Dynamically balanced synchronization-avoiding LU factorization with multicore and GPUs

Simplice Donfack
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
sdonfack@eecs.utk.edu

Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
tomov@eecs.utk.edu

Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
dongarra@eecs.utk.edu

ABSTRACT

Graphics processing units (GPUs) brought huge performance improvements in the scientific and numerical fields. We present an efficient hybrid CPU/GPU computing 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 high-end hybrid CPU/GPU systems show that our dynamically balanced synchronization-avoiding LU is both multicore and GPU scalable. Comparisons with state-of-the-art libraries like MKL (for multicore) and MAGMA (for hybrid systems) are provided, demonstrating significant performance improvements. The approach is applicable to other linear algebra algorithms. The scheduling mechanisms and tuning models can be incorporated into respectively dynamic runtime systems/schedulers and autotuning frameworks for hybrid CPU/MIC/GPU architectures.

Categories and Subject Descriptors

1. [**Applications**]: high performance numerical algorithms; 2. [**Programming Systems**]: solutions for parallel programming challenges—*scheduling, load balancing, hybrid programming, modeling*

Keywords

LU factorization, hybrid CPU/GPU programming, synchronization avoiding

1. INTRODUCTION

GPUs 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 GPUs 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 [20, 19, 15, 12, 18, 14]. 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 difficult to ensure their performance portability 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 [7, 11, 9], 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 [5] where we did not obtain expected improvements by just replacing the panel factorization in MAGMA by CALU while keeping the rest of the code unchanged. In fact, in this prior implementation, we used CALU to factorize the panel in MAGMA, and then we increased the panel size in order the keep the CPUs busier while the GPUs perform the updates of the trailing submatrices. First, we noticed that this approach was very difficult to tune; second, the panel size was severely majored by the bandwidth transfer rate between CPUs and GPUs, still leading to unbalanced work with the best implementation we obtained. So, faster factorization of the panel alone was not enough to maintain the scalability.

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 with Tesla S2050 GPUs, AMD Opteron 6180 with Tesla S2050 GPUs, and an Intel Xeon E5-2670 with Tesla M2090 GPUs, as well as on the latest NVIDIA Kepler (K20c) and Intel Xeon Phi architectures. 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× faster than MAGMA, and up to 1.9× faster than our prior implementation. A use of a single Tesla S2050 GPU accelerates the performance by 2.7× over the corresponding CPU routines of MKL using the same number of cores.

The rest of this paper is organized as follows. In section 2, we briefly introduce the LU factorization, its implementation in MAGMA, related work, and communication avoiding algorithms. In section 3, we present our new approach. Section 4 presets our model to guide the initial amount of work to transfer to the CPUs before the factorization. In section 5, we present experimental results, we show the modeled workload for underlying architecture, performance results, and the scalability of our implementation. Finally, in section 6, we give conclusions and future work directions.

2. BACKGROUND

2.1 LU factorization

The current standard for an LU factorization is the one implemented in LAPACK [2, 4]. 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.

2.2 LU factorization in MAGMA 1.3

MAGMA[3] is a well optimized numerical library which implements LAPACK routines for dense matrices on hybrid CPU/GPU systems. 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 has been shown that the panel factorization is more suitable for the CPUs [6] 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[13], 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, the routine proceeds as follows:

Step 0: The CPUs factorize the panel.

Step 1: The factorized panel is transferred from the CPUs to the GPU.

Step 2: The first column of the trailing submatrix is updated separately on the GPU in order to enable the look-ahead.

Step 3: The updated single column of the previous step is transfered to the CPUs. This step makes available the next panel factorization for the CPUs.

Step 4: The GPUs update the rest of the trailing submatrix associated with the current panel, while the CPUs factorize the next panel. We note that, at this step, 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.

The same process is repeated from Step 1 to 4 until the matrix is entirely factorized.

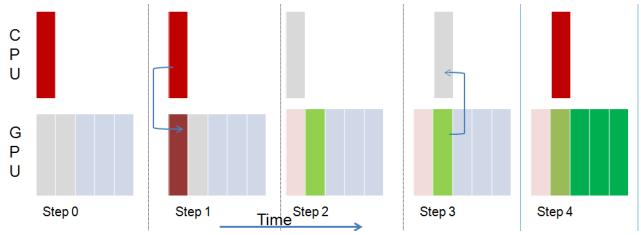


Figure 1: Example of an execution trace of magma_dgetrf on a square matrix. The matrix is partitioned into 5 x 5 blocs of columns.

The dgetrf routine in MAGMA as described in figure 1 is presented in Algorithm 1. In the algorithm, cpu_dgetrf represents the multithreaded routine called to factorize the panel. Device_setMatrix and device_getMatrix represent the routines which transfer the panel from the CPUs to the GPUs, and from the GPUs to the CPUs, respectively. These routines block the execution of the CPU until the transfer is completed. Magma_dtrsm and magma_dgemm represent the non blocking routines in MAGMA which compute, respectively, the upper part of the matrix and the update of the trailing submatrix at each step of the factorization.

Algorithm 1 magma_dgetrf

1: **Input:** $m \times n$ matrix dA, block size b, number of processors P, number of blocs for the CPU parts d
2: $M = m/b, N = n/b$
3: **workspace:** $m \times b$ matrix size $dAP = dA(1 : M, 1)$
4: **for** $K = 1$ to $N = n/b$ **do**
5: cpu_dgetrf(dAP, P)
6: device_setMatrix(dAP, $dA(K : M, K)$) /*Copy the factored panel from CPU to the GPU*/
7: magma_dtrsm($A(K, K + 1)$)
8: magma_dgemm($A(K + 1 : M, K + 1)$)
9: device_getMatrix($dA(K + 1 : M, K + 1)$, dAP) /*Copy next panel from the GPU to the CPU*/
10: cpu_synchronize(dAP) /*CPUs synchronization on the next panel*/
11: magma_dtrsm($A(K, K + d : N)$)
12: magma_dgemm($A(K + 1 : M, K + d : N)$)
13: **end for**

2.3 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. [20], and more recently Yaohung et al. [19], 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. [19]. 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. Unfortunately, this approach ignores possible interruptions such as system noise, data locality issues, and many more, which may vary from one execution to another and which may affect the estimated panel size values. 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 [5], we have evaluated the possibility of replacing the standard panel factorization in MAGMA by CALU, which is a more efficient approach. The reason for this evaluation was that MAGMA calls a multithreaded routine to factorize the panel. In the case of MKL for example, the corresponding multithreaded routine is known to be not well optimized for tall and skinny matrices [9]. 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. [12] 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 difficult to tune. The authors suggest 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. [18] 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, a formula is used to determine B_h, and then a simulation executing the routine (QR or Cholesky) on a small number of tiles (e.g., 3) in order to readjust the value of B_h for the global matrix. 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 every routine. Furthermore, to handle the large number of tiles created by the decomposition, one distinct thread is dedicated to manage each GPU. So, for a system with P cores and k GPUs, only $P - k$ threads will focus on the CPUs part.

Yulu et al. [14] proposed an approach based on cyclic distribution of data between CPUs and the GPUs. In their approach, column blocks are cyclically distributed to the GPUs k times (so called round robin) and then one column block is distributed to the CPUs, then the next k block columns are distributed to GPUs and so on. Indeed, the CPUs factorize the current panel but also participate in the update of some part of the trailing submatrix. First, this approach requires a full transformation of the input matrix to a more complex structure. Second, nothing other than the empirical results allows the user to determine the value of k.

Jakub et al.[15] proposed an approach based on dynamic load balancing of workload during LU factorization. First, a number of block columns is initially transferred to the CPU before the execution, and then a block column is transferred to the CPUs at each step of the factorization. Again, here the authors do not provide any idea about the choice of the initial amount of data to transfer to the CPUs; instead, they propose a strategy based on experimental tuning by following the performance improvements, making their approach difficult to port on different architectures.

2.4 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 [7, 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, as in classic algorithms implemented in ScaLAPACK for example, although the implementation is based on a block algorithm, the panel is still factorized 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. The algorithm can be described as follow for a panel B of size $m \times b$, where m is the number of rows and b is the block size:

Step 0: Partition the panel vertically into P parts, where P is the number of processors participating in the panel factorization, i.e., $B = [B_1; B_2; ...; B_P]$. Then, processor i owns block B_i.

Step 1: Each processor applies LU factorization with partial pivoting on its original block, i.e., $\pi_i B_i = L_i U_i$.

Step 2: Each processor applies the permutation vector π_i on its original rows and keeps the first b rows as a pivot candidate. That is, $C_i = \pi_i B_i(1 : b, b)$.

Step 3: The pivot candidates C_i are merged one on top of another using a reduction tree. At each node of the tree, LU factorization with partial pivoting is applied, the resulting permutation matrix is applied on the original merged block and then the first b rows are selected as a new pivot candidates. This is repeated until obtaining the pivots at the root of the reduction tree. Then these final pivot are selected as the good pivots for the whole panel.

Step 4: The final permutation is applied on the original panel to move the selected pivot at the top.

Step 5: Each processor applies LU factorization without partial pivoting on its part B_i.

By using a binary tree for the reduction operation through this approach, only $\log P$ messages are exchanged, which is less than ScaLAPACK's number of messages by a factor of b. 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, 7] and multicore environments [9, 8].

3. ASYNCHRONOUS LU FACTORIZATION WITH GPU ACCELERATORS

3.1 Method

Algorithm 2 gives 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. Part one is formed by the first d block columns of A and is transfered to the CPUs, while part two, formed by the remaining $N - d$ blocks, remains on the GPU. The input parameter d is determined using our model described in section 4. 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.

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 [16, 17]; 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, 9];

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. In order to enable the look-ahead technique during the execution, tasks are inserted in this order: tasks which update column $K+1$, then the tasks which factorize panel $K+1$ (in line $5-7$ of the algorithm), and finally the tasks which update columns $K+2$ to $K+d$.

At the end of the algorithm, the factorized matrix is stored entirely on the GPU memory.

3.2 Scheduling

Our algorithm, as described above, creates and inserts both CPU and GPU tasks in a queue of tasks. The 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. Although we have implemented a scheduler for the CPUs' queue of tasks, any existing task-based scheduler, e.g., QUARK[21], can be used instead. 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.

Algorithm 2 Asynchronous CALU

1: **Input:** $m \times n$ matrix dA, block size b, number of processors $P = P_r \times P_c$, column blocks for the CPUs 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**

Figure 2 shows an example of the direct acyclic graph (DAG) resulting from the insertion of tasks on a matrix partitioned into 5 column blocks, where 2 of the blocks are initially assigned to the CPUs. The DAG holds 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 group all P_r tasks created by the panel decomposition in one task, that is, the task **P** colored in red in the figure. The decomposition of the panel into tasks using a binary tree is well explained in [9]. It could be simplified into one task because, although the panel is parallelized, its outside dependencies are global for the entire panel. However, in the CALU implementation, the threads that execute the panel may do some tasks outside the panel, e.g., updates from previous steps of the factorization while the panel is still being factored. The **U** and **S** DAG vertices represent 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 to balance work at each step of the factorization. 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. The asynchronous behavior of our algorithm allows these transfers to be initiated at any time and without any synchronization with a thread in the CPUs part.

3.3 Runtime

Figure 3 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 step 0: The CPUs factorize the panel (represented by

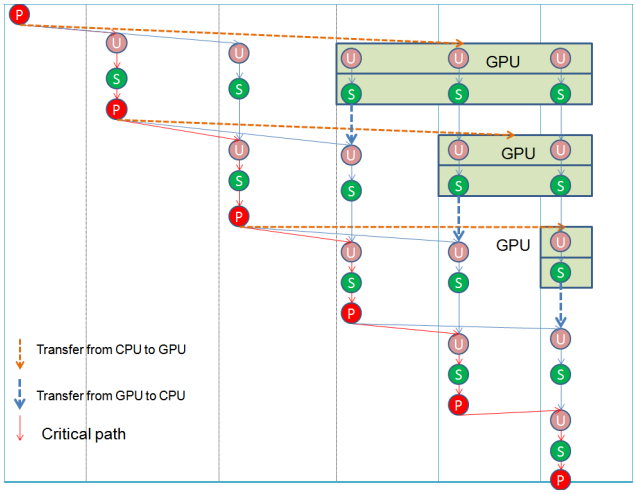


Figure 2: Example of a DAG of asynchronous CALU on a square matrix partitioned into 5 blocs of columns. The CPUs part is formed by 2 blocs of columns, i.e., 40% of the initial matrix.

a red bar) using P_r threads. This first step represents the only part of the factorization that is not overlapped with other computations;

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.

This process is repeated from step 2 to 3 on the remaining part of the trailing submatrix until the factorization is completed. We note that at each step the size of the trailing submatrix decreases by one block column. When the remaining matrix has less than d block columns, 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.

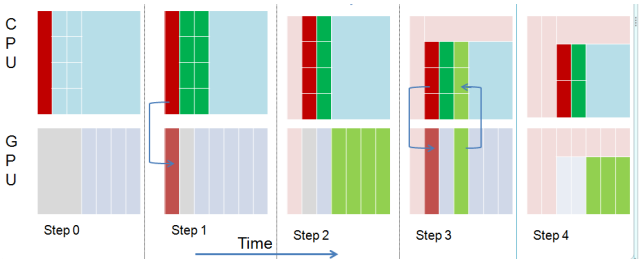


Figure 3: Example of an execution of the asynchronous CALU on a square matrix partitioned into 7 block columns. The CPUs part is formed by 3 blocs of columns, i.e., 42% of the initial matrix.

4. PERFORMANCE MODEL

As we will show in our experiments, 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 the GPU will become idle. The main question is how to determine the optimal number of block columns for this 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.

4.1 Theoretical model

Let A be the matrix of size $m \times n$ to be factored, b be the block size, and P be the number of CPU processors. The block LU algorithm partitions 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 4, 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,

$$W_1 = W_{1panel} + (d-1)W_{1update},$$

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 and therefore

$$W_2 = (N_K - d)W_{1update}.$$

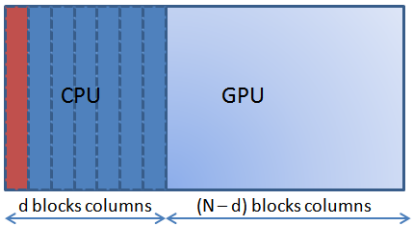


Figure 4: Initial decomposition of the matrix in two parts – first for the CPUs and second for the GPUs.

By definition, we can easily deduce that

$$T_1 = \frac{W_1}{P \times g_1} \quad \text{and} \quad T_2 = \frac{W_2}{g_2}.$$

For a synchronous algorithm where the threads in the CPU parts and the GPU synchronize at each step of the factorization, if $T_1 \neq T_2$, 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, where the threads in the CPU parts may switch immediately to the next step $(K+1)$, 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, if T_1 is lower than T_2, the threads will continue in step $K+1$ and therefore 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:

$$d = \frac{Pg_1N_K + g_2}{Pg_1 + g_2} - \frac{W_{1panel}}{W_{1update}} \frac{g_2}{Pg_1 + g_2}.$$

A well known approximation for the number of operations required to perform LU factorization on one block column of size $m \times b$ is $W_{1panel} = (m_K - \frac{b}{3})b^2$. The number of operations to update one block column of size $m \times b$ is a rank-b update, so $W_{1update} = m_K b^2$. Hence,

$$\frac{W_{1panel}}{W_{1update}} = \frac{(m_K - \frac{b}{3})b^2}{m_K b^2} = \frac{m_K - \frac{b}{3}}{m_k}.$$

Since the block size b is much smaller compared to the matrix size m_K, this quantity will tend asymptoticly to one, for which case we can obtain an expression for d:

$$d = \frac{Pg_1N_K}{Pg_1 + g_2}, \quad \text{and finally}$$

$$\frac{d}{N_K} = \frac{Pg_1}{Pg_1 + g_2}. \tag{1}$$

Here $\frac{d}{N_K}$ represents the largest percentage of the matrix for the CPUs part at iteration K. The expression $\frac{Pg_1}{Pg_1+g_2}$ does not depend on the matrix size, and can be used to determine the percentage of the matrix for the entire computation. To conform strictly with the model, the percentage of the matrix for the CPUs part should vary at each step of the factorization, but we have estimated that keeping it constant may introduce less imbalance, while keeping the implementation simple. The implementation of [19] shows that such a readjustment may increase just very slightly the performance of the entire algorithm.

4.2 Model analysis

Equation 1 guides the choice of the number of block columns to assign to the CPUs in order to balance work with the GPUs. The model suggests that the CPUs part (d) varies as follows:

- By increasing the number of processors P, it increases to keep the algorithm multicore scalable;
- By increasing the matrix size, it increases to balance work and keep equation 1 satisfied;
- By increasing the power of the GPUs when the power of the CPUs is kept constant, it decreases in order to balance the CPU-GPU work.

This is better illustrated in our experiments with some hardware parameters.

4.3 First panel factorization

It is necessary to take into account the fact that the first panel is not factorized in parallel with the update of the trailing submatrix associated with it, because of the dependencies. To take this case into account and make our model consistent, we need to handle the first panel separately. To do so, we consider that at step 0, only panel 0 is factorized. At step 1, the updates associated with panel 0 are performed and then panel 1 is factorized. We note that it is possible to factorize panel 1, just after the update associated with panel 0 is applied to column 1, even if the whole other updates are not completed. Following the same principle, at step 2, the update associated with panel 1 are performed and then panel 2 is factorized, and so on. So we deduce that, at step K of the factorization, the amount of work for the CPUs (W_1) can be expressed as the sum of the work for performing the update of the trailing submatrix associated

with panel K and factorizing panel $K+1$. But for simplicity, we keep W_1 unchanged in the model, i.e., we disregard its variation due to the change in amount of work introduced by replacing panel K by panel $K+1$.

4.4 Model accuracy in practice

Our model estimates the amount of work required for the first part of the matrix by $T_1 = \frac{W_1}{P \times g_1}$, where $W_1 = W_{1panel} + (d-1)W_{1update}$. $W_{1update}$ represents matrix products, so it can be easily decomposed into P_r pieces and updated separately. This is not the case for the panel, because of the synchronization introduced during its factorization. In other words, W_{1panel} should be expressed as the sequential amount of work to factorize a panel plus a communication overhead $\phi(P_r)$, where P_r is the number of threads working on the panel. Then T_1 becomes $\frac{W_1}{P \times g_1} + \phi(P_r)$. By using $T_1 = \frac{W_1}{P \times g_1}$, we assume that the communication overhead is negligible. In practice, the overhead may increase slightly the time to perform W_1, so the percentage of the CPUs should be slightly decreased depending on this overhead.

4.5 Adaptation with CALU

Our model assumes a use of an optimal classic algorithm for the panel factorization. In that case the expression $\frac{W_{1panel}}{W_{1update}}$ tends asymptoticly to one. If CALU is used, the number of flops, compared to the standard implementation, is higher as CALU performs more flops. In particular, 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 $\frac{O(b^3 \log P_r)}{W_{1update}} = \frac{O(b^3 \log P_r)}{m_K b^2} = \frac{O(b \log P_r)}{m_K}$. As $O(b \log P_r)$ can also be considered negligible compared to m_K, the model of equation 1 can be used as well to determine the percentage of the matrix for the CPUs in order to keep the model simple. Having a fast parallel panel factorization algorithm then helps to remove the panel on the critical path and then minimize its impact on the estimation of T_1.

5. 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 1.

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[5], 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_sync 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.

5.1 Performance of asynchronous CALU with fixed parameters

Figure 5 shows the performance of our algorithm (calu_async) when the number of block columns d in the CPUs part varies as 1, 2, 4, 8, 16, and 32. The factorization is performed on a matrix of size $M = N = 10112$ using 12 cores and one GPU. We compare each variation of calu_async to magma_dgetrf and magma_calu_sync. We note that the performance of magma_dgetrf and magma_calu_sync does not change when d varies; we repeat them in the diagram only for purposes of comparison. For this problem set, magma_calu_sync is 15% faster than magma_dgetrf. For $d = 1$, calu_async behaves as MAGMA with the difference that the panel is factorized using CALU and it does not use the look-ahead technique. Our algorithm does not use the look-ahead when the CPU holds only 1 panel, because the GPU needs to update the whole trailing submatrix before a new block column is transferred to the CPU. In this case, our approach is less efficient than magma_dgetrf and magma_calu_sync. For $d = 2$, i.e., for only one additional block column in the CPUs part, our approach becomes 20% faster than magma_dgetrf and 4% faster than magma_calu_sync. The best performance is obtained for $d = 8$, where it is 35% faster than magma_dgetrf and 17% faster than magma_calu_sync. We observe that for $d > 8$, the performance of calu_async decreases considerably. For example, for $d = 32$, it is 64% slower than magma_dgetrf. This case illustrates the impact of the load balancing on performance and the difficulty to manually tune applications based on a similar approach.

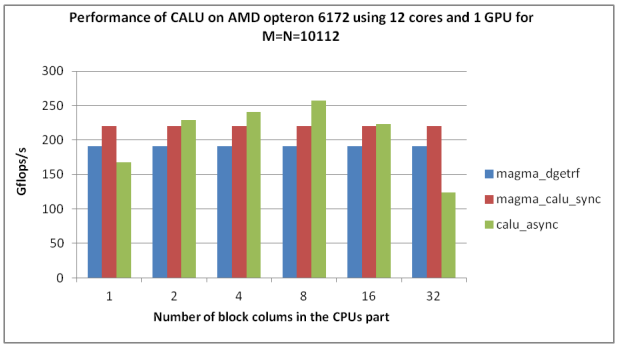


Figure 5: Performance of the variants of CALU for which the number of columns (panels) for the CPU part varies, on an AMD Opteron 6172 using 12 cores and one GPU with the matrix size kept constant.

5.2 Estimation of the percentage of the matrix for the CPUs part using the model

The adapted value of d depends not only on the matrix size but also on the number and the power of each computation unit. In this section, we study the estimated value of d for each matrix size and architecture using our model, that is:

$$\frac{d}{N_K} = \frac{Pg_1}{Pg_1 + g_2} \quad (2)$$

Figure 6 shows the estimated percentage of the matrix for the CPUs part when the number of processors varies. The two AMD machines in our test are equipped with the same

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 1: Peak performance of each machine in our test set.

Tesla S2050 GPUs with 504 GFlop/s peak performance. The AMD Opteron 6172 has a peak performance of 403.2 Gflops/s, while the AMD Opteron 6180 has 480 GFlops/s. Our model suggests to increase the percentage of the matrix on the AMD Opteron 6180 machine relative to the AMD Opteron 6172. On the Intel E5-2670, the CPUs' peak performance for 16 cores is 332.8 GFlops/s and the model suggests to use 30% of the matrix. We note that the Intel machine is equipped with 16 cores, but we plot the estimation through 48 cores in order to predict the recommended value of d as if the machine were equipped with 48, like the two AMD machines in our test. In that case, the CPUs' peak performance would be 998.4 GFlops/s and d would represent 57% of the matrix. We also note that in practice it is almost impossible for a parallel program to achieve the manufacturer's peak performance due to algorithm dependencies, possible system noises, etc. So, the estimated value of d in our experiments is computed using a realistic percentage value of peak performance. In our case, using 80% of the CPUs' peak and 90% of the GPUs' peak leads to reliable values.

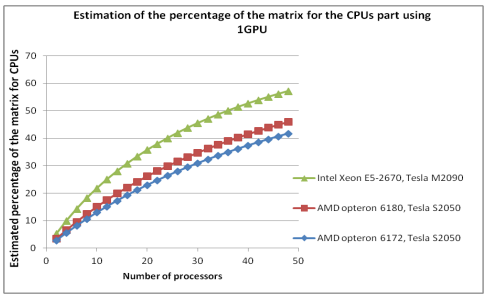


Figure 6: Modeled percentage of the matrix for the CPU part for the different architectures tested.

5.3 Performance comparison of CALU using modeled and fixed parameters

Figure 7 shows the scalability of the different variants of calu_async depending on the choice of d. The parameters for each variant are shown in brackets. "$d = x \quad panels$" indicates that the CPUs part is formed by x block columns and "$d = y\% \quad matrix$" indicates that the CPUs part is formed by y% of the number of block columns in the input matrix. "$d = estimated$" indicates the variant for which the value of d is computed using our model before the execution.

We observe that the variants calu_async ($d = 8 \quad panels$) and calu_async ($d = 16 \quad panels$) scale well for a small number of processors, but stagnate as the number increases. The reason is that, by increasing the processors while the CPUs part remains constant, the CPUs complete their portion of the factorization, while the GPU is still updating its part of the matrix. The same behavior is observed with calu_async($d = 26\% \quad matrix$). Calu_async ($d = estimated$) outperforms all the other variants and shows a good scalability with the increasing number of processors.

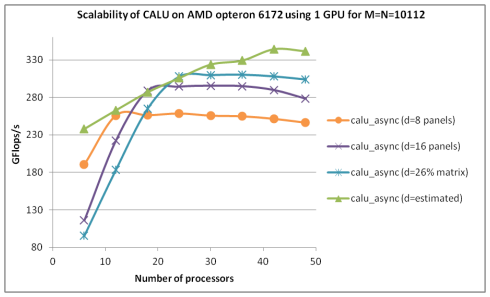


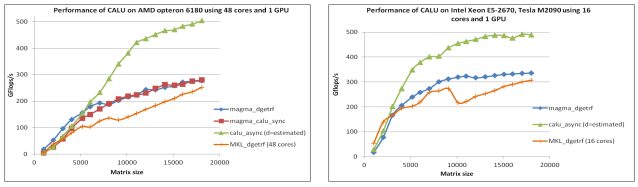
Figure 7: Scalability comparison of CALU with modeled parameters and some variants with fixed parameters on an AMD Opteron 6172 using one NVIDIA Tesla S2050 GPU.

5.4 Performance and scalability

5.4.1 *Performance for square matrices*

Figure 8 shows the performance of calu_async with the parameter d estimated using our model. Figure 8a. shows the performance on an AMD Opteron 6180 with 48 cores and one GPU. Here magma_dgetrf and magma_calu_sync achieve almost the same performance. 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 better. The reason is that, our approach is based on an estimation, so a small difference from the optimal number of blocks may show some performance degradation for very small matrices. For $M = N \geq 5184$, calu_async outperforms magma_dgetrf and magma_calu_async. The best improvement is otained for $M = N = 11008$, where calu_async is 2× faster than magma_dgetrf and magma_calu_sync, and also 2.7× faster than MKL_dgetrf. For the largest matrix, 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 8b shows the performance on Intel Xeon E5-2670, where the same behavior is observed and calu_async is up to 1.5× faster than magma_dgetrf and 2× 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.



a. 48 AMD Opteron 6180 cores and one S2050 GPU.

b. 16 Intel Xeon E5-2670 cores and one M2090 GPU.

Figure 8: Performance of CALU for square matrix.

Figure 9 shows the performance on high-end CPUs (Sandy Bridge; 16 core @2.6GHz, DP Peak 332 GFlop/s), GPUs (K20c; DP Peak 1,174 GFlop/s), and Intel Xeon Phi (DP Peak 1,046 GFlop/s).

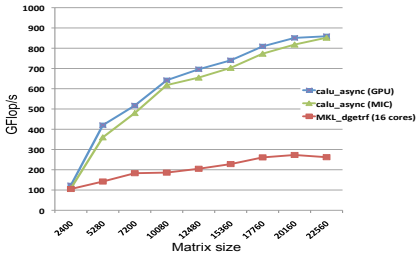
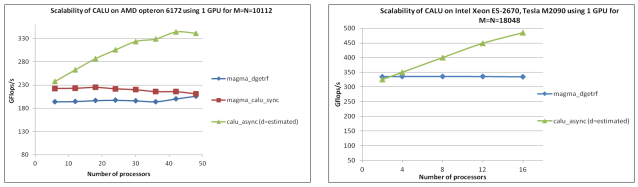


Figure 9: Asynchronous CALU on high-end CPUs (Sandy Bridge), GPUs (K20c), and Intel Xeon Phi.

5.4.2 *Scalability for square matrices*

Figure 10 shows the scalability of calu_async vs. magma_dgetrf and magma_calu_sync. 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 on both AMD and Intel systems.



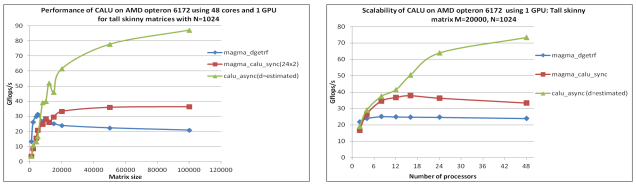
a. AMD Opteron 6172 using one Tesla S2050 GPU.

b. Intel Xeon E5-2670 using one Tesla M2090 GPU.

Figure 10: Scalability of CALU with modeled parameters vs. magma_dgetrf and magma_calu_sync.

5.4.3 *Results on tall skinny matrices*

Figure 11 shows the performance obtained for tall and skinny matrices. CALU is well suited for factorizing tall and skinny matrices, so combining it with a GPU may lead to important performance improvements as well. Figure 11 a. shows the performance of magma_dgetrf, magma_calu_sync, and calu_async for tall and skinny matrices when the number of columns is fixed to $N = 1024$ and the number of rows increases. We observe that, contrary to calu_async, magma_dgetrf and magma_calu_sync do not scale with the matrix size when the number of columns becomes larger than 20000. For this set of problems, calu_async is up to 4 times faster than magma_dgetrf and up to 2.5 times faster than magma_calu_sync. Figure 11 b. shows that calu_async also scales better with the number of processors. For a matrix of size $M = 20000$ and $N = 1024$, when $P \leq 12$, the model estimates to use only one block column in the CPUs part ($d = 1$). With one column in the CPUs part, calu_async behaves almost like magma_calu_sync and then shows similar performance. For $P > 12$, the model estimates a value of d greater than 1 and calu_async leads to important speedup.



a. Performance for N=1,024 and M increasing.

b. Strong scalability for $M = 20,000$, $N = 1,024$.

Figure 11: Performance and scalability of asynchronous CALU for tall skinny matrices on an AMD Opteron 6172 using one Tesla S2050 GPU.

6. 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 other algorithms 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 better scalable compared to the corresponding standard routine in MAGMA and our previous implementation of CALU for GPUs [5].

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 [10]. 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, and eigensolvers.

7. ACKNOWLEDGMENTS

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

8. 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] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[5] 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.

[6] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. 2008.

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

[8] 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.

[9] 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.

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

[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] 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.

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

[14] Y. Jia, P. Luszczek, and J. Dongarra. Multi-GPU implementation of LU factorization. *Procedia CS*, 9:106–115, 2012.

[15] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra. Lu factorization with partial pivoting for a multicore system with accelerators. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2012.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users guide.