Progressive Optimization of Batched LU Factorization on GPUs

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

***Abstract*—This paper presents a progressive approach for optimizing the batched LU factorization on graphics processing units (GPUs). The paper shows that the reliance on level-3 BLAS routines for performance does not really pay off, and that it is indeed important to pay attention to the memory-bound part of the algorithm, especially when the problem size is very small. In this context, we develop a size-aware multi-level blocking technique that utilizes different granularities for kernel fusion according to the problem size. Our experiments, which are conducted on a Tesla V100 GPU, show that the multi-level blocking technique achieves speedups for single/double precisions that are up to $3.28\times/2.69\times$ against the generic LAPACK-style implementation. It is also up to $8.72\times/7.2\times$ faster than the cuBLAS library for single and double precisions, respectively. The developed solution is integrated into the open-source MAGMA library.**

***Index Terms*—LU factorization, batch computation, GPU computing**

I. Introduction and Related Work

Primarily driven by scientific applications, the demand for high performance batched linear algebra routines has witnessed significant increase in the past five years. Such a demand was also motivated by the relatively poor performance of existing solutions for workloads that consist of batches of small matrices. Such workloads are popular in sparse direct solvers, tensor contractions, machine learning, quantum chemistry, and others.

The research community, as well as vendors of high-end computing systems, have recognized both the need and the challenge of optimizing batched linear algebra operations. This is why vendor math libraries, such as MKL [1], cuBLAS [2], and rocBLAS [3], have recently added some batched BLAS routines. While the coverage of batched routines varies from a vendor to another, some operations represent a common denominator across vendor libraries. As an example, the batched matrix multiplication routine (i.e., batched GEMM) exists in all of the aforementioned libraries as probably the most important operation in dense linear algebra, as well as deep learning.

Outside the vendor software landscape, there have been many developments in batched linear algebra across the research community. Some early research shows that scientists, motivated by lack of coverage of numerical software, have developed in-house batched routines for their specific needs. Such developments usually focus on one specific algorithm, and often target a specific range of sizes that the application requires. As an example, batched LU factorization has been used in subsurface transport simulation [4], [5], where the sizes are assumed to be up to 128×128. A batched Cholesky factorization and the triangular solve (for square sizes up to 100) have also been used to accelerate an alternating least square (ALS) solver that generates product recommendations on the basis of implicit feedback datasets [6], [7]. As a result, open-source libraries, such as MAGMA [8], provide optimized batched routines for many standard BLAS and LAPACK operations. The batched GEMM routine is at the core of many dense linear algebra algorithms, and its optimization and tuning is crucial for batched routines [9]. In particular, optimizing batched GEMM for extremely small sizes is very important in tensor contraction problems [10]–[12]. Other research efforts targeted batched matrix factorizations [13], singular-value decomposition (SVD) algorithms [14], and hierarchical matrices [15]. While most of the research efforts focus on batches of fixed problem size, there have been some efforts that targeted different problem sizes in the same batch [16]. We also observe that most of the research efforts target GPUs, since OpenMP-based solutions tend to deliver a very good performance on CPUs. However, there have been some contributions showing that dedicated batched routines on CPUs are still beneficial [17].

One particular challenge in optimizing batched routines is the problem size. In the past, performance optimizations used to target relatively large matrices, where the compute-bound BLAS operations (e.g., GEMM and triangular solve [TRSM]) dominate the execution time. However, batched routines specifically target small problem sizes, where the batched BLAS routines cannot operate close to the performance peak of the hardware. In fact, the performance of routines like batched GEMM can be bound by the memory bandwidth, especially for very small matrices [12]. In such situations, components of the algorithm other than batched BLAS become an important factor in achieving a high performance. As an example, batched one-sided matrix factorization could become dominated by the panel factorization step rather than by the batched rank-k updates. The batched panel factorization is a memory-bound problem, where data reuse is even more important for performance. In this regard, designing a single solution (i.e., GPU kernel) that assumes nothing about the input size is probably a bad strategy for most batched routines.

This paper studies the optimization of the batched LU factorization algorithm. The paper focuses on optimizations for graphics processing units (GPUs), where data reuse is more challenging due to the relatively small sizes of fast memory levels (e.g., caches and shared memories). The paper shows that the panel factorization becomes such a critical component for batch routines. We show that there are four different solutions for such a step in the LU algorithm, each having its assumptions about the problem size and the level of data reuse. We call this approach *the multi-level blocking technique*, which proves to be significantly faster than a generic design that assumes nothing about the input size. It also outperforms the vendor supplied routine by speedups up to 8.72×/7.2× in single/double precisions on a Tesla V100 GPU. The developed solution has been integrated into the publicly available MAGMA library.

II. Background

This section provides a background about the LU factorization algorithm, and the challenges of optimizing it for a batch of relatively small matrices. Throughout the paper, we will be showing experimental results in double precision only, except for the final performance results in Section IV, which are reported for single and double precisions.

The LU factorization is one of the most important algorithms in dense linear algebra for solving linear systems of equations. Its standard implementation in the LAPACK library can be found in the (**GETRF**) routine. The algorithm computes the L and U factors of a general matrix A, such that A = P×L×U. The matrix P is a permutation matrix which holds the pivoting information. The factors L and U are unit lower triangular and upper triangular, respectively. Instead of storing a permutation matrix P, the standard LAPACK implementation uses a *pivot vector* to store the row interchanges performed to maintain numerical stability.

Almost all factorization algorithms in LAPACK have at least two different designs. The *unblocked design* uses exclusively memory-bound building blocks from level-1 and level-2 BLAS, which makes it, in turn, a memory-bound design. There are four main steps in performing the unblocked LU factorization, which are summarized in Algorithm 1 (assuming double precision). At each iteration, the **IDAMAX** routine locates the maximum absolute value across the current column. The **DSWAP** step then exchanges the current row with the row holding the pivot. The third step scales the current column (**DSCAL**) and the forth step is a rank-1 update **DGER** of the trailing submatrix.

On the other hand, the blocked version of the algorithm is built to take advantage of the compute-bound routines in level-3 BLAS, especially matrix multiplication (**DGEMM**), in order to achieve high performance. Figure 1 and Algorithm 2 show a simplified version of the blocked algorithm. At each iteration, the design follows a *panel-update* design pattern. The panel stage uses the unblocked version in Algorithm 1, followed by performing the row interchanges necessary to the left and right

Algorithm 1: Unblocked LU factorization (**DGETF2**)

```
for i=1 to min(M, N) do
    piv = IDAMAX( abs(A[i:M,i]) )
    ipiv[i] = piv
    if abs(A[piv,i]) = 0 then
        // U is singular, report error.
    end
    DSWAP: Exchange A[i, 1:N] and A[piv, 1:N]
    DSCAL: A[i+1:M,i] ×= (1 / A[i,i])
    DGER: A[i+1:M,i+1:N] -= A[i+1:M,i]×A[i,i+1:N]
end
```

of the current panel. The update stage consists of a triangular solve (**DTRSM**) followed by a rank-k update (**DGEMM**). It is often best practice to replace **DGETF2** in Algorithm 2 with the recursive panel routine (**DGETRF2**) if large nb is required for performance. The **DGETRF2** routine further subdivides the wide panel recursively so that compute bound kernels (**TRSM** and **GEMM**) are used as much as possible.

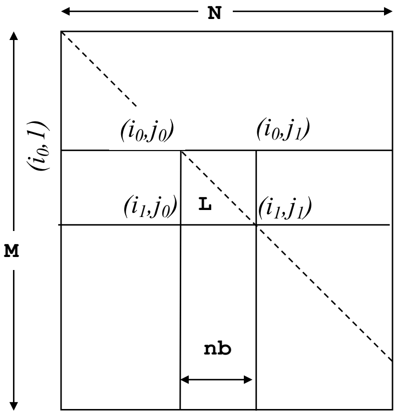

Fig. 1. LU factorization scheme

Algorithm 2: Blocked LU factorization (**DGETRF**) for the matrix in Figure 1

```
for k=1 to min(M, N) step nb do
    panel: DGETF2(A[$i_0$:M,$j_0$:$j_1$], ipiv)
    swap-left: DLASWP( A[$i_0$:M, 1:$j_0$], ipiv )
    swap-right: DLASWP( A[$i_0$:M, $j_1$:N], ipiv )
    DTRSM: A[$i_0$:$i_1$, $j_1$:N] = $L^{-1}$ × A[$i_0$:$i_1$, $j_1$:N]
    DGEMM: A[$i_1$:M, $j_1$:N] $-=$
            A[$i_1$:M, $j_0$:$j_1$]×A[$i_0$:$i_1$, $j_1$:N]
end
```

Perhaps the most important step in optimizing the batched LU factorization is to make sure that the batched GEMM routine delivers a relatively high performance for the typical use cases in the algorithm. The LU factorization requires a rank-k update of the form $C = C - A \times B$, such that A is tall and skinny (e.g., $m \times nb$), while B can be either short and wide (e.g., $nb \times n$), or square (e.g. $nb \times nb$). The former scenario is the typical use case for the blocked design shown in Algorithm 2, while the latter one is the typical use case for the update step in the recursive panel factorization. Figure 2 shows both scenarios. It is important, therefore, to make sure that the batched GEMM routine is properly tuned for such cases. The values of m and n are arbitrary (the problem size), while nb is a design parameter. Typical values are 8, 16, 32, or 128, depending on the problem size.

We investigate the performance of batched GEMM in cuBLAS and MAGMA for these use cases. Figure 3 shows sample results for the selected use cases when $nb = 16$. It turns out that the cuBLAS batched GEMM routine does not always deliver the best performance, especially when B is small and square. The MAGMA routine scores speedups up to $1.8x$ in such cases. This is due to a careful autotuning that pays attention to such typical use cases [9]. We also observe that it becomes beneficial to switch to cuBLAS when the value of nb is larger than 32. As for the batched TRSM routine, the MAGMA batched TRSM routine is always faster than cuBLAS. The former leverages the performance of the batched GEMM routine by utilizing a *solve-update* pattern, where the *solve* part finds the solution of a small triangular linear system, while the *update* part calls batched GEMM to update the remaining "unsolved" part of the right hand sides.

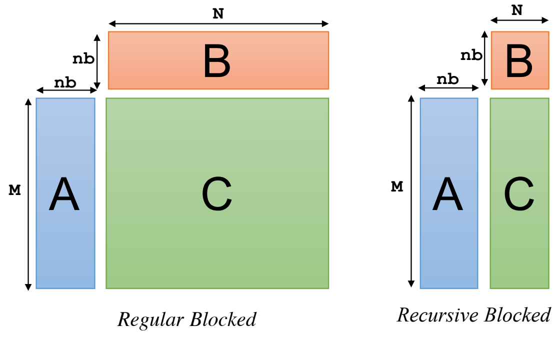

Fig. 2. Use cases of the batched GEMM routine inside the batched LU factorization.

The remaining two stages are the swapping routine (**DLASWP**), and the batched panel factorization (**DGETF2**). The swapping routine is purely memory-bound, and often leads to non-coalesced memory accesses due to the row interchanges on the column-major layout of the matrix. We adopt the "parallel swapping" technique [13] which can avoid non-coalesced memory writes. Now we are left with the batched panel factorization, which is the primary focus of the paper.

The panel factorization uses the recursive implementation (**DGETRF2**), which in turn calls the unblocked code of Algorithm 1 (**DGETF2**) when the panel width reaches a stopping criteria. Perhaps the most challenging part about the LU panel factorization is pivoting. As Algorithm 1 shows, each iteration requires finding a pivot across the entire column of the remaining submatrix. From a software library perspective, the problem size can be arbitrarily large, even for a batched

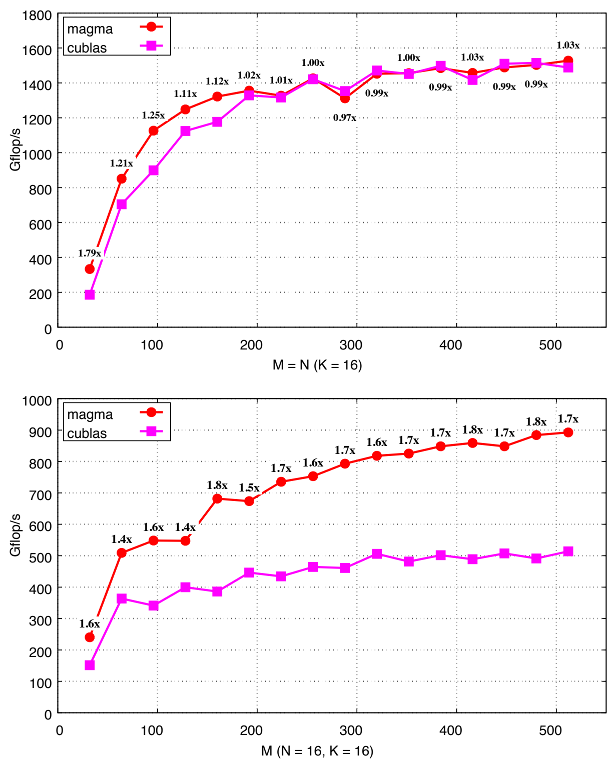

Fig. 3. Batched DGEMM performance for LU factorization with $nb = 16$, batchCount = 500, Tesla V100 GPU, CUDA-10.1.

routine. This means that the **IDAMAX** routine should deal with arbitrarily large vectors that may not fit into the GPU shared memory or registers. Such a generic design, though covering all matrix sizes, will be suboptimal for small sizes from a performance perspective.

The work done in [13] implements the generic (LAPACK-style) batched panel factorization, which uses separate routines for separate computational stages. We profiled the generic panel design to see how much time is spent during each computational phase. Figure 4 shows the normalized breakdown of the execution time for different matrix sizes. We identify five main categories:

1) **GEMM**: This is the rank-k update in **DGETRF2**, which is the recursive blocked shape shown in Figure 1.
2) **TRSM**: The triangular solve that precedes the rank-k update. Note that this is the "solve-only" component of the operation. The other component is matrix multiplication, which is included in the **GEMM** category.
3) **Rank-1 Updates**: This category combines the column scaling and the rank 1 update (**DSCAL + DGER**).
4) **SWAP**: This category combines two types of swapping kernels. The first one is **DSWAP**, which exchanges two rows at a time and exists in **DGETF2**. The second is the parallel swap version of **DLASWP** [13].
5) **IDAMAX**: This is the kernel that performs the pivot search.

Figure 4 shows that compute-bound kernels (**GEMM+TRSM**)

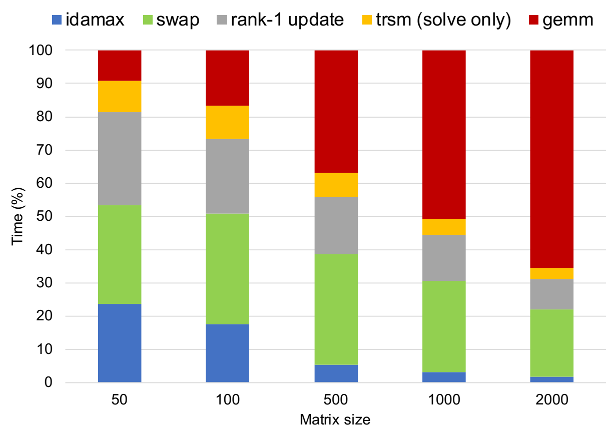

Fig. 4. Percentage time spent in different stages of the batched LU panel factorization. Results are shown for 500 square matrices on a Tesla V100 GPU (CUDA 10.1).

dominate the execution time for large matrices. Such a contribution diminishes consistently as the matrix size becomes smaller. For matrices of size 50 and 100, more than 80% and 70% of the total time is spent in memory-bound kernels, respectively. This means that even a carefully tuned and optimized level-3 batched BLAS routine may not be of the greatest importance for batched linear algebra algorithms, especially on very small sizes. In this work, we look at optimization techniques that can take advantage of small matrix sizes in the panel factorization stage. Such techniques assume that part of the matrix is small enough to be cached in fast memory levels, so that several computational stages are executed on such memory levels in a single context. This indeed leads to customized kernels that fuse two or more computational steps from the panel factorization, which deviates from the legacy LAPACK-style implementation. However, the final outcome is usually a significant gain in performance due to the increased data reuse.

III. Multi-level Blocking

In addition to the generic design discussed in the previous section, we discuss three different blocking levels that control the granularity of cached data, and the level of kernel fusion. More specifically, we recognize "column blocking," "panel blocking," and "matrix blocking." Some of those designs have been addressed in previous papers, which will be pointed out when they are discussed.

A. Column Blocking

This is the least restricted level of blocking, as it can apply to a relatively wide range of sizes. In this design, we assume that a column of the matrix is cached in shared memory while it is being updated and then factorized. The design uses a *left-looking* scheme which reorders the steps of Algorithm 1 so that **DGER** is the first step in each iteration (except for the first column). The column is first read into shared memory. The rank-1 update adds all the necessary accumulations to the cached column, which is then followed by a pivot search, a swap of two rows, a scaling operation, and a write back to the GPU global memory. All these operations are fused into a single kernel, which clearly saves unnecessary memory traffic regarding the current column.

In terms of limitations, our design assumes that one thread block performs the column factorization, such that a single thread is responsible for a single element of the column. It turns out that such a design is currently limited by the maximum number of threads in a single CUDA thread block, which is currently 1024. Shared memory is not a bottleneck for this kernel, since a thread block uses at most 15% or less of the available shared memory.

Figure 5 shows the performance improvements of column blocking against the generic (no blocking) design. We show the performance of the recursive panel factorization only. The performance gains vary between 10% and 40%. It is clear that the smaller the matrix, the larger the speedup, since memory traffic savings become more important. We notice that the asymptotic performance does not yield any gains. The reason is that column blocking saves memory traffic for only one column, which means that other dominant operations, like the rank-1 updates, do not really benefit from this blocking level. Another reason is that, as the column becomes larger, the shared memory requirements per thread block may limit the ability of the CUDA runtime to schedule many thread blocks on the same multiprocessor. This also explains the performance drops encountered in the performance graph of the column-blocking kernel.

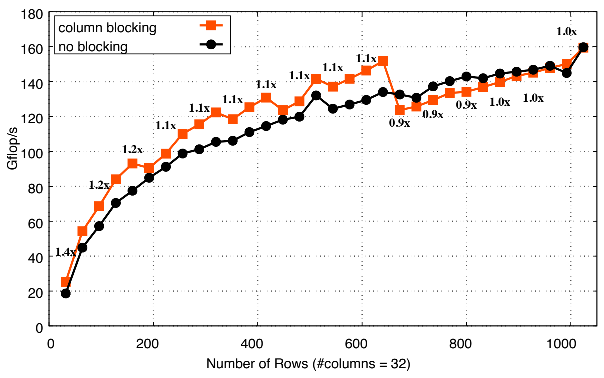

Fig. 5. Performance of column blocking vs. no blocking. Results are shown for 500 matrices on a Tesla V100 GPU (CUDA 10.1).

B. Panel Blocking

Assuming even smaller sizes, the $m \times nb$ panel may fit entirely in shared memory or registers. In such a case, all the computational stages of the **DGETF2** routine are fused into one kernel. This yields an optimal memory traffic for the panel (but not for the whole matrix), since it is read and written exactly once. In our design, we cache the entire panel in the register file. Using m threads per thread block, each thread holds a complete row of the panel of length nb. To ensure proper loop unrolling by the compiler for the main loop in Algorithm 1, the number of columns of the panel is assumed to be a compile-time parameter that is passed through C++ templates. Thanks to the recursive panel factorization technique, we can instantiate few kernel instances, in our case for $nb \in [1 : 32]$. The kernel makes use of a *lazy pivoting* technique [18], which delays all the row interchanges at the very end of the kernel before writing the factorized panel to the global memory of the GPU.

Figure 6 shows significant performance improvements for panel blocking vs. column blocking. The speedups range between $1.5\times$ and $4.8\times$. The oscillatory behavior of the panel blocking kernel is due to the occupancy of the kernel, which worsens as the panel gets bigger, which in turn reduces the number of concurrent factorizations per multiprocessor. The performance drops are more dramatic than column blocking since the memory requirements are $\mathcal{O}(m \times nb)$, unlike the $\mathcal{O}(m)$ memory requirement for column blocking. In terms of limitations, the panel-blocking kernel covers a smaller range of panel heights, and is limited by the capacity of the register file. Considering double precision as an example, the largest panel possible is 512×32.

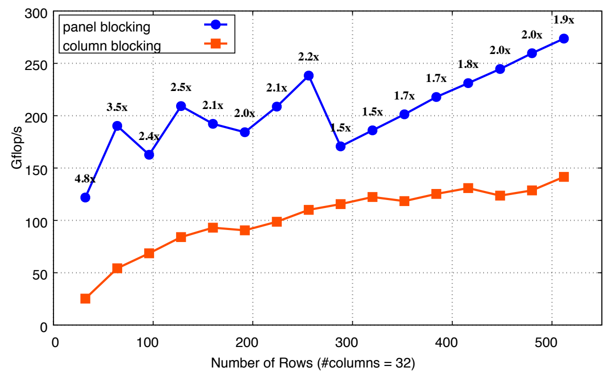

Fig. 6. Performance of panel blocking vs. column blocking. Results are shown for 500 matrices on a Tesla V100 GPU (CUDA 10.1).

C. Matrix Blocking

The final level of blocking deals with tiny matrices that can be fully cached throughout the whole factorization process. Since the LU factorization algorithm mostly deals with square matrices for solving linear systems, we consider only square tiny matrices. Such a kernels has been discussed in a previous effort [18], [19], and it ensures an optimal memory traffic for each matrix. The difference between this kernel and the panel-blocking kernel is that the former assumes square matrices, which makes it possible to have an unrolled code for the pivot search. The latter does not precompile information about the panel height, and so the pivot search is done through a generic CUDA device routine. Figure 7 shows that the matrix-blocking kernel is on average twice as fast as the panel blocking kernel. In terms of limitations, this kernel is limited by the size of the register file, and is considered only for sizes less than or equal to a warp (i.e., 32).

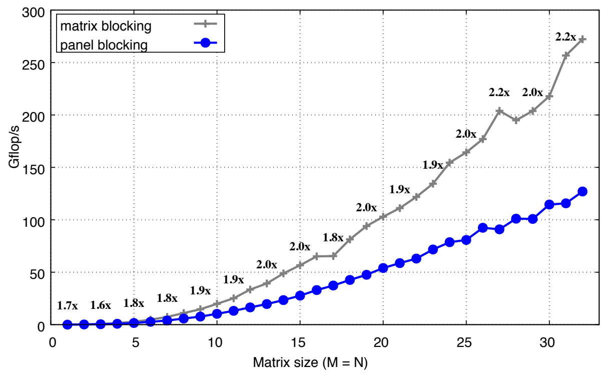

Fig. 7. Performance of matrix blocking vs. panel blocking. Results are shown for 500 matrices on a Tesla V100 GPU (CUDA 10.1).

IV. Final Performance Results

Figures 8 and 9 show the final performance results for the optimized batched LU factorization, for single and double precisions, respectively. The "magma-optimized" solution represents the combined implementation for all four designs (three levels of blocking + no blocking). Its performance is compared against the generic "no blocking" design, as well as against the vendor routine from the cuBLAS library. As expected, the improvements made by the magma-optimized routine are more significant for small sizes. In fact, as the problem size gets smaller, the performance gains grow from 1.6% up to $3.28\times$ for single precision, and from 1.7% to $2.69\times$ for double precision. These numbers do not include the tiny sizes (e.g., at 32), where we observe a $21.1\times$ speedup in single precision, and a $22.65\times$ speedup in double precision.

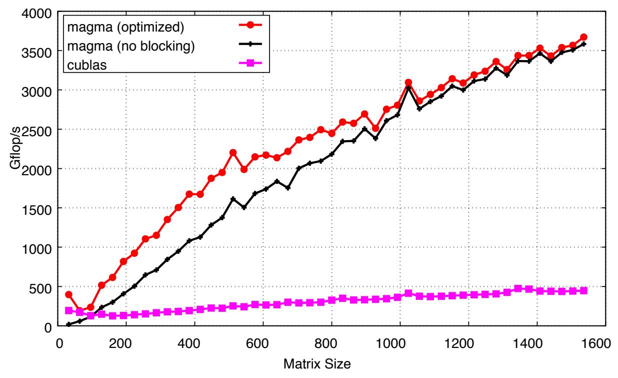

Fig. 8. Final performance of the batched LU factorization (single precision). Results are shown for 500 matrices on a Tesla V100 GPU (CUDA 10.1).

Such performance gains are due to the specialized kernels that can take advantage of the small panel sizes, and improve the data reuse accordingly. The magma-optimized routine is also significantly faster than cuBLAS, scoring speedups that range between $1.11\times$ and $8.72\times$ in single precision, and between $1.78\times$ and $7.22\times$ in double precision. The main reason behind such huge speedups against cuBLAS is that the latter does not use any level-3 BLAS routines, according to our profiling results.

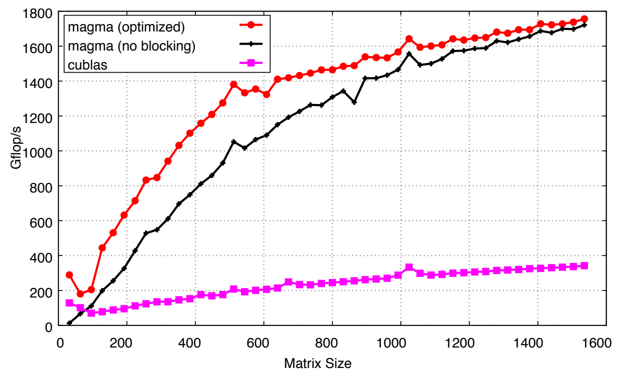

Fig. 9. Final performance of the batched LU factorization (double precision). Results are shown for 500 matrices on a Tesla V100 GPU (CUDA 10.1).

V. Conclusion and Future Work

In this paper, we showed that batch routines often require a set of different designs that target different size ranges. Each design has its own assumptions about the granularity of blocking, kernel fusion, and data reuse. Having a solution that is size-aware is often superior to a unified design that does not consider special optimizations for small sizes. By applying this methodology to the batched LU factorization problem, we show that a multi-level blocking scheme is up to 3.28×/2.69× faster than the unified "generic" design, and is up to 8.72×/7.2× faster than the cuBLAS library for single/double precisions. Future directions would target similar design strategies for other linear algebra algorithms, such as the QR factorization and SVD problems.

Acknowledgment

This work is partially supported by NSF grant No. OAC-1740250 and CSR 1514286, NVIDIA, and by the Exascale Computing Project (17-SC-20-SC).

References

[1] "Intel Math Kernel Library," available at http://software.intel.com/intel-mkl/.

[2] "NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS)," available at https://developer.nvidia.com/cublas.

[3] "rocBLAS, Next Generation BLAS Implementation for ROCm Platform," available at https://github.com/ROCmSoftwarePlatform/rocBLAS.

[4] O. Villa, M. Fatica, N. Gawande, and A. Tumeo, *Power/Performance Trade-Offs of Small Batched LU Based Solvers on GPUs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 813–825. [Online]. Available: https://doi.org/10.1007/978-3-642-40047-6_81

[5] O. Villa, N. Gawande, and A. Tumeo, "Accelerating subsurface transport simulation on heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, 2013, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CLUSTER.2013.6702656

[6] M. Gates, H. Anzt, J. Kurzak, and J. J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, 2015, pp. 667–676. [Online]. Available: https://doi.org/10.1109/BigData.2015.7363811

[7] J. Kurzak, H. Anzt, M. Gates, and J. J. Dongarra, "Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2036–2048, 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2015.2481890

[8] "MAGMA: Matrix Algebra on GPU and Multicore Architectures," available at http://icl.cs.utk.edu/magma/.

[9] A. Abdelfattah, A. Haidar, S. Tomov, and J. 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41321-1_2

[10] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302

[11] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor Contractions with Extended BLAS Kernels on CPU and GPU," in *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, 2016, pp. 193–202. [Online]. Available: https://doi.org/10.1109/HiPC.2016.031

[12] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices," *Parallel Computing*, vol. 81, pp. 1 – 21, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118301091

[13] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *IJHPCA*, vol. 29, no. 2, pp. 193–208, 2015. [Online]. Available: https://doi.org/10.1177/1094342014567546

[14] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, "Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression," *Parallel Computing*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819117301461

[15] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, *Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures*. Cham: Springer International Publishing, 2017, pp. 22–40. [Online]. Available: https://doi.org/10.1007/978-3-319-58667-0_2

[16] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs," in *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, 2017, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079103

[17] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing Vector-friendly Compact BLAS and LAPACK Kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 55:1–55:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126941

[18] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, 2017, pp. 606–615. [Online]. Available: https://doi.org/10.1016/j.procs.2017.05.250

[19] ——, "Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures," *Journal of Computational Science*, vol. 26, pp. 226–236, 2018. [Online]. Available: https://doi.org/10.1016/j.jocs.2018.01.005