# Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures

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

This paper presents new algorithmic approaches and optimization techniques for LU factorization and matrix inversion of millions of very small matrices using GPUs. These problems appear in many scientific applications including astrophysics and generation of block-Jacobi preconditioners. We show that, for very small problem sizes, design and optimization of GPU kernels require a mindset different from the one usually used when designing LAPACK algorithms for GPUs. Techniques for optimal memory traffic, register blocking, and tunable concurrency are incorporated in our proposed design. We also take advantage of the small matrix sizes to eliminate the intermediate row interchanges in both the factorization and inversion kernels. The proposed GPU kernels achieve performance speedups vs. CUBLAS of up to 6× for the factorization, and 14× for the inversion, using double precision arithmetic on a Pascal P100 GPU.

## 1 Introduction

There is a continuously increasing interest in providing high performance solutions of many small independent problems. This is a relatively new kind of computational workload in the realm of dense linear algebra. Instead of solving one big dense matrix, which is served well by software packages such as `LAPACK`, `ScaLAPACK`, `PLASMA`, `MAGMA`, and others, such workloads, which we call *batched workloads*, focus on very small dense matrices that are independent from each other. Such pattern of workloads appear in many large-scale scientific computing applications, e.g., quantum chemistry [8], sparse direct solvers [21], astrophysics [15], and signal processing [5].

Our focus in this paper is on the `LU` factorization and matrix inversion of millions of very small dense matrices. In addition to the applications already mentioned, the factorization is of particular importance to sparse direct solvers, such as the multifrontal LU solver that can be found in SuiteSparse [1]. Batched matrix inversion is also important in sparse matrix computation, such as the generation of block-Jacobi preconditioners.

The design of GPU kernels that perform this kind of tasks is not trivial. In fact, using existing numerical software for small matrix computation rarely results in good performance.

The reason is that such software is optimized to handle large sizes of data with embarrassingly parallel computational kernels such as matrix multiplication (GEMM). In dense linear algebra, the idea of blocking enables the use of compute-intensive trailing matrix updates that are well-suited for GPUs [19]. However, when the sizes are very small, the LAPACK-style blocking cannot be applied, since the blocking sizes will be very small, leading to memory-bound trailing matrix updates. We need a different design mindset in order to develop high performance GPU kernels to handle such type of workloads.

This paper presents highly optimized GPU kernels for batched LU factorization and matrix inversion. The kernels can handle millions of matrices of sizes up to 32. We show a step-by-step methodology, where incremental improvements in the kernel design lead to incremental performance gains. We justify all of our design choices by showing the performance before and after every incremental improvement. One of the main advantages of our design is the elimination of the expensive intermediate row interchanges by delaying them to the end of the kernel, which leads to a much faster kernel that produces the exact same result of a LAPACK-style LU-factorization and inversion. The performance results show significant speedups against the vendor-supplied CUBLAS kernels using a Pascal P100 GPU.

## 2 Related Work

GPUs are well suited for embarrassingly parallel tasks on large chunks of data, such as matrix multiplication(GEMM) [13][18]. The nearly optimal performance of GEMM on GPUs allowed the development of high performance LAPACK algorithms [19][11]. Following a growing interest in performing matrix computations on large batches of small matrices, an obvious start was to develop a batched GEMM routine for GPUs. This has already been addressed in literature [7][4], leading to developments of higher-level batched LAPACK algorithms. For example, Kurzak et al. [12] introduced batched Cholesky factorization for single precision up to sizes $100 \times 100$, while Dong et al. provided a more generic design [9]. Some contributions also proved the ability of GPUs to deal with variable size batched workloads [4][3].

Wang et al. [20] introduced FPGA-based parallel LU factorization of large sparse matrices, where the algorithm is reduced to factorizing many small matrices concurrently. Villa et al. [16] developed a GPU-based batched LU factorization, which has been used in subsurface transport simulation, where many chemical and microbiological reactions in a flow path are simulated in parallel [17]. Batched matrix inversion is also introduced in the context of generating block-Jacobi preconditioners [6]. The work presented by Haidar et al. [10] provides a batched LU factorization that can deal with any matrix size. It uses a blocking technique, similar to LAPACK, that achieves high performance based on an optimized batched GEMM kernel. However, such a blocking technique does not result in high performance for tiny sizes. For example, while the design proposed in [10] outperforms CUBLAS for the midrange of sizes, it trails CUBLAS in performance for the sizes we focus on (up to 32). The work presented by Masliah et al. [14], which proposed optimized batched GEMM, shows that a different approach is required for optimizing kernels for such small range of sizes. In this paper, we present an approach different from LAPACK-style blocking that achieves high performance for batched LU and inversion.

## 3 Background

The LU factorization computes the L and U factors of a general matrix A, such that A = P×L×U, where P is a permutation matrix that reflects the row interchanges associated with pivoting.

The matrix L is unit lower triangular, while U is upper triangular. The permutation matrix P is stored in a condensed format using a *pivot vector* (IPIV), such that for $i \in \{1, 2, \cdots, N\}$, row i has been exchanged with row IPIV(i).

Following an unblocked factorization, there are four main steps to perform the factorization. Assuming double precision arithmetic, these steps are, (1) locating the maximum absolute value in the current column (IDAMAX), (2) row swapping (DLASWP), (3) scaling the current column (DSCAL), and (4) rank-1 update (DGER). Algorithm 1 shows the factorization using the four steps. We point out that there is no need to use LAPACK-style blocking techniques. A very small matrix can be kept in registers or shared memory during the entire factorization. This means that it is makes no difference if blocking is used or not, since the data are accessed anyway from a fast memory level.

**Algorithm 1:** Unblocked LU factorization

```
for i=1 to N do
    IPIV[i] = max_id = IDAMAX( ABS( A[i:N,i] ) )
    if ABS(A[max_id,i]) = ZERO then
        U is singular, report error.
    end
    DLASWP: Exchange rows A[i, 1:N] and A[max_id, 1:N]
    DSCAL: A[i+1:N,i] = A[i+1:N,i] * (1 / A[i,i])
    DGER: A[i+1:N,i+1:N] = A[i+1:N,i+1:N] - A[i+1:N,i]×A[i,i+1:N]
end
```

In order to compute the matrix inverse, we solve for B, such that A×B = I, where I is the identity matrix. After performing the LU factorization, we have L×U×B = $P^{-1}$. The solution for B can be performed using two triangular solve (TRSM) operations. The first solves the lower triangular system of equations L×Y = $P^{-1}$, in order to get Y. The second solves the upper triangular sytem U×B = Y. Note that the first triangular solve with $P^{-1}$ is not a full TRSM, since nearly half of the resulting matrix is always zeros. This is the standard inversion in LAPACK using the GETRI routine.

However, the above methodology is too generic and does not take advantage of the small matrix sizes. Such a generic approach leads to four kernel launches on the GPU (LU factorization of A, row interchanges of I, and two TRSM calls). The generic approach, thus, leads to redundant memory traffic. We propose to use a fused approach where the factorization and inversion occur in the same kernel. The kernel performs an LU factorization on the augmented N×2N matrix [A | I], which is entirely stored on shared memory or registers. The factorization on the augmented matrix implicitly computes the result of the first triangular solve, meaning that the augmented matrix is overwritten by [LU | Y]. The only step left is to solve for the inverse using U and Y.

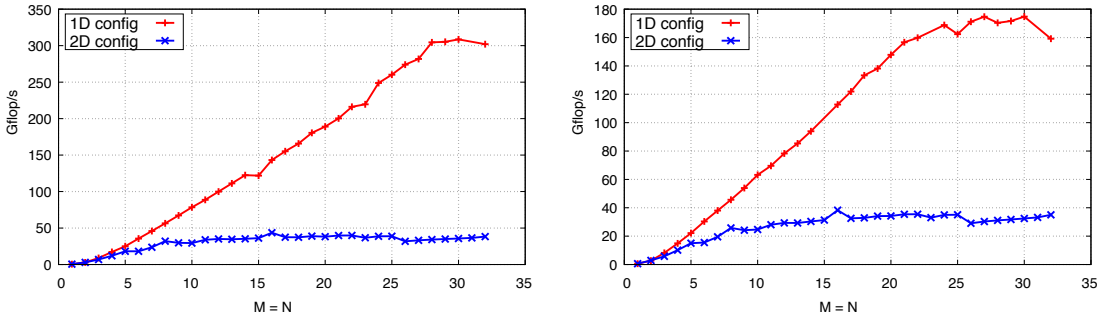
# 4 Algorithmic Design

**General Design Criteria:** For all the designs discussed throughout the paper, each matrix is factorized/inverted by exactly one thread block (TB). We launch as many TBs as the number of matrices. The grid of the kernel is, therefore, organized as a 1D array of TBs of size batchCount. Each TB has a unique batchid. Another common design criteria is that each matrix is read/written exactly once from/to the GPU global memory, leading to an optimal

memory traffic. The entire matrix will be kept in a fast memory level (registers or shared memory) throughout the lifetime of the TB assigned to it. Additionally, every kernel is written using C++ templates, with the matrix size being a template parameter that has to be specified during compile time. For such range of very small sizes, this is a crucial decision. It enables unrolling every loop in the kernel, thus minimizing the cost of executing integer and branch instructions. Finally, all kernels use unblocked computation, since the data is always access through fast memory levels.

**Thread Configuration: 1D vs. 2D:** The configuration of TBs can be either 1D or 2D. The 2D configuration simplifies the kernel design. For example, using an `N×N` threads, reading and writing the matrix is done using exactly one line of code, since each thread reads/writes one element. Similarly, the trailing matrix updates are fully parallelized among threads. However, there are some reasons to reject the 2D configuration. The first is using heavyweight TBs, in terms of the number of threads, which limits the ability of the CUDA runtime to schedule multiple TBs per streaming multiprocessor (SM), leading to poor occupancy. Another reason is the use of synchronization points. A 2D configuration certainly exceeds the warp size (32 threads) for most sizes, leading to necessary barriers whenever threads need to share data. This is an advantage of the 1D configuration, which keeps the code free of any synchronization points. Another advantage for the 1D configuration is that it assigns more work per thread, which eventually leads to a better instruction-level parallelism.

To better justify our judgement, we conducted a performance experiment for both configurations, where the matrix is kept in shared memory. Figure 1 shows the results, where we observe a clear advantage for the 1D configuration. We can observe huge speedups of at least 6× in single precision and 4× in double precision.




(a) Single precision (b) Double precision

Figure 1: Performance comparison between the 1D configuration and the 2D configuration of the LU factorization kernel. Both kernels use shared memory. Results are for 1M matrices on a Pascal P100 GPU.

**Register file vs. Shared memory:** Another design choice is to decide the storage resources. Considering the small sizes of interest, the whole matrix can be stored either in shared memory or in registers. Registers have faster acces time, but shared memory provides more flexible access patterns. Storing the matrix in registers leads to each thread possessing exactly one row of the matrix. If a thread wants to access a row that belongs to another thread, it needs to either use shared memory or the warp shuffle operations. This is a typical case in `LU` factorization, which relies on row interchanges for numerical stability. On the other hand, if the matrix is stored in shared memory, every thread can access any element of the matrix, and row swapping is easier to implement. However, shared memory is slower than registers and have limited number of ports. In addition, the shared memory resources per SM (e.g. 64 KB

on a P100 GPU) are much less than the register resources (256 KB). Therefore, using shared memory can limit the occupancy. To summarize, there is a clear tradeoff between using shared memory and registers. We decided to develop both versions for benchmarking purposes.

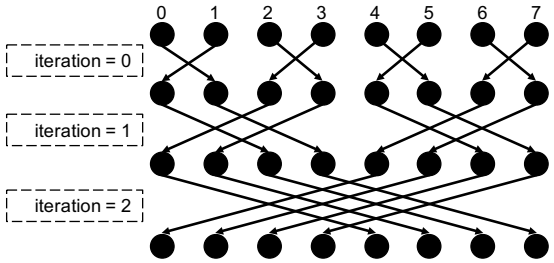


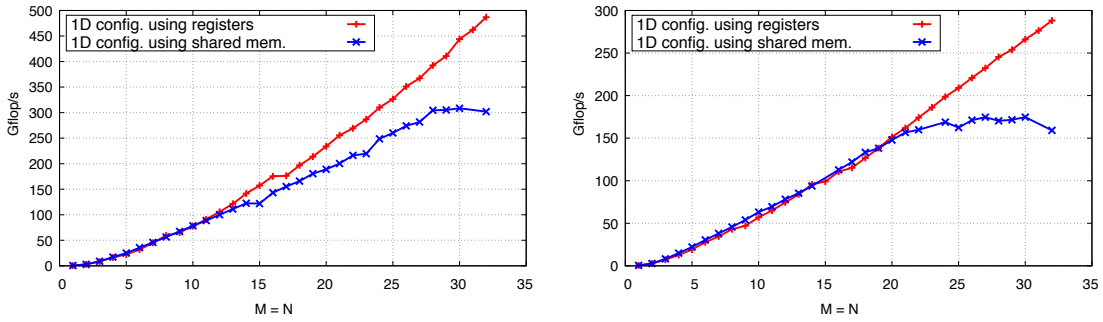

Figure 2: Data exchange among 8 threads using butterfly warp shuffle operations.

**Initial Designs:** The shared memory version is relatively simpler than the register version. The matrix is loaded into shared memory column-by-column to respect coalesced global memory access. At iteration `i`, every thread searches for the maximum absolute value in the column `A(i:N,i)`. Recall that threads within a warp are implicitly synchronized, and so there is no overhead of doing redundant work. After finding the pivot row, all threads collaboratively perform the swapping. They read the row `A(i,1:N)` (one element per thread), and swap it with the row `A(max_id,1:N)`, where `max_id` is the ID of the pivot row. Only threads who have thread IDs larger than `i` perform the trailing matrix updates, scaling the column `A(i+1:N, i)`, and updating the matrix `A(i+1:N, i+1:N)` column-by-column.

The register version mainly differs in finding the pivot row. This is because elements of the same column belong to different threads. In order to perform this step, we use the butterfly pattern of the warp shuffle operations, which performs the search in $log_2(\texttt{N})$ steps. Figure 2 shows the data exchange pattern of the butterfly shuffle operation for a vector of length 8. At the end of this step, all threads have individually determined the location of the pivot row. A hardware limitation of this technique is that it has to involve a number of threads that is power of 2. Another limitation is that the shuffle operations supports only 32-bit data exchange. Other precisions have to be handled on the software level. Figure 3 shows a performance comparison between the two design choices. While the shared memory version of the kernel is partly competitive, it fails to keep up the performance as the sizes go up. This is due to the limited shared memory resources compared to the size of the register file, which limits the kernel occupancy. The register version has asymptotic speedups of about 60% in single precision and 70% in double precision. Eventually, our decision was to continue improvement of the register version of the kernel.

# 5 Further Improvements

**Tunable Concurrency:** While the 1D configuration assigns more work per thread, the next optimization controls the amount of work and the level of concurrency on the level of TBs. The optimization is a generalization over the original design idea that assigns exactly one matrix per TB. Instead, we adopt a new kernel structure that allows a TB to factorize multiple matrices concurrently. The new TB configuration merges multiple TBs of the original design, to eventually have a 2D configuration of that performs independent factorizations. Figure 4 shows the new kernel structure, where a tuning parameter ($\texttt{nF}_{\texttt{TB}}$) controls the number of factorization per TB. The new kernel remains free of any synchronization barriers. The




(a) Single precision (b) Double precision

Figure 3: Performance comparison between performing the LU factorization in registers and in shared memory. Both kernels use a 1D thread configuration. Results are for 1M matrices on a P100 GPU.

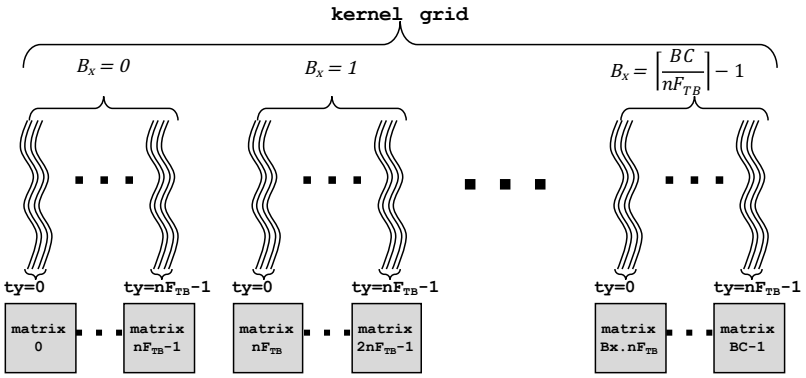
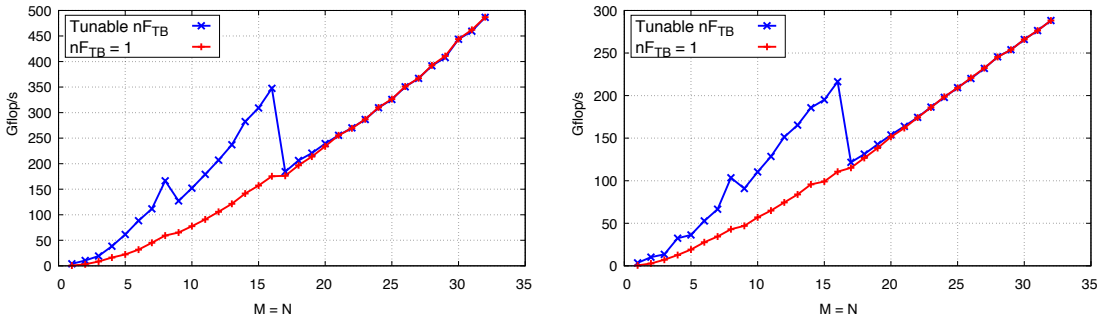



Figure 4: Structure of the GPU kernel using tunable concurrency.

motivation behind such structure is to improve the occupancy when the number of threads is too low. We conducted an autotuning experiment that performs a full sweep over values of $\texttt{nF}_{\texttt{TB}}$ from 1 to 16. We observed that after this range, there is no gain in performance. Figure 5 shows the performance gains obtained by having a tunable number of factorizations per TB. We only show the final results of tuning $\texttt{nF}_{\texttt{TB}}$, so that for each size, we choose the best performing value. We observe significant performance gains for sizes up to 16. Afterwards, there is no gain in performance. An explanation to such results is that our kernel uses a full warp for sizes 17 to 32 in order to use the butterfly shuffle operation. This also explains the performance drop starting size 17, since we are using a full warp on a $17 \times 17$ matrix. On the other side, when the sizes are within the range 1 to 16, the thread configuration is less than a warp, and so there is room for more occupancy. The performance gain are about 2-3× against using a fixed $\texttt{nF}_{\texttt{TB}} = 1$.

**Eliminating Intermediate Row Interchanges:** The row swapping step in the LU factorization is one of the most expensive steps in Algorithm 1. The main reason is that it has zero arithmetic intensity, and is performed at every iteration of the algorithm (*greedy swap*). We take advantage of the fact that the matrix is entirely factorized by one TB that has access to all its rows. Instead of performing a swap at each iteration using shuffle operations, each thread keeps track whether its row has been pivoted, so that only threads with non-pivoted rows




(a) Single precision (b) Double precision

Figure 5: Performance gains obtained by tuning $\texttt{nF}_{\texttt{TB}}$. Results are for 1M matrices on a P100 GPU.

perform the trailing matrix updates and search for new pivots in the next factorization step. Figure 6 shows the difference between the update steps in both cases. When the factorization is done, each thread knows the final destination of its row and writes it directly into global memory (*lazy swap*). Figure 7 shows the performance gains obtained by using the lazy swap technique.

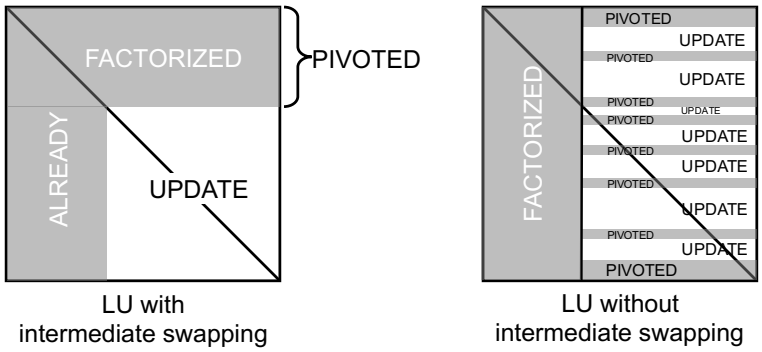
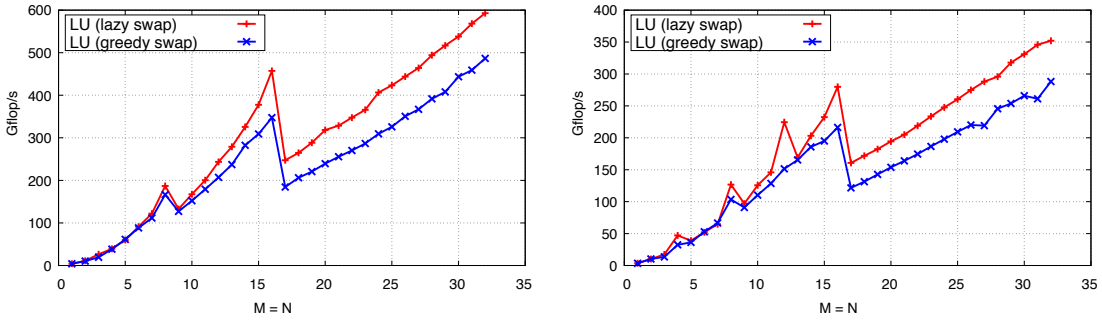



Figure 6: Trailing matrix updates in LU factorization with/without swapping.

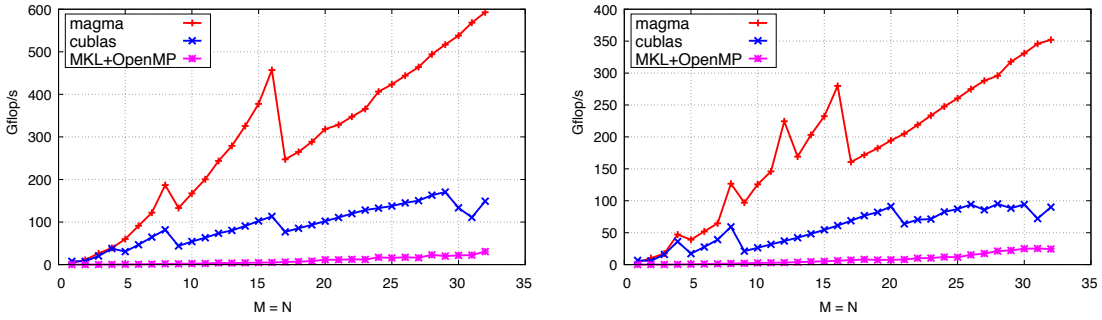



(a) Single precision (b) Double precision

Figure 7: Performance gains obtained by adopting a lazy swap technique. Results are for 1M matrices on a P100 GPU.

**Final Performance for LU factorization:** Figure 8 shows the final performance of the `LU` factorization against state-of-the-art vendor libraries. Results are shown on a Pascal

P100 GPU against the batched LU factorization from CUBLAS (CUDA 8.0), as well as against running the Intel MKL library on top of OpenMP on a 20-core Haswell machine (E5-2650 v3). In single precision, MAGMA is at least 19× faster than MKL+OpenMP, and achieves speedups between $1.4-5.1\times$ against CUBLAS. In double precision, MAGMA speedups against the CPU solution is asymptotically 14× and goes up to more than 40×. Against CUBLAS, MAGMA is faster by factors of $1.7-6\times$.




(a) Single precision (b) Double precision

Figure 8: Final performance of the LU factorization. Results are for 1M matrices on a P100 GPU.

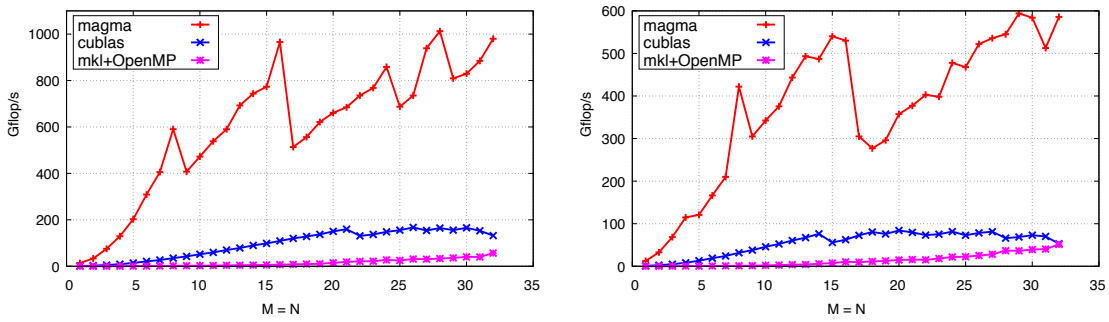
# 6 Computing the Matrix Inverse

As mentioned before, the matrix inversion is performed using LU factorization of the augmented matrix [A | I], and performing a triangular solve with respect to the upper triangular factor U. We leverage everything we learned from the LU factorization kernel and use it in the matrix inversion kernel. One property that affects the inversion is the absence of intermediate row interchanges. Since we are using a lazy swapping technique, the LU factorization overwrites the identity matrix part with a unit lower triangular matrix ($A_{iL}$). The inversion kernel takes advantage of such property while performing the trailing matrix updates. The lower triangular part of the correct inverse ($B_L$) can be obtained be performing column interchanges on $A_{iL}$ that reflects the pivot selections of the factorization.

We found out that using an N×2N register file is too much per TB and results in low occupancy for larger sizes. Instead, we use an N×N register file to perform the factorization, and overwrite the lower triangular factor L with $A_{iL}$. We write the matrix [$A_{iL}$\U] into shared memory after recovering all the required row interchanges. We then extract $A_{iL}$ back intro registers so that each thread has one column. This enables all thread to work independently on the second TRSM. Finally, the result of the second solve goes through some column interchanges (using shared memory) to recover the final correct inverse.

According to the LAPACK working notes [2], the LU factorization (GETRF) costs $(\frac{2N^3}{3}-\frac{N^2}{2}+\frac{5N}{6})$ FLOPS. While performing two triangular solves following an LU factorization (GETRS) would cost $(2N^3-N^2)$ FLOPs, performing the matrix inversion (GETRI), which takes into account the shape of the first solve, costs less FLOPs $(\frac{4N^3}{3}-N^2+\frac{5N}{3})$. We report our performance results in this section based on the more accurate formula, which sums the FLOPs of GETRF and GETRI, resulting in a total of $(2N^3-\frac{3N^2}{2}+\frac{15N}{6})$ FLOPs.

Figure 9 shows the performance of the batched matrix inversion kernel. We compare the performance of the proposed kernel against the batched inversion kernel provided by CUBLAS, as




(a) Single precision (b) Double precision

Figure 9: Final performance of the batched matrix inversion. Results are for 1M matrices on a P100 GPU.

well as against MKL running on top of OpenMP. The figure shows that MAGMA is significantly faster than CUBLAS, scoring speedups between $4.3 - 16.8\times$ in single precision, and between $3.4 - 14.3\times$ in double precision. If we compare the performance against MKL+OpenMP, we find out that MAGMA scores minimum speedups of $17.3\times$ and $11.4\times$ in single and double precisions, respectively. The speedups exceed $100\times$ for sizes less than 14.

# 7 Conclusion and Future Work

This paper presented optimized GPU kernels for batched `LU` factorization and inversion. The kernels are designed specifically for very small sizes. High performance is achieved using optimization techniques different from those used for large matrices. The proposed work is of great importance for scientific applications, including astronomy, sparse multifrontal solvers, and preconditioners. Future directions include studying other algorithms of interest to the scientific community, and designing an autotuning framework for such kernels.

# Acknowledgement

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

# References

[1] Suitesparse : A suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[2] LAPACK Working Note 41: Installation Guide for LAPACK, 1999. `http://www.netlib.org/lapack/lawnspdf/lawn41.pdf`.

[3] A. Abdelfattah, A. Haidar, S. Tomov, and 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*, pages 1249–1258, 2016.

[4] A. Abdelfattah, A. Haidar, S. Tomov, and 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*, pages 21–38, 2016.

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

[6] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. Batched gauss-jordan elimination for block-jacobi preconditioner generation on gpus. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, pages 1–10, New York, NY, USA, 2017. ACM.

[7] H. Anzt, B. Haugen, J. Kurzak, P. Luszczek, and J. Dongarra. Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience*, 27(17):5096–5113, 2015.

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

[9] 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.

[10] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, 29(2):193–208, 2015.

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

[12] 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.

[13] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, November 2012.

[14] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and 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*, pages 659–671, 2016.

[15] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[16] 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.

[17] 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.

[18] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.

[19] S. Tomov and J. Dongarra. Dense Linear Algebra for Hybrid GPU-based Systems. In J. Kurzak, D. A. Bader, and J. Dongarra, editors, *Scientific Computing with Multicore and Accelerators*. Chapman and Hall/CRC, 2010.

[20] X. Wang and S. G. Ziavras. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines. *Concurrency and Computation: Practice and Experience*, 16(4):319–343, 2004.

[21] S. N. YERALAN, T. A. DAVIS, S.-L. W. M, and S. RANKA. Algorithm 9xx: Sparse QR Factorization on the GPU. *ACM Transactions on Mathematical Software*, 2015.