive blocking in Figure 3, trailing matrix updates inside the blocked `POTF2` routine use the `HERK` operation instead of the memory-bound Hermitian rank 1 update (the `HER` routine in level-2 BLAS). In addition, blocking at the kernel level follows a left-looking Cholesky factorization, with a blocking size `ib`, as shown in Algorithm 1, which is known to minimize data writes (in this case from GPU shared memory to GPU main memory).

Algorithm 1: The left looking fashion.

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

4.2.1. Kernel optimization

Using a left-looking Cholesky algorithm, the update writes a panel of size `N`×`ib` in the fast shared memory instead of the main memory, so that the unblocked `POTF2` stage can execute directly in shared memory. Note that `N` and `ib` control the amount of the required shared memory We developed an optimized and customized *fused kernel* that first performs the update (`HERK`), and keeps the updated panel in shared memory to be used by the unblocked `POTF2` and the `TRSM` steps. The cost of the left looking algorithm is dominated by the update step (`HERK`). The panel `C`, shown in Figure 4, is updated as $C = C - A \times B^T$. A double buffering scheme is employed to perform the update in steps of `lb`,

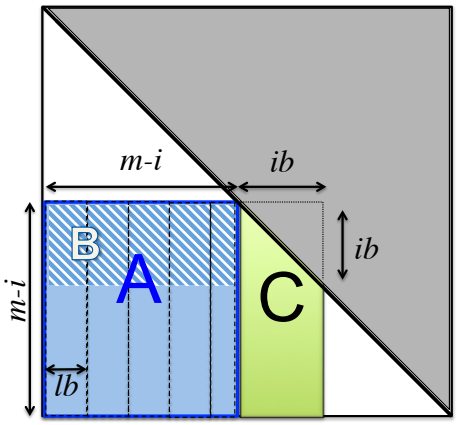


Figure 4: Left-looking Cholesky factorization

which minimizes the update cost, as described in Algorithm 2. For clarity, we prefix the data array by "r" and "s" to denote register and shared memory, respectively. We prefetch data from `A` into register a 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 into the shared memory array `sB`. Once the update is finished, the factorization (`POTF2` and `TRSM`) is performed as one operation on the panel `C`, held in shared memory.

4.2.2. Loop-inclusive vs. Loop-exclusive Kernels

In addition of fusing the computational steps of a single iteration in Algorithm 1, another level of fusion is to merge all iterations together into one GPU kernel. The motivation behind the *loop-inclusive* design is to maximize the reuse of data, not only in the computation of a single iteration, but also among iterations. For example, the factorized panel of iteration $i-1$ (which is in shared memory) can be reused 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 4) by directly accessing it from fast shared memory. However, such a design has a downside regarding occupancy, in terms of the number of factorizations that

Algorithm 2: The fused kernel correspond to one iteration of Algorithm 1.

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

can be performed on a single Streaming Multiprocessor (SM). A *loop-inclusive* kernel should be configured based on the tallest sub-panel (i.e., based on the size N). As we execute more iterations of Algorithm 1, more threads become idle and more of the reserved shared memory becomes unused. In other words, the kernel runs entirely on the occupancy level defined by the resource requirements of the first iteration.

The analysis of the occupancy and the throughput of the loop-fusion technique motivated the development of a more occupancy-oriented design, which we call the *loop-exclusive* kernel. In this regard, each iteration of Algorithm 1 corresponds to a kernel launch that has the exact resources required by this iteration, with no idle threads and no waste in shared memory. While this design leads to reloading the previous panel from the main memory, such extra cost is alleviated thanks to the double buffering technique in the update step. We conducted a tuning experiment for both kernels. The results, summarized in Figure 5, 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.

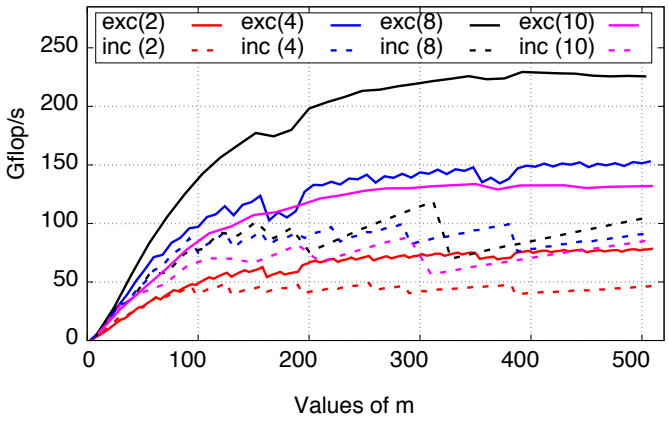


Figure 5: Performance tuning of *loop-inclusive*(inc) and *loop-exclusive*(exc) kernels on a K40c GPU, `batchCount` = 3000. The value of `ib` is shown between brackets. Results are shown for double precision.

4.2.3. Greedy vs. Lazy Scheduling for `potf2_vbatched`

Following a loop-exclusive design, the `potf2_vbatched` kernel is called as many times as required by the largest matrix in the batch. In this regard, there is a degree of freedom in determining when to start the factorization for smaller matrices. We present two different techniques for scheduling those factorizations. 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 Thread Block (TB) assigned to it in the following iterations becomes idle and is terminated using the ETM technique. With greedy scheduling, completion of factorization on individual matrices occurs at different iterations. A drawback of this problem is that smaller matrices are factorized alongside larger matrices in the same iteration. Since the shared memory allocation has to accomodate the tallest sub-panel, greedy scheduling results in wasted shared memory for smaller subpanels, which in turn results in low occupancy. The downside of greedy scheduling motivated the design of the opposite technique, which we call *lazy scheduling*. Individual factorizations start at different iter-

ations, such that they all finish at the last iteration. At each iteration, lazy scheduling considers only matrices with local sizes within the range `max_N -i` to `max_N -i+ib`, and ignores other matrices using ETMs. As a result, the resource allocation per iterations (number of threads and shared memory) is closest to the optimal configuration. In other words, lazy scheduling technique always ensures better occupancy than greedy scheduling, and is in fact more robust to the variations of sizes in the batch.

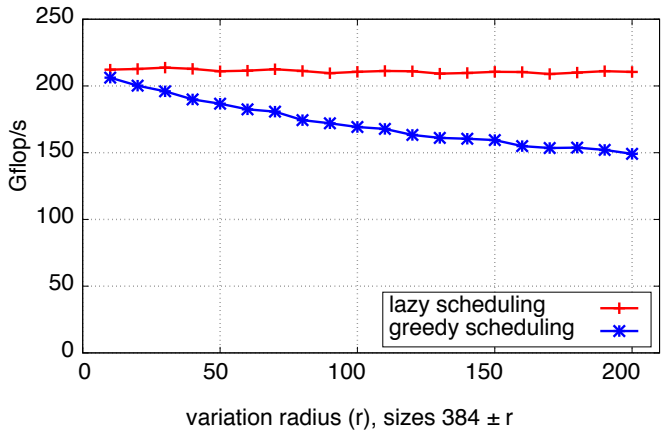


Figure 6: Performance robustness test of greedy and lazy scheduling techniques, `batchCount` = 3000. Sizes are randomly sampled within the (384±r) interval.

Figure 6 shows a performance robustness test for the greedy and the lazy scheduling techniques. We conducted performance tests on 3000 matrices, with a mean size of 384 and a variation of $\pm r$, so that the interval $(384 \pm r)$ is randomly sampled 3000 times to construct the batch. The figure shows that if the variation is small, both scheduling techniques score roughly the same performance. However, as we increase r, the greedy scheduling loses performance due to the larger variation in sizes, which causes bad occupancy. In fact, greedy scheduling loses up to 25% of its performance, while the lazy scheduling technique is capable of maintaining a stable performance regardless of the size variations.

The discussion of different scheduling techniques do not apply to the `TRSM` and `HERK` routines. Unlike the `potf2_vbatched` kernel which uses dynamic

shared memory allocation based on the `max_N`, both routines use static shared memory allocations based on tuning parameters rather than the input sizes. Therefore, their occupancy are controlled by the tuning parameters, and are minimally affected by size variations.

4.3. Triangular Solve (TRSM)

The `TRSM` routine starts by inverting square diagonals blocks of size `tri_nb` in the triangular matrix. The inversion is performed using a batched triangular inversion routine (`TRTRI`). The solution is, therefore, obtained by multiplying these inverses (which are stored in a workspace) with the corresponding sub-matrices of the right hand side matrix. A carefully tuned `GEMM` [25] is used to perform the multiplication. The value of `tri_nb` is chosen to let `GEMM` dominate the computation involved in the `TRSM` routine. Figure 7 shows the impact of the parameter `tri_nb` on performance. The figure represent a typical test case that is invoked by the Cholesky factorization if the panel size is set to 256. The best configuration of MAGMA is 10-17% times faster than a MKL+OpenMP, and 4-5× faster than CUBLAS.

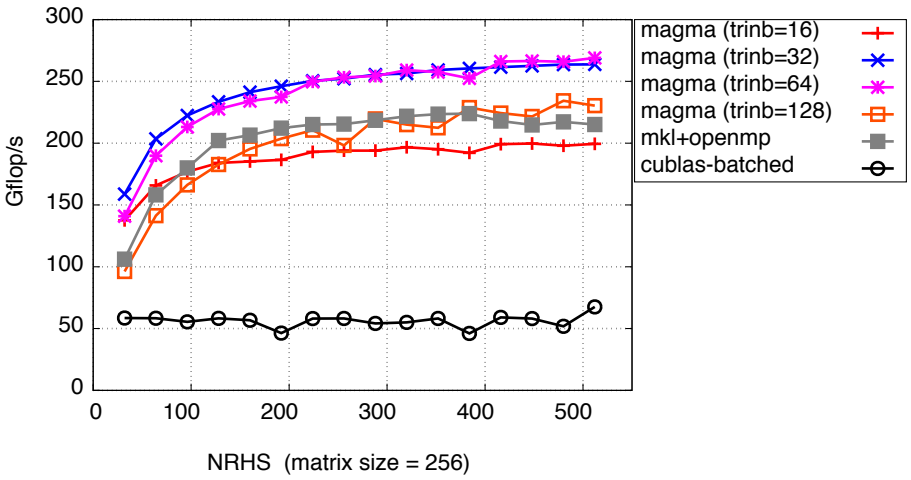


Figure 7: Performance tuning of batched `TRSM`, `batchCount` = 1000. Experiments are performed on a 1 K40c GPU and 16-core Intel Sandy Bridge CPU. Results are shown for double precision.

4.4. Hermitian Rank-k Update (HERK)

The HERK routine is a key to high performance in Cholesky factorization, as it dominates the trailing matrix updates, which represent the most compute-intensive phase of the computation if the matrix is larger than the crossover point C. MAGMA uses one of two HERK implementations based on the input size. The first one is a MAGMA kernel that uses the same code-base and tuning parameters of the GEMM kernel proposed in [25]. Such kernel performs a normal GEMM operation except for a preprocessing layer that terminates thread blocks writing to the upper/lower triangular part of the matrix. This means that the kernel inherits all the optimization techniques and tuning efforts that have been done for the GEMM kernel. The second implementation uses concurrent CUDA streams to launch multiple instances of the CUBLAS HERK kernel. The motivation behind the second implementation is that it achieves very high performance when the input size becomes relatively large. MAGMA transparently decides which approach to use based on the input size.

5. Performance Results

5.1. System Setup

Performance experiments are conducted on a 16-core Intel Sandy Bridge CPU (Intel Xeon E5-2670, running at 2.6 GHz), and three GPUs that are summarized in Table 1. The Titan-X and the GTX1080 GPUs do not support native double precision arithmetic, which means that it is emulated by software and is not expected to deliver any good performance. Our test environment uses Intel MKL Library 11.3.0 for CPU tests and CUDA Toolkit 8.0RC for GPU tests.

5.2. Performance of The Batched Routines

Figure 8 compares the performance of the blocked POTF2 kernel (when used solely), against the performance of the full POTRF routine (which internally calls POTF2). The figure shows that, if the matrix size is below some crossover point,

Name	Architecture	Compute Capability	CUDA Cores	Frequency
K40c	Kepler	3.5	2880	0.75 GHz
Titan-X	Maxwell	5.2	3072	1.08 GHz
GTX1080	Pascal	6.1	2560	1.73 GHz

Table 1: Summary of the GPUs used in performance tests.

it is better to perform the entire factorization using the blocked `POTF2` kernel only. Operating on such small sizes, most of the operations become more memory-bound. It is, therefore, important to save any unnecessary global memory traffic. The blocked `POTF2` routine does exactly that by fusing all operations into one kernel, and increasing data reuse in shared memory among the different computational stages. Considering matrix sizes less than 400, Figure 8 shows significant speedups against the full `POTRF`, ranging from 1.12× up to 4×, as the matrix size gets smaller.

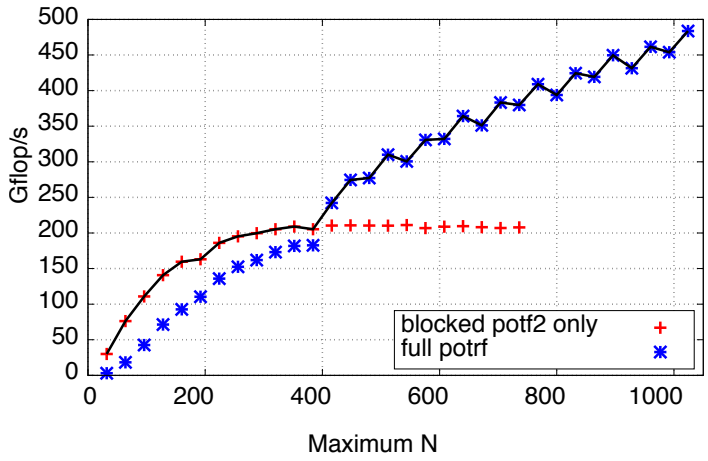
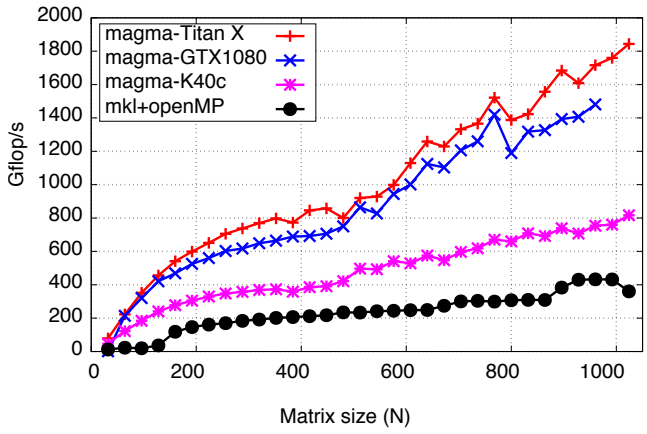
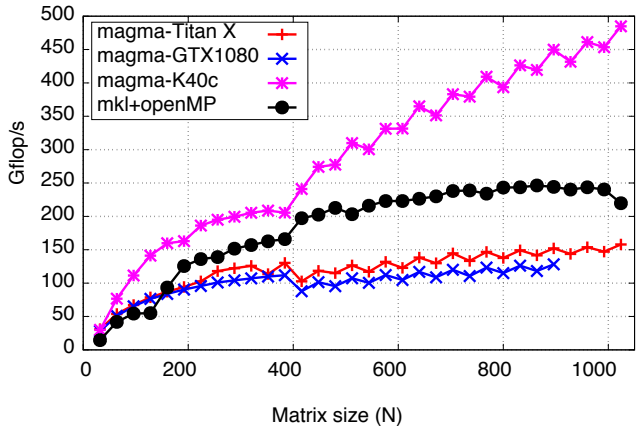


Figure 8: Example crossover point for `dpotrf_batched`, `batchCount`=1000.

Following the design strategy in Figure 3, the blocked `POTF2` routine cannot be used for any problem size, because its shared memory requirements are function of the matrix size. Moreover, Figure 8 shows that its performance starts to stagnate as the problem becomes more compute-bound. At this stage, our



(a) Single precision



(b) Double precision

Figure 9: Performance of the fixed size batched Cholesky factorization, `batchCount`=1000.

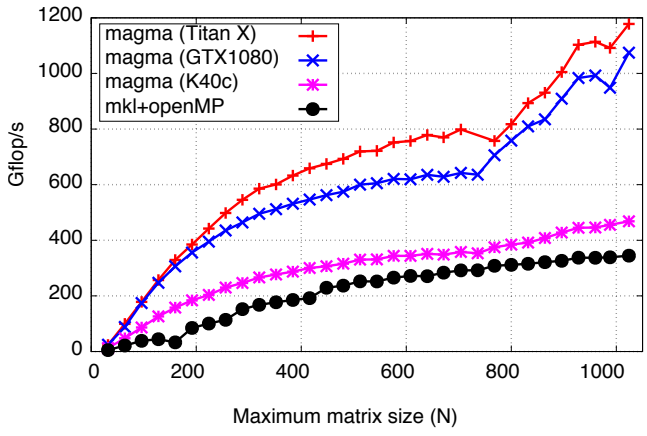
final solution switches to the full `POTRF` implementation. The crossover points in MAGMA are tunable according to the precision and the GPU model. Figures 9 and 10 shows the final performance of the batched Cholesky factorization for fixed and variable size problems, respectively. A first observation is that the MAGMA performance on the Titan-X is better than on the GTX1080, although the latter is the latest GPU architecture to date. The reason behind such a behavior is that GTX1080 is a low-end configuration of the Pascal architecture. In

fact, the Titan-X used in this work has more CUDA cores (refer to table 1), and, as our benchmarks show, has a higher memory bandwidth. We also point out that some graphs for the GTX1080 are not complete because it has less memory than the other two GPUs. Both figures show that the MAGMA performance in single precision is portable on three different GPU architectures. MAGMA is 1.75-2.3× faster than the CPU implementation using MKL and OpenMP. It also receives a performance boost on the newer architectures, scoring 3-4.4× and 4-5× speedups on the GTX1080 and the Titan-X GPUs, respectively. Despite the expected low performance achieved in double precision using the GTX1080 and the Titan-X GPUs, MAGMA outperforms the CPU implementation on the K40c GPU, scoring speedups ranging from 1.2× up to 2.5×.

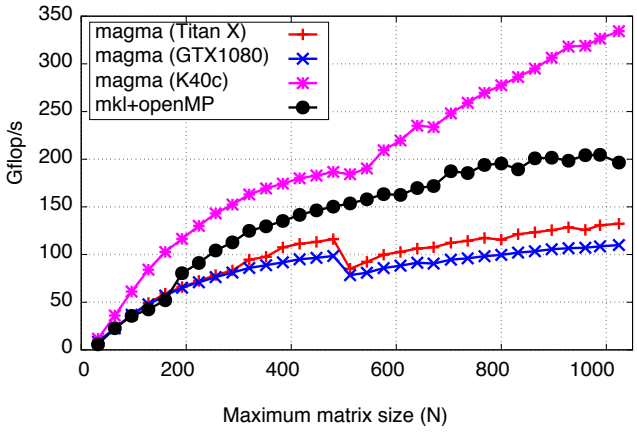
For the experiments for variable size batched problems, we constructed every test batch by randomly sampling the interval [1:N], where N is varied on the x-axis of Figures 10a and 10b. Similar to the fixed size batched routine, running the MAGMA *vbatched* routine on Titan-X/GTX1080 is 2-4× faster than MKL in single precision, and is 1.2-2.2× faster on the K40c GPU. In double precision, MAGMA achieves a similar 1.2-2× speedups against MKL when running on the K40c GPU.

5.3. Performance of The Native Routines

Figure 11 shows the performance of the MAGMA native Cholesky factorization. Since this test involves one factorization of a large matrix, we switch the CPU implementation to use all cores together to do the factorization, which means that the MKL configuration is switched to multithreaded. We also point out that the native MAGMA routines, while sharing the same code base with the batched routines, they usually use a larger panel size, in order to have a more compute intensive operation on the trailing updates. In single precision, the speedups scored by MAGMA are up to 4.2×, 8.8×, and 9.8× on the K40c, GTX1080, and Titan-X GPUs, respectively. Similar to the batched routines, speedups in double precision are scored on the K40c GPU only, where MAGMA is up to 4.1× faster than the multithreaded MKL implementation.



(a) Single precision

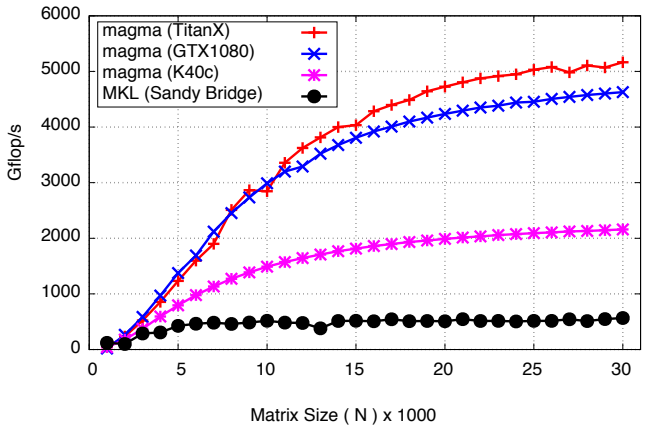


(b) Double precision

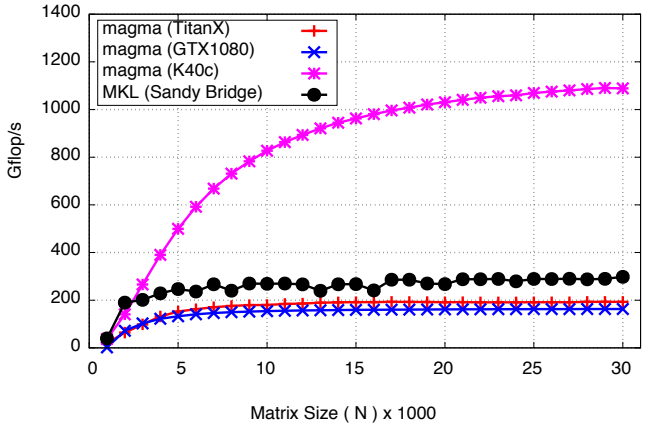
Figure 10: Performance of the variable size batched Cholesky factorization, `batchCount`=1000. Matrix sizes in each batch are randomly sampled between 1 and the maximum size shown on the x-axis.

6. Conclusion and Future Work

This paper introduced a high performance Cholesky factorization that is de-
330 signed for GPUs. The proposed work can operate in a batch mode, factorizing many small matrices of similar or different sizes, or in a native mode, factorizing one large matrix using the GPU only. The paper introduces a common code-base that can be used in both modes, and can deliver high performance against



(a) Single precision



(b) Double precision

Figure 11: Performance of the native GPU Cholesky factorization.

state-of-the-art solutions using multicore CPUs. Future directions include applying the same design concept to broader functionalities (e.g. LU and QR factorization), and developing an autotuning framework to guarantee portable performance across many GPU architectures.

Acknowledgement

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

References

[1] O. Messer, J. Harris, S. Parete-Koon, 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.

[2] 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. Sadayappanc, A. Sibiryakovc, Automatic code generation for many-body electronic structure methods: the tensor contraction engine, Molecular Physics 104 (2) (2006) 211–228.

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

[4] S. N. YERALAN, T. A. DAVIS, S.-L. W. M, S. RANKA, Algorithm 9xx: Sparse QR Factorization on the GPU, ACM Transactions on Mathematical Software.
URL `http://faculty.cse.tamu.edu/davis/publications_files/qrgpu_revised.pdf`

[5] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, J. Dongarra, A step towards energy efficient computing: Redesigning a hydrodynamic ap-

plication on CPU-GPU, in: IEEE 28th International Parallel Distributed Processing Symposium (IPDPS), 2014.

[6] E.-J. Im, K. Yelick, R. Vuduc, Sparsity: Optimization framework for sparse matrix kernels, Int. J. High Perform. Comput. Appl. 18 (1) (2004) 135–158. doi:10.1177/1094342004041296.
URL http://dx.doi.org/10.1177/1094342004041296

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

[8] M. Anderson, D. Sheffield, 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. Haidar, S. Tomov, P. Luszczek, J. Dongarra, Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing, in: 2015 IEEE High Performance Extreme Computing Conference (HPEC 15), (Best Paper Award), IEEE, IEEE, Waltham, MA, 2015.

[10] Matrix algebra on GPU and multicore architectures (MAGMA), available at http://icl.cs.utk.edu/magma/ (2014).

[11] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, Dense linear algebra solvers for multicore with GPU accelerators, in: Proc. of the IEEE IPDPS'10, IEEE Computer Society, Atlanta, GA, 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[12] A. Abdelfattah, A. Haidar, S. Tomov, J. J. Dongarra, Performance Tuning and Optimization Techniques of Fixed and Variable Size Batched Cholesky Factorization on GPUs, in: International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, 2016, pp. 119–130. doi:10.1016/j.procs.2016.05.303.

[13] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, in: W. mei W. Hwu (Ed.), GPU Computing Gems, Vol. 2, Morgan Kaufmann, 2010.

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

[15] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, On the Development of Variable Size Batched Computation for Heterogeneous Parallel Architectures, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016, 2016, pp. 1249–1258. doi:10.1109/IPDPSW.2016.190.

[16] V. Oreste, M. Fatica, N. A. Gawande, A. Tumeo, Power/performance tradeoffs of small batched LU based solvers on GPUs, in: 19th International Conference on Parallel Processing, Euro-Par 2013, Vol. 8097 of Lecture Notes in Computer Science, Aachen, Germany, 2013, pp. 813–825.

[17] V. Oreste, N. A. Gawande, A. Tumeo, Accelerating subsurface transport simulation on heterogeneous clusters, in: IEEE International Conference on Cluster Computing (CLUSTER 2013), Indianapolis, Indiana, 2013.

[18] I. Wainwright, Optimized LU-decomposition with full pivot for small batched matrices, gTC'13 – ID S3069 (April, 2013).
URL http://on-demand.gputechconf.com/gtc/2013/presentations/S3069-LU-Decomposition-Small-Batched-Matrices.pdf

[19] T. Dong, A. Haidar, S. Tomov, J. Dongarra, A fast batched Cholesky factorization on a GPU, in: Proc. of 2014 International Conference on Parallel Processing (ICPP-2014), 2014.

[20] J. Kurzak, H. Anzt, M. Gates, J. Dongarra, 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:10.1109/TPDS.2015.2481890.

[21] A. Haidar, T. Dong, S. Tomov, P. Luszczek, J. Dongarra, A framework for batched and gpu-resident factorization algorithms applied to block householder transformations, in: J. M. Kunkel, T. Ludwig (Eds.), High Performance Computing, Vol. 9137 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 31–47. doi:10.1007/978-3-319-20119-1_3.

[22] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched matrix computations on hardware accelerators based on gpus, IJHPCA 29 (2) (2015) 193–208. doi:10.1177/1094342014567546.

[23] A. Haidar, P. Luszczek, S. Tomov, 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, ACM, ACM, San Francisco, CA, 2015.

[24] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, J. J. Dongarra, High-Performance Matrix-Matrix Multiplications of Very Small Matrices, in: Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings, 2016, pp. 659–671. doi:10.1007/978-3-319-43659-3_48.

[25] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, 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, 2016, pp. 21–38. doi:10.1007/978-3-319-41321-1_2.