Parallelization of the LU Decomposition on Heterogeneous Systems

Carlos Martins, Pedro Tomás, and Ricardo Chaves

***Abstract*—With the appearance of GPUs as valid platforms, not only for graphics computation, but also general-purpose computations, applications that exploit hybrid/heterogeneous systems can be made available to the mass market due to the widespread availability of these systems. Correct distribution of the workload of these applications can lead way to significant performance boosts to complex applications. In this work, we study the LU factorization problem in heterogeneous systems and propose two different approaches developed with CUDA: a multi-device solution that distributes the workload as fairly as possible between the CPU and the GPU(s), and a master-slave approach in which the CPU does the task management while the GPU performs the necessary computations. Since one issue of the blocked factorization algorithm is the bottleneck created by the pivoting step, the proposed solutions make use of pre-processing pivoting strategies to mitigate this problem. The results obtained show that both solutions can thrive depending on the system architecture, obtaining speed-ups of up to 3 when compared to the Intel MKL implementation optimized for CPU.**

***Index Terms*—Heterogeneous Systems, Load Balancing, LU factorization, CUDA, Matrix computation**

I. Introduction

In the last decade, the physical limits of increasing the number of transistors on a single CPU chip efficiently have nearly reached its limits, triggering the appearance of multicore solutions. Multicore CPUs have two or more cores on the same die, increasing the processing power through parallelization while keeping the clock speed and power consumption at efficient levels.

A graphics processing unit (GPU) is a specialized parallel microprocessor designed to render 3D applications efficiently. The GPU design has, from its inception, been driven by the gaming industry and its parallel processor architecture, optimized for accelerating graphical computations, is a consequence of that. Until the last decade, programming general purpose applications in the GPU (GPGPU) was not easy without a background in graphics development, but with the introduction of CUDA in 2007 and CUDA enabled GPUs [6], developers are now able to exploit the GPU for building applications other than graphics rendering on NVIDIA architectures (and OpenCL introduced in 2010 allows applications to be developed for GPUs from multiple vendors).

The rise of GPGPU development and the widespread availability of GPUs on modern systems created a new paradigm of heterogeneous computing, where applications that can use both the sequential nature of multi-core CPUs and the high throughput of GPUs, to distribute the workload, to produce efficient solutions that can be targeted at a massive audience. The potential brought by the heterogeneous systems has spurred interest in the development of efficient solutions for linear algebra problems.

Hybrid architectures are becoming the dominant force and drive in innovation of today's and future's computer systems. The GPU is now present, alongside the CPU, in most devices available in the consumer market, allowing them to render high-quality 3D graphics, and to deliver high performance for highly regular applications. The advantages of GPU architectures for regular applications has lead the scientific community to try to use these systems' parallel design to improve performance of routines in many fields. This work will further aid that cause, studying a routine to perform LU factorization, taking advantage of the parallel nature of nowadays systems.

The chosen algorithm for this effect was LU factorization (on the LaPACK library, the routines responsible for the LU factorization are the *?getrf* routines), which factors an input matrix into an Lower triangular matrix and an Upper triangular matrix (and optionally a Permutation matrix). The LU decomposition routines are of great importance in the context of the LaPACK library, since most of the other routines, such as solving linear systems equations, require the input matrix to be factorized [7], meaning that an efficient LU factorization routine will have a great impact in a big part of the linear algebra functions of LaPACK. Two different approaches will be considered with the intent of finding the most efficient manner to balance the load in the heterogeneous environment: The multi-device approach that will distribute the workload between both the multicore CPU and the GPU, in an effort to take advantage of all the processing power of the whole heterogeneous system and the GPU-only implementation that will leave the bulk of the computations to be performed by the GPU and avoid most of the data transfers.

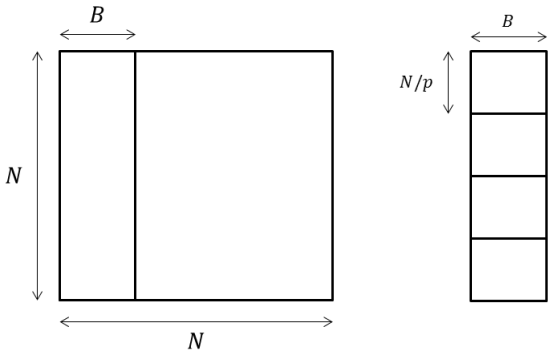
To maximize performance, the proposed set of algorithms are based on dynamic scheduling approaches, which take into account the device relative performance, by considering the execution time of previously executed tasks.

To accomplish the proposed work, existing library implementations (e.g., Intel MKL, NVIDIA CuBLAS and ViennaCL) were used, not only to efficiently execute specific functions in either the CPU or the GPU, but also as reference benchmarks, in order to evaluate the performance of the proposed algorithms.

II. Related work

For this work, existing implementations of LU factorizations were studied, some of them presenting variations of the previously presented blocked strategy, and are discussed here.

The solution proposed by [1] transforms the second step, the pivoting after the factorization into a pre-processing step, allowing the factorization to be parallelized in different processors. The paper focuses on multi-core and remote cluster systems and aims at reducing the communication time spent during data transfer between the different processors. This approach starts by splitting the first block-row of the matrix into N/p by B blocks. Each block is assigned to a different process (Figure 1(b)). Each process computes the pivot lines of its blocks and then a tournament between the threads starts: half of the processes will now merge the first half of their pivot lines with the first half of the remaining processes pivot lines and compute the pivot lines of that new N/p block. This process is repeated until the B pivot lines are found. The LU factorization of the first block-row can now be done without pivoting. In the typical approach where the factorization is done before the pivoting which cannot be parallelized and introduces a barrier between the factorization and the rest of the steps.



(a) Algorithm starts with first block-column

(b) The block-column is then split among the processors

Fig. 1. Matrix splitting by processors.

The work proposed in [1] is extended in [2], using the firsts idea for pivoting and applying it on a multi-core CPU + GPU architecture. In this approach, after the pre-processing pivoting tournament, the CPU starts by performing the LU factorization of the first block-column, and, after it completes this first step, one CPU thread starts sending the necessary data to the GPU and the remaining threads start to compute the trailing sub-matrix, overlapping communication with computation. When all data is transferred, the GPU, in parallel with the CPU, computes the rest of the trailing sub-matrix. This process repeats until the whole matrix is factorized. Figure 2 shows a step-by-step representation of this algorithm. The main focus of this method is to reduce the time spent with data transfers between the CPU and the GPU by overlapping them with computations. When presenting the results, the author notes that the performance is better when compared to the existing MAGMA implementations.

The authors of [3] present a similar approach to [1], having a pre-processing step eliminating the need to pivot after the factorization. In this case, a method called Partial Random Butterfly Transformation (PRBT) is used. This method starts

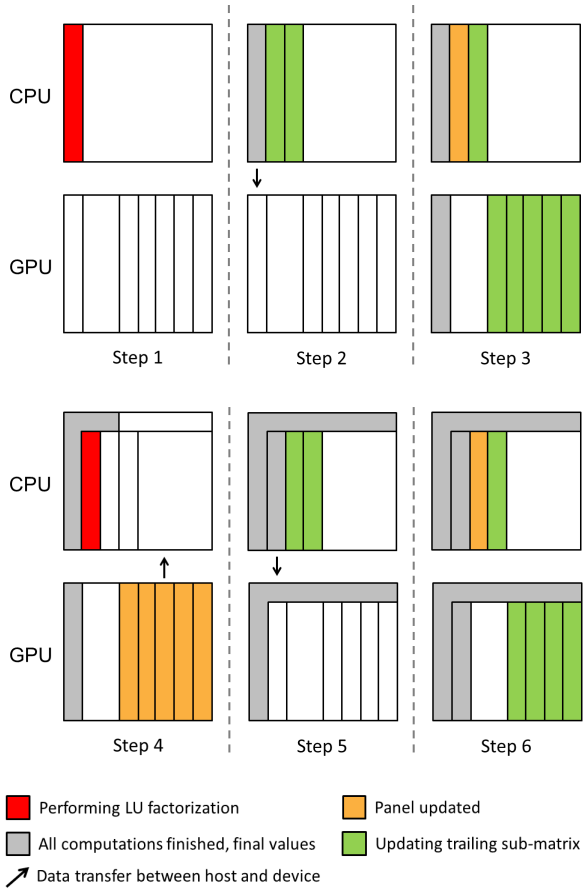
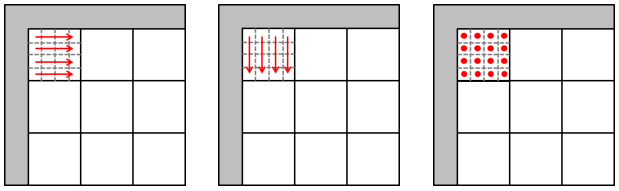


Fig. 2. Matrix splitting by processors.

by computing the randomized matrix $A_r = U^T AV$, where A is the input matrix, and U and V are recursive butterflies of depth d. The paper states that to get an accurate result, the randomized algorithms must be computed with butterflies of, at least, depth 2. The authors further state that there is an improvement in speed on the GEPP (Gaussian elimination with partial pivoting) routine compared to the one presented in the MAGMA library, but also states that the LU factorization is only worth applying in matrices with convenient parameters.

In [4], the load is split leaving the CPU responsible for computing the lower and upper matrices and the GPU only responsible for updating the trailing sub-matrix. Three different schemes for updating the sub-matrix are analyzed: Update through rows; update through columns; update each element. Each iteration of the routine computes a line and a column of the lower and upper resulting matrices and updates the trailing sub-matrix. The process of updating the trailing sub-matrix is done on the GPU by blocks where the threads compute it depending on the scheme by rows, columns, or associated to each element of the block. The author concludes that the best approach is to update the trailing sub-matrix having one thread associated with each element, but notes that no satisfactory performance results are obtained with this method.

The authors of [5] try to take advantage of coalescing memory by manipulating how the mapping of sparse matrices is done in the memory. The process is akin to the traditional



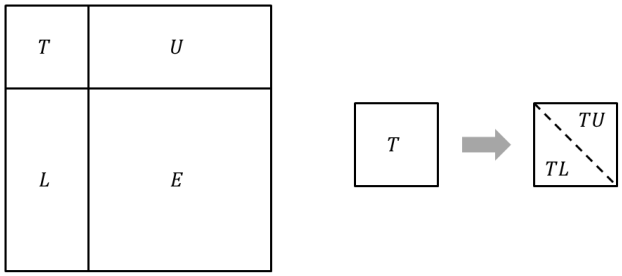
(a) Update by rows **(b)** Update by columns **(c)** Update by element

Fig. 3. The three different strategies used for updating the trailing sub-matrix.

blocked LU algorithm, and starts by performing the LU of a small block, then using the values from the first step, computing the values to its right and below it, and finally using those results to compute the rest of the matrix. This work presents GPU-only and CPU+GPU hybrid solutions with OpenMP. Despite acknowledging having lower speed-ups than expected, it does show performance improvements compared to the existing routines. The paper also concludes that on most cases, the GPU-only version shows better performance. However, this work does not mention the pivoting step and can only be applied to sparse matrices.

III. Blocked LU factorization

To allow for task parallelization and distribution among the heterogeneous system, a blocked strategy for the LU decomposition algorithm is used. Accordingly the $N \times N$ input matrix is subdivided in four partitions as depicted in Figure 4, where T is the first $B \times B$ tile of the matrix, L is the first $(N-B) \times B$ block-column, U the first $B \times (N-B)$ block-row and E the $(N-B) \times (N-B)$ square trailing sub-matrix. After the matrix partitioning, there are four major

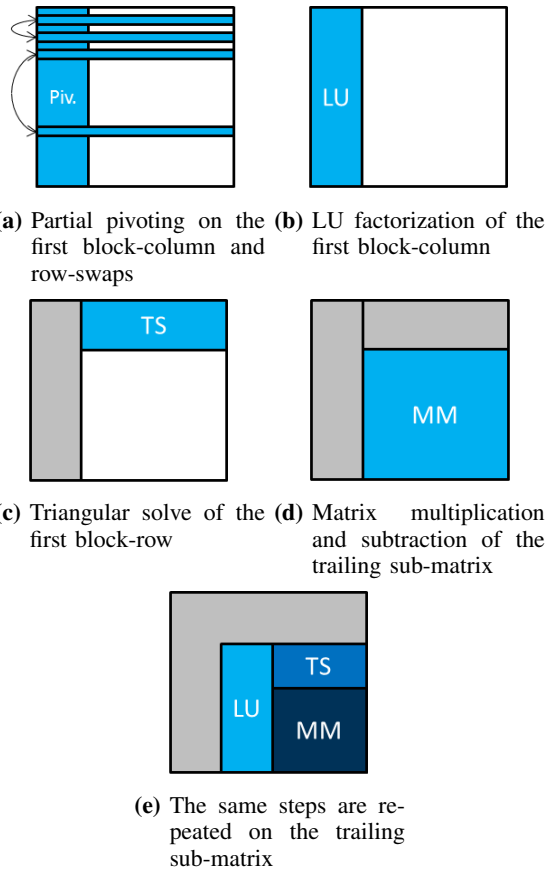


(a) Matrix partitioning for LU decomposition of a matrix **(b)** Identification of the sub-block of tile T

Fig. 4. Matrix partitioning strategy

mathematical operations that need to be realized to the sub-blocks to perform the blocked factorization, as depicted in Figure 5, namely:

1) **Partial pivoting** of the first $N \times B$ block-column, corresponding to blocks T and L (see Figs. 4(a) and 5(a)). The pivot lines are identified and the corresponding row swaps are performed.
2) **LU factorization** of the same block-column in which the partial pivoting was performed. The factorization operation rewrites the first tile as an upper triangular matrix TU and an lower triangular matrix TL (see Fig. 4(b)), and also computes the rest of the elements of the first block-column, L (see Fig. 5(b)).
3) The block-row U is computed by solving the $TL_0U_1 = U_0$ system, using the **triangular solve** operation, where TL_0 and U_0 correspond to blocks of the original matrix while U_1 is the resulting block-row (see Fig. 5(c)).
4) The trailing sub-matrix, E, is updated with a **matrix multiplication** followed by a subtraction, $E_1 = E_0 - L_1U_1$, where E_0 is the original square trailing sub-matrix partition and L_1, U_1 and E_1 are the resulting block-column, block-row and square trailing sub-matrix of the set of operations performed in the first iteration of operations (see Fig. 5(d)).



(a) Partial pivoting on the first block-column and row-swaps **(b)** LU factorization of the first block-column

(c) Triangular solve of the first block-row **(d)** Matrix multiplication and subtraction of the trailing sub-matrix

(e) The same steps are repeated on the trailing sub-matrix

Fig. 5. Main operations of the blocked LU factorization algorithm.

The second iteration is done by applying these steps to matrix E_1 (see Figure 5(e)), and the execution finishes when all the matrix's block-columns are factorized. Note that, as previously mentioned, the pivoting step is done before the factorization, unlike what is done in the typical block algorithm depicted in Section **??**, where the pivoting is done after the factorization step.

In the case of the LU factorization operation, matrix A can be rewritten as the product of two triangular matrices L and U as seen in expression (1).

$$A = LU \tag{1}$$

The elements of the main diagonal of matrix L are always one (see Eq. (2)), allowing for resulting matrices of the

factorization to be written in only one square matrix.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{10} & 1 & 0 \\ l_{20} & l_{21} & 1 \end{bmatrix} \begin{bmatrix} u_{00} & u_{01} & u_{02} \\ 0 & u_{11} & u_{12} \\ 0 & 0 & u_{22} \end{bmatrix} \quad (2)$$

As such, the factorization step is done in-place, meaning that there is no need to allocate further memory for the results as they just replace the original values, resulting in matrix depicted in Eq. (3). The in-place computation approach is also taken for the other operations as well, eliminating the need for further memory allocations.

$$\begin{bmatrix} u_{00} & u_{01} & u_{02} \\ l_{10} & u_{11} & u_{12} \\ l_{20} & l_{21} & u_{22} \end{bmatrix} \quad (3)$$

The devised Multi-device and GPU-only parallel solutions are based on the blocked LU algorithm, which, as previously stated, allows for dividing the computation work in sets of tasks. Each task may comprise more than one mathematical operation (as can be seen below with task LU1). The developed dispatching system herein described is used to issue these tasks, on the computational devices in the most efficient way possible. For this, it is important to note the dependencies between tasks which are described below. The tasks used to perform the algorithm in a blocked scheme are:

1) **LU factorization of a tile (LU1)** - In this step the pre-processing pivoting finds the pivots of the target block-column and the factorization of the first $B \times B$ tile is performed. The identification of the pivots of the block-column is done with the routines described later in Section IV and V for the multi-device and GPU-only solutions, respectively. The factorization done with the MKL implementation of *?getrf* in the multi-device solution, and with the kernel detailed in section V in the GPU-only solution.
2) **LU factorization of a tile (LU2)** - Factorization of the rest of the block-column. This factorization is done with the same routines as the previous task. The LU1 and LU2 tasks can only be performed once the trailing sub-matrix has been updated for the target block-column (with the exception of the first iteration).
3) **Triangular solve (TS)** - The triangular solve operations are performed with the *cublas?trsm* routine of the cuBLAS library. The triangular solver can only be applied once the LU1 task finishes.
4) **Matrix multiplication and subtraction (MM)** - The matrix multiplication is performed with the *cublas?gemm* routine of the cuBLAS library, while the subtraction is done by a developed subtraction kernel. The matrix multiplication and subtraction needs both the LU2 and the TS tasks to finish to be performed.

To execute the blocked tasks efficiently, a dispatching system was developed resorting to a priority queue that has all tasks available for execution at each point. This queue is updated dynamically during execution: whenever a task is finished, the queue is updated with the new tasks that are available for execution (e.g. after the LU factorization, triangular solve operations, to the right of the first tile are added). To have the highest possible degree of parallelization between all the tasks, the prioritization is done with the following aspects in mind:

1) LU factorizations (tasks LU1 and LU2) will always have highest priority. This is done because in the two presented solutions, the LU factorization is the operation that has the highest impact on performance.
2) Tasks on which the next factorization depends upon will be given higher priority. This results in: Tasks associated with earlier iterations being given higher priority; Within the same iteration, tasks whose localization within the matrix is closer to the top-left corner will be given higher priority.

The dispatching system resorts to a priority queue that has the necessary information about the tasks to allow the use of the defined prioritization strategy. To implement this queue, the tasks are defined in a structure that holds all the information concerning each task. The information held is comprised of the position in the matrix of block regarding the task, the task identifier (whether it's a factorization, a triangular solve or a matrix multiplication and subtraction), the current iteration and the task priority. The multi-device and GPU-only algorithms end when this queue is empty and all tasks finish their execution.

IV. Multi-device approach

The multi-device algorithm's core idea is to distribute the computation across all the processing devices so as to maximize performance. Although this may seem like the most obvious efficient approach at a first glance, the blocked LU factorization has many opportunities for bottlenecks to arise, mainly because of the constant memory transfers between the processing devices, but also because of the difference in the performance of each processing device when executing different tasks (e.g. the LU tasks in the CPU may take substantially more time to finish then all the tasks regarding the trailing sub-matrix in the GPU, or vice-versa, depending on the CPU and GPU on the system) and because of the dependencies between the tasks this might lead to high amounts of idle times. These dependencies refer to what is stated in the blocked algorithm presented in the previous section (e.g. the triangular solve uses the results of the previous factorization), and results in a need for synchronization between the tasks and a subsequent performance bottleneck. On the other hand, in order for host and devices to compute the operations, both must have up-to-date data, and thus the need for constant data communication between the processing units.

The proposed method for workload distribution between the available processing units in the system has into account the aforementioned limitations, and tries to mitigate their effects on the execution time. This approach aims at minimizing the idle time of each processing unit and overlapping, as much as possible, the computation of tasks with data communications.

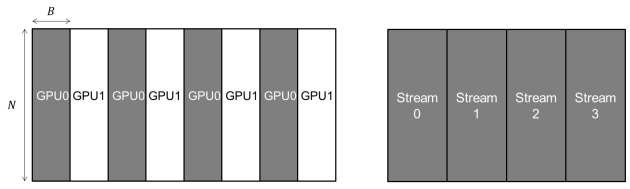
In the multi-device solution, the CPU is responsible for the pre-processing pivoting step and the LU factorization of block-columns, while the available GPUs will be updating the trailing sub-matrix corresponding to each factorization step. The reason for this workload distribution is the parallelizable nature of both matrix multiplication and triangular solve, making them suitable for computation in the GPU, and the difficulty associated with the parallelization of the factorization step, due to the data dependencies and irregular memory accesses. Since the operations (pivoting, LU factorization, triangular solve and matrix multiplication) are performed in blocks, it is possible to sub-divide the computation into several steps, which can be further distributed to different devices and at different time instances, in accordance with to the previously described task prioritization.

Accordingly, the execution of the multi-device LU decomposition starts by reading the input matrix and sending its data to the devices' memory, by relying on asynchronous CUDA call, to allow the data transfer to be done concurrently with the execution of the pivoting procedure. Data and work distribution is done in a round-robin fashion on the GPUs: the first block-column goes to the first device, the second block-column goes to the second device, and so on, cycling through the devices until all memory is transferred, an example of this for 2 GPUs can be found in Figure 6(a). The reason for the round-robin approach is to provide a fair distribution of workload across the available GPUs. At each iteration of the algorithm, the trailing sub-matrix size is decreased in B lines and columns. As such, considering execution in 2 GPUs, if instead of the round-robin approach, the matrix workload were split in the first half for one GPU and the second half for the other, the workload distribution would not be efficient, because once half of the block-columns were factorized, the GPU responsible for the first half of the matrix would have no more operations to execute.

All kernel calls are linked with CUDA streams, making it possible to call the kernels immediately, even if the GPU is busy, queueing them for execution. As such, the CUDA kernel can start computing, as soon as the kernel dependencies are solved, making the resources available for the computation. Each block-column of each GPU's matrix has one stream associated with it and whenever a task performed on that block-column is available for execution, it will be queued on its corresponding stream as depicted in Figure 6(b). After performing the factorization and queueing the rest of the tasks, the host will wait, if needed, for the CUDA stream responsible to send the concerning the updated trailing sub-matrix of the next block-column to begin the next pivoting and factorization step. This is done by associating streams to the *cudaMemcpyAsync()* calls.

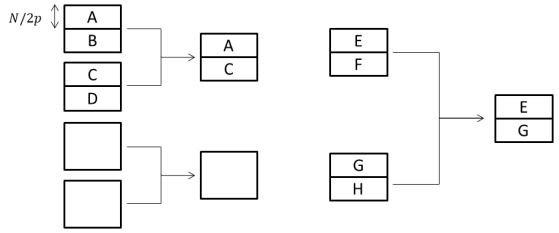
During the data transfer, the CPU starts computing the pivots of the first block-column, and the identified pivots are stored in the P vector that is then sent to the devices so that they can perform row swaps where needed to maintain data coherence.

In this implementation, the pivoting is done with the a tournament algorithm used to identify the pivots, used before the factorization step, similarly to [2]. At each iteration, i, the block-column that will be factorized, is split into p tiles of $N/p \times B$ elements, where p is the number of cores of processor.



(a) Work distribution on two GPUs. **(b)** Work distribution on the streams of one GPU.

Fig. 6. Work distribution by GPU and by CUDA streams.



(a) At step 1 p sub-matrices are pivoted **(b)** The process continues for $log(p)$ iterations

Fig. 7. Tournament pivoting strategy

Threads are assigned a tile and each thread computes the pivots of their respective tile. Each of the P thread computes the pivots of its respective partition using the following steps:

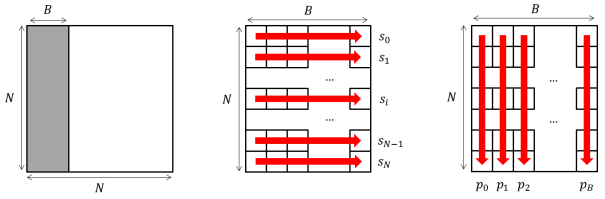
1) Loop over all elements of all the rows to find for their absolute maximum value, called the *scale factors*, these values are stored in the *scale factors vector* (Fig. 8(b)).

$$s_i = \max_{0<j<B} |a_{ij}| \tag{4}$$

2) All values from each column are evaluated and the value whose quotient with the corresponding scale factor is the maximum will define that column's pivot (Fig. 8(c)).

$$p_j = \max_{0<i<N} \frac{|a_{ij}|}{s_i} \tag{5}$$

This is done to all tiles of the target block-column in a tournament fashion until the final B pivots are found.



(a) Block-column is identified **(b)** From each row a scale factor is computed **(c)** From each column a pivot is identified

Fig. 8. Pivoting identification

The rows of each partition are then swapped according to the respective identified pivots. The B pivots lines of the first

thread, are then concatenated with the B pivot lines of the second, the third with the fourth and so on, beginning the tournament (the tournament scheme is depicted in Figure 7). The tournament continues until the final B pivot lines are found. Using the B pivots, the lines are swapped on the CPU before the LU factorization step starts execution. The lines are also sent to device memory to perform the row swaps on the trailing sub-matrix.

By having all the tasks blocked, communication and computation can easily be overlapped on the devices. As soon as a tile or block-column's result is available, a data transfer can begin concurrently with the factorization and the other tasks of the updating process of the trailing sub-matrix. Only the data necessary for either, the host or the devices computation to continue is transferred (e.g., the second factorization only needs the result of the first triangular solve and the first matrix multiplication).

After performing a task, new tasks may be available for execution (since their dependencies are now solved). Resorting to the priority queue of the previously mentioned dispatching system, tasks are executed in the order that minimizes the idle times of the host and the devices.

With one CPU and one GPU, the execution is performed depicted in Figure 9 and presented below:

1) The pivots of the first block-column are identified and the corresponding swap lines are performed as the resulting pivoting information is sent to the GPU who will swap lines of the matrix in device memory.
2) The host performs LU factorization of the first B by B block with the ?getrf routine and this information is sent to GPU as the host continues to compute the factorization of the rest of the block-column.
3) After step 2, computation can already start on the devices, so the factorized tile is sent to the GPU, asynchronously as the CPU continues to factorize the rest of the block-column, also using the ?getrf routine. The GPU uses the tile to compute the block-row with the triangular solver (with the cublas?trsm function).
4) After completing the factorization on the CPU the data is sent to the device. On the device, each block-column is computed independently. When the triangular solve (done with the cublas?trsm routine) finishes (may have finished while the CPU is performing the rest of the factorization), the matrix multiplication (done with cublas?gemm) and subtraction (executed by the previously mentioned subtraction kernel) of the block-column beneath begins. As the triangular solver function finished, its result is sent to the CPU.
5) When the computation of the results of the necessary trailing sub-matrix for the factorization of the second block-column is finished, the data is sent, while the rest of the sub-trailing matrix continues being computed. The CPU can start computing the pivots for the next iteration of the factorization algorithm.
6) As all the trailing sub-matrix is computed, the second iteration of factorization and can start on host and device. The first tile of the trailing sub-matrix of the previous iteration begins and the GPU performs the row swaps on device memory.
7) The factorization is applied to the rest of the second block-column as the GPU starts the triangular solver of the second block-row.
8) Once again, the triangular solver results are sent to the CPU when the computation finished and concurrently the matrix multiplication and subtraction operation update the trailing sub-matrix. These steps continue on all trailing sub-matrices.

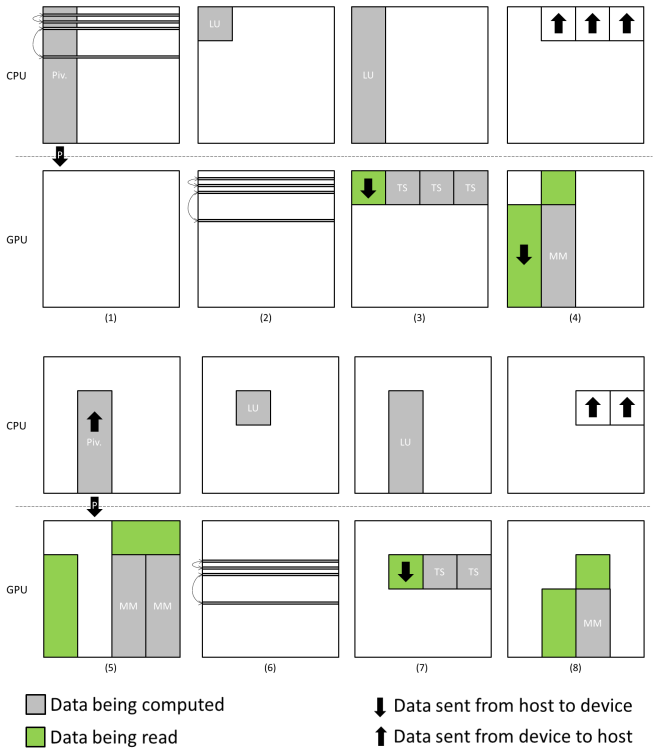


Fig. 9. Data transfer behaviour during multi-device execution.

The process ends when the priority queue used for task dispatching is empty and the last LU factorization task finishes execution.

V. GPU-ONLY APPROACH

The objective of this alternate solution is to mitigate the bottlenecks in the multi-device approach that may arise from the time the CPU takes to perform the factorization step, resulting in data transfers that cannot be overlapped with computation and GPU idle time. To mitigate this issue, we present a strategy focused on using the GPU (slave) to perform the computation of all tasks, while the CPU (master) simply manages the order in which the tasks are performed. The expected results are that, although the LU factorization step might suffer a loss in performance time, since the parallelization of the factorization with other tasks within the GPU is possible, the overall performance may increase. Another advantage of this approach is that there is no need for constant data transfers during execution, allowing the integration of the LU factorization in applications running solely on the GPU (removing the need to transfer data back to the CPU).

The dispatching system used in the multi-device solution is also used for the GPU-only implementation, but, in this case, all tasks and their respective priority are computed by the CPU, at the beginning of the execution. Doing this provides the CPU with all the necessary information to be able to queue, from start and finish, all the factorization tasks on the GPU in the beginning of the execution, removing the overhead of having to return to the host for each operation. Accordingly, all tasks are immediately dispatched to the GPU and are executed whenever there are resources available to do so.

The core idea of the pre-processing pivoting strategy used in Section V is re-used here, although in this case it's not done with the tournament algorithm, but rather in a way that takes advantage of the high number of cores of the GPU. This strategy starts by searching all rows for their maximum absolute value and storing them in a vector and then use the values from that vector, called scale factors vector, to find the pivot for each of the B columns.

This pre-processing pivoting kernel starts with the same method used by each core on their respective partition in the multi-device implementation. Each thread is therefore assigned with a row such as to find the *scale factors vector*, the vector is then used in a reduction to find the pivot lines. To find the scale factors vector, each thread starts by cycling through its respective row and stores the maximum absolute value on local memory (identical to what is seen in Figure 8(b)). After that, a reduction is performed on each column: in the first iteration the first $N/2$ threads compares the value of the corresponding element of the column with the element $N/2+tid$, where tid is the thread identifier (ranging from 0 to $N/2$ in the case of the first iteration) and the greatest value of the two is stored. In the second iteration, the comparison is done between the first $N/4$ and $N/4+tid$ elements. This process repeats $\log(N/2)$ times to find the pivot element, as seen Figure **??**. The reduction is performed over all columns to find the B pivots. The reduction operation is used because of its low arithmetic intensity that allows for high performance on operations over large arrays.

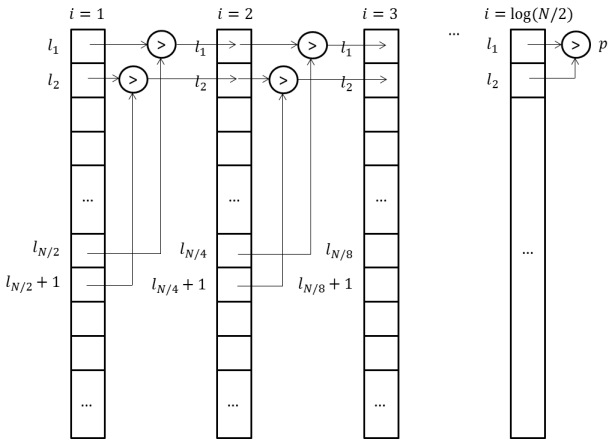


Fig. 10. Reduction strategy used to find the pivot elements.

After the factorization of the first tile, the triangular solve of the top tiles of the trailing sub-matrix begins. The routine used for the triangular solve is a variant of the cuBLAS's cublas?trsm, developed to be called from within the device, avoiding the overhead of having to return the results to the CPU to perform the kernel calls. Tasks are managed the same way as in the multi-device approach, so that when the triangular solve tile and the block-column factorization steps finish, the matrix multiplication and subtraction tasks begin, using the routine cublas?gemm of cuBLAS for device call. The matrix subtraction task is done with a developed kernel that has each thread responsible for a row, performing the subtraction for all B columns. This same approach is used in the kernel for the subtraction in the multi-device implementation.

All the factorization and trailing sub-matrix update tasks are performed until the priority queue is empty, and, once all execution on the device is finished, the resulting factorized matrix is transferred back to the CPU (Fig. 12(d)).

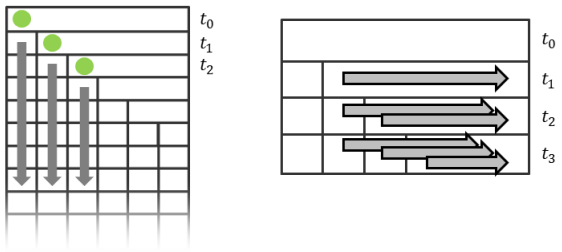


Fig. 11. GPU LU factorization execution

The developed pre-processing pivoting and LU factorization algorithms only use one of the available CUDA block in the GPU to perform the computations. This is done because of the difficulty in performing these tasks in more than one of the GPU's blocks, due to the data dependencies associated with these tasks and the fact that CUDA blocks run independently of each other (with access only its own shared memory) and cannot be synchronized. Although limiting the processing performance to that of a single simultaneous multiprocessor (SM), the other SMs can still be used to concurrently run other tasks, that are required by the LU factorization procedure.

The behaviour of the GPU-only approach is illustrated in Figure 12. The execution begins by sending the input matrix data to device memory with *cudaMemcpyAsync()* calls, and concurrently computing the priority queue with the task order for computation in the GPU (see Figure 12(a)). The priority system, presented in Section **??**, remains the same, so the LU factorization step is always done whenever available for execution. Since the factorization kernels only use one CUDA block, the remaining blocks can be used for concurrent execution of other tasks.

With the matrix data in the device, the process can begin. Once again, execution begins with the pre-pivoting applied to the first block-column, followed by the factorization of the first $B \times B$ square tile. Since the pivoting and the LU factorization kernels only use one block, the remaining blocks can be used for the operations used to update the trailing sub-matrix. Considering the LU factorization task is given the highest priority, the factorization and the other tasks will

operate concurrently through most of the execution (Figure 12(c)).

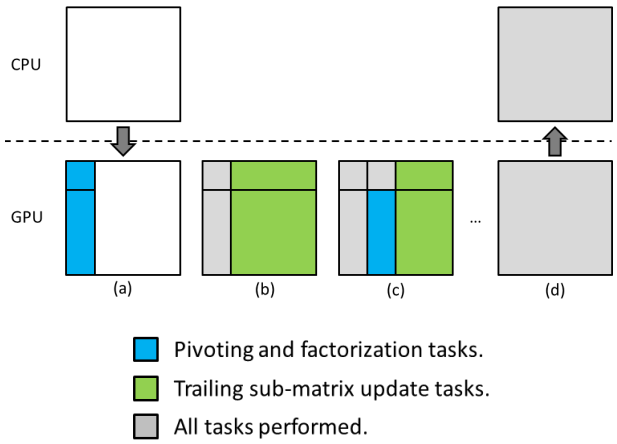


Fig. 12. GPU-only execution.

After the factorization of the first tile, the triangular solve of the top tiles of the trailing sub-matrix begins. The routine used for the triangular solve is a variant of the cuBLAS's cublas?trsm, developed to be called from within the device, avoiding the overhead of having to return the results to the CPU to perform the kernel calls. Tasks are managed the same way as in the multi-device approach, so that when the triangular solve tile and the block-column factorization steps finish, the matrix multiplication and subtraction tasks begin, using the routine cublas?gemm of cuBLAS for device call. The matrix subtraction task is done with a developed kernel that has each thread responsible for a row, performing the subtraction for all B columns. This same approach is used in the kernel for the subtraction in the multi-device implementation.

All the factorization and trailing sub-matrix update tasks are performed until the priority queue is empty, and, once all execution on the device is finished, the resulting factorized matrix is transferred back to the CPU (Fig. 12(d)).

VI. Results

A. Testing framework

To properly evaluate the proposed algorithms, hybrid systems are considered. For this purpose, two different machines are considered for the deployment of the several implementations and to evaluate the results achievable by each of the approaches. The testing framework is composed of the following machines:

1) Machine 1 - Xeon E5-2609 processor (64-bit quad-core at 2.4GHz) with 32GB of memory and two NVIDIA GTX 680 (Kepler) GPUs.
2) Machine 2 - Intel i7 4770K (64-bit quad-core at 3.5GHz), with 32 GB of memory and one Tesla K40c (Kepler) GPU.

With this setup, it is possible to evaluate the performance of the considered solutions in two different heterogeneous systems. In both cases, the computation time was obtained using the Performance API (PAPI) [9]. PAPI specifies an API for accessing the hardware performance counters available on most modern microprocessors. It uses specific signals related to the processor's function to facilitate the correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture, providing reliable time measurements. Matrix sizes (N) ranging 512 to 4096 lines and columns were used. The tile size (B) ranges from 32 to 1024 (when possible) on the performed tests. As the state of the art, the best known and with better performances implementations are considered, namely MKL and ViennaCL. The Intel Math Kernel Library accelerates math processing routines that increase application performance and reduce development time, providing an optimized implementation of linear algebra routines for the CPU. ViennaCL provides an API that uses OpenMP, OpenCL and CUDA backends to map linear algebra routines to heterogeneous systems.

TABLE I
Execution time (μs) of multi-device CPU routines. (In Machine 1)

(a) LU factorization

	B					
N	32	64	128	256	512	1024
512	3489	3548	4322	5325		
1024	3208	4259	5577	6943	7512	
2048	3881	4851	7183	9392	14891	21971
4096	4690	9991	11150	11750	20724	42342

(b) Tournament pivoting

	B					
N	32	64	128	256	512	1024
512	2053	2247	2806	3611		
1024	2249	2815	3532	4761	5021	
2048	2666	3028	4762	6514	8784	13430
4096	3341	6952	7890	8189	14490	29989

B. Performance results

Tables I illustrate the execution times of the CPU factorization and pivoting functions, and tables II depict the execution times for the matrix subtraction, matrix multiplication, triangular solve and LU factorization kernels, used in the multi-device approach, in μs on Machine 1.

The matrix multiplication, matrix subtraction and triangular solve kernels present execution times much lower than the ones seen in the factorization kernel. On the other hand it can be noted that each of these kernels are executed more times than to the factorization step (e.g.: for $N = 2096$ and $B = 128$, there are 16 factorization steps and 120 triangular solve, 120 matrix multiplication steps and 120 matrix subtraction steps). So even with considerably lower execution times, the kernels responsible for the trailing sub-matrix can be accountable for a high percentage of the total execution time. The obtained results suggest that for higher tile values (B), the majority of the execution is spent of the factorization and pivoting steps. For lower tile values, the kernels take the bulk of the execution time. For example, in the multi-device implementation, with $N = 4096$ and $B = 1024$, $53,54\%$ of the execution time is spent on the LU and pivoting steps, while

TABLE II
EXECUTION TIME (μs) OF MULTI-DEVICE GPU KERNELS. (IN MACHINE 1)

(a) Matrix multiplication

	B					
N + B	32	64	128	256	512	1024
512	53	62	88	187		
1024	56	81	132	217	315	
2048	76	124	154	307	626	1464
4096	115	148	207	475	1231	3797

(b) Matrix subtraction

	B					
N + B	32	64	128	256	512	1024
512	25	26	38	80		
1024	31	35	56	93	135	
2048	41	53	66	131	268	628
4096	50	64	89	203	527	1627

(c) Triangular solve

B	Time
32	97
64	117
128	156
256	303
512	855
1024	3110

(d) LU factorization

	Tile					
N	32	64	128	256	512	1024
512	736	1527	4871	17399		
1024	614	1849	6620	27563	88925	
2048	891	2952	11083	45624	185052	698001
4096	1478	5190	19883	86297	365373	1450183

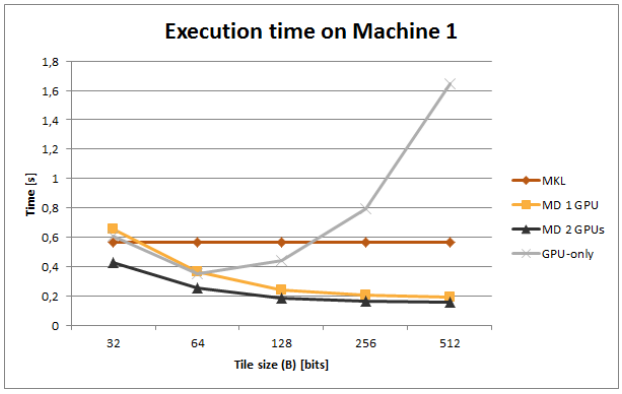


Fig. 13. Execution time for different values of B (Machine 1).

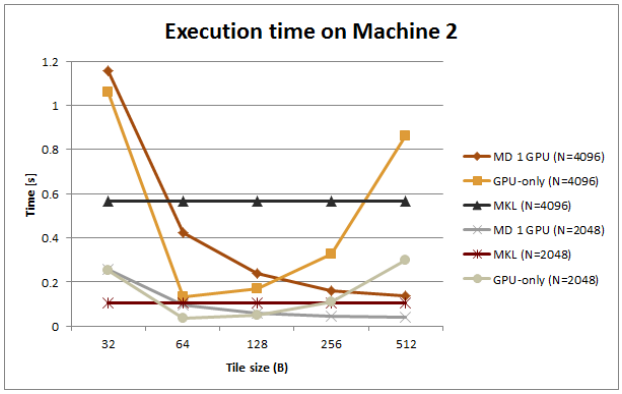


Fig. 14. Execution time for different values of B (Machine 2).

for the same N, with $B = 64$, the factorization step only takes 6% of the execution time.

The effect of the tile size in execution time is illustrated in Figures 13(Machine 1) and 14(Machine 2). These results illustrate the variation of the execution time of the algorithms when varying tile size, on the same matrix (considering $N = 4096$). Note that the execution time of the multi-device is lower when the tile size is higher, while the GPU-only approach achieves the peak performance with a tile size of $B = 64$, suggesting significantly higher execution times for higher tile values. The obtained results for the multi-device algorithm are a consequence of the lower number of operations (factorization, triangular solve, etc.) that are necessary to finish the execution for higher tile values. With fewer operations, there are fewer data transfers that leads to less time spent in communications and synchronizations. The GPU-only approach does not benefit from the lower number of operations, since the entire computation is performed on the GPU, thus not requiring additional data transfers. Moreover, as seen in Figure 4.2, the LU factorization kernel creates a bottleneck for high tile values due to its poor performance, resulting in high execution times. This makes the GPU-only approach more suited for tile values of $B = 64$.

Figures 15 and 16 depict the obtained speed-ups for Machine 1 and Machine 2. Considering the multi-device approach with 1 GPU (LB 1 GPU), the multi-device approach with 2 (LB 2 GPUs) and the GPU-only, approach, all compared with the MKL implementation.

On Machine 1, for the smaller matrices the multi-device performance is not as good as the MKL implementation, but as the matrix size increases the multi-device algorithm's performance surpasses the MKL implementation. These results reflect how well the distribution of the work-load can benefit the LU factorization so long as the input matrices have enough elements to be split into large tasks. As seen in Figure 4.2 the multi-device implementation suggests the best performance when the matrix is broken down into large blocks (high tile values), reducing the communications overhead. As expected this implementation benefits from distributing the trailing sub-matrix computation on the two GPUs, since the work-load for each GPU is reduced, allowing to achieve a speed-up of 3 on Machine 1 with two GPUs, in contrast with the speed-up of 2,3 for when considering only 1 GPU.

The GPU-only approach suggests identical execution times to that of the MKL implementation for lower matrix sizes but surpassing them for larger matrix dimensions. The speed-up of Machine 1 is half of the one achieved when considering the multi-device approach. However, on Machine 2, the GPU-only approach reveals to be faster in all cases. The contrast in the results on the different machines is a consequence of, the lack of a second GPU for the multi-device algorithm to further distribute the load, but, more importantly, the differences in the architectures of the GPUs. The processing units of

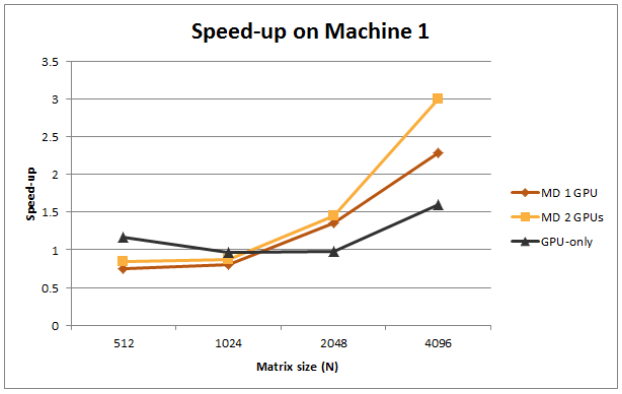


Fig. 15. Speed-up on Machine 1

Machine 2 have higher processing power which leads to better performances, especially on the GPU. As the GPU-only relies mainly on the GPU for computations, its performance benefits the most from the architecture of Machine 2.

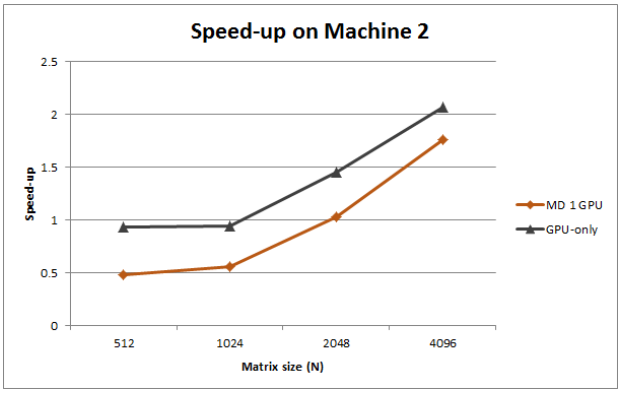


Fig. 16. Speed-up on Machine 2

VII. Conclusion

The main goal of this theses is to developed a load balance solution that can efficiently distribute the workload of linear algebra operations between the CPU and the GPU(s) of an heterogeneous system. The operation target of acceleration is the LU factorization due to its importance in the scientific and numeric fields but also in the LaPACK library, since most of the other routines, such as solving linear systems equations, require the input matrix to be factorized [7].

To do this two different solutions were proposed. The multi-device solution that aims at distributing the workload efficiently on the CPU and the available GPUs. The GPU-only approach is focused on performing the factorization solely on the GPU without the need for constant data transfers during the execution.

The main advantage of multi-device is its scalability. It can easily be mapped to take advantage of different devices with heterogeneous architectures, taking advantages of any CPU cores and/or GPU devices available. The limitations of this approach come from the possible difference in processing power of the different units in the system architecture. Even with the workload distribution done as uniformly as possible the several dependencies between tasks and the requirements for transferring data between devices, may not guarantee a maximal efficient execution and may even lead for some devices to stall while waiting for some data to become available.

Results for both algorithms were obtained in two different machines. The results suggest that both proposed implementations are more suitable for larges matrices than the MKL implementation. On Machine 1 speed-ups of 3 and 1,5 are obtained for the multi-device and GPU-only algorithms, respectively. In Machine 2 the observed speed-ups are of 2 and 1,7 for the GPU-only and multi-device algorithms, respectively. The results show how the different implementation can thrive on different architectures.

When comparing with the Multi-device approach, the GPU-only approach does not have some of the issues that caused a bottleneck for performance in the multi-device implementation. In particular, there is no need for data transfers between host and device during the execution. Also, since the LU factorization step only uses a part of the GPU resources, other tasks can be executed in parallel whenever possible. This implementation also has the benefit of having good performances on weaker CPUs (with the same GPU), since the host is only required to manage the task execution order.

Despite the benefits of this approach, it has its own limitations. The most glaring implications comes with the concept itself, in which the CPU is only using a small portion of its power. Another issue that arises in this implementation is the less efficient factorization step. Since the factorization algorithm is hard to map into the GPU architecture, it is implemented only using a sub-set of all the available threads in the GPU (only uses one CUDA block). Although some of this can be mitigated with the concurrent computation of the trailing sub-matrix, it leads to points where the factorization must finish before the other tasks can be performed, such as what happens in the first iteration.

References

[1] Laura Grigori and Jamws W. Demmel and Hua Xiang, *CALU: A communication optimal LU factorization algorithm*, Society for Industrial and Applied Mathematics, Jan. 2011.

[2] Simplice Donfack and Stanimire Tomov and Jack Dongarra, *Dynamically balanced synchronization-avoiding LU factorization with multicore and GPUs*, Jan. 2013.

[3] Marc Baboulin and Jack Dongarra and Julien Herrmann and Stanimire Tomov, *Accelerating linear system solution using randomization techniques*, 2011.

[4] H. M. D. M. Bandara and D. N. Ranasinghe, *Effective GPU strategies for LU decomposition*, 2011.

[5] Liu Li and Liu Li and Yang Guangwen, *A highly efficient GPU-CPU hybrid parallel implementation of sparse LU factorization*, Society for Industrial and Applied Mathematics, 2012.

[6] Erik Lindholm and John Nickolls and Stuart Oberman and John Montyrm, *NVIDIA TESLA: A unified graphics and computing architecture*, IEEE Computer Society, 2008.

[7] E. Anderson and Z. Bai and C. Bischof and L. S. Blackford and J. Demmel and J. Dongarra and J. Du Croz and A. Greenbaum and S. Hammarling and A. McKenney and D. Sorensen, *LAPACK Users' Guide*, 3rd ed, Aug. 1999.

[8] NVIDIA Corporation, *CUDA C Programming Guide*, docs.nvidia.com/cuda/cuda-c-programming-guide/, Jan. 2015.

[9] The University of Tennessee, *Performance Application Programming Interface*, http://icl.cs.utk.edu/papi/, Jan. 2015.