LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU

Tingxing Dong
Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, TN 37996
Email: tdong@utk.edu

Azzam Haidar
Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, TN 37996
Email: haidar@eecs.utk.edu

Piotr Luszczek
Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, TN 37996
Email: luszczek@eecs.utk.edu

James Austin Harris
Department of Physics Astronomy
University of Tennessee, Knoxville
Knoxville, TN 37996
Email: jharr100@utk.edu

Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, TN 37996
Email: tomov@eecs.utk.edu

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee, Knoxville
Oak Ridge National Laboratory, USA
University of Manchester M13 9PL, UK
Email: dongarra@eecs.utk.edu

***Abstract*—Gaussian Elimination is commonly used to solve dense linear systems in scientific models. In a large number of applications, a need arises to solve many small size problems, instead of few large linear systems. The size of each of these small linear systems depends, for example, on the number of the ordinary differential equations (ODEs) used in the model, and can be on the order of hundreds of unknowns. To efficiently exploit the computing power of modern accelerator hardware, these linear systems are processed in batches. To improve the numerical stability of the Gaussian Elimination, at least partial pivoting is required, most often accomplished with row pivoting. However, row pivoting can result in a severe performance penalty on GPUs because it brings in thread divergence and non-coalesced memory accesses. The state-of-the-art libraries for linear algebra that target GPUs, such as MAGMA, focus on large matrix sizes. They change the data layout by transposing the matrix to avoid these divergence and non-coalescing penalties. However, the data movement associated with transposition is very expensive for small matrices. In this paper, we propose a batched LU factorization for GPUs by using a multi-level blocked right looking algorithm that preserves the data layout but minimizes the penalty of partial pivoting. Our batched LU achieves up to 2.5-fold speedup when compared to the alternative CUBLAS solutions on a K40c GPU and 3.6-fold speedup over MKL on a node of the Titan supercomputer at ORNL in a nuclear reaction network simulation.**

I. Introduction

Various scientific applications use Gaussian elimination to solve dense linear systems. An important class of problems is when many small size systems, instead of few large ones, must be solved. Typically, the order of the linear systems is up to a few hundred, and their number is from a few thousand to millions. For example, subsurface transportation simulations have a number of reaction systems to solve. Each system involves computing a Jacobian matrix and iteratively applying the Gaussian elimination until an outer solver converges. The system size is typically around 100.

As another example, consider an astrophysics ODE solver with Newton-Raphson iteration [1]. Multiple zones are simulated in one MPI task and each zone corresponds to a small linear system with each one resulting in multiple sequential solves [1]. A sparse direct solver called MA48 solves a sparse unsymmetric system of m linear equations in n unknowns using Gaussian elimination. The typical matrix size is 150 by 150. If the matrix is symmetric and definite, the problem is reduced to batched Cholesky factorization [2]. Other examples include hydrodynamic simulations, e.g., where the need is to compute thousands of matrix-matrix multiplies (dgemm) for dimensions well below 100 by 100 [3].

The one-sided factorizations such as the Cholesky, LU, and QR factorizations are based on block outer-product updates of the trailing matrix. Algorithmically, this corresponds to a sequence of two distinct phases: the *panel factorization* and the *trailing matrix update*. Implementation of these two phases can be expressed as a straightforward loop shown in Algorithm 1.

Algorithm 1 Two-phase implementation of a one-sided factorization.

```
for P_i ∈ {P_1, P_2, ..., P_n} do
    PanelFactorize(P_i)
    TrailingMatrixUpdate(C^(i))
end for
```

The MAGMA library focuses on very large matrices by using a hybrid (CPU-GPU) algorithms [4]. Because the panel factorization is latency and memory-bound due to its predominant reliance on the Level 2 BLAS operations, MAGMA performs the panel factorization on the CPU and only uses the GPU to update the trailing matrix[5]. A data transfer of the factorized panel from the CPU to the GPU is required at each step of the loop in Algorithm 1 to perform the trailing matrix update. This overhead due to the data transfer can be overlapped in time with the GPU computation of the trailing matrix. This is possible because the panel matrix is small – on the order of hundreds of columns – while the trailing matrix is big – on the order of tens of thousands of columns.

In the batched LU implementation, however, we can not afford such a memory transfer at any step, since the trailing

matrix is small and the amount of computation is not sufficient to overlap it in time with the panel factorization. Many small data transfers will take away any performance advantage enjoyed by the GPU, especially due to the fact that the data for transfer are not continuous in the memory but instead are stored with a stride called a leading dimension.

Another challenge to achieving good performance is the pivoting, which is a source of thread divergence and non-coalescent memory accesses. This is the result of consecutive threads accessing the matrix elements with a stride of one column instead of one element stride when the matrix is stored in column-major format. To mitigate this issue, MAGMA transposes the whole matrix to the row-major format, performs the factorization, and then transposes the matrix back to column-major format. If the matrix is not square, extra storage is required of the same size as the original matrix. Additionally, the panel matrix must be transposed to the column-major format when factorizing the panel on the CPU. A special CUDA kernel performs these conversions between the two formats. Just as the CPU-GPU transfers, the transpose operations can be overlapped in time by the computation. However, the frequent transposes creates too large of an overhead for the batched factorization. Because of these restrictions, both panel factorization and the update have to be done on the GPU to avoid the data transfer overhead. Also, the data layout has to be preserved.

The rest of the paper is organized as follows. First, we give a brief overview of the related work on batched factorizations in Section II. Second, we describe a nuclear network astrophysics simulation called XNET that is used as an application background. The classic LU factorization algorithm and variants are examined in Section IV. Then, we detail our batched implementations in Section V. The performance obtained is presented and compared with existing implementations, including CUBLAS, in Section VII. The GPU accelerated result of XNET on the Titan supercomputer at ORNL is also presented. Finally, Section VIII concludes the paper.

II. Related work

Volkov et al. [6] implemented the right-looking algorithm for LU, Cholesky, and QR for GPUs. These implementations are similar to the ones in MAGMA [7] and target large problems. Villa et al. [8], [9] implemented batched LU targeting sizes up to 128 by 128 on the GPU. They did include not only partial pivoting but also complete pivoting. In their particular implementation, a single CUDA thread was used to solve one linear system of equations. However, their solution was not faster than the NVIDIA's batched LU implementation from CUBLAS [10].

III. Background

XNet is a fully implicit, general purpose solver for thermonuclear reaction networks in astrophysical applications [11]. Evolving the nuclear kinetics necessitates the choice of a suitable integration scheme. With the first-order backward Euler scheme, nuclear abundances, $\mathbf{y}$, are evolved by some change, $\Delta\mathbf{y}$, of the system over a time step, Δt, according to

$$\mathbf{y_{n+1}} = \mathbf{y_n} + \Delta\mathbf{y}. \tag{1}$$

This is done using the Newton-Raphson method, based on the Taylor series expansion of $\mathbf{y_{n+1}} = \mathbf{y_n} + f(\mathbf{y_{n+1}})$ about a known $f(\mathbf{y_n})$. This reduces to iteratively solving the N^2 dense matrix equation $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ in the form

$$\left(\frac{\tilde{\mathbf{I}}}{\Delta t} - \tilde{\mathbf{J}}\right)\Delta\mathbf{y} = f(\mathbf{y_n}), \tag{2}$$

where $\tilde{\mathbf{J}}$ is the Jacobian of $f(\mathbf{y_n})$. Iteration continues until the solution converges according to mass conservation or some more stringent abundance conservation test, the choice of which depends upon the desired accuracy. Each iteration requires computing the full set of abundance derivatives, calculating all reaction rates, evaluating the Jacobian, evaluating the right-hand side, and then performing one LU decomposition ($\sim \frac{2N^3}{3}$ floating-point operations) and backsubstitution ($\sim N^2$ floating-point operations). Double-precision floating-point arithmetic is required for the calculation, as the approach to equilibrium at various stages of the burning can lead to the near-cancellation of large reaction fluxes.

Extending the nuclear network approximation from an oversimplified 14-species α-network to one including 150 species or more in the nucleosynthesis evolution substantially extends the capability of the network to track a broad variety of particle captures. With the computational cost of evolving the network using a dense matrix solution being $\mathcal{O}(N^3)$, the advancement from the traditionally used 14-species α-network to a more realistic 150-species network makes the nucleosynthesis computation more expensive than the neutrino transport, because the number of species determines the size of the linear systems. Initial analysis reveals that increasing the number of species from 14 to 150 more than doubles the cost.

The evolution of the nuclear kinetics for any single time step on a single zone can be modeled as follows:

1) Choose a zone in the radial ray over which to subcycle, if necessary;
2) Calculate the reaction rates from the REACLIB database for nuclear reaction rates [12];
3) Calculate the necessary time step;
4) Build the left-hand side and right-hand side of Equation 2;
5) Perform LU decomposition (dgetrf) and triangular back-substitution (dgetrs) for each linear system;
6) Update the nuclear abundances with the solution vector, $\Delta\mathbf{y}$, representing the trial change in abundances for a single time step;
7) Continue the time step loop until desired time.

Step 5 of solving the linear systems (Equation 2) takes about 75% of the total time. Since each zone can be solved independently, there is a great deal of task-level (zone-level) parallelism to be exploited in this problem. However, relatively small problem sizes (14 or 150) limit the degree of data-level parallelism. This *occupancy problem* is at the crux of optimizing the nucleosynthesis solution and is a reflection on the imbalance between task-level and data-level parallelism. By grouping the systems of Equation 2 into batches of sufficient size, we can shift the imbalance so that we may exploit the computational tools at our disposal. Batching the zones in the

radial zone-loop directly address the issue of occupancy by only launching a kernel when there is sufficient computation for the GPU to perform. We can then frame the problem of optimization as an attempt to balance task-level parallelism on the CPU with threads and data-level parallelism on the GPU.

IV. Algorithmic Variants

The LU factorization (also called decomposition) is the first step in solving a dense linear system of equations $Ax = b$, where $A \in \mathbb{R}^{m \times n}$. The LU factorization of A with partial pivoting has the form $PA = LU$, where $L \in \mathbb{R}^{m \times n}$ is a lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), $U \in \mathbb{R}^{n \times n}$ is an upper triangular matrix (upper trapezoidal if $m < n$), and $P \in \{0,1\}^{m \times m}$ is the row permutation matrix.

A. The Blocked Right-Looking Algorithm

The blocked right-looking variant is shown in Algorithm 2 and its patterns of access to matrix elements is depicted in Figure 1. The factorization of the m by n matrix A proceeds in $\lceil n/nb \rceil$ steps of size nb except for the last one. The computation of the above steps in the LAPACK routine dgetrf, involves four operations: dgetf2, dtrsm, dgemm, and dlaswp.

For $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, where $A_{11} \in \mathbb{R}^{nb \times nb}$, the $\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix}$ submatrix is called a panel matrix. The panels are factorized by the dgetf2 routine:

$$P \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11}. \tag{3}$$

The L_{11} and U_{11} submatrices overwrite A_{11}. The transformations in this panel factorization, along with the pivoting P must be applied to the trailing matrix before the factorization proceeds to the next step. Related to the pivoting, the rows of A_{12} are permuted with the selected from the factorization pivot rows (from A_{12} and A_{22}). This is done by the dlaswp routine. The pivoting information is stored in a vector generated by dgetf2.

After the permutation, A_{12} is updated by a lower-triangular solve $A'_{12} \leftarrow L_{11}^{-1} A_{12}$ (dtrsm), and A_{22} is updated by the so called Schur complement: $A'_{22} = A_{22} - A'_{21} A'_{12}$ (dgemm) The trailing matrix A'_{22} is now considered as the new matrix to be factored in the next iteration of the loop. This algorithm keeps updating the right hand side – the trailing matrix – and hence it is called right-looking.

The dtrsm and the dgemm routines are known as Level 3 BLAS – they allow for cache-friendly implementations that scale well with computational load without overly taxing the main memory bus. Due to the use of Level 3 BLAS, the blocked implementations perform very well and reach high flops-per-second, and in particular much higher than a non-blocking implementation that relies on memory-bound operations such as the Level 2 BLAS [13].

Algorithm 2 The blocked right looking LU factorization.

for $i \in \{1, 2, 3, \ldots, n/nb\}$ **do**
  Panel Factorize $A_{ii} = L_{ii} U_{ii}$
  Compute $A_{ij} = L_{ii}^{-1} A_{ij}$
  Permutation P
  Trailing Matrix Update $A_{jj} = A_{jj} - A_{ji} A_{ij}$ where $A_{ij} = a(i \times nb : n, j \times nb : n)$
end for

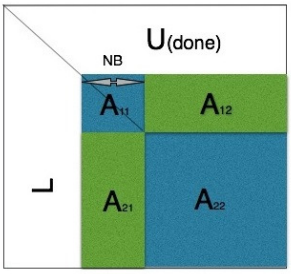

Fig. 1. Access patterns to matrix regions for the blocked right-looking LU factorization algorithm.

B. Multi-level blocked algorithm

The multi-level blocked algorithm is a variant of the blocked algorithm, depicted in Figure 2. The main difference is in the update of the trailing matrix. The right-looking variant operates on a current panel and updates all the way to the right. The multi-level blocked variant only applies the update to the next panel, but postpones the update of the rest of the trailing matrix after the "k-levels" of panels are factorized. Figure 2 shows the two-levels blocking of the LU factorization.

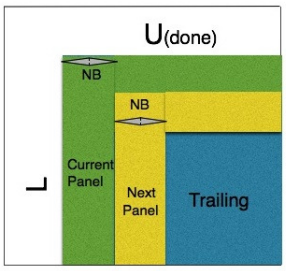

Fig. 2. Access patterns to matrix regions for the recursive blocked algorithm.

V. Batched Implementation

We target matrices of size less than or equal to 512, since most application candidates for batched execution are of this size [1], [2], [9]. Beyond the size of 512, we assume a single matrix is large enough to saturate the GPU's computational throughput and users can call MAGMA's standard dgetrf, even though our code can be extended in a straightforward fashion for sizes above 512.

A. Batched routines implementation

In a batched problem, each matrix is a separate problem that is solved independently. All of the routines discussed are batched and denoted by the corresponding LAPACK routine

name. We have implemented the routines in the four standard precision arithmetics – single real, double real, single complex, and double complex. For convenience, we use double precision routine name throughout the paper.

1) dgetf2: dgetf2 is used to factorize a panel of size $m \times nb$ at each step of the LU factorization. It consists of three Level 1 BLAS calls (idamax, dswap and dscal) and one Level 2 BLAS call (dger). Note that a natural way of implementing dgetf2 could be to load the panel to the GPU's shared memory and then do the entire computation before writing the result back to the main memory. However, this direction can not be easily implemented and can not provide good performance for two main reason. First, the size of the shared memory is limited currently to only 48KB per streaming multiprocessor (SMX) for the newest Nvidia K40, which limits the panels that can fit at once in it. Second, saturating the shared memory per SMX can decrease performance, since only one thread-block will be mapped to a SMX at a time. Indeed, due to a limited parallelism in the factorization of a small panel, the number of threads used in the thread block will be limited, resulting in low occupancy, and subsequently poor core utilization. In our implementation of dgetf2 , to perform the Gaussian elimination for the i^{th} column of the panel, we load only the column i to the shared memory. This is the only data that we reuse within one step. We found that such an implementation allows many thread-blocks to be executed by the same SMX in parallel, and thus taking a better advantage of its resources. Since the CUDA warp consists of 32 threads, it is recommended to develop CUDA kernels that use multiple of 32 threads per thread-block. For our batched algorithm, we discovered empirically that the best value for nb is 32.

2) dlaswp: To improve the numerical stability, pivoting is required. However, pivoting can be a performance killer for matrices stored in column major format. Indeed, a factorization directly in column-major format can be two times slower (depending on hardware and problem sizes) than implementations that transpose the matrix in order to internally use a row-major storage format [6]. Yet, experiments show that this conversion is too expensive for batched problems. In the LAPACK's dlaswp, the row swapping operations are serial, that is row by row. This limits the parallelism and is one of the factors for slow dlaswp for matrices in the column-major format. To minimize this penalty, we proposed a parallel swapping, detailed in Section VI-A.

3) dtrsm: After the panel factorization (3) and the row swapping, we compute the inverse of L_{11}, L_{11}^{-1}, with the dtrtri routine. Then, the A'_{12} update is accomplished by a dgemm, $A'_{12} = L_{11}^{-1} A_{12}$. Generally, computing the inverse of a matrix may suffer from numerical stability, but since A_{11} results from the numerically stable LU with partial pivoting and its size is just $nb \times nb$, or in our case 32×32, we do not have this problem [14].

4) dgemm: The goal of our batched LU is to reach the performance of the batched dgemm. Because of its importance, a lot of previous efforts have been focused on optimizing the dgemm routine. In particular for our case, dgemm is not only used in the trailing matrix updates but also in the implementation of the triangular matrix solvers (dtrsm). Since NVIDIA CUBLAS dgemm is written in assembly language and highly optimized on Kepler architecture, we call CUBLAS routines.

VI. Various factor impacts on the performance

A. Parallel swapping

We analyzed and evaluated the implementation as described above to find that more than 60% of the factorization time is spent in the swapping routine. Figure 3 shows the execution trace of 2,000 batched LU factorization of matrices of size 512. We can observe on the top trace that the classical dlaswp kernel is the most time consuming part of the algorithm. The swapping consists of nb successive interchanges of two rows of the matrices. The main reason that this kernel is the most time consuming is because the nb row interchanges are performed in a sequential order, and that the data of a row is not coalescent, thus the thread warps do not read/write it in parallel. It is clear that the main bottleneck here is the memory access. Indeed, slow memory accesses compared to high compute capabilities have been a persistent problem for both CPUs and GPUs. CPUs for example alleviate the effect of the long latency operations and bandwidth limitations by using hierarchical caches. Accelerators on the other hand, in addition to hierarchical memories, uses thread level parallelism (TLP) where threads are grouped into warps (e.g., of 32 threads) and multiple warps assigned for execution on the same SMX unit. The idea is that when a warp issues an access to the device memory, it will stall until the memory returns a value, but the accelerators scheduler switches to another warp. In this way, even if some warps stall, other warps can execute, keeping functional units busy while resolving data dependencies, branch penalties, and long latency memory requests. In order to overcome the bottleneck of swapping, we proposed to modify the kernel in order to apply all nb row swaps in parallel. This modification will also allow the coalescent write of the first nb rows of the matrix. So we changed the algorithm to generate two pivot vectors, where the first vector gives the final destination row indices for the first nb rows of the panel, and the second gives the row indices of the nb rows that must become the first nb rows of the panel. Figure 3 depicts the execution trace (bottom) when using our parallel dlaswp kernel. The experiment shows that this reduces the time spent in the kernel from 60% to around 10%. Note that the colors between the top and the bottom traces do not match each other; this is because the Nvidia profiler puts always the most expensive kernel in green. As a result, the gain obtained in terms of performance is around 50%, as shown in Figure 9.

B. Nested blocking

The panel factorization as described in V-A1 goes over the nb columns and factorizes them one after another, similarly to the LAPACK algorithm. At each of the nb steps, a rank-1 update is required to update the vectors at the right hand side of the factorized column i (this operation is done by the dger kernel). Since we cannot load the entire panel into the shared memory of the GPU, the right hand side vectors are loaded back and forth from the main memory at every step. Thus, one can expect that the rank-1 operation is the most time consuming of the panel factorization. A detailed analysis using the profiler reveals that the dger kernel consists

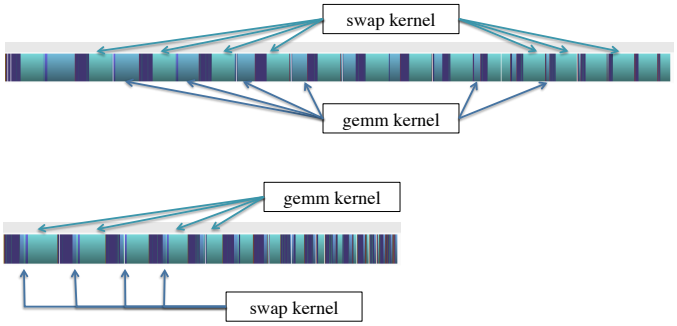

Fig. 3. Execution trace of the batched LU factorization using either classical swap (top) or our new parallel swap (bottom).

of more than 80% of the panel factorization time, and around 40% of the total LU factorization time. Similarly to the swapping kernel described above the main bottleneck here is the memory access. For that, we propose to improve the efficiency of this kernel and to reduce the memory access by using a recursive level of blocking techniques as depicted in Figure 4. In principle, the panel can be blocked recursively until a single element. Yet, in practice, 2-3 blocked levels are sufficient to achieve high performance. The above routines must be optimized for each blocked level, which complicates the implementation. The boost in performance obtained by this optimization is around 25%, as demonstrated in Figure 9.

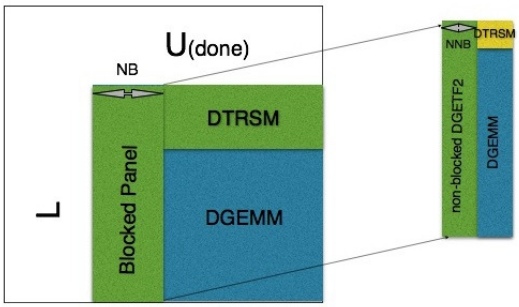

Fig. 4. Blocked panel factorization

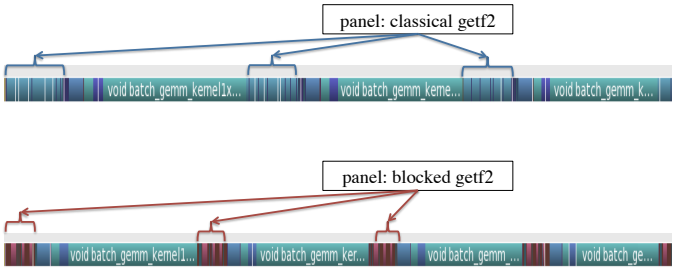

Fig. 5. Execution trace of the batched LU factorization using either classical getf2 (top) or our recursive getf2 (bottom).

C. Streamed dgemm

Our main goal is to achieve higher performance and to accomplish this we performed deep analysis of every kernel of the algorithm. We found that 70% of the time is spent in the batched dgemm kernel. An evaluation of the performance of the dgemm kernel using either batched or streamed dgemm is illustrated in Figure 6. The curves let us conclude that the streamed dgemm was performing better than the batched one for some cases, e.g., for $k = 32$ when the matrix size is of order of $m > 200$ and $n > 200$. We note that the performance of the batched dgemm is stable and does not dependent on k, in the sense that the difference in performance between $k = 32$ and $k = 128$ is minor. However it is bound by 300 Gflop/s. For that we proposed to use the streamed dgemm whenever it is faster, and to roll back to the batched one otherwise. Figure 7 shows the trace of the batched LU factorization of 2,000 matrices of size 512 each, using either the batched dgemm (top trace) or the combined streamed/batched dgemm (bottom trace). We can see that the use of the streamed dgemm (when the size allows it) can speed up the factorization by about 20% and this is confirmed by the performance curve plotted in Figure 9.

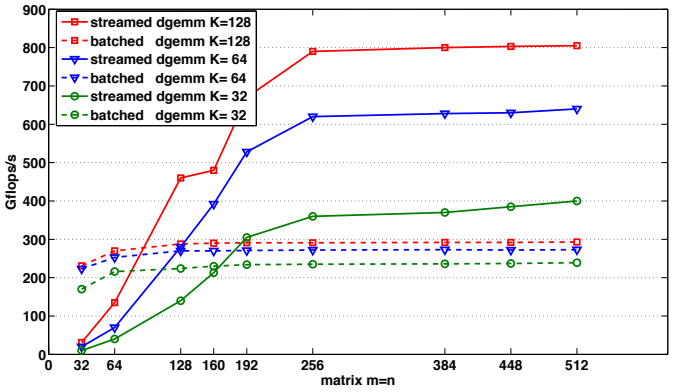

Fig. 6. Performance comparison between the streamed and the batched dgemm kernel for different value of K and different matrix sizes where $m = n$.

D. Multi-level blocking of the update

The performance of the streamed dgemm kernel as shown in Figure 6 is highly dependent on the size of the matrices. In particular, this affects the trailing matrix updates in the LU factorization which consist of rank-k operations. The performance of the streamed dgemm kernel is around twice higher for $k = 128$ than for $k = 32$. Since our panel size is limited to 32, the performance of the trailing matrix update is limited by the performance of the dgemm for $k = 32$. However, in order to achieve higher performance, we propose to use multi-level of blocking of the trailing matrix update. The idea here is to use multi-level of blocking during the trailing matrix update. This means that at step i we only update the next panel and delay the subsequent portion of the update till step $i + l$ where we reach a value of k that is acceptable to perform the whole update of the delayed portion and then start over again. For example, at step $i = 0$ the update is performed only on the next 32 columns and then at the next step $i = 1$ the dgemm will be using $k = 64$ and so on. The impact of this strategy can be seen only for large matrices of size $m > 128$. Figure 8 shows the multi-level technique for two values of $k = 64$ (top graph) and $k = 128$ (bottom graph). The top graph shows the trace when the value of k is equal to 64. The

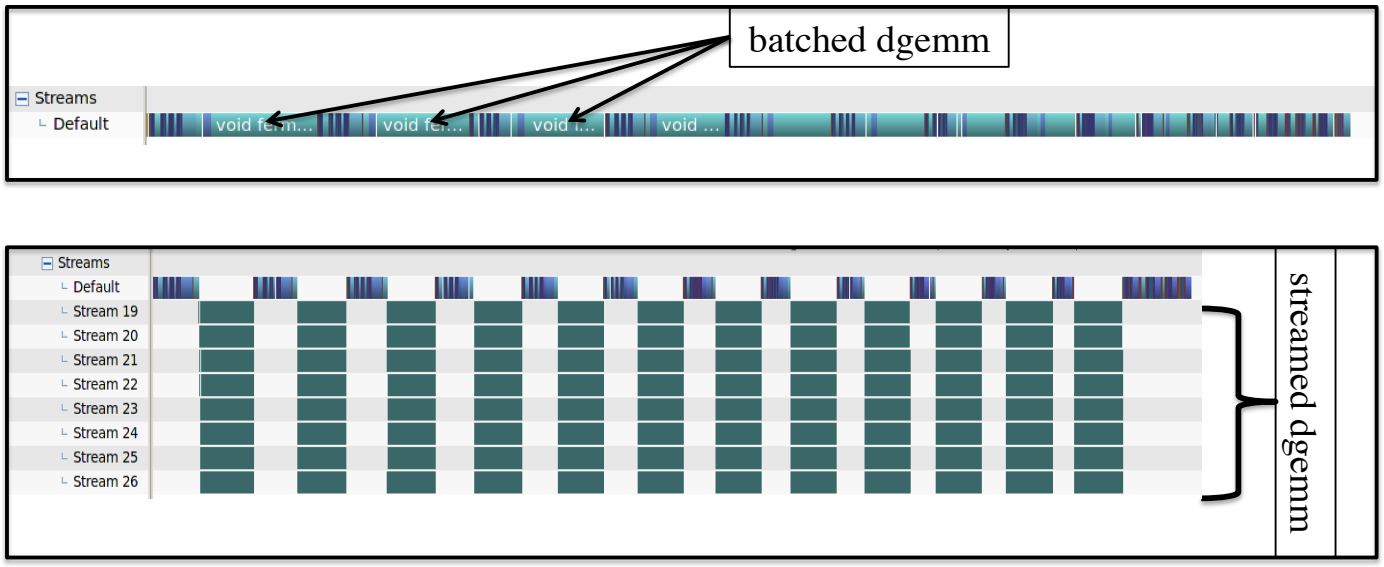

Fig. 7. Execution trace of the batched LU factorization using either batched dgemm (top) or streamed/batched dgemm (bottom).

update is delayed one step and thus at step i only the next panel is updated using a batched dgemm because it is faster when $n = k = 32$ and then at step $i+1$ a streamed/batched dgemm is performed using $k = 64$. The bottom graph shows the trace when the value of k is chosen to be 128. Here we can see that the trailing matrix update is delayed 3 steps until we perform a dgemm with $k = 128$. We can observe that for this range of small matrices, increasing the value of acceptable "k" for example to 128 gives us the advantage of performing dgemm at higher speed but it reduce the number of such dgemm operations (3 operations for a matrix of size 512 and $k = 128$ *vs.* 6 operations for $k = 64$). The performance observed is similar for both $k = 64$ and $k = 128$ for matrices of size 512 while $k = 64$ is always outperforming $k = 128$ for smaller sizes. As a result a trade-off value of k need to be chosen depending on the matrix size. The improvement obtained by this technique is around 15%, as shown in Figure 9.

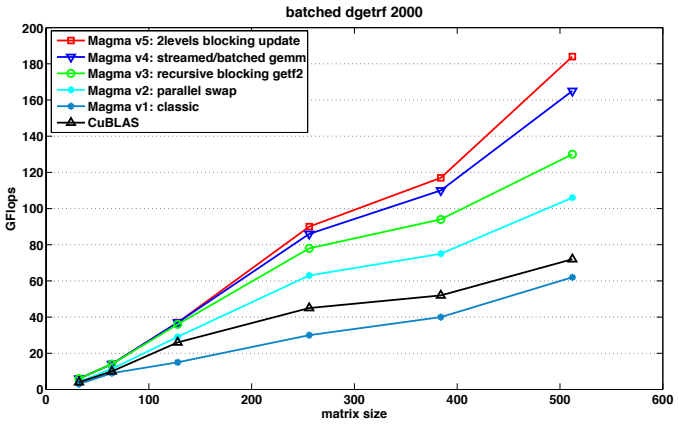

Fig. 9. Performance in Gflops/s of our different version of the batched LU factorization compared to the CUBLAS implementation.

VII. Performance Results

A. Comparison with CUBLAS on a K40c

We conducted our experiments on NVIDIA K40c cards with 11.6 GB of GDDR memory per card running at 825 MHz. The cards were connected to the host via two PCIe I/O hubs with 6 GB/s bandwidth.

CUBLAS version 5.5 features a dgetrfBatched routine. By comparison, our batched LU is up to 2.5× faster than the CUBLAS routine as shown in Figure 9. The slowest code in the figure has performance below 60 Gflop/s and is marked as "classic" – it corresponds to the performance of the MAGMA [4] library, which was optimized for large matrices. The *classic* implementation is improved upon by CUBLAS' dgetrfBatched version (marked as "CUBLAS" in Figure 9) and the performance exceeds 70 Gflop/s. To go beyond 100 Gflop/s, we used the code that optimizes pivoting with *parallel swap*. Next step in performance improvement is the use of variable blocking (also called *recursive blocking* getf2), which enables performance levels that go slightly above 130 Gflop/s. The final two improvements are *streamed/batched gemm*, which moves the performance level beyond 160 Gflop/s, and finally, *2-levels blocking update* completes the set of optimizations and takes the performance beyond the 180 Gflop/s mark. Each of these optimizations is described in detail in Section V.

B. XNET Application with GPU acceleration on Titan

We performs experiments for a 2D test of the nuclear network application described by Equation 2 on a single node of Titan [**?**]. We used three different solvers to solve the linear system involved in the simulation. We used 1) the LU factorization of a dense matrices dgetrf from the Intel Math Library MKL [15], 2) the MA48 factorization from the Harwell Subroutine Library [16], which solves a sparse unsymmetric linear systems using Gaussian elimination, and finally our batched GPU implementation of the LU factorization for dense matrices. The 2D XNET test consists of 419 zones, which also represents the number of linear systems that need to be

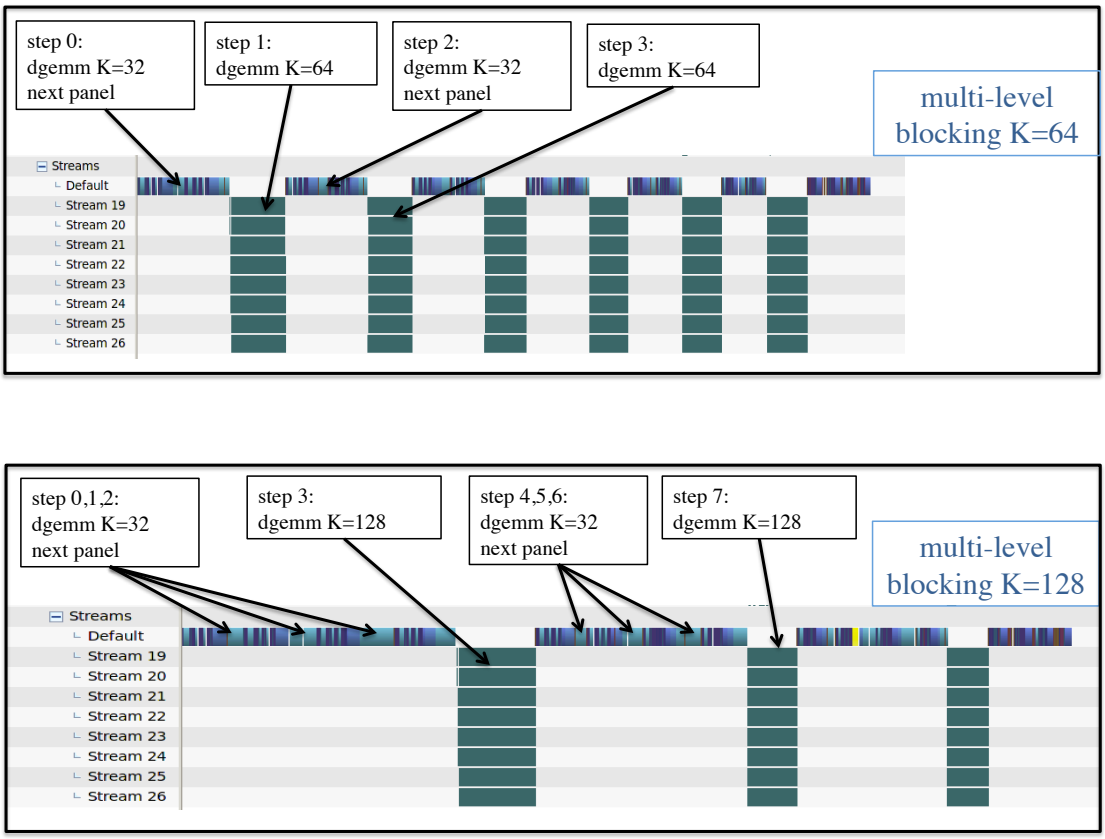

Fig. 8. The multi-level technique for $k = 64$ (top graph) and $k = 128$ (bottom graph).

solved in a set of 64 zones each. We note that when the domain size and dimension increases, the number of zones can be tens of thousands, but the linear system computation is always local. For example, in a large 3D simulations, we can distribute more MPI tasks across the nodes, and there would be 3200 zones per GPU. The solver time includes both the LU factorization and the backward substitution, which is done on the CPU. Both MKL and MA48 are optimally implemented and called, especially considering such a small data size are comparable with the L3 Cache, which is 16MB, on Titan AMD CPU. The GPU is a Nvidia K20x. The speedup is shown in Figure 10. Although MAGMA batched dgetrf only accelerated the factorization part of the solver, it achieved a 3.6× overall speedup over MKL and 1.8× over MA48.

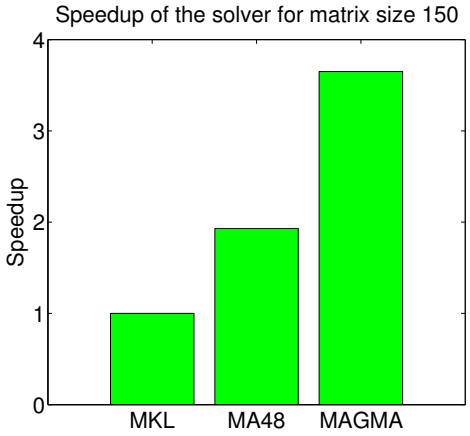

Fig. 10. XNET Solver speedups using either the sparse solver MA48 or our MAGMA batched LU factorization on a K20x GPU versus the MKL library.

VIII. Conclusions

The need to solve large number of small linear systems often arises in scientific computing applications. Examples vary from electronic structure calculations to nuclear reaction network simulations to electromagnetism and radar simulations, to just mention a few. In contrast to large linear system which can expose data parallelism and can be efficiently implemented on either GPUs or CPUs, solving small linear systems is memory bound. This is due to the fact that the ratio of the computation to the data needed is very small compared to the one for large matrices. Existing numerical libraries for the highly parallel GPU architectures can not perform well on such small problems. We demonstrated that GPU architectures can be used efficiently for solving many small size problems. In particular, we developed different algorithm variants and optimization techniques for the batched LU factorization on GPUs and analyzed their impacts on performance. These techniques can be used by other high level linear algebra solvers, for example, QR, Cholesky, as well. Our performance exceeded the CUBLAS dgetrfBatched by up to 2.5×. By integrating our batched LU in a nuclear network simultation, we achieved up to 3.6× speedup over the MKL Library for solving hundreds of matrices of size 150×150 on

a node of the Titan supercomputer at ORNL.

ACKNOWLEDGMENT

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA and MAGMA project support. The authors thank the testing support of the Performance End Station PEAC Project sponsored by DOE under Contract No. DE-AC05- 00OR22725.

REFERENCES

[1] 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.

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

[3] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, "A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU," in *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

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

[5] I. Yamazaki, S. Tomov, and J. Dongarra, "One-sided dense matrix factorizations on a multicore with multiple GPU accelerators in MAGMA," in *International Conference on Computational Science ICCS*, 2012.

[6] V. Volkov and J. W. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," Tech. Rep. LAPACK Working Note 202.

[7] S. Tomov, R. Nath, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2014.

[8] 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*, Aachen, Germany, August 26-30 2013.

[9] 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.

[10] "CUBLAS," http://docs.nvidia.com/cuda/cublas/.

[11] W. Raphael and Friedrich-Karl, "Silicon burning II: Quasi-equilibrium and explosive burning," *ApJ*, vol. 511, pp. 862–875, February 1999.

[12] R. H. Cyburt, A. M. Amthor, R. Ferguson, Z. Meisel, K. Smith, S. Warren, A. Heger, R. D. Hoffman, T. Rauscher, A. Sakharuk, H. Schatz, F. K. Thielemann, , and M. Wiescher, "The JINA REACLIB database: Its recent updates and impact on Type-I X-ray bursts," ., vol. 189, pp. 240–252, July 2010.

[13] K. Gallivan, W. Jalby, and U. Meier, "The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory," *SIAM J. Sci. Stat. Comp.*, vol. 8, 1987, 10791084.

[14] D. Croz, J. J. Dongarra, and N. J. Higham, "Stability of methods for matrix inversion," *IMA J. Numer. Anal.*, vol. 12, no. 119, 1992.

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

[16] "HSL. A collection of Fortran codes for large scale scientific computation," 2013, http://www.hsl.rl.ac.uk".