Dynamically balanced synchronization-avoiding LU factorization with multicore and GPUs

Simplice Donfack*, Stanimire Tomov* and Jack Dongarra*
Email: sdonfack@eecs.utk.edu, tomov@eecs.utk.edu, dongarra@eecs.utk.edu
**Innovative Computing Laboratory, University of Tennessee, Knoxville, USA*

***Abstract*—Graphics processing units (GPUs) brought huge performance improvements in the scientific and numerical fields. We present an efficient hybrid CPU/GPU approach that is portable, dynamically and efficiently balances the workload between the CPUs and the GPUs, and avoids data transfer bottlenecks that are frequently present in numerical algorithms. Our approach determines the amount of initial work to assign to the CPUs before the execution, and then dynamically balances workloads during the execution. Then, we present a theoretical model to guide the choice of the initial amount of work for the CPUs. The validation of our model allows our approach to self-adapt on any architecture using the manufacturer's characteristics of the underlying machine. We illustrate our method for the LU factorization. For this case, we show that the use of our approach combined with a communication avoiding LU algorithm is efficient. For example, our experiments on a 24 cores AMD opteron 6172 show that by adding one GPU (Tesla S2050) we accelerate LU up to $2.4\times$ compared to the corresponding routine in MKL using 24 cores. The comparisons with MAGMA also show significant improvements.**

***Keywords*-LU factorization, hybrid CPU/GPU programming, synchronization avoiding.**

I. Introduction

General purpose graphics processing units (GPGPUs) brought a huge performance acceleration to the scientific and numerical fields. Since their appearance, many applications were successfully readapted in order to take into account the advantages that GPGPUs offer. However, even if the GPUs' effectiveness is well established, their efficient usage depends on the ability to fully exploit them in a heterogeneous CPU/GPU environment while doing a good balancing of workload.

The LU factorization is one of the most important algorithms in the scientific and numerical fields, and one of the most difficult to parallelize because of its irregular data movements introduced by the pivoting. Its block version partitions the matrix to factorize into blocks of columns, and then factorizes each block of columns by steps. At each step of the factorization a block of columns, referred to as a panel, is factorized and the corresponding trailing submatrix is updated. This decomposition allows to separate the panel factorization (which is not efficiently parallelizable) and the updates. These updates are based on matrix products, and therefore can be performed by optimized Level 3 BLAS [1] operations, which are easily parallelizable. The approach used in MAGMA [3], a well known and optimized numerical library in linear algebra for GPUs, is based on this principle. It factorizes the panel on the CPUs and performs the update of the trailing submatrices on the GPUs. This ensures efficient updates and optimal use of the GPUs. However, performing a prefixed amount of work on the CPUs (a panel at each iteration), even when the power of the CPUs available is comparable to that of the GPUs, is inefficient because the performance of a panel factorization is data-bound. Thus, the performance potential/scalability that is expected when growing the number of CPUs, will be lost. This lack of load balancing can therefore lead to strong underutilization of the CPUs available.

We focus on the LU factorization because of the constraints related to the synchronization of the processes that are involved during the panel factorization. For this case, the load balancing problem has been well studied by several authors [19], [18], [13], [17]. For most implementations, the main idea is to determine empirically the amount of work to assign to the different computational units, or perform some necessary adjustments depending on the problem size in order to keep CPUs busy. Unfortunately, these approaches require high efforts for tuning. Also, they are difficult to evaluate, and therefore not portable on a variety of architectures. To our knowledge, none of these approaches propose a realistic model to guide the load balancing between CPUs and GPUs.

In this paper, we propose a new efficient and portable approach that balances the load between the CPUs and the GPUs in numerical algorithms. In particular, we developed a theoretical model for determining the amount of work to assign to each computational unit to ensure the scalability of our algorithm. First, our approach determines the initial amount of work to assign to the CPUs before the execution. Then during the execution, a part of work is dynamically transferred from the GPUs to the CPUs in order to maintain load balance. The data transfers associated with this dynamic load balancing are asynchronous and overlapped with computations. Our model and implementation self-adapts to any architecture by using underlying machine characteristics, and it does not require further tuning.

We use the LU factorization to illustrate our load balancing method, but we believe that our approach can be adapted to several other algorithms in numerical linear algebra such as the QR factorization, the SVD decomposition, and Cholesky factorization. We propose to use CALU [6], [11], [8], one of the algorithms recently introduced for the LU factorization, which aims to minimize communications during the factorization of the panel by doing some redundant computations. There are two motivations of using CALU to factorize the panel in our work: first, CALU allows an efficient parallelization of the panel, which is suitable for the CPUs in this hybrid approach; Second, we remove the bottleneck from our prior work [4] where we did not obtain expected improvements for square matrices by just replacing the panel factorization in MAGMA by CALU while keeping the rest of the code unchanged.

Our new approach removes the bottleneck in the standard MAGMA implementation by keeping CPUs busy without changing the panel size, but by giving more updates to the CPUs asynchronously. We tested our approach on an AMD opteron 6172 equipped with Tesla S2050 GPUs, an AMD opteron 6180 equipped with Tesla 6180 GPUs, and an Intel xeon E5-2670 equipped with Tesla M2090 GPUs. Our implementation, tested on a single Tesla S2050 GPU with up to 48 CPU cores, is multicore scalable. In particular, it is up to $2\times$ faster than MAGMA, and up to $1.9\times$ faster than our prior implementation. A use of a single Tesla S2050 GPU accelerates the performance by $2.7\times$ over the corresponding CPU routines of MKL using the same number of cores.

The rest of this paper is organized as follows. In section II, we briefly introduce the LU factorization, its implementation in MAGMA, related work, and communication avoiding algorithms. In section III, we present our new approach. Then, in section IV, we present our simple and yet very accurate model to guide the initial amount of work to transfer to the CPUs before the factorization. In section V, we present experimental results, we show the modeled workload for underlying architecture, performance results, and the scalability of our implementation. Finally, in section VI, we give conclusions and future work directions.

II. Background

A. LU factorization

The current standard for an LU factorization is the one implemented in LAPACK [2]. This is the function GETRF, prefixed with an S, D, C, or Z, representing the arithmetic to be used (correspondingly single real, double real, single complex, or double complex floating point arithmetic). GETRF computes the LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges, which is stable in practice. Moreover, the algorithm is known as blocking, in a sense that a block of columns, referred to as panel, is factored at a time. The transformations used during the panel factorization are not applied immediately on the trailing matrix (i.e., the rest of the matrix after the current panel), but delayed and applied at once after the panel is factored. Thus, the algorithm groups together a block of inefficient (memory-bound) Level 2 BLAS transformations and applies them at once as Level 3 BLAS updates, which can be done very efficiently on current architectures.

B. Communication avoiding LU factorization

It is well known that hardware improvements in terms of communications are very low compared to the ones in terms of arithmetic operations. Based on these observations, a new class of algorithm referred to as communication avoiding algorithms [6], [11], whose main idea is to reduce communications by doing some redundant computations, have been introduced. For LU factorization, a communication avoiding algorithm LU (CALU) performs the panel factorization as a block operation. This is because, classic algorithms as implemented in ScaLAPACK for example, factorize the panel column by column. Hence, pivoting requires communication among processors participating in the operations for each column. So, at least $b \log P$ messages are exchanged during this step, where b is the number of columns in the panel being factorized and P is the number of processors. CALU introduces a new pivoting strategy referred to as TSLU, which performs the panel factorization in two steps. First, it identifies the good pivot at a reduced communication cost and then applies the corresponding permutation matrix to the panel. Second, it applies LU factorization without partial pivoting on the panel. By using a binary tree for a reduction operation through this approach, only $\log P$ messages are exchanged, which is less than ScaLAPACK number of messages with a factor of b. The algorithm is well described in [11], [8]. Once a panel is factorized using TSLU, the update of the trailing submatrix is updated as in classic LU factorization.
CALU is shown to be stable in practice [11] and has been adapted to distributed memories [11], [6] and multicore environments [8], [7].

C. LU factorization implementation in MAGMA

MAGMA[3] is a well optimized numerical library which implements several LAPACK routines for dense matrices. The library takes advantage of the hybrid CPU/GPU programming to achieve better performance. The key principle lies in the optimal use of the GPUs in these routines. To do so, the highly parallel part of each routine is identified and scheduled on the GPUs, while the part with less parallelism is scheduled on CPUs. For the LU factorization, it is shown that the panel factorization is more suitable for the CPUs than for the GPUs. What is implemented in MAGMA for the LU factorization embraces this principle, the panel is factorized on the CPUs using a multithreaded routine such as dgetrf from the MKL vendor library[14], while the update of the trailing submatrices is performed on the GPUs using highly optimized kernels.

Figure 1 shows an example of execution of the dgetrf routine as implemented in MAGMA. This example is shown for a matrix decomposed into 5 block columns using P processors and 1 GPU. The bars at the top represent CPUs computations, while the bars at the bottom represent GPU computations. Red and green bars show the part of the matrix where current operations are performed. As shown in the figure: at the first step (step 0), the CPUs factorize the panel. At step 1, the factorized panel is transferred from the CPUs to the GPU. At step 2, the first column of the trailing submatrix is updated separately on the GPU in order to enable the look-ahead. At step 3, the updated single column of the previous step is transfered from the GPU to the CPUs. This step makes available the next panel factorization for the CPUs, and finally at step 4, the GPUs update the rest of the trailing submatrix associated with the current panel, while the CPUs factorize the next panel.
The same process is repeated from step 1 to 4 until the matrix is entirely factorized.
We note that, at step 4, the CPUs are likely to finish their computations very early because of the relatively small size of the panel compared to the size of the trailing submatrix. This early end of the CPUs work prevents inactivity on the GPU, because the panel is likely to be ready when the GPU finishes its computations, but it may result in important inactivity for the CPUs.

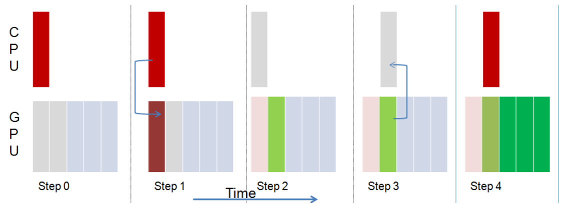

Figure 1. Example of execution of magma_dgetrf on a square matrix. The matrix is partitioned into 5 x 5 blocs of columns.

D. Related work

The load balancing problem for hybrid CPU/GPU applications is well recognized in the literature and presents a challenge because of the heterogeneity of the CPU/GPU systems. The approach used in MAGMA leads to optimal use of the GPUs at the price of possible CPUs underutilization. Based on this approach, the work performed by the CPUs is relatively small to prevent GPUs idle time. The fact that this can create load imbalance was identified by several authors in the literature, and several approaches have been suggested in order to solve the problem and improve performance. In this section, we detail some of them. In particular, we focus on the LU and QR factorizations.

To solve the possibility of load imbalance in the QR factorization Volkov et al. [19], and more recently Yaohung et al. [18], proposed an approach based on variable panel block size. It consists of choosing an adapted block size for the panel at each step of the factorization in order to keep CPUs busy. The approach would have been optimal if the CPUs and the GPUs ended their work simultaneously, but this is not possible in general, as mentioned by Yaohung et al. [18]. Instead, they merely search the largest panel size for the CPUs for which the GPUs' execution is not impacted. The validation of the results is based on empirical tests. Furthermore, if a large panel size is used in order to achieve a good load balancing, an additional time due to bandwidth limitations may be paid for transferring it to the GPUs. This restriction represents an upper bound for the panel size.

In our previous work [4], we have evaluated the possibility of replacing the standard panel factorization in MAGMA by CALU, which is a more efficient approach. The idea behind this implementation was to efficiently factorize the panel and then slightly increase the panel block size to keep the CPUs busy. Once the panel size is increased, it is split into relatively small tasks in order to enable enough parallelism for the whole set of cores participating in the operation. We observed that this approach gave better results for tall and skinny matrices involving the CPU/GPU, but unfortunately the performance improvements for square matrices were not significant.

Horton et al. [13] noted the ineffectiveness of using too many threads for the panel factorization in the QR factorization in MAGMA. To improve the efficiency of the CPUs, they determine the optimal number of threads required for the panel factorization, then they dedicate a fixed rightmost part of the matrix to the remaining threads for the updates. This approach introduces several new parameters which appear further difficult to tune such as: the panel width, the number of threads working on the panel, the size of the rightmost part of the matrix dedicated to the remaining threads and the inner block size of the panel. The author suggests an approach based on extensive experiments to compute the possible values of these parameters for a range of matrices in order to reuse them later, which may require up to two hours of computations as also mentioned by the authors.

Song et al. [17] proposed an approach based on the decomposition of the input matrix into tiles of size B. These tiles are cyclically distributed to the GPUs in ScaLAPACK fashion. A portion, B_h, of each tile is assigned to the CPUs. First, they use a formula to determine B_h, and then they simulate the execution of the routine (the QR or Cholesky factorization) on a small number of tiles (e.g., 3 tiles in practice) in order to readjust the value of B_h for the global matrix. Another empirical search is required to determine the tile size B which achieves best performance for the GPU kernel.Such empirical experiments are required before the execution of any of their routines.

III. Asynchronous LU factorization with GPU accelerators

A. Method

Algorithm 1 describes our dynamically balanced synchronization-avoiding LU factorization. A general matrix A of size $m \times n$ is partitioned into blocks of columns, and factored iteratively, similar to the block LAPACK algorithm. To establish an initial load balance between the CPUs and the GPU, A is split into two parts. The first part formed with the first d block columns of A is transfered to the CPUs, while the second part formed with the remaining $N - d$ block columns, where N represents the number of block columns, remains on the GPU as presented in figure 2. The input parameter d is determined using our model that we describe in section IV. The first part is further partitioned in a 2-dimensional (2D) way; each block column is 1D partitioned into P_r blocks of rows, where P_r is the number of CPU cores participating in a panel factorization. The structure of the second part of the matrix is kept unchanged.

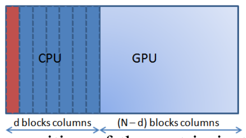

Figure 2. Initial decomposition of the matrix in two parts. First part for the CPUs and second for the GPUs.

This decomposition creates fine and coarse grain computational tasks, correspondingly in the first and second part of A. There are two advantages for this decomposition. First, the GPUs are massively parallel and can be used efficiently on regular computations like the matrix-matrix product updates in the second part of the matrix [15], [16]; while the CPUs can better handle (than the GPUs) less parallel and irregular computations, and therefore are more suitable for operations such as the panel factorizations (in the first part). Second, the use of fine grain computational tasks is associated with increased parallelism, while the use of coarse grain tasks is associated with high efficiency. By combining and properly scheduling both fine and coarse grain tasks, we achieve balance and hardware efficiency; the panel factorization (which is the critical path of the algorithm) is fine grain partitioned in order to be accelerated through parallelism, while the update of trailing submatrix (which is the bulk of the computation) is coarse grain partitioned in order to exploit the resources.

Before the factorization, d block columns of the matrix are transfered (line 3 of the algorithm) from the GPU to a workspace allocated on the CPUs. Then, for each step K of the factorization, the algorithm proceeds as follows:**Lines** 5 **to** 7**:** A panel is decomposed into P_r tasks and each task is inserted in the CPUs' queue of tasks. The associated routine is calu_dgetrf which factorizes a portion of the panel and then performs some reduction steps conducted by a binary tree as described in [11], [8]; **Line** 8**:** The new factored panel is asynchronously transfered to the GPU; **Lines** 9 **and** 10**:** A dtrsm and a dgemm task on the coarse part of the matrix are inserted in the GPU's tasks queue. The associated routines are gpu_dtrsm and gpu_dgemm, respectively; **Line** 11**:** A new block column is asynchronously transfered to the CPU to balance work; **Lines** 12 **to** 15**:** dtrsm and dgemm tasks are inserted in the CPUs' tasks queue. This is done for each block column in the CPUs part. The associated routines are respectively cpu_dtrsm and cpu_dgemm. At the end of the factorization, the resulting matrix is stored entirely on the GPU memory.

Algorithm 1 Asynchronous CALU

```
 1: Input: $m \times n$ matrix $dA$, block size $b$, number of processors $P = P_r \times P_c$,
    number of blocs for the CPU parts $d$
 2: $M = m/b, N = n/b$
 3: workspace: $A = dA(1 : m, 1 : d * b)$   /*part of the matrix for the CPUs*/
 4: for $K$ = 1 to $N = n/b$ do
 5:     for $I$ = 1 to $P_r$ do
 6:         cpu_insert_task(calu_dgetrf, $A(K + I.\frac{M-K}{P_r} : K + (I + 1).\frac{M-K}{P_r}, K)$)
 7:     end for
 8:     gpu_insert_task(device_setMatrix, $A(K : M, K)$, $dA(K : M, K)$)
        /*Copy the factored panel from CPU to the GPU*/
 9:     gpu_insert_task(gpu_dtrsm, $A(K, K + d : N)$)
10:     gpu_insert_task(gpu_dgemm, $A(K + 1 : M, K + d : N)$)
11:     gpu_insert_task(device_getMatrix, $dA(K + 1 : M, K + d)$, $A(K + 1 : M, K + d)$)   /*Copy one column from the GPU to the CPU*/
12:     for $J$ = $K$ to $K + d$ do
13:         cpu_insert_task(cpu_dtrsm, $A(K, J)$)
14:         cpu_insert_task(cpu_dgemm, $A(K + 1 : M, J)$)
15:     end for
16: end for
```

B. Scheduling

Our algorithm, as described above, creates and inserts tasks in both a CPUs or GPU' queue of tasks. These tasks must be executed as soon as their data dependencies allow it. We use dynamic scheduling to map the tasks in the CPUs' queue to threads, while tasks in the GPU' queue are executed by the GPU. The main goal of the scheduler is to check the dependencies of the tasks in the queue of tasks and schedule them to the available threads as soon as they are ready to be executed.

Figure 3 shows an example of the direct acyclic graph (DAG) resulting from the insertion of tasks and their execution on a matrix partitioned into 5 column blocks, where 2 of the blocks are initially assigned to the CPUs. The figure represents the different tasks and the dependencies among them. Each circle represents a task and each arrow represents the dependency between two tasks. For simplicity, we represent all P_r tasks created by the panel decomposition as one task, that is, the task **P** colored in red in the figure. The **U** (colored in purple) and **S** (colored in green) DAG vertices in figure 3 represents respectively, the dtrsm and dgemm tasks of the algorithm. The GPU part of the computation is represented by rectangular areas in the same figure. We represent tasks inside these rectangular areas to show the parts of the matrix which are detached from the GPU and sent to the CPU dynamically during the execution. Dashed arrows in the figure represent the transfer of data, orange arrows show CPU to GPU panel transfers while blue ones represent GPU to CPU block column transfers.

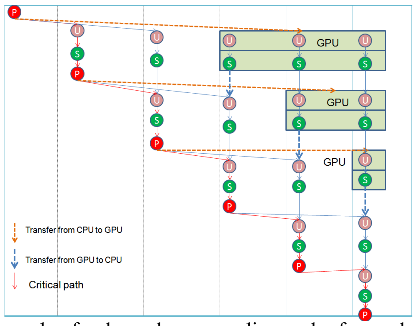

Figure 3. Example of a dependency acyclic graph of asynchronous CALU on a square matrix. The matrix is partitioned into 5 blocs of columns. The CPUs part is formed by 2 blocs of columns, i.e., 40% of the initial matrix.

C. *Runtime*

Figure 4 shows an execution of our algorithm for a matrix partitioned into 7 block columns, where 3 block columns are assigned to the CPUs before the factorization. The initial decomposition is shown in **step** 0 of the figure. At that step the CPUs factorize the panel (represented by a red bar) using P_r threads. **At step** 1**:** One thread initiates a transfer of the panel to the GPU, while the other $P - 1$ threads start the update of the CPUs part of the trailing submatrix. This illustrates how our approach overlaps computations and communications. **At step** 2**:** The GPU starts the update of its corresponding part of the trailing submatrix, while the next P_r available threads may start a new panel factorization, and the other $P - Pr$ threads continue to perform the update of the CPU's trailing submatrix. With this approach, each panel is always factorized in advance, so to avoid GPU stall. **At step** 3**:** The algorithm performs as in step 1, with the difference that a new block of columns is transfered from the GPU to the CPUs. This transfer is asynchronous and helps to equilibrate work with CPUs in order to replace the panel which is being sent from the CPUs to the GPUs. This step shows one of the best features for our algorithm: overlapping communications in both directions with computations. We note that, the time it takes to transfer a block of data from the CPUs to the GPUs differs from the time it takes to perform computations. This step illustrates only how communications are overlapped with computations. As our algorithm is asynchronous, these computations may represent the ones introduced either by the panel from step 0 or by the new factorized panel at step 2.
This process is repeated from step 2 to 3 on the remaining part of the trailing submatrix until the factorization is completed. When the remaining matrix became very small, the GPU is completely removed from the execution and does not participate anymore in the computation. There are two reasons for doing that: first, when the matrix becomes very small, the GPU does not achieve better performance; and second, when there are not enough computations to overlap communications, the time to transfer data between the CPUs and the GPU would dominate the time to simply perform the computation associated with these data on the CPUs.

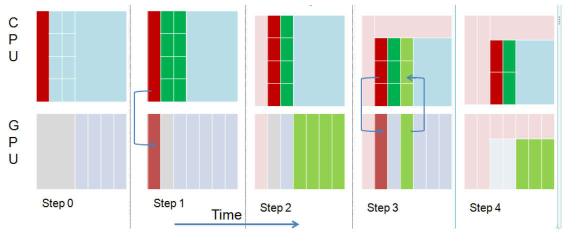

Figure 4. Example of an execution of the asynchronous CALU on a square matrix. The matrix is partitioned into 7 block columns. The CPUs' part is formed by 3 blocs of columns, i.e., 42% of the initial matrix.

IV. Performance model

The initial number of block columns for the CPUs is crucial for performance, if it is relatively small then at least some CPUs will become idle, and if it is too large then the GPU will become idle. The main question is how to determine the optimal number of block columns for each part. We propose a simple, yet accurate model for determining the percentage of the matrix to assign to the CPUs so that the work between the CPUs and the GPU is balanced. Our approach incorporates a trade-off between simplicity and accuracy. The goal is to implement an algorithm that self-adapts on the underlying architecture and achieves better performance.

Let A be the matrix of size $m \times n$ to be factored, b be the block size, and P be the number of processors in the CPUs. The block LU algorithm partitions the matrix A into $M \times N$ block columns, where $M = m/b$ and $N = n/b$. Also, let $\mathbf{d}$ be the initial number of block columns for the CPUs' part of the matrix. We denote by $\mathbf{g_1}$ and $\mathbf{g_2}$ the peak performances of one CPU and one GPU, respectively. These parameters correspond to the maximum number of operations per second which can be executed by one CPU or one GPU, respectively, and are indicated by the architecture's manufacturers.

At each step K of the factorization, we consider $\mathbf{W_1}$ to be the total amount of work required to complete the CPUs part, and $\mathbf{T_1}$ to be the theoretical minimum time to do so. Similarly, let $\mathbf{W_2}$ be the total amount of work required to complete the GPU part, and $\mathbf{T_2}$ be the theoretical minimum time to do so. $\mathbf{M_K}$ and $\mathbf{N_K}$ are taken to be, correspondingly, the number of block rows and block columns in the entire trailing submatrix at the corresponding step.

As shown in figure 2, at each step K of the factorization the CPUs factor one panel and perform $d-1$ updates of the trailing submatrix in the first part of the matrix. Therefore, W_1 can be computed as: $W_1 = W_{1panel} + (d-1)W_{1update}$, where W_{1panel} and $W_{1update}$ are the amount of work required to factor and update one block column, respectively. At the same step, the GPU performs the update of its corresponding part of the trailing submatrix in the second part of the matrix. Therefore, W_2 can be computed as: $W_2 = (N_K - d)W_{1update}$. By definition, we can easily deduce that $T_1 = \frac{W_1}{P \times g_1}$ and $T_2 = \frac{W_2}{g_2}$.

For a synchronous algorithm, the threads in the CPU parts and the GPU synchronize at each step of the factorization, if T_1 is smaller or larger than T_2, then at least some CPUs or the GPU will become idle. Therefore, the goal is to have $T_1 = T_2$.

For an asynchronous algorithm, the threads in the CPU parts may switch immediately to the next step $(K + 1)$ thanks to the look-ahead, both CPUs and GPU may continue working even if the GPU is still updating the trailing submatrix corresponding to step K. In that situation, new tasks would be inserted in the GPU' queue without impacting its execution. Contrary to the synchronous case, if T_1 is lower than T_2, the threads will continue to insert tasks in the GPU' queue. In that situation, the CPUs may stall only if they compute the entire factorization of the d columns while the GPU is still updating the trailing submatrices of previous steps. On the other hand, if T_1 is very large, then the GPU will complete its computations early and become inactive while waiting for some panels to be factorized. So, to avoid GPU idle time, we must split the work so that $T1 \leq T2$.

Regardless of the algorithm being synchronous or asynchronous, by solving $T_1 = T_2$, we determine the optimal number of block columns to assign to the CPUs. In particular, the following relations hold $\frac{W_1}{P \times g_1} = \frac{W_2}{g_2}$
$\Rightarrow g_2(W_{1panel} + (d-1)W_{1update}) = Pg_1(N_K - d)W_{1update}$
By solving for d, we obtain the relation: $d = \frac{Pg_1 N_K + g_2}{Pg_1 + g_2} - \frac{W_{1panel}}{W_{1update}} \frac{g_2}{Pg_1 + g_2}$.

A well known approximation for the number of operations required to perform LU factorization on one block column of size $m_K \times b$ is $W_{1panel} = (m_K - \frac{b}{3})b^2$. The number of operations to update one block column of size $m_K \times b$ in the trailing submatrix is a rank-b update, so $W_{1update} = m_K b^2$. Hence, $\frac{W_{1panel}}{W_{1update}} = \frac{(m_K - \frac{b}{3})b^2}{m_K b^2} = \frac{m_K - \frac{b}{3}}{m_k}$. If CALU is used for the panel factorization, the number of flops required for the panel (W_{1panel}) is increased by $O(b^3 \log P_r)$, where P_r is the number of processors participating in the panel factorization. This results in an increasing of the fraction $\frac{W_{1panel}}{W_{1update}}$ by $\frac{O(b^3 \log P_r)}{W_{1update}} = \frac{O(b^3 \log P_r)}{m_K b^2} = \frac{O(b \log P_r)}{m_K}$ and becoming $\frac{W_{1panel}}{W_{1update}} = \frac{m_K - \frac{b}{3} + O(b \log P_r)}{m_K}$. Since the block size b is much smaller compared to the matrix size m_K, the fraction $\frac{m_K - \frac{b}{3} + O(b \log P_r)}{m_K}$ will tend asymptoticly to one, for which case we can obtain an expression $d = \frac{Pg_1 N_K + g_2}{Pg_1 + g_2} - \frac{g_2}{Pg_1 + g_2} = \frac{Pg_1 N_K}{Pg_1 + g_2}$, and finally,
$\frac{d}{N_K} = \frac{Pg_1}{Pg_1 + g_2}$.(1)

Here $\frac{d}{N_K}$ represents the largest percentage of the matrix for the CPUs' part at iteration K of the factorization. The expression $\frac{Pg_1}{Pg_1 + g_2}$ does not depend on the matrix size, and it can be used to determine the percentage of the matrix for the entire computation. The model suggests that the CPUs' part (d) varies with the number and the peak performance of the CPUs, and the peak performance of the GPU.

Communications between CPUs and GPUs are irrelevant for our model because the model aims to balance computations between CPUs and GPUs such that both finish their computations at the same time at each step of the factorization. At the end of each step, the CPUs and GPUs then exchange data before moving to the next step. For synchronous algorithm, these communication time have an impact on the overall factorization time, while for asynchronous algorithm they are hidden by the computations of the next steps. Hence, in both case they do not have any significant impact on the estimation given by the model.

V. Experiments

In this section, we evaluate the performance of our algorithm on three different machines running on linux. The two first machines use a four-socket, twelve core configuration based on an AMD Opteron processors with a Tesla S2050 GPU, and the third machine uses a two-socket, eight core configuration based on an Intel Xeon E5-2670 processor with a Tesla M2090 GPU. The processors' frequency and the peak performance of each machine is shown in Table I. We refer to magma_dgetrf as the routine implemented in MAGMA 1.3, which performs LU factorization using the indicated number of cores and one GPU. Magma_calu_sync refers to our previous implementation[4], that is, the approach which consists of replacing the standard panel factorization in magma_dgetrf with CALU, and for which the tuned panel block size is chosen to achieve best performance. Calu_async refers to our new implementation, additional parameters are indicated in brackets. MKL_dgetrf refers to LU factorization with partial pivoting routine, implemented in MKL using the indicated number of cores. We use MKL 11.1.069 to compute MKL_dgetrf results. MAGMA and CALU are linked with the BLAS version in MKL.

A. Performance and scalability

1) Performance: We present the performance of calu_async with the parameter d estimated using our model. Figure 5 shows the performance on an AMD Opteron 6180 with 48 cores and one GPU. For very small matrices, magma_dgetrf is better than magma_calu_sync, while for larger matrices, magma_calu_sync becomes slightly better but with no more than 5% improvement. Magma_dgetrf is better than MKL_dgetrf but the performance of MKL keeps scaling because it fully exploits the number of cores available. For very small matrices $(M = N \leq 4032)$, the performance of calu_async is close to magma_dgetrf but not

CPUs			GPU		CPUs + GPU
Processor Model	Single core frequency	Total peak performance (double precision)	Model	Peak performance (double precision)	Peak performance (double precision)
AMD Opteron 6172	2.1 Ghz	403.2 GFlops/s	Tesla S2050	504 GFlops/s	907.2 GFlops/s
AMD Opteron 6180	2.5 Ghz	480.0 GFlops/s	Tesla S2050	504 GFlops/s	984.0 GFlops/s
Intel Xeon E5-2670	2.6 Ghz	332.8 GFlops/s	Tesla M2090	665 GFlops/s	997.8 GFlops/s

Table I
PEAK PERFORMANCE OF EACH MACHINE IN OUR TEST SET.

better. This because of the overhead caused when scheduling very few number of tasks on large number of cores. For $M = N \geq 5184$, calu_async outperforms magma_dgetrf and magma_calu_async. The best improvement is obtained for $M = N = 11008$, where calu_async is $2\times$ faster than magma_dgetrf and magma_calu_sync, and also $2.7\times$ faster than MKL_dgetrf. For the largest matrix in our test, that is, $M = N = 18048$, calu_async reaches 503 GFlops/s, which represents 51% of the total peak performance (CPUs + GPU), while magma_dgetrf reaches 276 GFlops/s which represents 28% of the total peak performance. Figure 6

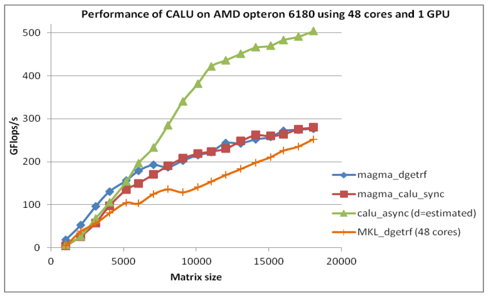

Figure 5. Performance for square matrix using 48 AMD Opteron 6180 cores and one S2050 GPU.

shows the performance on Intel Xeon E5-2670, we observe that calu_async is up to $1.5\times$ faster than magma_dgetrf and $2\times$ faster than mkl_dgetrf, for $M = N = 13056$. For $M = N = 18048$, it achieves 489 GFlops/s, while magma_dgetrf achieves 336 GFlops/s, which corresponds, respectively, to 49% and 33% of the total peak performance. Figure 7 shows the performance on high-end CPUs (Sandy

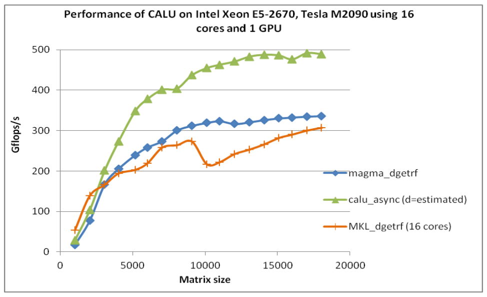

Figure 6. Performance for square matrix using 16 Intel Xeon E5-2670 cores and one M2090 GPU.

Bridge; 16 core @2.6GHz, DP Peak 332 GFlop/s), GPUs (K20c; DP Peak $1,174$ GFlop/s), and Intel Xion Phi (DP Peak $1,046$ GFlop/s). We observe that our implementation using one Intel MIC behaves a good as the one using one NVIDIA K20c GPU having approximately the same peak performance in double precision.

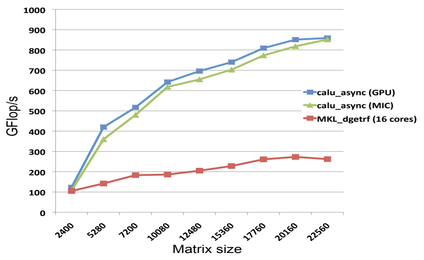

Figure 7. Asynchronous CALU on high-end CPUs (Sandy Bridge), GPUs (K20c), and Intel Xion Phi.

2) Scalability: Figure 8 shows the scalability of calu_async vs. magma_dgetrf and magma_calu_sync on the AMD Opteron machine. We observe that when increasing the number of processors, magma_dgetrf does not scale. On the same problem size, magma_calu_sync increases the performance by 5% compared to magma_dgetrf, but it does not scale either, while calu_async scales very well. The same behavior is also observed on the Intel Xeon machine.

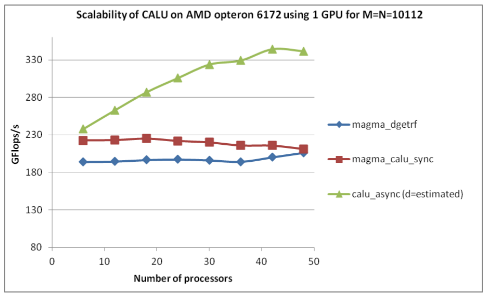

Figure 8. Scalability of CALU with modeled parameters vs. magma_dgetrf and magma_calu_sync.

VI. CONCLUSION

In this paper, we have introduced a new LU factorization approach for hybrid CPU/GPU systems which balances work between CPUs and the GPU. The main contribution of this work was to propose an approach that can self-adapt on any architecture by using its manufacturer's peak performances. We suggested a simple and yet accurate model to compute the initial amount of work for the CPUs. The advantage of our model is that it can be easily extended to several others approaches in dense linear algebra. We have used CALU for the panel factorization because of its possibility to parallelize the panel, but also because of the increasing popularity of such a class of algorithms. On AMD opteron and Intel Xeon machines in our test set, our experiments show that our algorithm is faster and scalable compared to the corresponding standard routine in MAGMA and our previous implementation of CALU for GPUs presented in [4].

This work has several directives. First, we plan to extend our implementation to multi-GPUs. This can be done easily by broadcasting each computed panel to the GPUs and by transferring dynamically each block column from the appropriate GPU to the CPUs during the factorization. Second, we will implement the same approach with the classic partial pivoting; to do so, we plan to use a technique such as parallel recursive LU factorization [12], [9]. Third, our work is being incorporated into MAGMA. Finally, we plan to extend this concept to several other dense algebra routines such as QR, CAQR, cholesky, eigensolvers, and much more.

Acknowledgment

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] Basic linear algebra subprogram. http://www.netlib.org/blas/.

[2] LAPACK. http://www.netlib.org/lapack/.

[3] Magma. http://icl.cs.utk.edu/magma/.

[4] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov. A class of communication-avoiding algorithms for solving general dense linear systems on cpu/gpu parallel machines. *Procedia Computer Science*, 9:17–26, 2012.

[5] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey. Designing and dynamically load balancing hybrid lu for multi/many-core. *Computer Science-Research and Development*, 26(3-4):211–220, 2011.

[6] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Implementing communication-optimal parallel and sequential qr factorizations. *Arxiv preprint arXiv:0809.2407*, 2008.

[7] S. Donfack, L. Grigori, W. Gropp, and V. Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 496–507. IEEE, 2012.

[8] S. Donfack, L. Grigori, and A. Gupta. Adapting communication-avoiding lu and qr factorizations to multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[9] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Exploiting fine-grain parallelism in recursive lu factorization. In *International Conference on Parallel Computing*, 2011.

[10] N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3. IEEE Computer Society, 2005.

[11] L. Grigori, J. Demmel, and H. Xiang. Communication avoiding gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 29. IEEE Press, 2008.

[12] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

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

[14] Intel. Math kernel library (mkl). http://www.intel.com/software/products/mkl/.

[15] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2010. Springer.

[16] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.

[17] F. Song, S. Tomov, and J. Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *ICS*, pages 365–376, 2012.

[18] Y. Tsai, W. Wang, and R. Chen. Tuning block size for qr factorization on cpu-gpu hybrid systems. In *Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium on*, pages 205–211. IEEE, 2012.

[19] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.