Providing GPU Capability to LU and QR within the ScaLAPACK Framework

Peng Du*, Stanimire Tomov*, and Jack Dongarra*†‡
*University of Tennessee Innovative Computing Laboratory
†Oak Ridge National Laboratory
‡University of Manchester

***Abstract*—In the field of dense linear matrix computations on distributed memory systems, ScaLAPACK has established its importance over the years with its high performance and scalability. Since the introduction of CUDA based GPGPU computing in 2008, methods to efficiently use such computing power on distributed memory systems equipped with multicore CPUs, have attracted much attention. In this work we integrate the CUDA computing directly into the ScaLAPACK framework and demonstrate that good speedup could be achieved on routines like LU and QR by carefully managing the GPU-CPU data transfers. The objective is to eventually convert most of the ScaLAPACK routines to support GPU computing so that current application codes that already utilize ScaLAPACK get a "free" (automatic) speedup when GPUs are present.**

I. Introduction

Dense linear algebra computations such as the LU and QR factorizations are widely used in scientific computation. For instance, dense linear system of equations are normally solved by the LU factorization. The AORSA fusion energy simulation program [6] and the High Performance LINPACK (HPL) benchmark [12] are such examples. The QR factorization can be used to solve the linear least squares and eigenvalue problems.

In order to get better performance, ScaLAPACK [13] was introduced in 1997 for distributed memory cluster systems. By using block algorithms that are normally seen in BLAS/LAPACK [2], high computing performance is achieved by casting computations to level 3 BLAS operation like matrix-matrix multiplication [4]. Also, the two-dimensional block-cyclic distribution allows the performance to scale well on large scale systems. ScaLAPACK uses a modular design which is built on top of HPC packages such as BLAS/LAPACK. On multicore machines, high performance implementations such as Intel's Math Kernel Library (MKL) can be used. For inter-node communication, BLACS [11] is developed which packs MPI communications into matrix oriented interface. And on top of BLACS, similar to to the way in which LAPACK uses BLAS, ScaLAPACK relies on the PBLAS [10] library for basic matrix operations. Till today, ScaLAPACK is still under active development.

In 2008, NVIDIA introduced the Compute Unified Device Architecture (CUDA) [20]. With CUDA, programming the graphics processing unit (GPU) became much less painful and CUDA gained the attention of the HPC community immediately. Powered by CUDA, the NVIDIA's GPU unleashes the significant computing "horsepower" which used to be exclusively enjoyed by graphics related application like gaming and video rendering. In 2009, the Fermi architecture was introduced which further improved the GPU's capability in both performance and dependability [19]. Many cluster system, including large scale system such as the Kraken Supercomputer at the Oak Ridge National Lab, are being equipped with the GPUs. In 2010, the "Tianhe-1A" supercomputer from China won the first place on the TOP500 [17] achieving a performance level of 2.57 petaflop/s with the help of 7,168 NVIDIA Tesla M2050 general purpose GPUs running the HPL benchmark.

Despite the great rate of integration, the matching development in software has not been as fast. Several methods have been proposed to port certain routines to use the GPU on cluster systems, but a good way to provide as many of routines as those supported by ScaLAPACK still has not come to shape. In this work, by integrating the GPU support directly into the framework of ScaLAPACK, we attempt to convert the majority of ScaLAPACK routines to support GPUs so that ScaLAPACK users can benefit from the speedup on cluster systems automatically and instantly when GPUs become available. Minimizing the amount of code changes is also a design requirement for this work. Further, the routines' interfaces are not changed so that all the users must do is to re-compile their code and link with the appropriate CUDA libraries. LU and QR are selected as the testbed because they represent two typical categories of computational routines in ScaLAPACK. GPU's global memory can be viewed as another level of the memory hierarchy of a system, and because of the relatively low CPU-GPU bandwidth, it is important that the CPU-GPU data transfers be minimized or hidden. The QR factorization, similar to the Cholesky factorization, has a predictable memory access pattern while the pivoting in LU requires more "random" inter-process communication that is dictated by the input matrix, leading to more difficulty to amortize the data transfer latency.

The rest of the paper is organized as follows: Section II discusses the related work in this field. Section III introduces the ScaLAPACK framework. Section IV describes the integration of the GPU support in the ScaLAPACK. Performance experiment results are shown in section V, and section VI concludes the work.

II. Related Work

Dense linear algebra routines for single-GPU have been developed with great performance speedups, for example [28], [27], [1], [18] focus on shared memory machines with both multicore CPU and GPUs. In [25], by treating a heterogeneous system as a distributed-memory machine, hybrid methods are proposed that can utilize all CPU cores and all GPUs on heterogeneous multicore and multi-GPU systems. A new runtime system has also been designed for the Cholesky and QR factorization, which has shown great scalability and performance on either single node or clusters.

For distributed memory platforms, the HPL banchmark was among the first targets converted to use GPUs [5], [21], [15]. [29] describes the efforts that turned HPL for the Tianhe-1A Petascale Supercomputer. Specifically, a software pipelining technique is proposed to hide the communication overhead caused by the low-bandwidth between the CPU and GPU communication. GPU is also considered as an energy efficient alternative to using multicore CPUs [22]. HPL is the linear system solver based on the right-looking version of the LU factorization. Special techniques like lookahead are used to reduce the performance impact from the slow panel factorization by overlapping it with the trailing matrix update. Lookahead is not implemented in the ScaLAPACK LU but in our work we adopt its hybridization version. In [14], three methods of designing parallel LU factorization on cluster systems are discussed, including a “thunking” approach that links ScaLAPACK LU with CUBLAS using a software emulation layer; This method exhibits worse performance than the existing ScaLAPACK LU using only CPUs because of the large data transfer latency overhead between the CPU and GPU. Even replacing CUBLAS with MAGMA BLAS did not help the GPU version to win out. An out-of-core version of LU is also discussed to solve large problems that can not fit into the GPU’s global memory. This method requires major change to the code, although great speedup was obtained compared to the CPU version. With a start-from-scratch but highly scalable runtime system, [24] proposed a tile-based design to run dense linear algebra algorithms on GPU-based heterogeneous clusters efficiently. Hybrid-size tasks are generated by running the serial version of the algorithm, which are then distributed through the runtime system powered by a novel distributed task-assignment protocol. This method, showing unprecedented performance on cluster systems with the GPUs, demands an inside-out redesign of both the mathematical algorithm and software implementation, which has yet shown wide extendability to other routines in the field.

In our work, we attempt to make a compromise by sacrificing some of the performance speedup that comes from using GPUs for better extendability, which will allow large quantity of dense linear algebra routines to benefit from using GPUs on cluster systems with much less pain in code transitioning.

III. The ScaLAPACK Framework

ScaLAPACK is designed to work on cluster systems, which nowadays normally comprise of computing nodes that have multi-core CPUs connected by high-speed interconnect such as InfiniBand, and high computing performance relies on the usage of fast single-node dense linear algebra library, efficient inter-node communication algorithm, and well-balanced computing load. To achieve high performance on each node, fast BLAS implementation such as MKL is usually used. The CPU cores on each node are utilized by spawning threads on each core in a fork-join fashion[8]. The inter-node communication performance is at the mercy of MPI implementation. For the sake of load balancing, two-dimensional block-cyclic distribution is adopted.

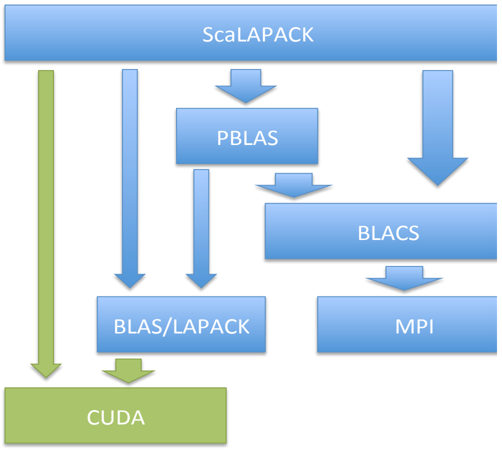


Fig. 1: The ScaLAPACK calling stack

ScaLAPACK is built on top of several software modules, and maintains a close code appearance to the LAPACK counterparts. Fig. 1 shows the software stack of ScaLAPACK. The blue boxes represent the original software packages that run on CPUs only. Because the majority of the computation takes place in BLAS/LAPACK routines on each node, in this work some of these routines are replaced by their counterparts in CUDA directly to run on the GPU, which introduces another level of memory space in addition to the CPU memory space. Since data transfer to and from the GPU takes non-negligible time, careful design is required to remove or hide this overhead within the ScaLAPACK framework. This section discusses the two-dimensional data distribution and the block algorithms for LU and QR that are implemented with in ScaLAPACK. The GPU support based on such framework will be introduced in section IV.

A. Two-Dimensional Block-cyclic Distribution

Data layout plays an important role in the performance of parallel matrix operations on distributed memory systems [9], [16]. In 2D block-cyclic distributions, data is divided into equally sized blocks, and all computing units are organized into a virtual two-dimensional grid of size P by Q. Data blocks are distributed to computing units in round robin following the two dimensions of the virtual grid. This layout helps with load balancing and reduces data communication frequency

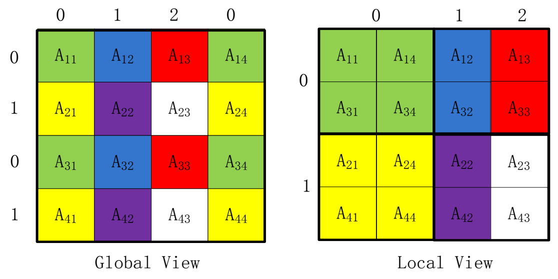


Fig. 2: Example of 2D block-cyclic data distribution

since in each step of the algorithm as many computing units can be engaged in computations and most of the time, each computing unit only works on its local data. Fig. 2 is an example of $P = 2, Q = 3$ and a global matrix of 4×4 blocks. The same color represents the same process and numbering in A_{ij} indicates the location in the global matrix. ScaLAPACK assumes input data is already in 2D block-cyclic distribution. Mapping between local blocks and their global locations can be found in [13].

B. Block Algorithm for LU and QR

On top of the 2D block-cyclic data distribution, ScaLAPACK also implements dense linear algebra routines using the block algorithm. The block algorithm casts more computation into level 3 BLAS operations such as matrix-matrix multiplications. These operations have high data/cache reuse due to their $O(N^3)$ floating point operation counts versus $O(N^2)$ data counts, and hence a high performance.

1) LU Factorization: For a dense matrix A, the LU factorization of A produces $PA = LU$, where P is a pivoting matrix, L is a unit lower triangular matrix, and U is upper triangular matrix. The LU factorization is popular for solving systems of linear equations. Having the L and U factors, the linear system $Ax = b$ is solved by first solving $Ly = b$ and then $Ux = y$. ScaLAPACK implements the right-looking version of LU with partial pivoting based on a block algorithm and 2D block-cyclic data distribution.

We split A an $N \times N$ into 2×2 blocks. A_{11} has size $NB \times NB$, A_{12} is $NB \times (N - NB)$, A_{21} is $(N - NB) \times NB$, and A_{22} is $(N - NB) \times (N - NB)$, where NB is known as blocking size and A_{22} as the "trailing matrix". Decompose A as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & L_{22} \end{bmatrix}$$

and therefore

$$\begin{cases} \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11} & \rightarrow PDGETF2 \\ A_{12} = L_{11}U_{12} & \rightarrow PDTRSM \\ L_{22}U_{22} = A_{22} - L_{21}U_{12} & \rightarrow PDGEMM \end{cases} \quad (1)$$

PDGETF2, PDTRSM, and PDGEMM are the names of the ScaLAPACK routines that perform the corresponding operations on the left. This poses as one iteration (step) of the factorization, and pivoting is applied on the left and right of the current panel. The routines names in the ScaLAPACK LU are listed after "$\rightarrow$".

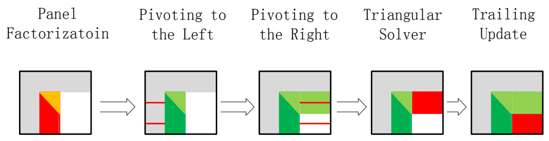


Fig. 3: LU factorization diagram; Green: Just finished; Red & Orange: being processed; Gray: Finished in previous steps

Fig. 3 is a diagram of the components of LU. QR factorization has a similar recursive execution process.

2) QR Factorization: The QR factorization decomposes a matrix A into a product $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. QR factorization is often used to solve the linear least squares problem, and also in the QR algorithm – an eigenvalue algorithm to calculate the eigenvalues and eigenvectors of a matrix.

Several methods exist for computing the QR factorization, such as the Gram-Schmidt process, Householder transformations, and Givens rotations. In today's high performance math libraries, for instance, LAPACK [3], ScaLAPACK [9], and MAGMA, a block version of the Householder transformations is adopted to achieve high performance. In particular, given an input matrix A, a Householder matrix Q_1 is multiplied to A such that

$$Q_1 A = \begin{bmatrix} r_{11} & r_{12} \cdots r_{1n} \\ 0 & \\ \vdots & A' \\ 0 & \end{bmatrix}$$

This zeros out the elements under the diagonal in the first column. The next step is carried out on the trailing matrix A' with

$$Q_2{}' = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{bmatrix}$$

ScaLAPACK uses a block version of the QR factorization by accumulating a few steps of the Householder matrix. This method is rich in level 3 BLAS operations and therefore can achieve high performance. Q is stored the lower triangular matrix below the diagonal of the input matrix in the form of a WY representation of the Householder transformation products[23], [7].

ScaLAPACK implements the block QR factorization as follow. At step i , an $m \times m$ submatrix A_i is partitioned and factorized as

$$A_i = \begin{bmatrix} A_1 & A_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = Q \times \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

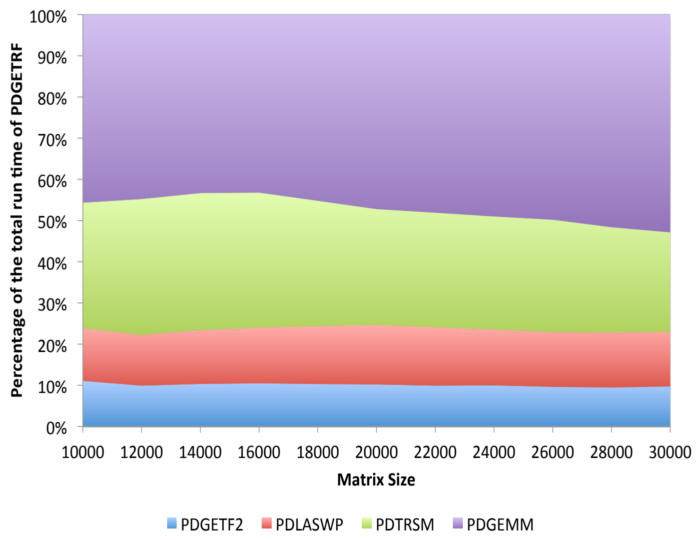


Fig. 4: The run time breakdown of PDGETRF
2×3 MPI process grid, 1 process (8 MKL threads)/node

Here A_{11} is of size $NB \times NB$, where NB is the block size. A_{21} is of size $(m-NB) \times NB$. $A_1 = [A_{11}, A_{12}]^T$ constitutes the area for the panel QR factorization. Since ScaLAPACK uses the Householder method, Q is expressed as a series of Householder transformations in the form $H_i = I - \tau_i v_i v_i^T, i = 1 \cdots NB$. v_i has 0 for the first $i-1$ entries, 1 on the $i-th$ entry and $\tau_i = 2/v_i^T v_i$. In ScaLAPACK, v_i is stored below the diagonal of A and when Q is applied to the trailing matrix $A_2 = [A_{21}, A_{22}]^T$, Q is computed by $Q = H_1 \cdots H_{NB} = I - VTV^T$, where T is an upper triangular matrix of size $NB \times NB$, and V has v_i as its $i-th$ column. With this expression, the trailing matrix update becomes

$$\tilde{A}_2 = \begin{bmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{bmatrix} = Q^T A_2 = (I - VT^T V^T) A_2. \tag{2}$$

This finishes one iteration of the block QR factorization. This process is repeated from $\tilde{A}_{22}$ until the whole matrix is factorized.

IV. Shifting Computing Load to the GPU

In this section, we discuss the procedure of adding GPU support within the ScaLAPACK framework.

The GPU memory is added to the cluster system as another level of memory space. All data that is to be operated needs to be transferred to the GPU global memory before computation, and the result needs to be transferred back to the CPU memory. The computation on the GPU takes place mostly in the GPU's shared memory and register space. However since these are similar to the space of the cache hierarchy of multicore CPU and we adopt the CUBLAS library from NVIDIA as the GPU implementation of BLAS, only the data transfer between CPU and the GPU global memory is considered as the level of extra memory space that requires optimization.

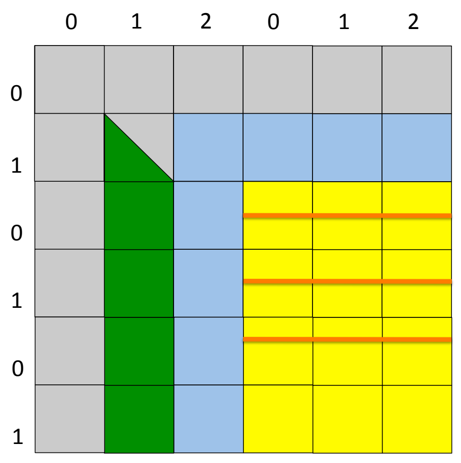


Fig. 5: The hybrid version of PDGETRF

A. LU Factorization

We start by understanding the run time of ScaLAPACK routines. Fig. 4 shows the breakdown of PDGETRF, obtained on the Dancer cluster (details in Section V). As shown in Section III, QR has a similar code structure and therefore the breakdown analysis for LU also applies to it. Both the panel factorization PDGETF2 and pivoting PDLASWP take roughly 10% of the run time while the triangular solver PDTRSM and the trailing update in the form of matrix-matrix multiplication PDGEMM occupy the rest. At large size, trailing matrix updates account for more than 50% of the computational time. Because PDTRSM and PDGEMM are both level 3 BLAS, they are more likely to benefit from using the CUBLAS library to run on the GPU. In this work we replace the PDGEMM, and PDTRSM will be converted in future work.

Since the panel factorization PDGETF2 has less parallelism than the rest of the routines, we choose to use the CPU to run it. This is similar to the hybridization methodology normally found in MAGMA routines [26]. To meet this design, the panel data is transferred from the GPU to the CPU during the trailing update step. Processes that own the blocks that are the next panel factorization split the trailing matrix blocks into two parts. The blocks making the next panel are directly transferred to the CPU where they are updated by the CPU version of DGEMM, and the rest of the trailing matrix blocks are updated on the GPU using cublasDgemm.

Fig. 3 shows that there are two pivoting procedures that swap rows in the LU factorization. The "left" pivoting accesses and modifies data on the left of the panel, i.e., the submatrix that has been already factorized, while the "right" pivoting touches matrix data in the PDTRSM's right hand side area and the trailing matrix of that factorization step. Because PDGETF2 is carried out on the CPU and the result is left on the CPU as well, the left pivoting does not require any change. The right pivoting, on the other hand, operates on the trailing matrix that still resides on the GPU, and therefore the trailing matrix data involved in the pivoting needs to be brought back to the CPU. In the hybrid version of LU in

MAGMA, this situation also exists but since all the data resides on the GPU memory and no inter-node communication is involved, pivoting can be performed directly on the GPU. In this work, no such guarantee exists, and it is not unusual that rows are frequently swapped between processes on different nodes, making it difficult to avoid the data transfer between the CPU and GPU memory.

To describe the GPU algorithm, suppose that the matrix to be factored is of size $M \times N$ and that the block size is NB. The matrix after factoring the first K columns using the right-looking LU can be represented as follows:

$$\left[\begin{array}{c||c|c} L_{11}\backslash U_{11} & \multicolumn{2}{c}{U_{12}} \\ \hline\hline L_{21} & A & \begin{array}{c} B \\ \hline C \end{array} \end{array}\right]$$

Here L_{11} and U_{11} are $K \times K$ blocks. U_{12}, L_{21} are $K \times (N-K)$ and $(M-K) \times K$ blocks, respectively. These four submatrices represent the blocks that have been finished by previous steps of the factorization. A is the next panel factorization area which is of size $(M-K) \times NB$. B is the right hand side area of the next PDTRSM (of size $NB \times (N-K-NB)$), and C is the next trailing matrix update area (of size $(M-K-NB) \times (N-K-NB)$).

Algorithm 1 is the GPU version of the ScaLAPACK LU proposed.

Algorithm 1 GPU resident ScaLAPACK LU

```
Transfer the input matrix to the GPU global memory;
First step (K=0) only transfers $C$;
for All the panels: do
    Copy $A$ and $B$, GPU $\rightarrow$ CPU, except the first step;
    CPU: PDGETF2 and PDLASWP (to Left);
    CPU: PDTRSM to $B$;
    Copy $\tilde{C}$, GPU $\rightarrow$ CPU;
    PDLASWP in $B$ and $C$;
    Copy $\tilde{C}$, CPU $\rightarrow$ GPU;
    if (My process owns blocks in the next panel) then
        Copy the next panel blocks that I own, GPU $\rightarrow$ CPU
        CPU: PDGEMM to the next panel blocks
        GPU: trailing update with cublasDgemm on the rest of the blocks other than the next panels blocks
    else
        GPU: trailing update with cublasDgemm
    end if
end for
```

In this algorithm, $\tilde{C}$ has the options of being the whole trailing matrix of the next step or only those rows that will be involved in the pivoting. Because the sole purpose of transferring C back and forth is for the pivoting to be valid, the second option could largely reduce the amount of data transferred and therefore leading to much less overhead. Fig. 5 shows an example of the second option. Once the panel factorization (green) is finished, the blue blocks are transfered back to the CPU which are the next panel blocks and the PDTRSM right hand side blocks. The red rows are those that will be involved in the pivoting according to IPIV so only these three (red) rows are transferred back. When the PDLASWP is done, these three rows are transferred back to the GPU. In ScaLAPACK, the pivoting information is stored in a global vector called IPIV, and IPIV(i) stores which global row the local row i needs to be swapped with. For example, on a process, if IPIV(2)=18, then the *local* row 2 starting from a base row is to be swapped with *global* row 18. The base is determined according to the current block-iteration step. Specifically, after the s-th panel factorization, the base local position in IPIV is calculated by

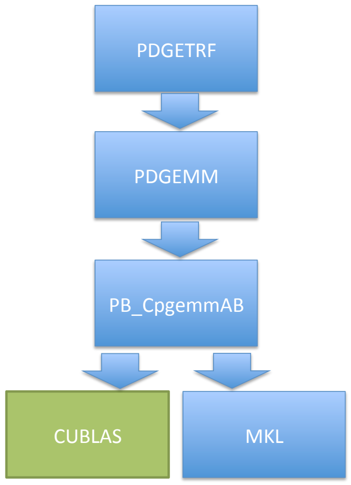


Fig. 6: Code Change to LU

```
infog2l_(&s, &s, descA, &nprow, &npcol,
&myrow, &mycol, &iic, &jjc, &icrow, &iccol);
```

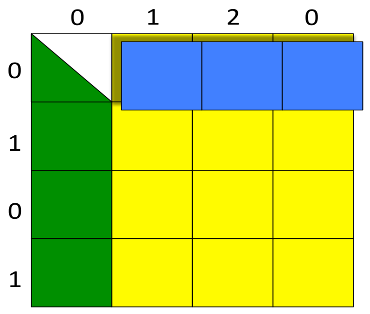
On each process, the base location is $iic-1$. IPIV[$iic-1$] to IPIV[$iic-1+NB$] have the pivoting information for this step.

The input matrix and the factorization result in Algorithm 1 reside on the CPU memory. From the application developer's point of view, no explicit GPU access is seen and the interface to PDGETRF looks exactly the same as the original version, except now it runs with the support of GPU for better performance. The code change made to PDGETRF is shown in Fig.6.

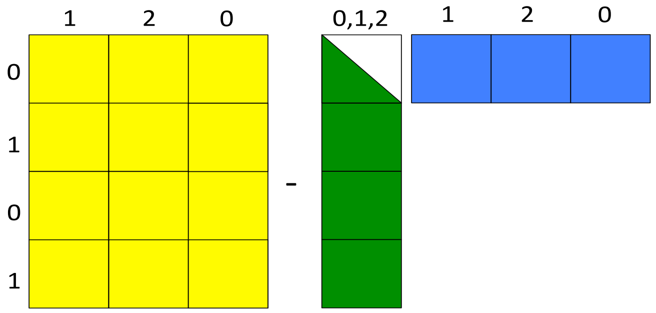
B. QR factorization

The major difference between LU and QR factorizations is the pivoting. Pivoting swaps rows that can only be determined at runtime, forcing a less efficient data transfer pattern between CPU and GPU. QR factorization has a much cleaner structure and this allows an implementation that has less CPU-GPU data transfers.

In (2), the trailing update perform the following computation:



(a) After the first step: $A_2^T V$



(b) $A_2 - V\tilde{W}^T$

Fig. 7: PDLARFB

$$\begin{aligned} \tilde{A}_2 &= Q^T A_2 = (I - VT^T V^T) \times A_2 \\ &= A_2 - V \times ((A_2^T \times V) \times T)^T \\ &= A_2 - V\tilde{W}^T \end{aligned} \tag{3}$$

In PDGEQRF, PDLARFB implements $Q^T A_2 = (I - VT^T V^T)A_2$ in three steps (W is a temporary buffer):

1) $W \leftarrow V^T A_2$
2) $\tilde{W} \leftarrow T^T \times W$ (depicted in Fig.7(a))
3) $\tilde{A}_2 \leftarrow A_2 - V\tilde{W}$ (Fig.7(b))

In Fig.7(a) and Fig.7(b), the blue blocks represent the temporary buffer W, and the green trapezoid is V. A different W is created in each iteration, and V is broadcasted row-wise from the processes that own the current PDGETF2 panel. T is a $NB \times NB$ block that is updated in each iteration. W are created on the GPU. T stays on the CPU while V are transferred to the GPU during each trailing update.

When it comes to A_2, a naive way is to keep the data on the CPU memory and transfer only the involved blocks to and from GPU during the trailing update. This method is referred to as the "Non-GPU-resident QR". This method is straightforward in coding but has no way to avoid the data transfer latency. Comparing to the case of LU factorization, we can also opt to keep the main matrix on the GPU global memory, and only transfer back the final result. This leads to Algorithm 2 for QR.

Algorithm 2 GPU resident ScaLAPACK QR

```
Transfer the input matrix to the GPU global memory;
for All the panels: do
  CPU: PDGEQR2 and PDLARFT;
  Copy $V$ and $T$ into the GPU global memory;
  if (My process is in the same process column as the next panel) then
    Copy the next panel blocks that I own, GPU $\rightarrow$ CPU;

    CPU: PDLARFB to the next panel blocks;
    GPU: PDLARFB on the rest of the blocks other than the next panels blocks;
  else
    GPU: PDLARFB for the trailing matrix blocks;
  end if
end for
```

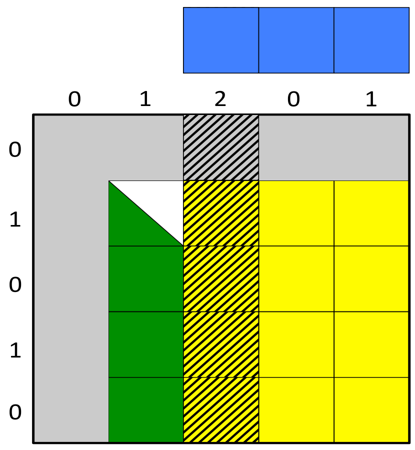


Fig. 8: The hybrid version of PDLARFB

Fig. 8 is an example of the GPU resident QR. The gray area is the part that is already finished and stays on the CPU. The green trapezoid is the current panel, and Processes (0,2) and (1,2) have the next panel blocks. Before executing PDLARFB, these two processes copy the shaded blocks from GPU to CPU to perform trailing update on CPU, while the yellow blocks are updated on the GPU. Since processes (0,2) and (1,2) have both CPU and GPU blocks to update, the GPU update is performed using the cublasDgemm call, which is non-blocking and therefore the GPU and CPU update can be overlapped with each other.

The code involved in the QR factorization is shown in Fig.9. In PDLARFB, the three steps in (3) are implemented with direct calls to the BLAS routines. Calls to cublasSetMatrix, cublasGetMatrix, and cublasDgemm are added to handle the GPU part of the trailing update.

V. Performance Evaluation

In this section experimental results are shown to verify and analyze the performance of the converted ScaLAPACK

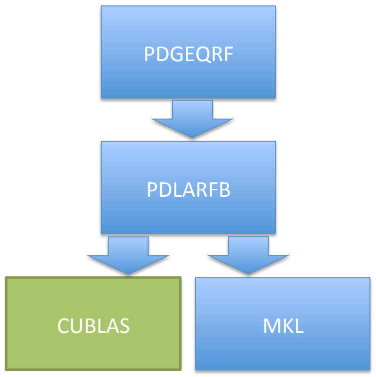


Fig. 9: Code Change to QR

LU and QR routines. Experiments were performed on two platforms. For small size experiments, we used a small cluster at the University of Tennessee, Knoxville named “Dancer”, which is a 12-node cluster with an Infiniband 20G interconnect. Each node has two quad-core Intel 2.27GHz Xeon CPUs, and an NVIDIA C2050 or C2070 GPU. For large scale experiments, we use the Keeneland system. Keeneland consists of computing nodes that have two hex-core CPUs and 3 NVIDIA GPUs, connected by a Qlogic QDR InfiniBand interconnect. The current system has a total of 120 nodes, 240 CPUs, and 360 GPUs. In all the experiments, one MPI process is run on each node. Multiple cores in each node are utilized by MKL threads. One GPU per node is used even if multiple ones might exist.

Fig.10 is the performance of LU factorization on the Dancer cluster. Both options of trailing matrix memory copy mentioned in Section IV-A have been implemented and labelled as “Option #1” and “Option #2”. From the experiment result, the first option are barely able to keep up with the CPU version of the ScaLAPACK while the second option outperforms CPU ScaLAPACK after matrix size $24,000 \times 24,000$. At the largest size of $56,000 \times 56,000$, a speedup of 70% was achieved at 489.34 Gflop/s against the 289.18 Gflop/s from the CPU version. To confirm the analysis on option 1, we profiled the run time of each component in the GPU PDGETRF and the result is shown in Fig. 11. From this figure, it is obvious that data transfer between CPU and GPU has taken a serious toll on the performance. When matrix is large enough, it takes more time to ship the data back and forth than performing the major parts of the computation such as PDGEMM and PDTRSM. This shows that only communicating the rows involved in the pivoting largely reduces this overhead and enables the eventual speedup.

With the same MPI process configuration, Fig. 12 shows the experiment of QR factorization on Dancer. Two flavors of GPU QR are implemented. One is the “naive” version where data stays on the CPU and is transferred to and from GPU only when it is required in the trailing matrix update. The other

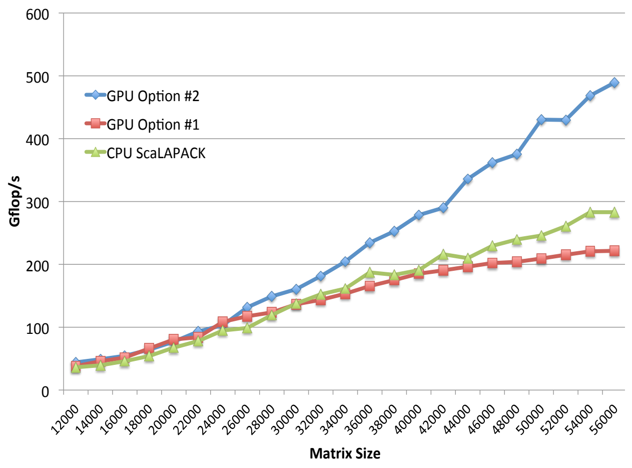


Fig. 10: ScaLAPACK LU with GPU on Dancer
4×3 MPI process grid, 8 MKL threads per node

version keeps the data on the GPU for computation and only transfers it back to the CPU memory one panel in each step. The comparison is similar to the LU factorization experiment because constantly exchanging large amount of data between the CPU and the GPU also leads to large overhead, making the GPU version even slower than the CPU version. At size of $8,000$ the GPU-resident version takes the lead in performance and increases to 1,03 Tflop/s at the largest size, achieving a 92% speedup over ScaLAPACK. Note that the crossover point where the GPU version passes the CPU version is much smaller for QR factorization than for LU. This is because of the data transfer overhead from pivoting in LU. Larger matrix, hence more FLOPs, is required to amortize this overhead. It is reasonable to foresee that for Cholesky, similar speedup can be seen to that of the QR factorization due to the absence of such frequent inter-process communication.

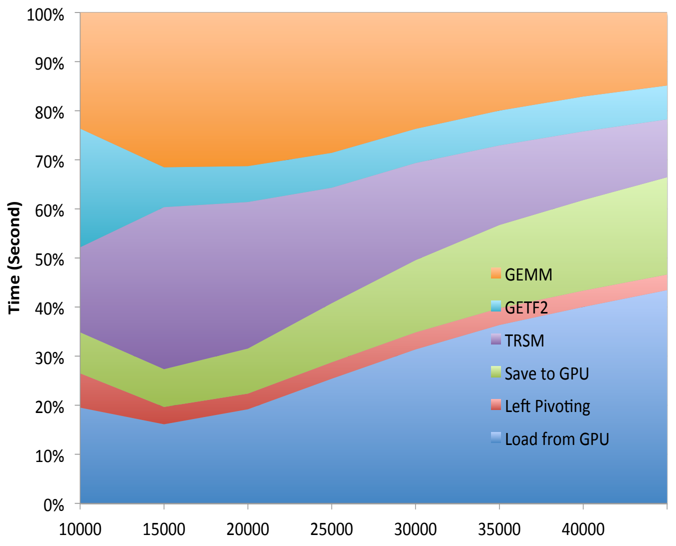


Fig. 11: The run time breakdown of the GPU-PDGETRF
2×3 MPI process grid, 8 MKL threads/node

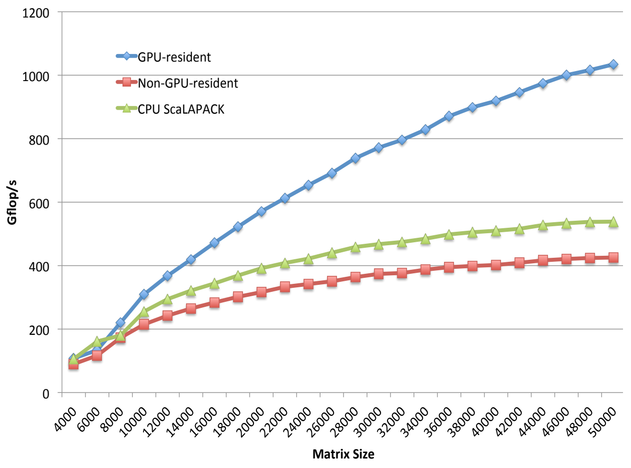


Fig. 12: ScaLAPACK QR with GPU on Dancer
4×3 MPI process grid, 8 MKL threads per node

Fig. 13 and 14 are the performance result on the Keeneland cluster system. Larger MPI process configuration is used for both experiments. Due to a software issue of the cluster system, the grid and matrix size are limited for LU factorization. This will be resolved in the future. Both experiments use the faster GPU version of LU and QR to compete against the CPU version. Because Keeneland system has faster CPU and more cores, the speedup for both factorizations is relatively smaller than on the Dancer cluster. Speedup of %40.4 and %68.2 was achieved for LU and QR respectively.

VI. Conclusion

In this work, by integrating the CUDA computing directly into the ScaLAPACK framework, we demonstrated that good speedup could be achieved by routines like LU and QR by carefully managing the GPU-CPU data transfer. Generally speaking, it is beneficial to keep data onto GPUs as much as possible. For LU factorization where pivoting forces more frequent data transfer, minimizing the data amount helps largely to reduce the performance impact. As future work, multiple GPUs per node will be taken into consideration, and more algorithms will be converted. Larger scale experiments will be conducted to further confirm the design. Other key components such as the triangular solver PDTRSM will be ported to use GPUs.

Acknowledgments

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and the MathWorks for supporting this research effort.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[2] E. Anderson, Z. Bai, and C. Bischof. *LAPACK Users' guide*. Society for Industrial Mathematics, 1999.

[3] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[4] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Technical report, Technical Report LAPACK working note 19, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[5] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr. Optimized hpl for amd gpu and multi-core cpu usage. *Computer Science-Research and Development*, pages 1–12, 2011.

[6] R.F. Barrett, THF Chan, E.F. D'Azevedo, E.F. Jaeger, K. Wong, and RY Wong. Complex version of high performance computing linpack benchmark (hpl). *Concurrency and Computation: Practice and Experience*, 22(5):573–587, 2010.

[7] C.H. Bischof and C. Van Loan. The wy representation for products of householder matrices. In *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing*, pages 2–13. Society for Industrial and Applied Mathematics, 1985.

[8] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

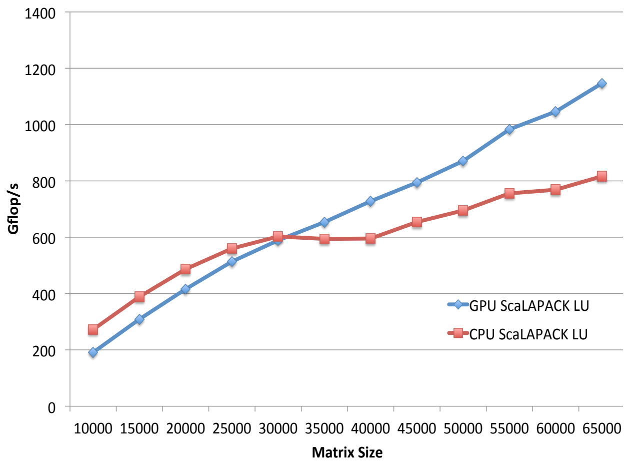


Fig. 13: ScaLAPACK LU with GPU on Keeneland
4×4 MPI process grid, 12 MKL threads per node

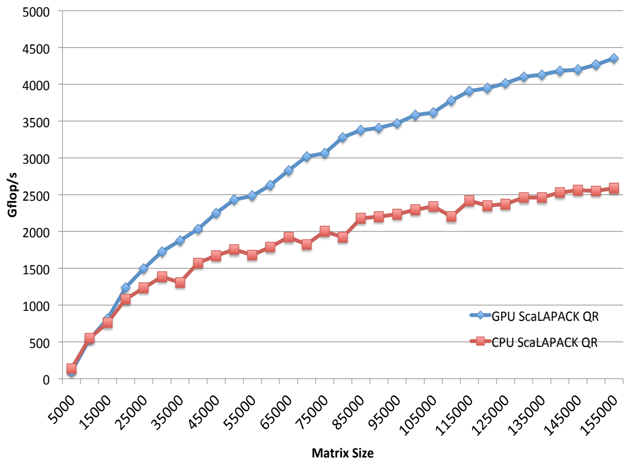


Fig. 14: ScaLAPACK LU with GPU on Keeneland
6×6 MPI process grid, 12 MKL threads per node

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[10] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. A proposal for a set of parallel basic linear algebra subprograms. *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 107–114, 1996.

[11] Jack Dongarra and R. Clint Whaley. A user's guide to the BLACS v1.1. Technical Report UT-CS-95-281, University of Tennessee Knoxville, March 1995. LAPACK Working Note 94 updated May 5, 1997 (VERSION 1.1).

[12] J.J. Dongarra. *LINPACK: users' guide*. Society for Industrial Mathematics, 1979.

[13] J.J. Dongarra, L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, et al. ScaLAPACK user's guide. *Society for Industrial and Applied Mathematics, Philadelphia, PA*, 1997.

[14] E. DAzevedo and JC Hill. Parallel lu factorization on gpu cluster. *Procedia Computer Science*, 9:67–75, 2012.

[15] M. Fatica. Accelerating linpack with cuda on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 46–51. ACM, 2009.

[16] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings, 1994.

[17] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst D. Simon. *TOP500 Supercomputer Sites*, 36th edition, November 2010. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html).

[18] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[19] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi v1.1. Technical report, NVIDIA Corporation, 2009.

[20] C. Nvidia. Nvidia cuda c programming guide. *NVIDIA Corporation*, 2011.

[21] J. Ohmura, T. Miyoshi, I. Hidetsugu, and T. Yoshinaga. Computation-communication overlap of linpack on a gpu-accelerated pc cluster. *IEICE TRANSACTIONS on Information and Systems*, 94(12):2319–2327, 2011.

[22] D. Rohr, M. Bach, M. Kretz, and V. Lindenstruth. Multi-gpu dgemm and hpl on highly energy efficient clusters. *Micro, IEEE*, (99):1–1, 2011.

[23] R. Schreiber and C. Van Loan. A storage-efficient wy representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[24] F. Song and J. Dongarra. A scalable framework for heterogeneous gpu-based clusters. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 91–100. ACM, 2012.

[25] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 365–376. ACM, 2012.

[26] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[27] V. Volkov and J. Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May*, pages 2008–49, 2008.

[28] V. Volkov and J.W. Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[29] F. Wang, C.Q. Yang, Y.F. Du, J. Chen, H.Z. Yi, and W.X. Xu. Optimizing linpack benchmark on gpu-accelerated petascale supercomputer. *Journal of Computer Science and Technology*, 26(5):854–865, 2011.