Journal of Computational Science 26 (2018) 226–236

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures☆

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

ARTICLE INFO

Article history:
Received 14 October 2017
Received in revised form
30 November 2017
Accepted 28 January 2018
Available online 13 February 2018

Keywords:
GPU computing
Matrix factorization
Batch computation

ABSTRACT

The use of batched matrix computations recently gained a lot of interest for applications, where the same operation is applied to many small independent matrices. The batched computational pattern is frequently encountered in applications of data analytics, direct/iterative solvers and preconditioners, computer vision, astrophysics, and more, and often requires specific designs for vectorization and extreme parallelism to map well on today's high-end many-core architectures. This has led to the development of optimized software for batch computations, and to an ongoing community effort to develop standard interfaces for batched linear algebra software. Furthering these developments, we present GPU design and optimization techniques for high-performance batched one-sided factorizations of millions of *tiny* matrices (of size 32 and less). We quantify the effects and relevance of different techniques in order to select the best-performing LU, QR, and Cholesky factorization designs. While we adapt common optimization techniques, such as optimal memory traffic, register blocking, and concurrency control, we also show that a different mindset and techniques are needed when matrices are tiny, and in particular, sub-vector/warp in size. The proposed routines are part of the MAGMA library and deliver significant speedups compared to their counterparts in currently available vendor-optimized libraries. Notably, we tune the developments for the newest V100 GPU from NVIDIA to show speedups of up to 11.8×.

1. Introduction

Batch computations apply the same numerical algorithm to a fairly large number of relatively small problems. The batched computing workload is quite different than the common workload for just one, typically large, matrix. The latter is served well by many software packages, including LAPACK [2], ScaLAPACK [3], PLASMA [4], and MAGMA [5]. The former, however, is relatively recent, and gained a lot of attention in many scientific communities, e.g., quantum chemistry [6], sparse direct solvers [7], astrophysics [8], and signal processing [9]. Software libraries such as Intel's MKL [10], NVIDIA's cuBLAS [11], and MAGMA recently started to provide highly optimized batched routines for many of the BLAS and LAPACK operations.

Existing numerical linear algebra software packages rarely achieve good performance on matrices of small sizes, because most of the optimization techniques that they carry out pay off only on large matrices. For example, the hybrid *lookahead* technique in MAGMA [12] is used to overlap the panel factorization (on the CPU) with the trailing matrix update (on the GPU). This design strategy is not efficient for small sizes, since the updates are no longer compute intensive, and therefore fail to overlap the panel factorization and the CPU-GPU communication. This is why new developments with different design strategies are needed.

While there have been new developments for GPU accelerated batched computations, for example the work done by Haidar et al. [13] and Abdelfattah et al. [14], there is still room for significant improvements when the matrix sizes are *tiny*. For these extremely small problems, the LAPACK-style blocking cannot achieve high performance, even if it is carried out on the GPU solely. Since the computation becomes memory bound on such small sizes, the cost of writing the factorized panel and then reading it back to perform the update becomes significant. Furthermore, the parallelization (to achieve sufficiently high occupancy) and the vectorization (for efficient warp use) become more challenging when sizes are less

☆ This is an extended version of our conference paper [1] that was invited to the JoCS special issue (https://doi.org/10.1016/j.procs.2017.05.250).
∗ Corresponding author.
E-mail addresses: ahmad@icl.utk.edu (A. Abdelfattah), haidar@icl.utk.edu (A. Haidar), tomov@icl.utk.edu (S. Tomov), dongarra@icl.utk.edu (J. Dongarra).

https://doi.org/10.1016/j.jocs.2018.01.005
1877-7503/© 2018 Elsevier B.V. All rights reserved.

than 32, and must, for example, be done across matrices. Therefore, other design, parallelization, and vectorization strategies must be discovered in order to resolve the aforementioned issues regarding memory traffic and parallelization.

This paper presents highly optimized GPU kernels for batched one-sided factorizations. The paper extends the previous work by the authors for batched LU factorization and matrix inversion [1], and applies the same design principles to the QR, and Cholesky factorizations. In terms of the workload size, we consider one million matrices of tiny sizes, typically from 1 up to 32. In addition to the applications already mentioned, these factorizations are of particular importance to sparse direct solvers, such as the multifrontal solvers that can be found in SuiteSparse [15]. We adopt a step-by-step methodology, where incremental improvements in the kernel design lead to incremental performance gains. Such a methodology automatically justifies all of our design choices. While all the kernels share the same optimization techniques, our design for the LU factorization adopts a unique *lazy swap* strategy, which eliminates the expensive intermediate row interchanges, thus leading to a much faster kernel, but that is still numerically equivalent to an LAPACK-style LU-factorization. The performance results show significant speedups against the vendor-supplied cuBLAS kernels on a Pascal P100 GPU, as well as on the new Volta V100 GPU.

2. Related Work

The design of high performance dense linear algebra (DLA) software for GPUs was originally motivated by the high performance GPUs can achieve in embarrassingly parallel, compute intensive tasks, most notably on the matrix-matrix multiplication (GEMM) [16–18]. The high performance GEMM enabled the development of high performance DLA in libraries like MAGMA [5], where many of the LAPACK numerical algorithms are designed in a hybrid style to take advantage of both CPUs and GPUs [12].

The growing demand for high performance dense linear algebra on large batches of small matrices has led to early developments for batch matrix multiplication [19,20], which were then followed by more optimized kernels being available in cuBLAS, starting with version 8.0. The development of the batched GEMM in MAGMA enabled the development of batched one-sided factorizations routines based on the LAPACK-style blocking [13], but on a smaller scale. For example, while non-batched routines use a large blocking size (e.g., 512 to 1024) to get asymptotically optimal performance, batched routines block by much smaller sizes (e.g., 8 to 32), and rely on batched GEMM that is specifically tuned for small sizes in order to extract performance [21][22]. The developments for extremely small matrices, however, are more challenging. An approach that relies on separate panel/update stages [13] leads to redundant memory traffic. This cost can be affordable for medium sizes (e.g., 64 up to 256), but becomes significant for smaller sizes.

This is why other research efforts followed a *one-kernel approach*, where all computations are fused into a single GPU kernel. For example, Wang et al. [23] introduced FPGA-based parallel LU factorization of large sparse matrices, where the algorithm is reduced to factorizing many small matrices concurrently. Villa et al. [24] 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 [25]. Kurzak et al. [26] developed batched Cholesky factorization in single precision for sizes up to 100×100, which was used in an Alternating Least Squares (ALS) solver. Masliah et al. developed batched GEMM for very small sizes for both CPUs and GPUs [27]. Batched matrix inversion has been also introduced in the context of generating block-Jacobi preconditioners [28]. Batched QR factorization is of particular importance to H-matrix computation, as highlighted by Akbudak et al. [29], and by Boukaram et al. [30]. Kim et al. also introduced batched GEMM, triangular solve, and LU (no pivoting) for CPUs and Intel's Xeon Phi architectures based on a compact interleaved data layout [31].

This paper follows the same one-kernel approach to improve the MAGMA performance on very small sizes. It complements the work by Haidar et al. [13], which outperforms cuBLAS for medium and large sizes, but trails it for the sizes we focus on (up to 32).

3. Contributions

Below is a list of contributions for this paper.

1. Highly optimized GPU kernels for one-sided factorization on batch workloads. The developed kernels significantly outperform the state of the art designs from the vendor provided software. We typically consider single-node workloads that involve millions of extremely small matrices.
2. A set of unified design techniques that are oblivious to the three algorithms considered (LU, QR, and Cholesky factorizations). The paper manages to find a common ground among the three algorithms to achieve high performance.
3. The paper presents a detailed study of the different choices for every aspect of the kernel design, including thread configuration, matrix storage, occupancy, and others. The paper justifies the final design choice by showing intermediate performance results for different choices. Such a detailed study can be considered as a guide for designing other algorithms on similar workloads.

4. Background

This section introduces the computational steps for the LU, QR, and Cholesky factorizations on square matrices of size N×N. The description follows the LAPACK notations in double precision arithmetic.

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

There are four main steps in performing the unblocked LU factorization. Namely, these are: (1) locate the maximum absolute value in the current column (IDAMAX); (2) swap current row with the row with maximum absolute value in the current column (DLASWP); (3) scale the current column (DSCAL); and (4) rank – 1 update (DGER). Algorithm 1 shows the factorization using the four steps. According to LAPACK working notes [32], the LU factorization of a square matrix performs $(\frac{2N^3}{3} - \frac{N^2}{2} + \frac{5N}{6})$ floating point operations (FLOPs).

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] *= (1 / A[i,i])
    DGER: A[i+1:N,i+1:N] -= A[i+1:N,i]×A[i,i+1:N]
end
```

While the LU factorization is able to factorize symmetric positive definite (SPD) matrices, the Cholesky factorization, shown in Algorithm 2, introduces a much faster algorithm for such matrices. The algorithm factorizes an SPD matrix A = LL^T, where L is a lower triangular matrix. In each iteration, the algorithm (1) computes the square root of the current diagonal element (`A[i,i]`), (2) scales all the elements below that element (`A[i+1:N,i]`), and (3) applies a symmetric rank – 1 update to the lower triangular part of the trailing matrix (`A[i+1:N,i+1:N]`). The Cholesky factorization algorithm performs $(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6})$ FLOPs.

Algorithm 2. Unblocked Cholesky factorization.

```
for $i$=1 to N do
    if A[i, i] $\leq$ 0 then
        // report error for non-positive definiteness.
    end
    A[i, i] = sqrt(A[i, i])
    DSCAL: A[i+1:N,i] *= (1 / A[i,i])
    // symmetric rank-1 update (DSYR)
    A[i+1:N,i+1:N] -= A[i+1:N,i]$\times$A[i+1:N, i]$^T$
end
```

The third algorithm that we consider is the QR factorization, where a matrix A is represented as the product Q×R, where Q is an orthogonal matrix, and R is an upper triangular matrix. Following the LAPACK implementation of the algorithm, the Q matrix is not computed explicitly. It is rather represented as a product of elementary Householder reflectors, namely Q = H(1) H(2) ··· H(N), where $H(i) = I - \tau v v^T$. Considering the reflector H(i), the corresponding vector v has $v[1:i-1]=0$, $v[i]=1$, which is not explicitly stored, and $v[i+1:N]$, which is stored in $A[i+1:N, i]$. Algorithm 3 shows the basic steps of the QR factorization: (1) generate an elementary reflector ($H(i)$) by computing v and τ (the DLARFG routine); and (2) apply $H(i)$ to the trailing submatrix. The QR factorization requires $(\frac{4N^3}{3} + 2N^2 + \frac{14N}{3})$ FLOPs.

Algorithm 3. Unblocked QR factorization

```
for $i$=1 to N do
    // Generate elementary reflector H(i)
    // to annihilate A[i+1:N,i]
    (v, $\tau$) $\leftarrow$ DLARFG(A[i:N,i])
    // DLARF: Apply H(i) to A[i:N,i+1:N]
    A[i:N,i+1:N] -= $\tau$ v v$^T$A[i:N,i+1:N]
end
```

As pointed out before, there is no need to use LAPACK-style blocking techniques, which is the strategy adopted for LU, QR, and Cholesky factorizations in the DGETRF, DGEQRF, and DPOTRF routines, respectively. The reason is that a very small matrix can be kept in registers or shared memory during the entire factorization. This means that it makes no difference if (additional) blocking is used or not, since all the data is blocked and accessed from fast memory until the factorization is complete.

5. System setup

We illustrate our findings on two systems accelerated with high-end NVIDIA GPUs. The first system is equipped with two 10-core Intel Haswell processors (E5-2650 v3, running at 2.3 GHz) and a Pascal GPU (Tesla P100). The GPU has 56 streaming multiprocessors, with a base clock of 1.189 GHz. The second system is equipped with an identical CPU, but the GPU is a Volta V100, which features 80 multiprocessors, running at 1.38 GHz. Both GPUs have 16GB of CoWoS Stacked HBM2 memory, and are attached to the host CPU through a PCIE interconnect. All results are obtained using the CUDA 9.0RC toolkit. We point out that to quantify the effect of our techniques, and to justify design choices, we show incremental results on the P100 GPU only, while the final performance results are shown on both GPUs.

6. General design criteria

All the kernels discussed in this paper have some common design aspects.

First, each matrix is factorized using exactly one CUDA thread block (TB). Upon launch, the kernel is configured with as many TBs as the number of matrices, i.e., the batch size (`batchCount`). The kernel grid in CUDA is, in general, a three dimensional array (`gx`, `gy`, `gz`). We use the `gx` dimension only to launch a 1D array of TBs. The maximum value for `gx` is $2^{31} - 1$, which means that our kernels can solve billions of matrices in a single kernel launch.

Second, we adopt an optimal memory traffic strategy. Each matrix will be read and written exactly once. Since the computation is memory bound on such small sizes, it is an important design choice to keep the entire matrix cached in a fast memory level in order to avoid any redundant memory traffic.

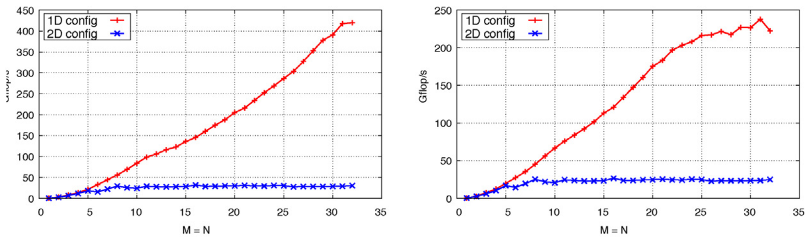
Third, we use C++ templates to generate fully unrolled codes. Since the sizes of interest are finite, the size of the input matrix is passed as a template parameter. For such a range of very small sizes, this is a crucial decision. Fully unrolled loops get rid of integer and branch instructions, which can be quite an overhead [27].

Finally, all kernels use unblocked computations. There is no need to factorize a panel of width $nb > 1$ so that the trailing updates use L3 BLAS operations that are rich in data reuse. As pointed out earlier, each matrix is entirely kept in registers or shared memory, which means that data reuse is preserved anyway. Every kernel factorizes one column at a time and carries out the required transformation to the trailing matrix.

7. Design choice 1: thread configuration

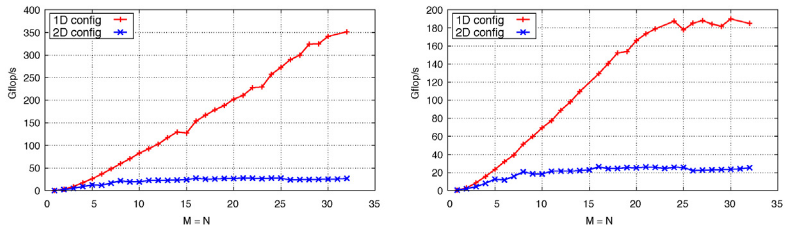
For most linear algebra kernels, the configuration of TBs can be 1D or 2D, which is one of our design parameters. A 2D configuration simplifies the coding effort. To use an N × N thread configuration to factorize an N × N matrix, each thread is responsible for one element of the matrix. Thus, reading and writing the entire matrix can be done through one line of code each. In addition, the trailing matrix update is fully parallelized among threads. On the other hand, reading, writing, and updating the matrix will be written in loops if a 1D configuration of N threads is used. Our analysis shows that there are multiple reasons to reject the 2D configuration in favor of the 1D alternative. In batch workloads, it is important to optimize the throughput of the processed matrices, which is achieved by maximizing the number of resident TBs per multiprocessor. Using a 2D configuration, the occupancy level of each multiprocessor are far from optimal, and thus resulting in more limited parallelism, which is likely to produce a very poor performance. For example, the 2D configuration requires 256 threads for a 16 × 16 matrix. This limits the number of resident TBs per multiprocessor to 8, which can host a maximum of 2048 threads. A modern Kepler GPU can have up to 16 resident TBs per multiprocessor, while later generations can host up to 32. This means that the occupancy at size 16 is brought down by a factor of 2 on a Kepler GPU, and by a factor of 4 on later GPUs. Using a 1D configuration, the occupancy is no longer limited by the number of threads, but rather by the amount of memory resources required by each TB.

Another motivation to abandon the 2D configuration is synchronization. The use of N^2 threads is likely to produce too many barriers in the kernel. At each iteration, for example, threads possessing the trailing matrix always wait for the threads performing the factorization of the current column, and threads performing the factorization in the next iteration have to wait for the update to finish. On the contrary, a 1D design can be free of synchronization points. Even with the use of the new `__syncwarp()` function introduced in CUDA 9.0, we observe that this function is obviously



(a) Single precision (b) Double precision

Fig. 1. Performance comparison between the 1D configuration and the 2D configuration of the Cholesky factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.



(a) Single precision (b) Double precision

Fig. 2. Performance comparison between the 1D configuration and the 2D configuration of the LU factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.

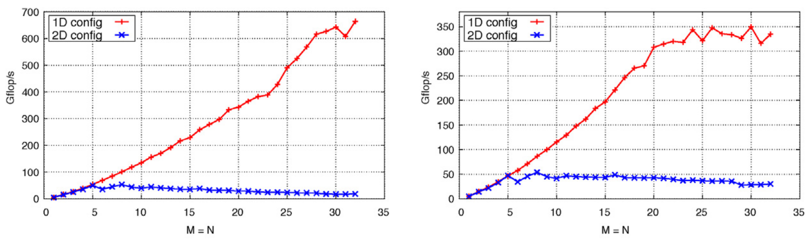
more lightweight than the legacy `__syncthreads()` function. In addition, a 1D design assigns more work per thread, which results in a better instruction-level parallelism (ILP).

We conducted a performance test for both design choices to quantify the differences in the two techniques and to verify our analysis. For every algorithm, we designed two kernels based on both configurations. The matrix is stored in shared memory in both situations. Figs. 1–3 show the performance for the three factorizations using both single and double precision arithmetic. All performance graphs show a clear advantage for the 1D configuration over the 2D configuration. Fig. 1 shows speedups of 10.5×/9.6× in single/double precision against the 2D configuration for the Cholesky factorization. A similar 9× (or more) speedup is obtained for the LU factorization in both precisions (Fig. 2). Fig. 3 also shows similar significant speedups for the QR factorization. It is clear that, regardless of the numerical algorithm, the 1D configuration is the winning strategy for batch workloads of small sizes. From now on, we continue to carry out further enhancements on the 1D configuration.

8. Design choice 2: matrix storage

The second design parameter is the matrix storage in fast memory, which can be either the register file, or the shared memory. In general, the access time of registers is faster than shared memory, but the latter provides more flexible access patterns. In either case, the 1D array of threads reads the matrix column by column from the global memory to preserve a coalesced memory access. If threads read into registers, each thread has direct access to an entire row of the matrix. In order to access other elements, communication among threads is required, either by using shuffle operations, or through shared memory. All three factorization algorithms require accessing data from other threads, e.g., as in the row interchanges in the LU factorization, and the rank – 1 updates in all algorithms. On the other hand, if the threads read the matrix into shared memory, any thread has access to the entire matrix. This comes at the cost of synchronization points (recall that CUDA now deprecates implicit warp synchronous codes, especially on the Volta GPUs). In addition, shared memory is slower than registers, and its capacity is smaller than the register file (e.g., 64KB on the P100 GPU *vs.* a 256KB register file). The capacity of the chosen storage is important, since it directly impacts the occupancy on the multiprocessor. Since there is a tradeoff between the two design choices, we developed two versions for each factorization algorithm to better understand the consequences of each choice per algorithm and matrix size. The first version uses shared memory for storage and communication among threads. The second uses registers for storage and warp shuffle operations for communication.

First, we consider the Cholesky factorization algorithm. The shared memory version loads the matrix into shared memory column-by-column to respect coalesced global memory access. At each iteration `i`, all threads compute the square root of `A[i,i]`, scale the column `A[i+1:N,i]` accordingly in shared memory, and then use the result to update the lower triangular part of the trailing



(a) Single precision (b) Double precision

Fig. 3. Performance comparison between the 1D configuration and the 2D configuration of the QR factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.

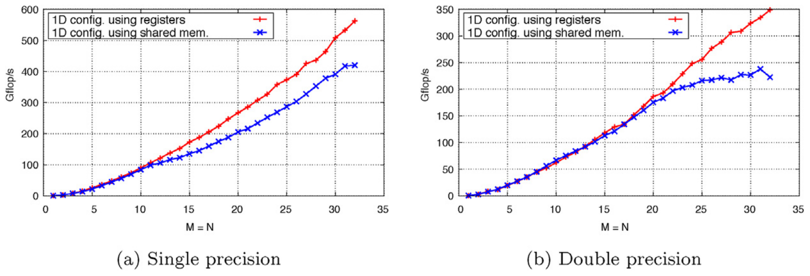


Fig. 4. Performance comparison between performing the Cholesky factorization in registers and in shared memory. Results are for 1M matrices on a Pascal P100 GPU.

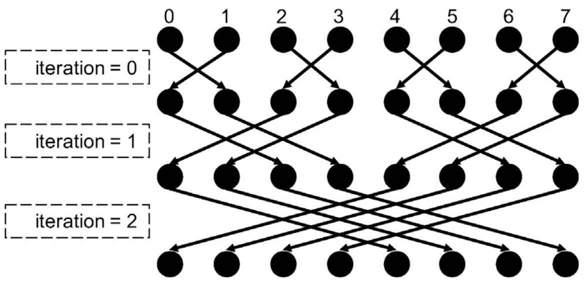


Fig. 5. Butterfly shuffle operation in CUDA among eight threads.

matrix `A[i+1:N, i+1:N]` in shared memory, column by column. Both the scale and the update phases are fenced by synchronization points (i.e. `__syncwarp()`). On the other hand, the register version must start by a shuffle operation to broadcast `A[i,i]` to all threads. After computing the square root, each thread scales its respective element in the current column. Before the update takes place, threads must perform shuffle operations to share the elements `A[i+1:N,i]` among all threads. Afterwards, each thread runs independently to update its respective row. It is clear that synchronization points in the shared memory version are replaced by shuffle operation in the register version. Fig. 4 shows the results for the two versions on the Cholesky factorization algorithm. For both precisions, the shared memory version fails to keep up the same performance with the register version. As the sizes become larger, the shared memory capacity and bandwidth become a bottleneck, which gives the advantage to the register version.

The `LU` factorization requires a pivot search at each iteration `i`. On shared memory, this can be done by a linear search on the elements of the column `A[i:N,i]`. In the register version, this step is done through a binary search, which can be implemented using the butterfly warp shuffle operations shown in Fig. 5. While this operation requires just $\mathtt{log_2(N)}$ steps, it must be carried out by a number of threads that is equal to power of 2. Another limitation is that shuffle operations support only 32 bit exchanges, which means that shuffles on double precision is twice the cost of single precision. Once the pivot is found, the swap operation can be done using shared memory or using shuffle operations. The scale and update operations of the `LU` algorithm are similar to that of the Cholesky algorithm except that the update requires communication of the pivot row `A[i,i+1:N]`, and that all the trailing matrix is updated. Fig. 6 shows the performance of the two `LU` versions. We observe a similar to the Cholesky factorization case advantage for the register version over the shared memory version.

Finally, for the `QR` factorization, we begin by discussing the results first, which are shown in Fig. 7. Unlike the `Cholesky` and `LU` factorizations, we observe that the shared memory version is faster than the register version, which is the opposite of our initial expectations. The explanation of this finding requires a deeper analysis for the `QR` factorization. As explained in Algorithm 3, the `DLARFG` routine in the `i`th iteration begins with a reduction operation to compute the norm of the current column `A[i:N,i]`. The reduction step can be done in exactly the same two ways as the pivot search in the `LU` algorithm. The main difference, however, for the `QR` factorization comes in the update phase, where an extra matrix-vector multiplication is required before the rank-1 update. The multiplication computes $\mathtt{y = v^T C}$, where `C = A[i:N, i+1:N]`. Assuming that `C` has a size of `P`×`Q`, the multiplication costs 2`PQ` FLOPs. However, there becomes a difference in the number of reduction steps required compute `y` depending on the storage type. If registers and butterfly shuffle operations are used, then all threads collaborate to compute one element of `y` at a time. Therefore, assuming that $\mathtt{N_{p2}}$ is the nearest power of 2 larger than or equal to `N`, the total number of reduction steps is $\mathtt{Q\times log_2(N_{p2})}$, and not $\mathtt{Q\times log_2(P)}$. Recall that shuffle operations require that the number of matrices is always equal to a power of 2. On the other hand, shared memory seems to provide a better alternative. If both `v` and `C` are in shared memory, we can assign each element of `y` to a single thread. Since threads are totally parallel, and using `Q` threads, it would take only `P` steps to find the entire vector `y`. This means that it requires much fewer steps to compute `y` in shared memory. Since the computation of `y` is at the innermost loop of the kernel, the shared memory version can be faster than the register version.

The results of the `QR` factorization in Fig. 7 motivated us to try a *hybrid storage design*, which means to use registers as the main storage for the matrix, but use shared memory (instead of shuffle operations) for communication among threads. Another motivation for this strategy is that CUDA 9.0 deprecates all the previous

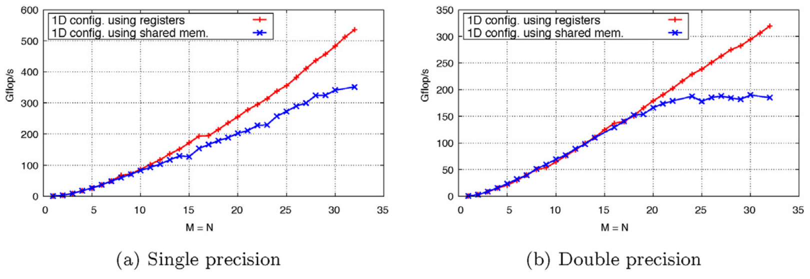


Fig. 6. Performance comparison between performing the `LU` factorization in registers and in shared memory. Results are for 1M matrices on a Pascal P100 GPU.

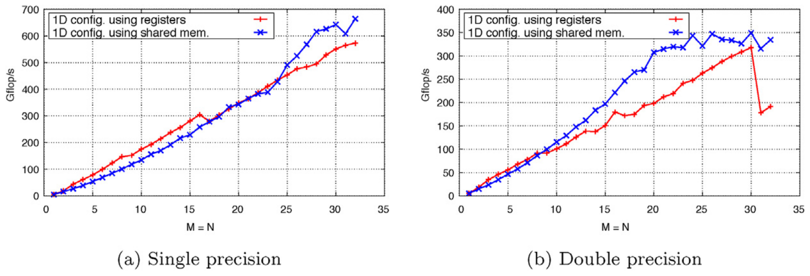


Fig. 7. Performance comparison between performing the QR factorization in registers and in shared memory. Results are for 1M matrices on a Pascal P100 GPU.

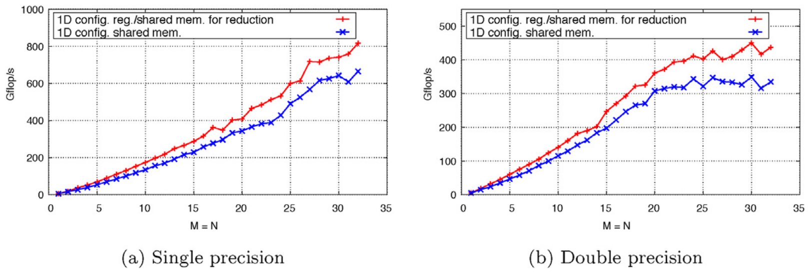


Fig. 8. Performance comparison between performing the QR factorization using hybrid storage and using shared memory only. Results are for 1M matrices on a Pascal P100 GPU.

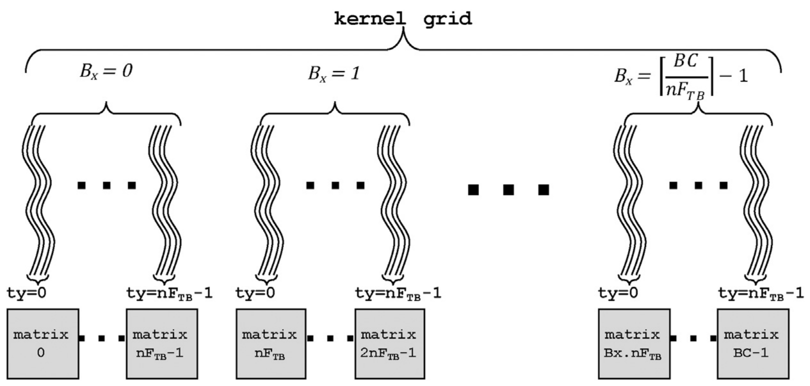


Fig. 9. Concurrency control in the kernel grid configuration.

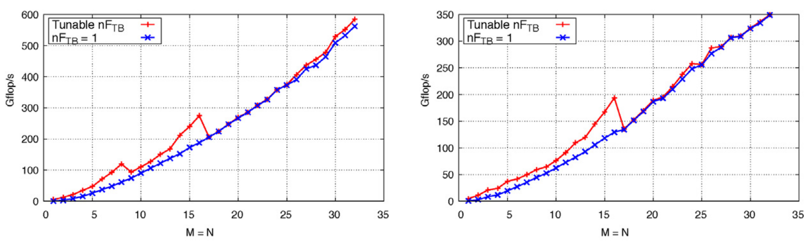
shuffle operations, and replaces them by new ones that are relatively slower (significantly slower on the Volta GPU). Fig. 8 shows that the register version with shared memory communication beats the pure shared memory implementation. We tried the same technique with Cholesky and LU factorization, and barely found any impact on performance.

9. Design choice 3: concurrency control

The general design strategy in Section 6 states that a matrix is processed using exactly one TB. However, it does not restrict the number of matrices assigned to the same TB. In fact, we can assign multiple matrices to the same TB to be factorized concurrently. The motivation behind this strategy is the 1D configuration of TBs, which brings very low occupancy if the matrix size is small, and in particular, if smaller than the warp size, which is the case at hand. For example, a batch of 8×8 matrices requires a TB configuration of $(8, 1)$ threads. Such configuration does not use a full warp, which obviously wastes resources. The second issue is that such a configuration makes the CUDA runtime in full control of the number of concurrent TBs per multiprocessor. Even if the runtime does the optimal decision, it will not be able to schedule more than 32 TBs per multiprocessor, which is the hardware limit on modern GPUs. Instead, we adopt a different configuration that aggregates multiple TBs into one, thus using an $(8, \mathtt{nF}_{\mathtt{TB}})$ TB configuration, where $\mathtt{nF}_{\mathtt{TB}}$ controls the number of concurrent factorizations per TB. Assuming an optimal runtime behavior with $\mathtt{nF}_{\mathtt{TB}} = 4$, the 32 TBs will be able to factorize 128 matrices per multiprocessor instead of 32, thus achieving a 4× speedup. This example assumes that there is enough memory resources to host 128 matrices of size 8×8, which is a valid assumption for double precision if the register file is used for storage. The $\mathtt{nF}_{\mathtt{TB}}$ value is a tuning parameter, with an optimal value that depends on many factors, including the matrix size and the storage type.

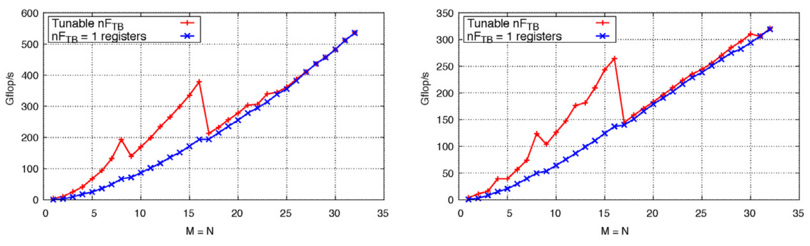
Fig. 9 shows the general idea of concurrency control. In general, a batch of size BC can be factorized using $\left\lceil \frac{\mathrm{BC}}{nF<ce:inf>TB</ce:inf>} \right\rceil$, so that each TB handles $\mathtt{nF}_{\mathtt{TB}}$ matrices. The value of the tuning parameter $\mathtt{nF}_{\mathtt{TB}}$ does not need to be known at compile time. We conducted an autotuning experiment that performs a full sweep over values of $\mathtt{nF}_{\mathtt{TB}}$ from 1 to 16. We observe that after this range, there is no gain in performance. Figs. 10–12 show the impact of concurrency control on the three factorization algorithms in single and double precision.

We observe that a tunable $\mathtt{nF}_{\mathtt{TB}}$ does not affect the performance for sizes larger than 16. For such range, the hardware rounds up



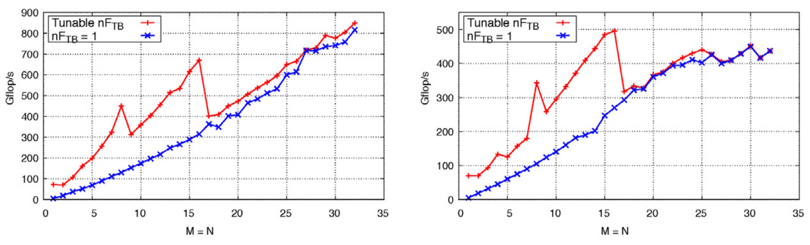
(a) Single precision (b) Double precision

Fig. 10. The impact of concurrency control on the Cholesky factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.



(a) Single precision (b) Double precision

Fig. 11. The impact of concurrency control on the LU factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.



(a) Single precision (b) Double precision

Fig. 12. The impact of concurrency control on the QR factorization kernel. Results are for 1M matrices on a Pascal P100 GPU.

the number of threads to use a full warp anyway, which means that there are no wasted resources in the TB configuration. It also means that, as long as full warps are used, the runtime is able to schedule TBs efficiently among the multiprocessors. For sizes 1 through 16, we observe performance gains in both single and double precision. The smaller the sizes, the more the speedups. For example, the Cholesky factorization has speedups that range from 1.2× to 4.3× in single precision and from 1.2× to 3.9× in double precision. The impact is more apparent in the LU and QR factorizations, with the LU factorization enjoying speedups that range from 2.0× to 3.5× in single precision, and from 1.8× to 3.7× in double precision. Similarly, the QR factorization performance is enhanced by factors that range from 2.0× to 3.8× in single precision, and from 1.8× to 3.8× in double precision.

10. Lazy swapping for LU Factorization

The LU factorization has a unique step that is necessary for its numerical stability. At each iteration, two rows potentially exchange their positions in the matrix. This is an expensive step with pure data movement, and zero arithmetic intensity. Originally, the exchange occurs at each iteration in a *greedy* style. While the *greedy swap* is necessary for general sizes, we can take advantage of the small sizes of interest and introduce a *lazy swap* technique, where all the interchanges are carried once just before writing the matrix back into the global memory. Instead of explicitly changing positions in registers, each thread keeps track of the final position of its row. A row that has been pivoted in a previous iteration is marked with a flag so that it is excluded from the trailing matrix update, as shown in Fig. 13. Pivoted rows are also excluded from the pivot searches in the following factorization steps. Fig. 14 shows the performance gains of the lazy swap technique, that are up to 21% and 32% in single and double precision, respectively.

11. Final performance results

In this section, we show the final performance results of the three algorithms against the competitive routines from the vendor library (cuBLAS). All kernels are tuned and scheduled for the next release of the MAGMA library. The cuBLAS library provides batched routines for the LU and QR factorizations, but not for the batched Cholesky factorization. Therefore, the results for the Cholesky factorization use the cuBLAS batched LU routine instead, which is still a valid option (though suboptimal) to factorize SPD matrices. We also include a reference batched CPU implementation that we developed using sequential calls to the factorizations in the MKL library (version 11.3.0) from within an OpenMP `parallel for` loop. The number of OpenMP threads is set to 20, which is equivalent to the core count of the CPU.

A normal behavior that we observe in most performance graphs are the spikes at sizes that are power of 2, and the drops that come right after those sizes. This is a normal behavior in GPUs, which always execute threads in groups of 32 (warps). A size that is a power of 2 allows the hardware to use full warps (especially when

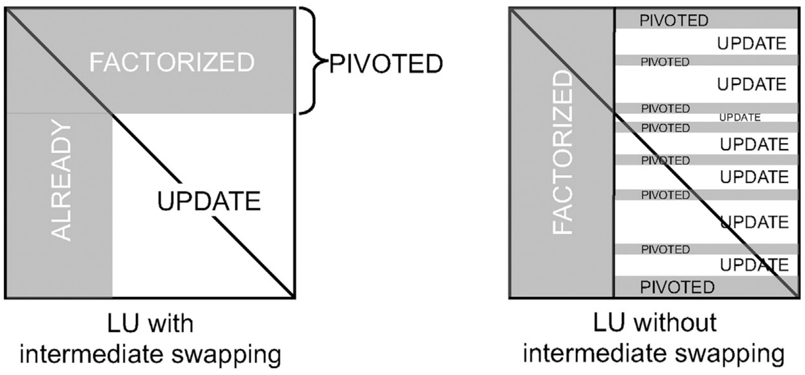

FACTORIZED
PIVOTED
ALREADY
UPDATE
LU with intermediate swapping
PIVOTED
UPDATE
FACTORIZED
LU without intermediate swapping

Fig. 13. Trailing matrix updates in LU factorization with/without swapping.

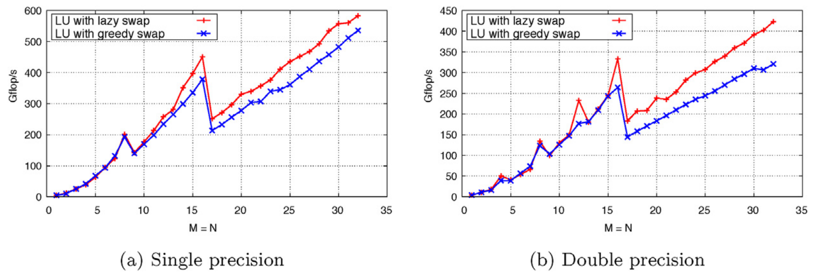


Fig. 14. The impact of the lazy swap on the LU factorization. Results are for 1M matrices on a Pascal P100 GPU.

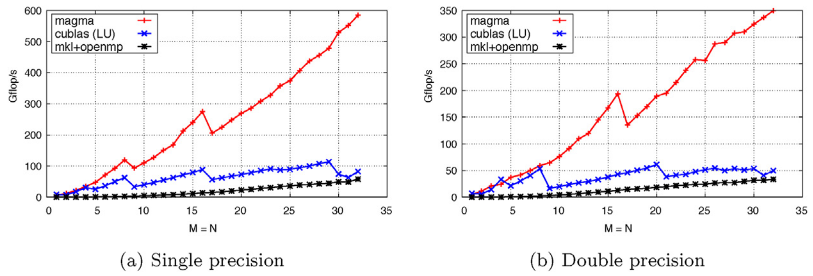


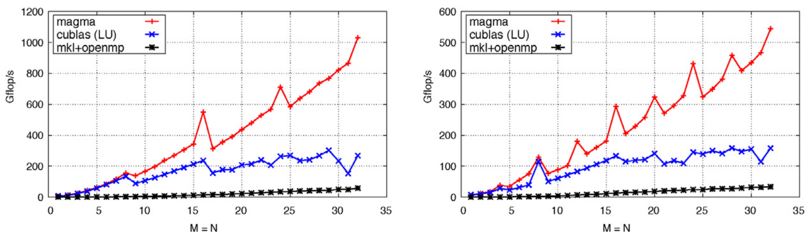
Fig. 15. Final performance of the Cholesky factorization on the P100 GPU (`batchCount` = 1M).

using the concurrency control technique discussed in Section 9). For other sizes, a round up to the nearest power of 2 is imposed either by the kernel configuration, or implicitly by the hardware. Even on the Volta V100 GPU, which has separate program counters and stacks for every thread, the same behavior is still observed.

Fig. 15 shows the performance of the Cholesky factorization on the P100 GPU. The MAGMA kernels are significantly faster than cuBLAS (which uses the LU algorithm), scoring speedups that are up to 8.8× in single precision, and 8.2× in double precision. Against the reference CPU implementation, MAGMA is at least 9.3× faster in both precisions. The performance of the Cholesky factorization on the V100 GPU is also summarized in Fig. 16, where we observe up to 5.7×/4.1× speedups against cuBLAS in single/double precision. The MAGMA performance numbers on the V100 GPU are at least 13.2× faster than the CPU performance.

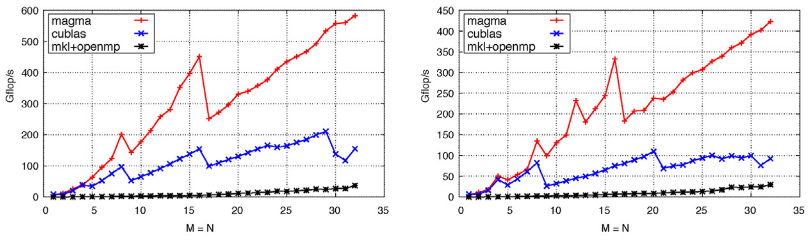
Considering the results for the LU factorization on the P100 GPU, which are summarized in Fig. 17, the MAGMA kernel is faster than cuBLAS by up to 4.8× in single precision, and 5.2× in double precision. It is at least 14.3× faster than the CPU reference implementation. Looking into the results on the V100 GPU, MAGMA is still faster than cuBLAS, scoring speedups that are up to 3.8× in single precision, and up to 2.9× in double precision. The MAGMA performance on the V100 GPU is at least 22× faster than the CPU performance.

The performance gains in the QR factorization results are also significant. Considering the P100 GPU (Fig. 19), MAGMA is 2.1×–11.04× faster than cuBLAS in single precision. It is also 2.9× to 9.4× faster in double precision. The speedup against the reference implementation is at least 16.1×. We observe a kind of stagnation in the double precision performance for MAGMA. Recall that this kernel uses both registers and shared memory to achieve the best performance. The shared memory requirements for this kernel (N×N) seem to limit the multiprocessor occupancy. The results in single precision support this explanation, since we observe no stagnation for the single precision results, as the shared memory requirements are almost half of what is needed in double precision. Considering the results on the V100 GPU (Fig. 20), MAGMA is up to 11.8× faster than cuBLAS in single precision. It is also up to 10.8× faster in double precision. The minimum speedup obtained against the CPU reference implementation is 32.95× across both precisions.



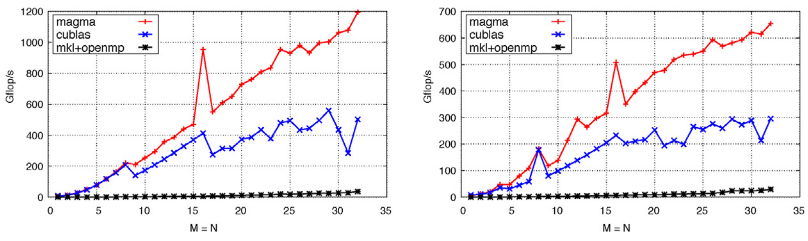
(a) Single precision (b) Double precision

Fig. 16. Final performance of the Cholesky factorization on the V100 GPU (`batchCount` = 1M).



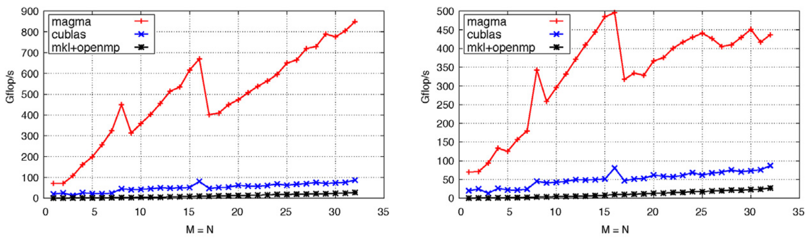
(a) Single precision (b) Double precision

Fig. 17. Final performance of the LU factorization on the P100 GPU (`batchCount` = 1M).



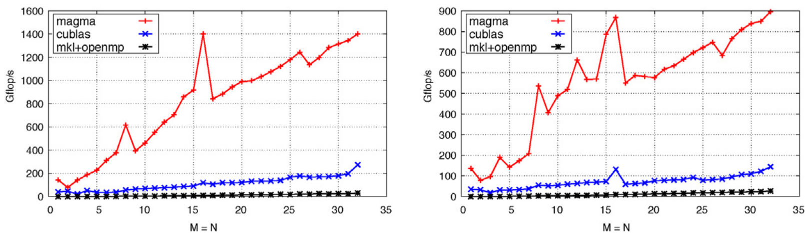
(a) Single precision (b) Double precision

Fig. 18. Final performance of the LU factorization on the V100 GPU (`batchCount` = 1M).



(a) Single precision (b) Double precision

Fig. 19. Final performance of the QR factorization on the P100 GPU (`batchCount` = 1M).



(a) Single precision (b) Double precision

Fig. 20. Final performance of the QR factorization on the V100 GPU (`batchCount` = 1M).

12. Conclusion and future work

This paper introduced a progressive design methodology for optimizing batched one-sided factorizations using GPUs. The paper mainly addresses extremely small matrix sizes, and introduces design techniques that are different from those used for larger matrices. Significant performance gains were achieved against the vendor library. The proposed work is of great importance for scientific applications, including astronomy, sparse multi-frontal solvers, and preconditioners. Future directions include variable size batched workloads, developing an autotuning framework for such kernels, and integration with scientific applications where high performance batched routines have a great impact on performance.

Acknowledgements

This work is partially supported by NSF Grants SI2:SSE 1740250 and CSR 1514286, NVIDIA, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

References

[1] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Factorization and inversion of a million matrices using GPUs: challenges and countermeasures, Procedia Comput. Sci. 108 (Supplement C) (2017) 606–615, http://dx.doi.org/10.1016/j.procs.2017.05.250, International Conference on Computational Science, ICCS 2017, 12–14 June 2017, Zurich, Switzerland. URL http://www.sciencedirect.com/science/article/pii/S1877050917308785.
[2] LAPACK – Linear Algebra PACKage, "http://www.netlib.org/lapack/".
[3] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
[4] PLASMA. Parallel Linear Algebra Software for Multicore Architectures, Available at: https://bitbucket.org/icl/plasma (October 2017).
[5] MAGMA. Matrix Algebra on GPU and Multicore Architectures, available at http://icl.cs.utk.edu/magma/ (October 2017).
[6] 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, Mol. Phys. 104 (2) (2006) 211–228.
[7] S.N. Yeralan, T.A. Davis, W.M. Sid-Lakhdar, S. Ranka, Algorithm 980: sparse QR factorization on the GPU, ACM Trans. Math. Softw. 44 (2) (2017), http://dx.doi.org/10.1145/3065870, 17:1–17:29. URL http://doi.acm.org/10.1145/3065870.
[8] O. Messer, J. Harris, S. Parete-Koon, M. Chertkow, Multicore and accelerator development for a leadership-class stellar astrophysics code, Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing" (2012).
[9] M. Anderson, D. Sheffield, K. Keutzer, A predictive model for solving small linear algebra problems in GPU registers, IEEE 26th International Parallel Distributed Processing Symposium (IPDPS) (2012).
[10] Intel Math Kernel Library, available at http://software.intel.com/intel-mkl/.
[11] NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS), available at https://developer.nvidia.com/cublas.
[12] S. Tomov, J. Dongarra, Dense linear algebra for hybrid GPU-based systems, in: J. Kurzak, D.A. Bader, J. Dongarra (Eds.), Scientific Computing with Multicore and Accelerators, Chapman and Hall/CRC, 2010.
[13] 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, http://dx.doi.org/10.1177/1094342014567546.
[14] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Fast Cholesky factorization on GPUs for batch and native modes in MAGMA, J. Comput. Sci. 20 (Supplement C) (2017) 85–93, http://dx.doi.org/10.1016/j.jocs.2016.12.009,. URL http://www.sciencedirect.com/science/article/pii/S1877750316305154.
[15] SuiteSparse. A Suite of Sparse Matrix Software, Available at http://faculty.cse.tamu.edu/davis/suitesparse.html.
[16] V. Volkov, J. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, in: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–11.
[17] R. Nath, S. Tomov, J. Dongarra, An improved MAGMA GEMM for Fermi graphics processing units, Int. J. High Perform. Comput. Appl. 24 (4) (2010) 511–515, http://dx.doi.org/10.1177/1094342010385729.
[18] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, 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, ACM, New York, NY, USA, 2011, http://dx.doi.org/10.1145/2063384.2063431, pp. 35:1–35:11.
[19] H. Anzt, B. Haugen, J. Kurzak, P. Luszczek, J. Dongarra, Experiences in autotuning matrix multiplication for energy minimization on GPUs, Concurr. Comput.: Practice Exp. 27 (17) (2015) 5096–5113, http://dx.doi.org/10.1002/cpe.3516.
[20] 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, kFrankfurt, Germany, June 19–23, 2016, 2016, pp. 21–38, http://dx.doi.org/10.1007/978-3-319-41321-1_2.
[21] 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, http://dx.doi.org/10.1109/IPDPSW.2016.190.
[22] T. Dong, A. Haidar, S. Tomov, J. Dongarra, A fast batched Cholesky factorization on a GPU, Proc. 2014 International Conference on Parallel Processing (ICPP-2014) (2014).
[23] X. Wang, S.G. Ziavras, Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines, Concurr. Comput.: Practice Exp. 16 (4) (2004) 319–343, http://dx.doi.org/10.1002/cpe.748.
[24] V. Oreste, M. Fatica, N.A. Gawande, A. Tumeo, Power/performance trade-offs 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.
[25] 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.
[26] J. Kurzak, H. Anzt, M. Gates, J. Dongarra, Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs, IEEE Transactions on Parallel and Distributed Systems, PP (99) (2015), http://dx.doi.org/10.1109/TPDS.2015.2481890, 1-1.
[27] 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, 2016, pp. 659–671, http://dx.doi.org/10.1007/978-3-319-43659-3_48.
[28] H. Anzt, J. Dongarra, G. Flegar, E.S. Quintana-Ortí, Batched Gauss–Jordan Elimination for Block–Jacobi Preconditioner Generation on GPUs, in: in: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'17, ACM, New York, NY, USA, 2017, pp. 1–10, http://dx.doi.org/10.1145/3026937.3026940.
[29] K. Akbudak, H. Ltaief, A. Mikhalev, D.E. Keyes, Tile low rank Cholesky factorization for climate/weather modeling applications on manycore architectures, High Performance Computing – 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017 (2017) 22–40, http://dx.doi.org/10.1007/978-3-319-58667-0_2.
[30] W.H. Boukaram, G. Turkiyyah, H. Ltaief, D.E. Keyes, Batched QR and SVD Algorithms on GPUs with Applications in Hierarchical Matrix Compression, Parallel Computing, https://doi.org/10.1016/j.parco.2017.09.001, http://www.sciencedirect.com/science/article/pii/S0167819117301461.
[31] K. Kim, T.B. Costa, M. Deveci, A.M. Bradley, S.D. Hammond, M.E. Guney, S. Knepper, S. Story, S. Rajamanickam, Designing vector-friendly compact BLAS and LAPACK kernels, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, 2017, http://dx.doi.org/10.1145/3126908.3126941, pp. 55:1–55:12.
[32] LAPACK Working Note 41. Installation Guide for LAPACK, http://www.netlib.org/lapack/lawnspdf/lawn41.pdf (1999).

Ahmad Abdelfattah received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). He is currently a research scientist in the Innovative Computing Laboratory at the University of Tennessee. He works on optimization techniques for different linear algebra workloads in the MAGMA library. Ahmad has B.Sc. and M.Sc. degrees in computer engineering from Ain Shams University, Egypt.

Azzam Haidar received a Ph.D. in 2008 from CERFACS, France. He is Research Scientist at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests focus on the development and implementation of parallel linear algebra routines for scalable distributed multi-core and GPU architectures, for large-scale dense and sparse problems, as well as new algorithms for singular value (SVD) and eigenvalue problems as well as approaches that combine direct and iterative algorithms to solve large linear systems.

Stanimire Tomov received a M.S. degree in Computer Science from Sofia University, Bulgaria, and Ph.D. in Mathematics from Texas A&M University. He is a Research Director in ICL and Adjunct Assistant Professor in the EECS at UTK. Tomov's research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software for emerging architectures for HPC.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a Senior Scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Department of Electrical Engineering and Computer Science at the University of Tennessee, has the position of a Distinguished Research Staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow in the Computer Science and Mathematics Schools at the University of Manchester, and an Adjunct Professor in the Computer Science Department at Rice University.