# A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines

Marc Baboulin, Simplice Donfack, Laura Grigori, Adrien Rémy, Stanimire Tomov, Jack Dongarra

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines

Marc Baboulin — Simplice Donfack — Laura Grigori — Adrien Rémy — Stanimire Tomov — Jack Dongarra

N° 7854

January 2012

Domaine Informatique/Analyse numérique

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7854--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

# A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines

Marc Baboulin[*] , Simplice Donfack[†] , Laura Grigori[‡] , Adrien Rémy[§] , Stanimire Tomov[¶] , Jack Dongarra[‖]

Domaine :
Équipe-Projet Grand-Large

Rapport de recherche n° 7854 — January 2012 — 16 pages

**Abstract:** We study several solvers for the solution of general linear systems where the main objective is to reduce the communication overhead due to pivoting. We first describe two existing algorithms for the LU factorization on hybrid CPU/GPU architectures. The first one is based on partial pivoting and the second uses a random preconditioning of the original matrix to avoid pivoting. Then we introduce a solver where the panel factorization is performed using a communication-avoiding pivoting heuristic while the update of the trailing submatrix is performed by the GPU. We provide performance comparisons for these solvers on current hybrid multicore-GPU parallel machines.

**Key-words:** hybrid multicore/GPU computing, dense linear algebra libraries, linear system solvers, communication-avoiding algorithms, LU factorization.

[*] Inria and Université Paris-Sud, France (marc.baboulin@inria.fr).
[†] Inria and Université Paris-Sud, France (simplice.donfack@lri.fr).
[‡] Inria and Université Paris-Sud, France (laura.grigori@inria.fr).
[§] Inria and Université Paris-Sud, France (adrien.remy@lri.fr).
[¶] University of Tennessee, USA (tomov@eecs.utk.edu).
[‖] University of Tennessee and Oak Ridge National Laboratory, USA, and University of Manchester, United Kingdom (dongarra@eecs.utk.edu).

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

# Une classe d'algorithmes à communication minimale pour résoudre les systèmes linéaires denses sur machines parallèles multicœurs-GPU

**Résumé :** Nous étudions plusieurs solveurs de systèmes linéaires avec comme objectif de réduire le surcoût de communication lié au pivotage. Nous décrivons d'abord deux algorithmes existants pour la factorisation LU sur architectures hybrides CPU/GPU. Le premier est basé sur le pivotage partiel et le second utilise un préconditionnement aléatoire de la matrice originale pour éviter de pivoter. Puis nous présentons un solveur où la factorisation du panel est effectuée en utilisant une heuristique de pivotage à communication minimale alors que la mise à jour du reste de la matrice est réalisée par le GPU. Nous proposons des comparaisons de performance pour ces solveurs sur des machines hybrides multicœurs-GPU actuelles.

**Mots-clés :** calcul hybride multicœurs-GPU, bibliothèques d'algèbre linéaire dense, solveurs de systèmes linéaires, algorithmes à communication optimale, factorisation LU.

# 1 Introduction

There have been several main changes in the development of dense linear algebra libraries over the years. These changes have always been triggered by major hardware developments. For example, LINPACK [11] in the 70's targeted the vector machines at the time for which cache reuse was not essential, and as a result LINPACK had relied on just Level 1 BLAS. In the 80's LINPACK had to be rewritten, leading to LAPACK [1], that would rely on Level 3 BLAS for cache based machines. In the 90's it was extended to ScaLAPACK [4] for parallel platforms, relying on the PBLAS [7] message passing. Now, in the 00's, with the explosion in parallelism and heterogeneity as well as the ever increasing data-communication costs, the old libraries had to be redesigned once again. An example is the MAGMA[1] library - a collection of this next generation linear algebra libraries [19, 21, 22].

MAGMA, similarly to LAPACK, is being build as a community effort, incorporating the newest developments in hybrid algorithms and scheduling, and aiming at minimizing synchronizations and communication in these algorithms. The goal of these efforts is to redesign the dense linear algebra algorithms in LAPACK to fully exploit the power of current heterogeneous systems of multi/manycore CPUs and accelerators, and deliver the fastest possible time to an accurate solution within given energy constraints. Indeed, the algorithms included so far in MAGMA 1.1 manage to overcome bottlenecks associated with just multicore or GPUs, to significantly outperform corresponding packages for any of these components taken separately. MAGMA's one-sided factorizations for example (and linear solvers) on a single Fermi GPU (and a basic CPU host) can outperform state-of-the-art CPU libraries on high-end multi-socket, multicore nodes (e.g., using up to 48 modern cores). In parallel to the success stories in the development of hybrid algorithms, there have been a number of new developments related to minimizing the communication in one-sided factorizations (e.g. [2]). Such improvements have become essential due to the increasing gap between communication and computation costs.

For the linear system solvers on current multicore or GPU architectures, a bottleneck in terms of communication cost and parallelism comes from the pivoting, a technique used to prevent divisions by too-small numbers in the Gaussian Elimination (GE) process. The commonly used method of Gaussian Elimination with partial pivoting (GEPP) is implemented in current linear algebra libraries for solving square linear systems $Ax = b$ resulting in very stable algorithms. These systems are in general solved using the well-known LU factorization that decomposes the input matrix $A$ into the product $L \times U$, where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix. Current libraries like LAPACK implement GE using a block algorithm, which factors the input matrix by iterating over its blocks of columns (panels). At each iteration, the LU factorization of the current panel is computed, and then the trailing submatrix is updated. In the LAPACK implementation, pivoting is used during the factorization of each column of a panel, and leads to swapping rows. Pivoting not only requires communication (or synchronization in a shared memory environment), but it also limits the exploitation of asynchronicity between block operations. This is because the update of the trailing submatrix can be performed only

[1] Matrix Algebra on GPU and Multicore Architectures, `http://icl.cs.utk.edu/magma/`

when the panel factorization is completed. We can find in [3] an evaluation of the communication overhead due to partial pivoting using MAGMA on a given CPU/GPU architecture. This cost can represent on some hybrid architectures up to 40% of the global factorization time, depending on the matrix size. The communication cost of GEPP is asymptotically larger than the lower bounds on communication [13]. Other classical pivoting strategies can be used, as rook pivoting or complete pivoting (see [16] for a comprehensive review), but they always require between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons to search for the pivot. In this paper we consider two alternative strategies to these pivoting techniques, that have the property of reducing the communication in the LU factorization while providing a satisfying accuracy.

The first alternative is tournament pivoting, introduced in the last years in the context of CALU, a communication-avoiding LU factorization algorithm [14]. It is shown in [13] that tournament pivoting is as stable as partial pivoting in practice and that CALU minimizes communication. With this strategy, the panel factorization, referred to as TSLU (Tall Skinny LU), can be efficiently parallelized as follows. The panel is partitioned into $P_r$ blocks. From each block, a set of local pivots is selected in parallel using GEPP. A tournament is used on the $P_r$ local sets to select a set of global pivots. These global pivots are moved to the diagonal positions, and then the LU factorization with no pivoting of the entire panel is performed. The tournament is implemented as a reduction operation, with GEPP being the operator used at each step of the reduction.

The second alternative is proposed in [3] where the communication overhead due to pivoting is completely removed by considering a randomization technique referred to as Partial Random Butterfly Transformation (PRBT). This method revisits the approach initially proposed in [20] where the original matrix $A$ is transformed into a matrix that would be sufficiently “random" so that, with a probability close to 1, pivoting is not needed. It consists of a multiplicative preconditioning $U^T AV$ where the matrices $U$ and $V$ are chosen among a particular class of random matrices called *recursive butterfly matrices.* Then Gaussian Elimination with No Pivoting (GENP) is performed on the matrix $U^T AV$. It has been shown in [3] that a small number of recursions (2 in practice) is sufficient to obtain a satisfying accuracy. Moreover, by exploiting the particular structure of the recursive butterflies, this technique of randomization is attractive in terms of Gflop/s ($\mathcal{O}(n^2)$ operations) and data storage. Note that since in this approach we know in advance that we are not going to pivot, GENP that follows randomization is implemented as a very efficient fully BLAS 3 algorithm. Note also that when the initial matrix is randomized, we systematically add iterative refinement in the working precision for better stability, as indicated in [16, p. 232].

We show in this paper that the usage of these techniques in the context of hybrid CPU/GPU architectures allows us to take advantage of each computational unit.

The paper is organized as follows. First we describe in Section 2 two existing algorithms for LU factorization on hybrid CPU/GPU machines. The first one is implemented in the parallel library MAGMA and is based on the partial pivoting technique. The second algorithm uses randomization to avoid pivoting and performs GENP on the randomized matrix. Then in Section 3 we introduce tournament pivoting, a strategy based on CALU that we adapt specifically for CPU/GPU architectures. In this new implementation, the panel is factored on the CPU using a modified CALU factorization while the update of the trailing submatrix is performed on the GPU. The resulting solver is called H-CALU solver. Finally we propose in Section 4 preliminary performance results where we compare H-CALU with LU solvers using partial pivoting (MAGMA) and PRBT. Concluding remarks are given in Section 5.

# 2 Two existing hybrid algorithms for LU factorization

## 2.1 LU factorization based on partial pivoting

Let us illustrate how the hybrid multicore + GPU approach can be applied to the LU factorization by describing the algorithm as it is implemented in the MAGMA library. The method is based on splitting the computation as shown in Figure 1 that represents a current matrix factored via a right looking block LU factorization [9, p. 85], where the dark part has been already factored. The initial matrix has been downloaded to the GPU and we describe here a current iteration:

1. The current panel (1) containing $b$ columns is downloaded to the CPU.

2. (1) is factored by the CPU and the result is sent back to the GPU.

3. The GPU updates (2) that corresponds to the next panel (first $b$ columns of the trailing submatrix).

4. The updated panel (2) is sent to the CPU and asynchronously factored on the CPU while the GPU updates the rest of the matrix (3).

The technique consisting of factoring (2) while still updating (3) is often referred to as "look-ahead" technique [18]. In the current implementation of MAGMA, the panel factorization is performed using GEPP but this algorithm is general enough to be applicable to many forms of LU factorizations, where the distinction can be made based on the form of pivoting that they employ. In Section 3.2 we will use a different pivoting strategy that turns out to be very efficient for factoring the panel due to its particular "tall and skinny" structure. Depending on the problem size $n$ and on the hardware used, MAGMA proposes a default value for the parameter $b$ (width of the panel).

Note that the design of the hybrid LU in MAGMA avoids communicating by having only panels transferred between CPU and GPU ($\mathcal{O}(n * b)$ data vs $\mathcal{O}(n * n * b)$ computation in the updates), enabling also the total overlap of the panel computation by the updates for $n$ large enough.

## 2.2 PRBT solver

Using the PRBT solver, we solve the general linear system $Ax = b$ by the following steps:

1. Compute the randomized matrix $A_r = U^T AV$, with $U$ and $V$ recursive butterflies.

2. Factorize $A_r$ with GENP.

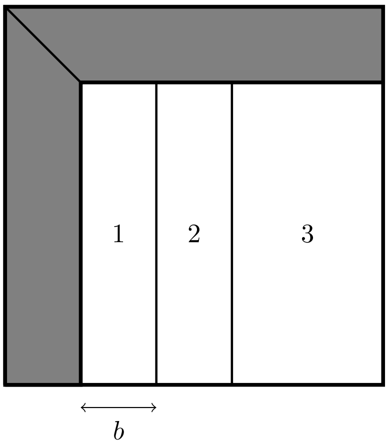


Figure 1: Block splitting in hybrid LU factorization

3. Solve $A_r y = U^T b$.
4. Solution is $x = Vy$.

In this section, we present an implementation of a PRBT solver for hybrid CPU/GPU architectures. The randomization by recursive butterflies uses 2 recursions and the resulting computational cost for randomization is $4n^2$ flops (see [3] for more details on the computational kernels that exploit the particular structure of the recursive butterflies). The PRBT solver for hybrid CPU/GPU architectures performs the following tasks:

- Random generation and packed storage of $U$ and $V$ on the CPU, while sending A to the device memory.
- The packed representation of $U$ and $V$ are sent from the host memory to the device memory.
- Randomization is performed on the GPU, updating $A$ in-place (no additional memory needed) on the device memory.
- The randomized matrix is factorized with GENP on the GPU, the panel factorization being performed on the CPU host.
- We compute $U^T b$ on the GPU, $A_r y = U^T b$ is solved on the GPU, followed by the solution $x = Vy$.
- The solution is sent to the host memory, followed if necessary by iterative refinement on the CPU.

When using the PRBT solver, the bulk of the computation corresponds to the GENP factorization (the randomization represents less than 2% of the global computational time). Then we expect that the Gflop/s performance of the PRBT solver will provide us with an upper bound for other LU solvers on hybrid CPU/GPU architectures.

# 3 Hybrid communication-avoiding LU factorization

## 3.1 Communication-avoiding LU

The poor evolution of latency and memory bandwidth that we observed over recent years for parallel architectures is a major bottleneck for algorithms that require communication like GEPP. New approaches have been recently proposed to minimize the amount of communication due to pivoting. Among them are the communication-avoiding algorithms introduced for distributed-memory machines [14] and for multicore architectures [8]. These algorithms are effective in the sense that they reduce significantly the communication while being stable in practice. CALU is an algorithm that partitions the input matrix into block of columns (panels), iteratively factors the panel and updates the trailing submatrix. The factorization of the panel is one of the most important task in the LU fatorization since it is part of the critical path in the diagram of tasks and its effective execution influences the performance of the algorithm. In CALU, the panel factorization is performed by the TSLU algorithm which factors a block column of size $b$ with the following steps:

1. Partition the block column in $P_r$ threads where $P_r$ is the number of threads participating in the panel factorization. Let $A_i$, the block assigned to each thread.

2. Each thread applies GEPP to its block and we have $\Pi_i A_i = L_i U_i$. We denote by $C_i = (\Pi_i A_i)(1 : b, 1 : b)$ the block containing the potential pivots, where $\Pi_i$ is the permutation matrix resulting from the factorization of $A_i$.

3. For each step of the reduction tree (see [8] for more details on the most adapted reduction tree):
   If $j$ is the thread number exchanging data with thread $i$ and $C'_i = [C_i; C_j]$, then we compute $\Pi_i C'_i = L_i U_i$ and $C_i = (\Pi_i C'_i)(1 : b, 1 : b)$ is the new pivot candidate.

4. Apply the latest permutation computed in 3) to the original panel in order to compute the block column of $L$ and to compute the block row of $U$.

Once the panel is factored using TSLU then we update the trailing submatrix. Following the approach presented in [5, 6], the CALU algorithm can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on one or several $b \times b$ blocks and where edges represent the dependencies among them. A dependency occurs when a task must access data that is the output of another task, either to further update or just read that data. In Figure 2 we represent an example of LU factorization with CALU as a sequence of DAGs using 2 threads. The panel is partioned into 3 column blocks. Red tasks represent the factorization of the panel via TSLU, the green tasks represent the update of the trailing submatrix, and the blue tasks are those that are not executed so far.

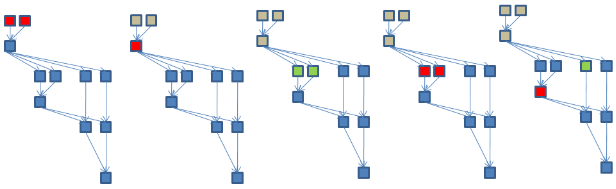


Figure 2: Example of asynchronous LU factorization using multithreaded CALU (2 threads).

### 3.2 Hybrid version of CALU (H-CALU)

As explained in Section 2.1, the LU algorithm implemented in MAGMA factors each block of columns iteratively. Each step is essentially decomposed into two distinct phases: the factorization of the panel by the CPU followed by the update of the trailing submatrix by the GPU. The algorithm's design minimizes the CPU-GPU communications. In the following we describe an algorithm that further improves the algorithm in MAGMA by minimizing the communication associated with the panel factorizations.

At each step, a panel of size $B$ is factored on the CPU by applying CALU to a rectangular matrix and the update of the trailing submatrix is performed by the GPU. CALU factors the panel by splitting the initial block of columns into smaller blocks containing $b$ columns that are factored iteratively using TSLU. Thus, the factorization of the panel is considered as a variant of the algorithm at the first level where we factor a rectangular matrix using only the CPU. The use of this second level of blocking is important for performance on hybrid CPU/GPU architectures because the CPU and GPU processors have different size of cache. The block size $B$ is chosen in order to optimize the performance of the matrix-matrix product on the GPU and to ensure a good grain for increasing parallelism. Then the block size $b$ is tuned in order to optimize the utilization of the multicore cache. This decomposition of the algorithm into small tasks allows us to operate on block of data that fit into the cache. It results in an asynchronous and dynamic execution of the panel factorization on the CPU, yielding to good performance on multicore machines [8]. This asynchronous execution keeps busy most of the CPU threads. When $b = B$, CALU behaves simply as TSLU. If $B$ is large enough (which will be the case for our hybrid implementation), the panel is factored using CALU rather than TSLU because CALU can be executed asynchronously [8]. Our approach also uses the well known technique referred to as *look-ahead* [18] but adapted here so that the CPU and the GPU can work together while minimizing the number of memory transfers. In this approach, we start factoring the next panel as soon as possible.

Figure 3 shows an example of the factorization of a matrix at the top level. We consider that the matrix is initially stored on the GPU. Red tasks represent the factorization of the panel using multithreaded CALU and the green tasks represent the update of the trailing submatrix in the GPU. At each step of the factorization, the block corresponding to the panel is transfered to the CPU and factored using CALU. Once a panel is factored, it is transfered again to the GPU to update the trailing submatrix. The GPU updates in priority the column block corresponding to the next panel. Note that, similarly to [21], the data transfer between CPU and GPU is, as much as possible, overlapped by computation.

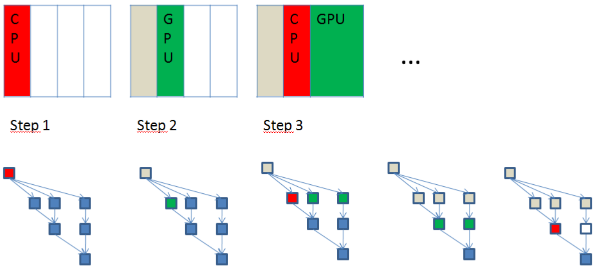


Figure 3: Hybrid CALU factorization (4 panels).

# 4 Numerical experiments

In this section we present performance results for the H-CALU algorithm described in Section 3.2. These numerical experiments were carried out using a hybrid CPU/GPU system where:

- The GPU device is an NVIDIA Fermi Tesla S2050 with 448 CUDA cores running at 1.15 GHz and 2687 MB memory.
- The multicore host is a 48 cores system (4 sockets × 12 cores) AMD Opteron 6172 (2.1 GHz).

For experiments involving only the multicore host (panel factorization), comparisons are made against the MKL [17] multithreaded library. For experiments involving both CPU and GPU (factorization of the whole matrix), comparisons are made against version 1.1 of the MAGMA library. All computations are performed in double precision arithmetic.

## 4.1 Panel factorization

As described in Section 3.2, the panel factorization is performed by the CPU while the update of the trailing submatrix is executed on the GPU. Let us evaluate specifically the performance for the panel factorization phase in an LU factorization. This performance is measured by summing the total number of flops executed in factoring successively each panel throughout the factorization and dividing it by the time spent during these steps. This performance (expressed in Gflop/s) is plotted in Figure 4 for the factorization of four random square matrices, each associated with a given panel size (parameter $B$ defined in Section 3.2, corresponding to the number of columns for the panel). For factoring the panel, we consider different number of threads (one CPU core being used for each thread) varying from 1 to 26. Note that using more than 26 threads does not provide us with better performance, due to the too-large amount of communication involved in the panel factorization. The panel size $B$ considered in Figure 4 for each matrix size corresponds to a value empirically tuned in order to provide the best global factorization time for each matrix when using a hybrid implementation.

In these experiments, we compare the performance of the following routines:

- CALU factorization routine modified for the H-CALU solver and linked with the sequential version of MKL for the required BLAS and LAPACK routines.
- MKL implementation of the LAPACK routine `dgetrf`.
- A recursive routine for GEPP `rgetf2` (linked with MKL multithreaded BLAS) described in [15] which is known to give good performance on "tall and skinny" matrices.
- GENP routine `dgetrf_nopiv` (no pivoting).

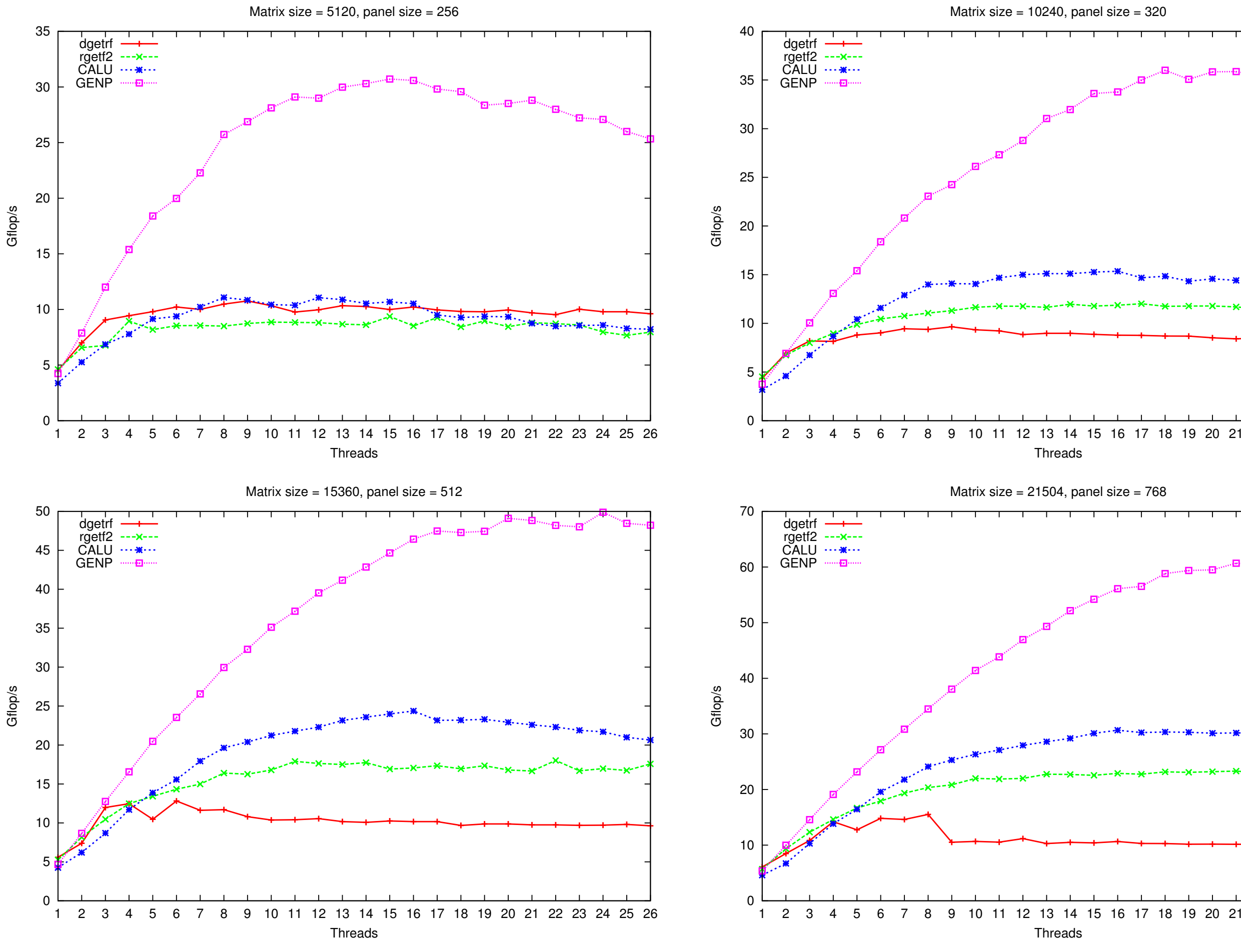
The routines compared in this section have been selected on the fact that they can be used as kernels for our hybrid CPU/GPU implementation. If we use only multicore machines without GPU, then other solvers can be considered (see e.g. recursive tile version in [10]).

In these experiments we do not mention the PRBT solver since the panel factorization in PRBT is performed using GENP. The performance of the GENP routine (`dgetrf_nopiv`) can be considered here as a "peak" performance for the panel factorization. In this respect, we observe that, in percentage of this peak performance and depending on the matrix size $n$, CALU achieves between 36 % ($n = 5120$) and 48 % ($n = 21504$), `dgetrf` achieves between 35 % ($n = 5120$) and 23 % ($n = 21504$), and `rgetf2` achieves between 31 % ($n = 5120$) and 38 % ($n = 21504$). We also observe that CALU is even faster for larger ratios rows/columns. Moreover, CALU and GENP have better scalability properties. This can be explain by the fact that CALU minimizes communication thanks to its pivoting strategy and GENP does not pivot at all. The plateau observed for each curve after a certain number of threads corresponds to cases where the volume of communication becomes too large and cannot be overlapped by computation. For $n = 5120$, CALU, `dgetrf` and `rgetf2` give similar performance. However, when the matrix size increases and then the panel becomes more "tall and skinny", CALU outperforms the two other solvers and achieves a reasonable fraction of the GENP rate. This good behavior of CALU for factoring the panel was already mentioned in [8]. In particular this better scalability of CALU enables us to use more CPU threads in factoring the panel and then to improve the overall performance of a hybrid LU solver.

## 4.2 Performance of hybrid LU implementations

In this section we study the performance of LU factorization routines that utilize resources from multicore (16 threads) and one GPU. We compare in Figure 5 the following routines, applied to square matrices of various sizes:

- The MAGMA routine `magma_dgetrf`, where the panel is factored by the CPU using the MKL routine `dgetrf`.


Matrix size = 5120, panel size = 256

Matrix size = 10240, panel size = 320

Matrix size = 15360, panel size = 512

Matrix size = 21504, panel size = 768

Figure 4: Comparison of CPU multi-threaded panel factorizations.

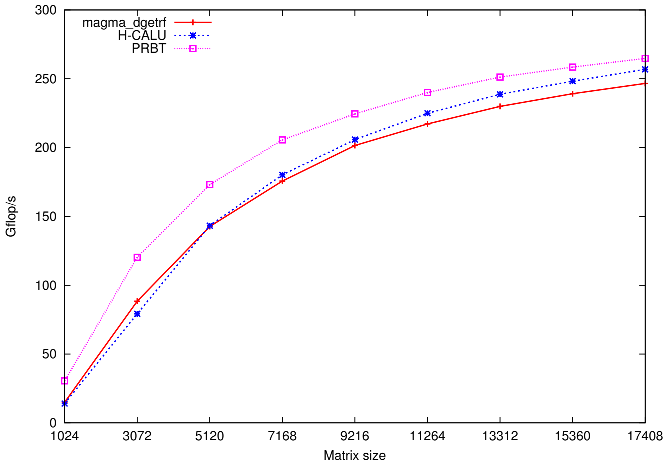


Figure 5: Performance on AMD + Tesla 2050, 16 threads

- H-CALU, where the panel is factored using the CALU routine mentioned in Section4.1.
- The PRBT solver (randomization + GENP).

In these experiments, we do not mention the MAGMA routine in which the panel would be factored by `rgetf2` since it gives performance results similar to `magma_dgetrf` for square matrices.

As expected, PRBT outperforms the routines because it does not pivot and the randomization time is negligible. We can also observe that in the range 1024-5120, H-CALU gives similar performance as MAGMA but it is slightly faster for matrix sizes larger than 5120. This trend can be explained by the fact that, for matrix sizes smaller than 5120, the panels are not enough "tall and skinny" to take advantage of the CALU algorithm. We notice that the difference of performance observed for the panel in Section 4.1 has a moderate impact on the whole factorization since the update phase performed on the GPU represents the bulk of the computation. Note that asymptotically, the performance of the three routines should be close because communication becomes negligible compared to the $O(n^3)$ computations for large dimensions.

In Figure 6 we compare the performance of hybrid LU factorization routines for rectangular matrices of size $m \times n$ with $m > n$, using 16 threads. Such an LU factorization exists when $A(1:k; 1:k)$ is nonsingular for $k = 1:n$ (see [12, p. 102]). In our experiments $n = 2048$ and $m$ varies from 3072 to 21504. Comparisons are made against MAGMA routines `magma_dgetrf` and `magma_dgetrf_nopiv` (instead of PRBT since the latter has no implementation for rectangular matrices). We also compare with the MAGMA routine `magma_dgetrf` modified by factoring the panel using the recursive GEPP kernel (this routine is named H-`rgetf2` in our graph). On this type of matrices, H-CALU outperforms `magma_dgetrf` and H-`rgetf2`. Indeed, for rectangular matrices, the proportion of computation performed during the panel factorization is bigger. Hybrid factorization on rectangular matrices could be for

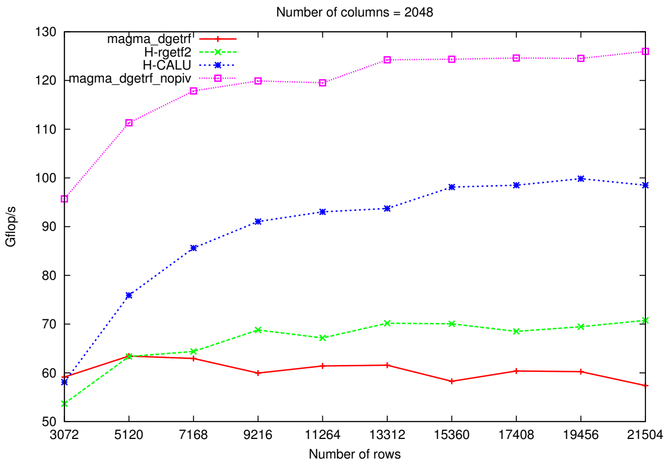


Figure 6: Performance on AMD + Tesla 2050 16 threads

instance useful in a future hybrid factorization with multiple GPUs where the (rectangular) panel could be factored using CPU and a GPU.

In the next experiment we show that, when using only a small number of cores on square matrices, there is much less interest in using a communication-avoiding approach and that factoring the panel with only 4 cores does not provide enough parallelism to obtain better performance than the original MAGMA code. In Figure 7, we present performance results on a machine composed with an AMD Phenom 9850 Quad-Core Processor each core running at 2.5GHz and an NVIDIA GeForce GTX 260 with 192 CUDA cores running at 1.3 GHz and 895 MB of memory. Performance of the three solvers is very similar. Even PRBT, which does not pivot, is only 5% faster than MAGMA. This can be explained by the small number of CPU cores involved in the panel factorization, which does not allow us to take advantage of the communication-avoiding characteristics of CALU and PRBT.

## 5 Conclusion

In this paper we presented different LU factorization routines using a multicore machine accelerated with one GPU. The difference beween these approaches comes from the pivoting strategy chosen for factoring the panel. We proposed a new hybrid communication-avoiding solver H-CALU where the panel is factored on the CPU while the update is performed by the GPU. In our experiments, this solver turns out to be faster than the classical GEPP implementation in MAGMA for square matrices larger than 5120 and when using a sufficient number of threads. H-CALU is more scalable, allowing us to use larger panels and thus to limit the amount of transfer between the CPU and the GPU memory. We point out that further optimizations are possible with e.g. additional tuning and scheduling but our experiments give a general trend for the performance of algorithms as dictated by the amount of communication that they perform. However, the solver based on randomization always outperforms other solvers since we do not pivot at all and the randomization cost is small. The good per-

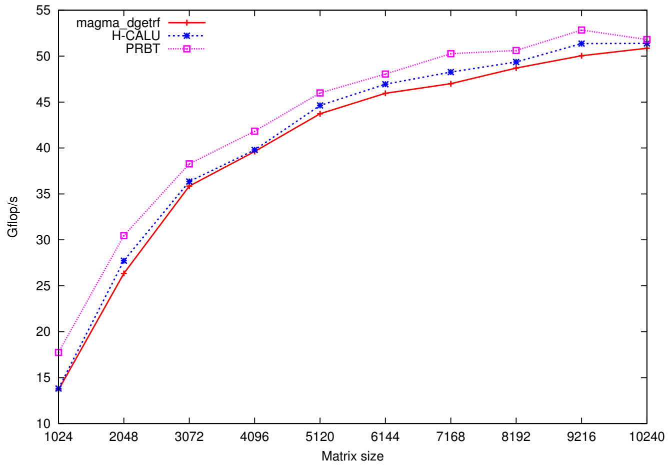


Figure 7: Performance on AMD + GTX260, 4 threads

formance of H-CALU on rectangular matrices is promising in the perspective of extending this approach to multiple GPUs.

## References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1999. Third edition.

[2] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-Avoiding QR decomposition for GPUs. 2011. LAPACK Working Note 240, proceedings of IPDPS'11.

[3] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. 2011. LAPACK Working Note 246.

[4] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[5] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. 2006. In Proceedings of PARA 2006, Workshop on state-of-the art in scientific computing.

[6] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20:1573–1590, 2007.

[7] J. Choi, J. Dongarra, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical report, 1995. LAPACK Working Note 100.

[8] S. Donfack, L. Grigori, and A. K. 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, I. Duff, D. Sorensen, and H. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998.

[10] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszcsek. Achieving numerical accuracy and high performance using recursive tile LU factorization. LAPACK Working Note 259.

[11] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[12] G. H. Golub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. Third edition.

[13] L. Grigori, J. Demmel, and H. Xiang. Calu: a communication optimal lu factorization algorithm. *SIAM Journal on Matrix Analysis and Applications*, 32:1317–1350, 2011.

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

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

[16] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002. Second edition.

[17] Intel. *Math Kernel Library (MKL)*. `http://www.intel.com/software/products/mkl/`.

[18] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multicore processors with pipelining and a look ahead. 2006. LAPACK Working Note 178. Also available as University of Tennessee Technical Report UT-CS-06-581.

[19] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi GPUs. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[20] D. S. Parker. Random butterfly transformations with applications in computational linear algebra. Technical Report CSD-950023, Computer Science Department, UCLA, 1995.

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

[22] S. Tomov, R. Nath, and J. Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.

# Contents

**1 Introduction 3**

**2 Two existing hybrid algorithms for LU factorization 5**

2.1 LU factorization based on partial pivoting . . . . . . . . . . . . . 5

2.2 PRBT solver . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

**3 Hybrid communication-avoiding LU factorization 7**

3.1 Communication-avoiding LU . . . . . . . . . . . . . . . . . . . . . . 7

3.2 Hybrid version of CALU (H-CALU) . . . . . . . . . . . . . . . . . . 8

**4 Numerical experiments 9**

4.1 Panel factorization . . . . . . . . . . . . . . . . . . . . . . . . . . 9

4.2 Performance of hybrid LU implementations . . . . . . . . . . . . 10

**5 Conclusion 13**

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399