Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

# One-sided dense matrix factorizations on a multicore with multiple GPU accelerators in MAGMA[1]

Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra
University of Tennessee, Knoxville, TN37996, USA
{iyamazak, tomov, dongarra}@cs.utk.edu

**Abstract**

One-sided dense matrix factorizations are important computational kernels in many scientific and engineering simulations. In this paper, we propose two extensions of both right-looking (LU and QR) and left-looking (Cholesky) factorization algorithms to utilize the computing power of current heterogeneous architectures. We first describe a new class of non-GPU-resident algorithms that factorize only a submatrix of a coefficient matrix on a GPU at a time. We then extend the algorithms to use multiple GPUs attached to a multicore. These extensions enable the factorization of a matrix that does not fit in the aggregated memory of the multiple GPUs at once, and provide potential of fully utilizing the computing power of the architectures. Since data movement is expensive on the current architectures, these algorithms are designed to minimize the data movement at multiple levels. These algorithms are now parts of the MAGMA software package, a set of the state-of-the-art dense linear algebra routines for a multicore with GPUs. To demonstrate the effectiveness of these algorithms, we present their performance on a single compute node of the Keeneland system, which consists of twelve Intel Xeon processors and three NVIDIA GPUs. The performance results show the scalable performance of our multi-GPU algorithms and the negligible overheads of our non-GPU-resident algorithms due to the communication-avoiding techniques employed.

*Keywords:* dense linear algebra; one-sided factorization; GPU accelerators;

## 1. Introduction

Moore's law predicted that the number of transistors on a chip would double every 18 months. This trend has continued for more than half a century and is expected to continue through the end of the next decade. However, to continue this trend under physical limitations (e.g., power constraint), emerging architectures are based on multicore processors like homogeneous x86-based multicore CPUs, NVIDIA GPUs, and AMD Fusion and Intel MIC architectures. Furthermore, leading high-performance computers are based on heterogeneous architecture consisting of these different types of processors. For example, an SGI Altix UV 100 system at the National Institute for Computational Sciences (NICS) consists of 1024 cores of Intel Nehalem EX processors, with 427TB of shared memory,

[1]This research was supported by DoE DE-SC0003852, DoE DE-SC0004983, and Georgia Institute of Technology RA241-G1 grants. We used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. We also thank NVIDIA and MATLAB for supporting our research efforts.

coupled with 8 GPUs. Another example is the Keeneland system at the Georgia Institute of Technology, whose single compute node has two six-core Intel Xeon processors and three NVIDIA GPUs. These different types of processors on the heterogeneous architectures are adapted for particular types of tasks. For instance, GPUs are designed to maximize the throughput of multiple tasks, and they are particularly adapted to handle tasks that exhibits strong data or thread-level parallelism. On the other hand, a CPU is designed to minimize the latency of a single task using deep memory-hierarchy and instruction-level parallelism. To fully utilize the computing power of such architectures, software must be re-designed to exploit the different performance strengths of different processors.

Linear Algebra PACKage (LAPACK) [1] is a set of dense linear algebra routines for a multicore, and is commonly used in scientific and engineering simulations. LAPACK performs most of its computation using Basic Linear Algebra Subroutines (BLAS) [2], which implements basic vector-vector, matrix-vector, and matrix-matrix operations, which are respectively classified as BLAS-1, BLAS-2, and BLAS-3 operations. There are implementations of BLAS that are optimized for specific multicore architectures.

To utilize the computing power of heterogeneous architectures, Matrix Algebra on GPU and Multicore Architectures (MAGMA) [3] extends LAPACK routines using a hybrid programming paradigm. One important set of such routines implement one-sided factorization algorithms (like LU, Cholesky, and QR factorizations), which consist of sequences of two distinct phases: a panel factorization and submatrix update. Most of the computation in the submatrix update is performed using BLAS-3, which exhibits more data parallelism than BLAS-1 or BLAS-2 and is ideal for running on GPUs. On the other hand, the panel factorization is mainly based on BLAS-1 and BLAS-2, and is often faster on a CPU [4, 5]. Hence, the current version of MAGMA stores the whole coefficient matrix on a GPU and uses the single GPU for the submatrix update, while a multicore CPU is used for the panel factorizations. Significant speedups have been obtained using MAGMA over a vendor-optimized LAPACK [6].

In this paper, we extend the LU, Cholesky, and QR factorization algorithms of MAGMA to further exploit the computing power of the heterogeneous architectures. We first describe our non-GPU-resident factorization algorithms that store only a part of the matrix on the GPU at a time. We then extend the algorithms to use multiple GPUs attached to the multicore. These extensions enable the factorization of a matrix that does not fit in the aggregated memory of the multiple GPUs at once, and provide potential of fully utilizing the computing power of the architecture. Since data movement is expensive on the current architecture, these algorithms are designed to minimize the data movement at multiple levels. To demonstrate the effectiveness of these algorithms, we show their performance on a single compute node of the Keeneland system.

The rest of the paper is organized as follows: In Sections 2 and 3, we first describe the one-sided factorization algorithms of LAPACK and MAGMA, respectively. Then, in Sections 4 and 5, we respectively describe our non-GPU-resident and multi-GPU factorization algorithms. The performance results are presented in Section 6, and in Section 7, we conclude with final remarks.

## 2. Block-based factorization algorithms

In this section, we describe the one-sided factorization algorithms of LAPACK.

*Right-looking LU and QR factorizations.* An LU factorization of an $m$-by-$n$ matrix $A$ with partial pivoting is of the form

$$PA = LU, \tag{1}$$

where $P$ is an $m$-by-$m$ permutation matrix, $L$ is an $m$-by-$n$ unit lower-triangular matrix, and $U$ is an $n$-by-$n$ upper-triangular matrix. The LAPACK routine **xGETRF** computes this LU factorization, where **x** can be either **S**, **D**, **C**, or **Z** denoting either single, double, single-complex, or double-complex precision used for computing the factorization. In LAPACK, all the matrices are stored in column-major.

The first step of **xGETRF** computes the factorization

$$P_1 A = P_1 \begin{pmatrix} a_{1,1} & a_{1,2:n_b} \\ a_{2:m_b,1} & a_{2:m_b,2:n_b} \end{pmatrix} = \begin{pmatrix} \ell_{1,1} & \\ \ell_{2:m_b,1} & I \end{pmatrix} \begin{pmatrix} I & \\ & \widehat{A} \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2:n_b} \\ & I \end{pmatrix}, \tag{2}$$

where $P_1$ represents the pivoting for the first $b$ columns of $A$; $a_{1,1}$, $\ell_{1,1}$, and $u_{1,1}$ are the leading $b$-by-$b$ submatrices of $A$, $L$, and $U$, respectively; and $m_b$ and $n_b$ are the respective numbers of blocks in the row and column of $A$ (i.e., $m_b = \frac{m}{b}$

and $n_b = \frac{n}{b}$).[2] This factorization (2) is computed by the following two-stage algorithm, where the corresponding LAPACK or BLAS routines are shown in bold case:

1. *Panel factorization.* **xGETRF2** computes an LU factorization of the leading $m$-by-$b$ block column $a_{:,1}$ of $A$ with partial pivoting; i.e.,
$$P_1 \begin{pmatrix} a_{1,1} \\ a_{2:n_b,1} \end{pmatrix} = \begin{pmatrix} \ell_{1,1} \\ \ell_{2:n_b,1} \end{pmatrix} (u_{1,1}),$$
where $a_{:,j}$ is the submatrix consisting of the $((j-1)b+1)$-th through the $(jb)$-th columns of $A$ and referred to as the $j$-th panel of the factorization.
2. *Submatrix update.* The transformation computed by `xGETRF2` is applied to the $m$-by-$(n-b)$ trailing matrix $a_{:,2:n_b}$, which is the submatrix consisting of the $(b+1)$-th through the $n$-th columns of $A$, i.e.,
   (a) **xLASWP** applies the pivoting $P_1$ to the trailing submatrix:
   $$\begin{pmatrix} a_{1,2:n_b} \\ a_{2:m_b,2:n_b} \end{pmatrix} := P_1 \begin{pmatrix} a_{1,2:n_b} \\ a_{2:m_b,2:n_b} \end{pmatrix}.$$
   (b) **xTRSM** computes the off-diagonal blocks $u_{1,2:n_b}$ of $U$:
   $$(u_{1,2:n_b}) = (\ell_{11})^{-1}(a_{1,2:n_b}).$$
   (c) **xGEMM** updates the trailing submatrix $a_{2:m_b,2:n_b}$:
   $$\widehat{A} := (a_{2:m_b,2:n_b}) - (\ell_{2:m_b,1})(u_{1,2:n_b}).$$

To compute the LU factorization of $A$, the transformation (2) is recursively applied to the $(n-b)$-by-$(n-b)$ submatrix $\widehat{A}$. Notice that after the $j$-th panel factorization, the pivoting $P_j$ must be applied to both $a_{:,1:(j-1)}$ and $a_{:,(j+1):n_b}$. The above algorithm is referred to as a right-looking algorithm since at each step, the panel is used to update the trailing submatrix, which is on the right of the panel. The upper-triangular part of $U$ and the strictly lower-triangular part of $L$ are stored in the corresponding parts of $A$. An additional $\min(n,m)$-length vector is required to store the pivots $P$.

The QR factorization of an $m$-by-$n$ matrix $A$ is of the form
$$A = QR, \tag{3}$$
where $Q$ is an $m$-by-$m$ orthonormal matrix, and $R$ is an $m$-by-$n$ upper-triangular matrix.

The LAPACK routine for computing the QR factorization is **xGEQRF** whose first step computes the factorization
$$A = \begin{pmatrix} a_{1,1} & a_{1,2:n_b} \\ a_{2:m_b,1} & a_{2:m_b,2:n_b} \end{pmatrix} = H_1 \begin{pmatrix} r_{1,1} & r_{1,2:n_b} \\ 0 & \widehat{A} \end{pmatrix}, \tag{4}$$
where $H_1$ is an $m$-by-$m$ Householder matrix that transforms the 1-st panel $a_{:,1}$ into a upper-triangular form (i.e., $r_{1,1}$ is upper triangular). This factorization is computed by the following two-stage right-looking algorithm:

1. *Panel factorization.* **xGEQR2** computes the Householder matrix $H_1$ such that
$$H_1^T(a_{:,1}) = \begin{pmatrix} r_{1,1} \\ 0 \end{pmatrix}.$$
2. *Submatrix update.* The transformation computed by **xGEQR2** is applied to the trailing submatrix $a_{:,2:n_b}$:
   (a) **DLARFT** computes a block representation of the transformation $H_1$, i.e.,
   $$H_1 = I - V_1 T_1 V_1^H,$$
   where $V_1$ is an $m$-by-$b$ matrix and $T_1$ is a $b$-by-$b$ upper-triangular matrix.
   (b) **DLARFB** applies the transformation computed by **xLARFT** to the trailing submatrix:
   $$\begin{pmatrix} r_{1,2:n_b} \\ \widehat{A} \end{pmatrix} := (I - VTV^H) \begin{pmatrix} a_{1,2:n_b} \\ a_{2:m_b,2:n_b} \end{pmatrix}. \tag{5}$$

Then, the QR factorization of $A$ is computed by recursively applying the transformation (4) to the submatrix $\widehat{A}$. The column reflectors $V_j$ are stored in the lower-triangular part of $A$, while $R$ is stored in the upper-triangular part. Additional $m$-by-$b$ storage is required to store $T_j$.

[2] To simplify our discussion, we assume that $m$ and $n$ are multiples of $b$, but the discussion can be easily extended to other cases.

*Left-looking Cholesky factorization.* The Cholskey factorization of a Hermitian positive-definite matrix $A$ is of the form

$$A = RR^H, \tag{6}$$

where $R$ is an $n$-by-$n$ lower-triangular matrix with positive real diagonals. The LAPACK routine for computing the Cholesky factor $R$ is **xPOTRF**. At the $j$-th step of **xPOTRF**, the $j$-th block column $r_{:,j}$ of $R$ is computed by updating and factorizing the $j$-th panel $a_{:,j}$ as follows:

1. *Panel update.* The $j$-th panel $a_{:,j}$ is updated using the previously-computed columns $r_{:,1}, r_{:,2}, \ldots, r_{:,j-1}$ of $R$,
   (a) **xSYRK** updates the diagonal block $a_{j,j}$ based on a symmetric low-rank updates,
   $$(a_{j,j}) := (a_{j,j}) - (r_{j,1:(j-1)})(r_{j,1:(j-1)})^H.$$
   (b) **xGEMM** updates the off-diagonal blocks $a_{(j+1):n_b,j}$ of $a_{:,j}$,
   $$(a_{(j+1):n_b,j}) := (a_{(j+1):n_b,j}) - (r_{(j+1):n_b,1:(j-1)})(r_{j,1:(j-1)})^H.$$
2. *Panel factorization.* The $j$-th panel $a_{:,j}$ is factorized,
   (a) **xPOTRF2** computes the Cholesky factor $r_{j,j}$ of the diagonal block $a_{j,j}$,
   $$(a_{j,j}) = (r_{j,j})(r_{j,j})^H.$$
   (b) **xTRSM** computes the off-diagonal blocks $r_{(j+1):n_b,j}$,
   $$(r_{(j+1):n_b,j}) = (r_{j,j})^{-1}(a_{(j+1):n_b,j}).$$

This is known as a left-looking algorithm since at each step, the panel is updated using the previous columns, which are on the left of the panel. If a right-looking algorithm is used, then to update the lower-triangular part of the trailing matrix $a_{2:n_b,2:n_b}$, **xSYRK** and **xGEMM** must be called on each block column of $a_{2:n_b,2:n_b}$. On the other hand, the left-looking algorithm updates the panel with single calls to **xSYRK** and **xGEMM**, and often exhibits more regular, and hence efficient, data access. The Cholesky algorithm above references only the lower-triangular part of $A$, which is overwritten by $R$. Alternatively, given the upper-triangular part of $A$, **xPOTRF** can compute $R^H$ by block-rows.

These LAPACK routines consist of block operations which can be performed using highly-tuned BLAS to improve the data locality of accessing data through the memory hierarchy of a specific multicore architecture. In particular, the panel factorizations are based on BLAS-1 or BLAS-2, while most of the computation in the submatrix or panel updates is performed using BLAS-3.

## 3. One-sided factorizations on a multicore with a GPU

MAGMA extends the LAPACK routines to utilize the computing power of modern heterogeneous architectures. It is based on a hybrid programming paradigm and exploits the different performance strengths of a CPU and a GPU. For instance, during one-sided factorizations, the submatrix updates based on BLAS-3 exhibit strong data parallelism and are ideal for performing on a GPU. On the other hand, panel factorizations are based on BLAS-1 and BLAS-2 and are often faster on a CPU. Hence, MAGMA stores the whole matrix $A$ and updates its submatrix on the GPU while the panel is copied to and factorized on the CPU. To be concrete, in this section, we outline the one-sided factorization algorithms of MAGMA. The name of the MAGMA routine appends **magma_** in front of that of the corresponding LAPACK or BLAS routine (e.g., **magma_xGETRF** for **xGETRF**). MAGMA uses the same matrix storage as LAPACK and BLAS on the CPU (i.e., column-major), and a user can switch from LAPACK to MAGMA by adding **magma_** to the corresponding routine calls in most cases. Here, we focus on the amount of the data communicated between the CPU and GPU since the communication can be expensive on the current architecture.

**Algorithm 3.1**
1. copy $A$ from CPU to GPU.
2. `for` $j = 1, 2, \ldots, n_b$ `do`
   a. use **xGETRF** to factorize panel on CPU.
   b. copy the panel from CPU to GPU.
   c. apply pivoting to $L$ on GPU.
   d. use **magma_xGETRF** to compute $u_{j,(j+1):n_b}$ on GPU.
   e. use **magma_xGEMM** to update trailing submatrix on GPU.
   f. copy $a_{(j+1):n_b,j+1}$ from GPU to CPU.
   `end for`
3. copy LU factors from GPU to CPU.

(a) LU factorization.

**Algorithm 3.3**
1. copy $A$ from CPU to GPU.
2. `for` $j = 1, 2, \ldots, n_b$ `do`
   a. copy $r_{1:(j-1),j}$ from GPU to CPU.
   b. copy panel $a_{j:n_b,j}$ from CPU to GPU.
   c. use **xGEQRF** to factorize the panel on CPU.
   d. use **xLARFT** to compute $T_j$ and $V_j$.
   e. copy $T_j$ and $V_j$ from CPU to GPU.
   f. use **magma_xLARFB** to apply the transformation on GPU.
   `end for`
3. synchronize copying of $R$ from step 2.a.

(b) QR factorization.

**Algorithm 3.2**
1. copy $A$ from CPU to GPU.
2. `for` $j = 1, 2, \ldots, n_b$ `do`
   a. use **magma_xSYRK** to update $a_{j,j}$ on GPU.
   b. copy $a_{1:(j-1),j}$ from GPU to CPU.
   c. copy $a_{j,j}$ from GPU to CPU.
   d. use **magma_xGEMM** to update $a_{(j+1):n_b,j}$ on GPU.
   e. use **xPOTRF** to compute $r_{j,j}$ on CPU.
   f. copy $r_{j,j}$ from CPU to GPU.
   g. use **magma_xTRSM** to compute $r_{(j+1):n_b,r}$ on GPU.
   `end for`
3. synchronize copying of $R$ from step 2.b.

(c) Cholesky factorization.

Figure 1: MAGMA one-sided factorization algorithms.

*LU and QR factorizations.* The MAGMA routine for computing an LU factorization (1) of $A$ is **magma_xGETRF** whose algorithm is outlined in Figure 1(a). This algorithm first copies the matrix $A$ from the CPU to the GPU (step 1 of Algorithm 3.1). Then, at the beginning of each step, the current panel is factorized on the CPU (step 2.a) and copied to the GPU (step 2.b) for the submatrix update (steps 2.d and 2.e). Once the whole $A$ is factorized, the LU factors are copied to the CPU (step 3). As in LAPACK, $A$ is overwritten by its LU factors on both CPU and GPU. The algorithm communicates total of $4mn - (n^2 - nb)$ matrix elements between the GPU and CPU, where $n^2 - nb$ is the total number of elements in the strictly-upper block triangular parts of the panels that are not copied between the CPU and GPU.

Since the panels are factorized on the CPU, if the whole panel $a_{:,j}$ is copied to the CPU at step 2.f of Algorithm 3.1, then the final copying of LU factors to the CPU (step 3) can be avoided. The reason for having step 3 in Algorithm 3.1 is that the GPU is used to apply the pivoting to the previous columns of $L$ (step 2.d). For an efficient data access for applying the pivoting on the GPU, the matrix is stored in row-major on the GPU, and each thread swaps an element of a row in parallel. On the other hand, the matrix is stored in column-major on the CPU to call **xGETRF**. This requires us to transpose the matrix every time the matrix is copied between the GPU and CPU, but the overall performance is improved [7]. Furthermore, the application of pivoting is overlapped with the copying of the panel to the GPU. Finally, to overlap the copying of the panel to the CPU and its factorization with the submatrix update, a look-ahead of depth one is implemented, i.e., when GPU is updating the trailing submatrix using the $j$-th panel, the $(j+1)$-th panel is asynchronously sent to the CPU as soon as it is updated with the $j$-th panel. Hence, the CPU does not wait for the completion of the remaining trailing submatrix update before starting the $(j+1)$-th panel factorization.

**magma_xGEQRF** computes the QR factorization (3) of $A$ as in Figure 1(b), which has the same algorithmic flow as that of **magma_xGETRF** in Figure 3(a). In comparison to **magma_xGETRF**, **magma_xGEQRF** copies the additional $nb$ elements of $T_j$ from the CPU to the GPU, which is used to apply the transformation (5) on the GPU (step 2.f). On the other hand, once the panel is factorized on the CPU (step 2.c), it is not updated by the later steps. Hence, at each step, the algorithm copies the whole panel to the CPU (steps 2.a and 2.b), and avoids the final copying of the computed factors, which is needed by **magma_xGETRF** (step 3 of Algorithm 3.1). To overlap this copying of $R$ to CPU with the proceeding steps, a designated stream is used, which is synchronized only at the end (step 3). The total of $3mn - (n^2 - 2nb)$ matrix elements are copied between the CPU and GPU by **magma_xGEQRF**.

*Cholesky factorization.* Figure 1(c) shows the pesudocode of **magma_xPOTRF** that computes the Cholesky factor $R$ of $A$. The algorithm first copies the lower-triangular part of $A$ from the CPU to the GPU (step 1 of Algorithm 3.2). Then, at the beginning of the $j$-th step, the $j$-th diagonal block $a_{jj}$ is updated on the GPU (step 2.a of Algorithm 3.2) and copied to the CPU for the factorization (step 2.b). While $a_{j,j}$ is being copied to the CPU, the GPU updates the off-diagonal blocks $a_{(j+1):n_b,j}$ (step 2.c). When the CPU receives $a_{j,j}$, it computes its Cholesky factor $r_{j,j}$ (step 2.d) and copies it back to the GPU (step 2.e). Finally, $r_{j,j}$ is used to compute $r_{(j+1):n_b,r}$ on the GPU (step 2.g). To avoid explicit synchronization, $r_{(j+1):n_b,r}$ is computed on the same GPU stream as that is used to send $r_{j,j}$. Furthermore, similarly to **magma_xGEQRF**, to hide the time required to copy the computed $R$ to the CPU, at the $j$-th step, **magma_xPOTRF**

**Algorithm 4.1**
`for` $J = 1, 2, \ldots, n_B$ `do`
1. copy $A_{:,J}$ from CPU to GPU, and transpose it.
2. update $A_{:,J}$ with previous columns (left-looking).
   a. `for` $k = 0, 1, \ldots, (J-1)B_b$ `do`
   b. apply pivoting $P_k$ to $A_{:,J}$.
   c. copy previous columns $\ell_{:,k}$ and $u_{:,k}$ from CPU to GPU.
   d. use **magma_xTRSM** and **_xGEMM** to update $A_{:,J}$ with $\ell_{:,k}$ and $u_{:,k}$ on GPU.
   e. `end for`
3. use **magma_xGETRF_gpu** to factorize $A_{:,J}$; i.e., $P_J A_{:,J} = L_{:,J} U_{:,J}$ (right-looking).
4. copy the LU factors of $A_{:,J}$ from GPU to CPU.

`end for`

(a) LU factorization.

**Algorithm 4.2**
`for` $J = 1, 2, \ldots, n_B$ `do`
1. copy $A_{J:n_B,J}$ from CPU to GPU.
2. update $A_{J:n_B,J}$ with previous columns (left-looking).
   a. `for` $k = 0, 1, \ldots, (J-1)B_b$ `do`
   b. copy previous columns $r_{(J-1)B_b:n_b,k}$ from CPU to GPU.
   c. a update $A_{:,J}$ with $r_{(J-1)B_b:n_b,k}$ on GPU.
      `for` $\ell = 0, 1, \ldots B_b - 1$
         use **magma_xSYRK** to update $a^{(J)}_{\ell,\ell}$ and **_xGEMM** to update $a^{(J)}_{\ell+1:n_b,\ell}$
      `end for`
   d. `end for`
3. use **magma_xPOTRF_gpu** to factorize $A_{:,J}$; i.e., $A_{J:n_B,J} = R_{J:n_B,1:J} R^H_{J,J}$ (left-looking).
4. copy $A_{:,J}$ from GPU to CPU.

`end for`

(b) Cholesky factorization.

Figure 2: non-GPU-resident factorization algorithms.

sends the already-computed off-diagonal blocks $r_{j,1:(j-1)}$ of $R$ to the CPU through a designated stream (step 2.b). Only when the whole $A$ is factorized, we synchronize the stream (step 3).

The $j$-th step of **magma_xPOTRF** updates only the $j$-th panel and has much less parallelism to be exploited on the GPU than that of **magma_xGETRF**. However, the CPU is used only to factorize the diagonal block, and the ratio of the computation performed on the GPU over that on the CPU is greater in **magma_xPOTRF**. Even though **magma_xGETRF** overlaps the panel factorization with the submatrix update, in some cases, the GPU is idle waiting for the completion of the panel factorization on the CPU. As a result, the panel factorization often becomes the bottleneck. This is especially true when the ratio of the computation required by the CPU over that required by the GPU increases at a later step of the factorization and with more GPUs used to update the submatrix (see Section 5), Hence, **magma_xPOTRF** may obtain greater performance than **magma_xGETRF** on a large matrix.

MAGMA performs most of operations on the CPU and GPU through LAPACK and MAGMA BLAS [8] that are optimized for a specific multi-core CPU and GPU architectures, respectively. However, the whole matrix $A$ must be stored on the GPU, and the size of $A$ that can be factorized by MAGMA is limited by the amount of the memory available on the GPU. To overcome this limitation, in the next two sections, we respectively describe our non-GPU-resident and multi-GPU factorization algorithms that factorize a submatrix of $A$ at a time and use multiple GPUs attached to a multicore.

## 4. Non-GPU-resident factorization algorithms

In this section, we describe our non-GPU-resident factorization algorithms which factorize a part of a matrix $A$ on the GPU at a time and enable the factorization of $A$ that is too large to fit in a GPU's memory at once. The $J$-th step of the algorithm consists of the following four stages: 1) copy from the CPU to the GPU the $m$-by-$B$ submatrix $A_{:,J}$ consisting of the $((J-1)B+1)$-th through the $(jB)$-th column of $A$, 2) update $A_{:,J}$ with the previously-factorized columns of $A$, 3) use the standard MAGMA algorithms in Section 3 to factorize $A_{:,J}$, and 4) copy $A_{:,J}$ back to the CPU. In our current implementation, the algorithms dynamically select $B$ as the maximum number of columns of $A$ that can fit in the GPU's memory, but $B$ can be a user-specified parameter.

*LU and QR factorizations.* The new MAGMA routine **magma_xGETRF_ngr** implements a non-GPU-resident LU factorization algorithm. The pseudocode of the algorithm is shown in Figure 2(a), where $n_B$ is the number of submatrices in $A$ and $B_b$ is the number of block columns in the submatrix (i.e., $n_B = \frac{n}{B}$ and $B_b = \frac{B}{b}$).[3] To copy and transpose

[3] Our analysis assumes $n$ is a multiple of $B$. If $n$ is not a multiple of $B$, then in our implementation, the first $n_B - 1$ submatrices contain $B$ columns, while the last submatrix contains $n - (n_B - 1)B$ columns.

$A_{:,J}$ on the GPU (step 1), two GPU streams and two buffers of size $m$-by-$b$ are alternately used such that transposing the first block column on the GPU is overlapped with copying the second block column to the GPU. In comparison to copying the entire $A_{:,J}$ to the GPU and then transposing it, our incremental copy-and-transpose algorithm reduces the size of the buffer, allowing a larger value of $B$, and is shown to be more efficient. Then, the current submatrices are updated using the previous submatrices (step 2). Finally, the routine **magma_xGETRF_gpu** takes $A_{J:n_B,J}$, which is stored in row-major on the GPU, and computes its LU factors using the algorithm in Section 3 (step 3).

Algorithm 4.1 does not apply the pivoting $P_J$ to the previously-computed submatrices $L_{:,1}, L_{:,2}, \ldots, L_{:,J-1}$ of $L$. There are two approaches to apply the pivoting. The first approach applies $P_J$ to the previous submatrices on the CPU. This computes the LU factors which are equivalent to the ones computed by LAPACK, and the LAPACK routine **xGETRS** can be used for the forward and backward substitutions. The second approach applies the pivoting to the right-hand-sides (RHSs) during the forward substitution. This is the approach used in software packages like LINPACK [9] and PLASMA [10]. When the number of RHSs is relatively small, the second approach may be more efficient than the first approach. However, LAPACK does not provide a routine that alternately applies the pivoting and substitution to RHSs, and a new routine must be included in MAGMA. Finally, as discussed in Section 2, if the whole panel is copied to the CPU at each step of **magma_xGETRF_gpu** (step 2.f of Algorithm 3.1), then the final copying of LU factors to the CPU (step 5 of Algorithm 4.1) can be avoided. However, the pivoting for each block column of $L_{:,J}$ must be applied to the previously-computed columns of $L_{:,J}$ on the CPU. At the $j$-th step of **magma_xGETRF_ngr**, the application of the $j$-th pivoting to the previous columns of $L_{:,J}$ on the CPU might be overlapped with the trailing submatrix update on the GPU. However, as $j$ increases, the pivoting must be applied to more blocks in the $j$-th block row of $L_{:,J}$, while the computation required for the submatrix update reduces. Hence, especially when multiple GPUs are used to update the submatrix, it becomes difficult to hide the time to apply the pivoting on the CPU.

The total number of matrix elements copied between the CPU and GPU by Algorithm 4.1 is

$$4mn - (n^2 - nb) + \sum_{J=1}^{n_B}(J-1)mB = \left(4 + \frac{n_B - 1}{2}\right)mn - n^2 + nb. \tag{7}$$

If the final copying of the LU factors to the CPU is avoided, then the number of copied elements is reduced by $\frac{n^2+nb}{2}$. Note that the number of submatrices, $n_B$, appears in (7), but the block size $B$ does not. Hence, the total communication volume is minimized by setting $B$ to be the maximum number of columns of $A$, which can fit in the GPU's memory.[4]

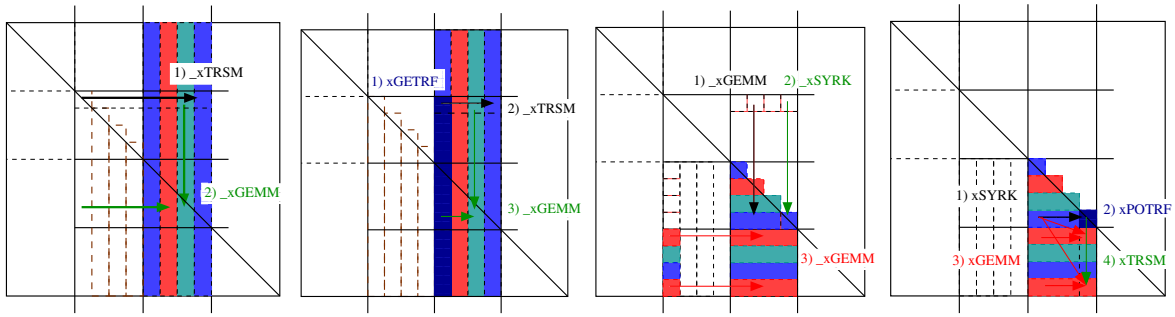
The routine **magma_xGEQRF_ngr** implements our non-GPU resident QR factorization algorithm. Since both **magma_xGEQRF_ngr** and **magma_xGETRF_ngr** are right-looking algorithms, **magma_xGEQRF_ngr** follows the same algorithmic flow as Algorithm 4.1. In comparison to **magma_xGETRF_ngr**, **magma_xGEQRF_ngr** copies from the CPU to the GPU extra matrix elements of triangular factors $T_j$ of the previous submatrices, which are then used to update the current submatrix. Hence, the total number of matrix elements copied between the CPU and GPU by **magma_xGEQRF_ngr** is given by

$$3mn - (n^2 - 2n_b b^2) + \sum_{J=1}^{n_B}(J-1)(m+b)B = \left(3 + \frac{n_B - 1}{2}\right)mn - n^2 + \frac{n_B + 3}{2}nb.$$

In order to reduce the memory requirement, the triangular factors $T_j$ of the previous block reflectors are recomputed on the CPU at each $J$-th step of the algorithm.

*Cholesky factorization.* Figure 2(b) shows the pseudocode of our non-GPU-resident algorithm that computes the Cholesky factor $R$ and is implemented in the routine **magma_xPOTRF_ngr**. In the pseudocode, to update only the lower-triangular part of $A$, each block-column of $A_{:,J}$ is updated by a previously-computed block column of $L$ at a time (step 2.c), where $a_{i,j}^{(J)}$ is the $(i,j)$-th block of $A_{J:n_B,J}$ (i.e., $a_{i,j}^{(J)} = a_{(J-1)B_b+i,(J-1)B_b+j}$). Then, the routine **magma_xPOTRF_gpu** uses the algorithm in Section 3 to compute the Cholesky factor of the submatrix $A_{J:n_B,J}$ which is already on the GPU.

[4] If $n$ is not a multiple of $B$, then in our implementation, the first $n_B - 1$ submatrices have $B$ columns, and the last submatrix contains the remaining $n - (n_B - 1)B$ columns. Hence, the formula (7) can be easily extended to the case where $n$ is not a multiple of $B$. However, the total communication volume between the CPU and GPU may be reduced if the first submatrix contains these remaining columns instead of the last submatrix.




(a) LU factorization. (b) Cholesky factorization.

Figure 3: multi-GPU factorization in non-GPU-resident algorithm. For each figure (a) or (b), the left figure shows the updates with the previous columns, and the right figure shows the factorization of the submatrix. MAGMA routines are identified by the underscore, _, in front of their names.

At step 1 of Algorithm 4.2, a single stream is used to asynchronously send the lower-triangular part of $A_{:,J}$ to the GPU one block-column at a time. After all the columns are sent, the stream is synchronized before the submatrix update with the previous columns (step 2.c). The total number of matrix elements copied between the CPU and GPU is given by

$$\begin{aligned} n^2 + 3nb + \sum_{J=0}^{n_B-1} \sum_{j=0}^{JB_b-1} b(n - jb) &= n^2 + 3nb + \left(nB + \frac{Bb}{2}\right) \sum_{J=0}^{n_B-1} J - \frac{B^2}{2} \sum_{J=0}^{n_B-1} J^2 \\ &= \left(\frac{n_B}{2} + \frac{1}{3}\right) n^2 + \left(\frac{B}{12} + \frac{11b}{4}\right) n. \end{aligned}$$

At the $J$-th step, only the $((J-1)B_b)$-th through the $n_b$-th block rows of $L_{:,1:(J-1)}$ are copied to the GPU.

## 5. Multiple-GPU factorization algorithms

In this section, we describe our multi-GPU factorization algorithms that take advantage of multiple GPUs attached to a multicore. In these algorithms, as before, the panel is factorized on the CPU, but the submatrix is now updated using multiple GPUs. For the discussion, here, we use $\bar{A}$ to denote the matrix that is factorized by our multi-GPU algorithm. These multi-GPU algorithms are used to factorize the submatrix $A_{J:n_B,J}$ in the non-GPU-resident algorithms in Section 4; i.e., $\bar{A} = A_{J:n_B,J}$.

*LU and QR factorizations.* **magma_xGETRF_mgpu** implements our multi-GPU LU factorization algorithm that distributes the matrix $\bar{A}$ in a column-wise 1D block-cyclic layout. Figure 3(a) illustrates the algorithm. At the $j$-th step, the GPU owning the $(j+1)$-th panel performs the look-ahead and asynchronously sends the next panel to the CPU for the factorization (steps 2.e and 2.f of Algorithm 3.1). After the panel factorization is completed on the CPU, the panel is asynchronously sent to all the GPUs (steps 2.b and 2.c). Then, each GPU applies the pivoting and updates their local trailing submatrix independently from each other (steps 2.d, 2.e and 2.f). Only the GPU owning the $(j+1)$-th block column of $\bar{A}$ can store the panel in its local storage of $\bar{A}$, and the other GPUs store the panel in their local buffers. We alternately use two buffers to store the panels on each GPU so that copying of the next panel can be started before the completion of the submatrix update with the current panel. We also use GPU streams to maximize the parallelism and to overlap communications and computations. For instance, during the look-ahead, one stream is used to update the next block column, and another is used to update the remaining trailing submatrix.

To update the current submatrix with the previous submatrices in our non-GPU-resident LU algorithm (step 3 of Algorithm 4.1), each block column of the previous submatrices is first sent to all the GPUs. Then, the GPUs apply the pivoting and perform the update with the block column to their local submatrices in parallel. The copying of previous block to the GPU is overlapped with the application of the pivoting. As before, our multi-GPU QR factorization algorithm follows the same algorithmic flow as our multi-GPU LU factorization algorithm.

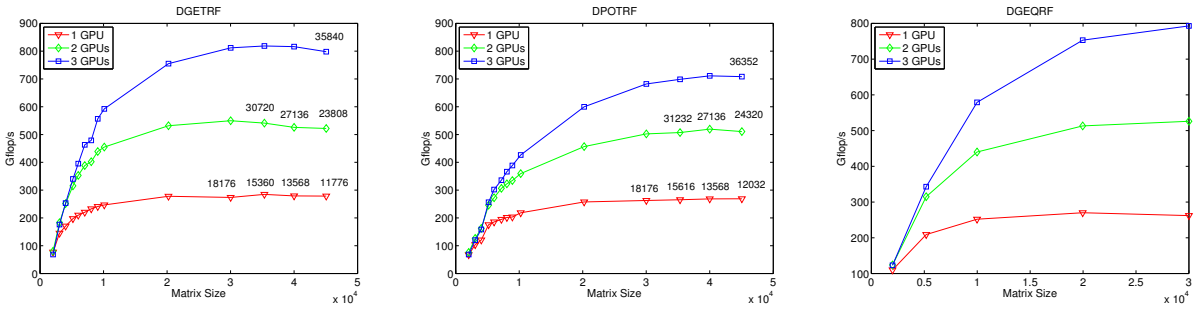



Figure 4: Performance of non-GPU resident multi-GPU factorization algorithms.

*Cholesky factorization.* Since only one block-column is updated and factorized at each step of the right-looking Cholesky factorization, our multi-GPU Cholesky algorithm distributes the matrix $\bar{A}$ among GPUs in a row-wise 1D block-cyclic layout (see Figure 3(b)). At the beginning of the $j$-th step, the GPU owning the $j$-th diagonal block updates the diagonal block (step 2.a of Algorithm 3.2). Since all the blocks required for the diagonal update (i.e., those in the $j$-th block row) are on this GPU, this step does not require any communication. Then, the updated diagonal block is sent to the CPU for the factorization (steps 2.b and then 2.d). After the completion of the factorization, the resulting Cholesky factor is asynchronously sent to all the GPUs (step 2.e).

Since the off-diagonal blocks $r_{(j+1):n_j,j}$ of the $j$-th block column is distributed among multiple GPUs, each GPU requires the $j$-th block row $r_{j,1:(j-1)}$ to update $r_{(j+1):n_j,j}$ (step 2.c). Hence, when the block $r_{j,j-1}$ is computed at the end of the $(j-1)$-th step (step 2.f), the block row $r_{j,1:(j-1)}$ is sent to the CPU. Then, at the beginning of the $j$-th step, the CPU waits for this $j$-th block row and broadcasts to all the GPUs (except to the one owning the $j$-th row). This copying of the block row to the GPUs is overlapped with the diagonal updates on the GPU. Furthermore, updating of the off-diagonal block is overlapped with the copying of the diagonal block to the CPU, panel factorization, and copying of the Cholesky factor back to the GPUs. Once a GPU receives the Cholesky factor, it independently computes the off-diagonal blocks of $r_{:,j}$ (step 2.f).

To use multiple GPUs in our non-GPU-resident Cholesky algorithm, each block column of the diagonal submatrix $a^{(J)}_{\ell,\ell}$ is updated at a time using **magma_xSYRK** on the diagonal block and **magma_xGEMM** on the off-diagonal block (step 2.c of Algorithm 2.c). Once the diagonal submatrix is updated, the remaining part of $\bar{A}$ can be updated with a single call to **magma_xGEMM**.

## 6. Performance results

Here, we show the performance of the proposed algorithms on a compute node of the Keeneland system [11]. For our experiments, we used all of the twelve Intel Xeon 5660 processors and three Tesla M2070 GPUs available on a single compute node. Even though each GPU has only 6GB of memory, our non-GPU-resident algorithms enable the factorization of the matrix that does not fit in the aggregated GPUs' memory. However, total page-cache available on a CPU, which can be allocated using CudaHostAlloc, is 18GB, and the largest matrix that can be stored in the cache-page had the dimension of about 4500. Figure 4 shows the Gflop/s obtained by our algorithms in double precision, where the user has provided the matrix $A$ stored on the CPU and allocated the memory to store $A$ on the GPUs. The numbers above markers are the numbers of the columns in the submatrices, $n_B$, used by our non-GPU-resident algorithms (i.e., for the markers without the numbers, the whole matrix fit in the GPUs' memory). We see that in comparison to using one GPU (equivalent to the released version of MAGMA), for DGETRF, DPOTRF, and DGEQRF, our multi-GPU algorithms respectively obtained the speedups of up to 2.0, 1.9, and 2.0 using two GPUs, and the speedups of up to 2.9, 2.6, and 2.9 using three GPUs. Furthermore, our non-GPU-resident algorithms maintained the high-performance of our multi-GPU algorithms even though they had to reload the previous submatrices to GPUs to factor each submatrix.

Finally, Figure 5 shows the trace of our multi-GPU LU factorization. The pink trace reprsents the panel factorization on the CPU (step 2.a of Algorithm 3.1). The remaining three pairs of traces represent the submatrix updates

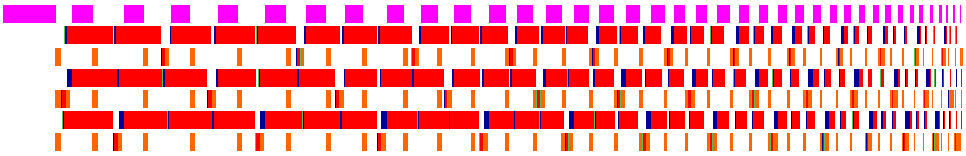


Figure 5: Trace of multi-GPU LU algorithm ($n = 10,000$).

using two streams on each of the three GPUs (one for look-aheads and the other for remaining submatrix updates). Green, blue, and red traces represent steps 2.c, 2.d, and 2.e of Algorithm 3.1, respectively, while the orange trace is the copying of the panels between the CPU and GPUs. Using two streams, copying the panel was overlapped with the remaining submatrix updates. Furthermore, initially, the panel factorization was completely hidden behind the submatrix updates, while at a later step, it was more expensive than the submatrix update, and the GPUs had to wait for the completion of the panel factorization. For a larger matrix, the panel factorization can be hidden behind the submatrix updates for a greater number of steps.

## 7. Conclusion

We described a non-GPU-resident and multi-GPU extensions of one-sided factorization algorithms of MAGMA. Our performance results on the Keeneland system have shown that these algorithms can factorizes a matrix which does not fit in the aggregated memory of multiple GPUs, and provide the potential of fully utilizing the computing power of a compute node. We are currently studying several techniques to further optimize the performance of the algorithms. For instance, in many cases, especially when multiple GPUs are used, panel factorization on the CPU can be the bottleneck. To overcome this, we are investigating the integration of panel factorization algorithms (e.g., [12, 13]), which are more scalable than that of LAPACK on a multicore. Other optimization techniques include look-ahead of depth greater than one on the CPU to reduce the idling time of the CPU, and usage of GPU streams to eliminate explicit synchronizations. We are also extending two-sided factorization algorithms of MAGMA to use multiple GPUs.

## References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' guide, 3rd Edition, Society for Industrial and Applied Mathematics, 1999.

[2] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Soft. 5 (1979) 308–323.

[3] S. Tomov, R. Nath, P. Du, J. Dongarra, MAGMA version Users' guide, available at `http://icl.eecs.utk.edu/magma/` (2009).

[4] M. Baboulin, J. Dongarra, S. Tomov, Some issues in dense linear algebra for multicore and special purpose architectures, Tech. Rep. UT-CS-08-200, University of Tennessee (2008).

[5] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Orti, Solving dense linear systems on graphics processors, in: Euro-Par 2008 Parallel Processing, Vol. 5168 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 739–748.

[6] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, Dense linear algebra solvers for multicore with GPU accelerators, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.

[7] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, S. Tomov, LU factorization for accelerator-based systems, in: 9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11), 2011.

[8] R. Nath, S. Tomov, J. Dongarra, An improved magma gemm for fermi graphics processing units, Int. J. High Perform. Comput. Appl. 24 (2010) 511–515.

[9] J. Dongarra, J. Bunch, C. Moler, G. Stewart, LINPACK Users' Guide, Society for Industrial and Applied Mathematics, 1979.

[10] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, PLASMA version Users' guide, available at `http://icl.eecs.utk.edu/plasma/`.

[11] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, S. Yalamanchili, Keeneland: Bringing heterogeneous gpu computing to the computational science community, IEEE Computing in Science and Engineering 13 (2011) 90–5, available also at `http://dx.doi.org/10.1109/MCSE.2011.83`.

[12] M. Horton, S. Tomov, J. Dongarra, A class of hybrid lapack algorithms for multicore and gpu architectures, in: Proceedings of Symposium for Application Accelerators in High Performance Computing (SAAHPC), 2011.

[13] J. Dongarra, M. Faverge, H. Ltaief, P. Luszczek, Achieving numerical accuracy and high performance using recursive tile LU factorization, Tech. rep., Innovative Computing Laboratory, University of Tennessee (2011).