LU Factorization for Accelerator-based Systems

Emmanuel Agullo*, Cédric Augonnet*, Jack Dongarra†, Mathieu Faverge†,
Julien Langou‡, Hatem Ltaief† and Stanimire Tomov†
*INRIA, LaBRI, University of Bordeaux, France
Email: Emmanuel.Agullo,Cedric.Augonnet@inria.fr
†Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA
Email: Dongarra,Faverge,Ltaief,Tomov@eecs.utk.edu
‡University of Colorado Denver, CO 80202, USA
Email: julien.langou@ucdenver.edu

***Abstract*—Multicore architectures enhanced with multiple GPUs are likely to become mainstream High Performance Computing (HPC) platforms in a near future. In this paper, we present the design and implementation of an LU factorization using tile algorithm that can fully exploit the potential of such platforms in spite of their complexity. We use a methodology derived from previous work on Cholesky and QR factorizations. Our contributions essentially consist of providing new CPU/GPU hybrid LU kernels, studying the impact on performance of the looking variants as well as the storage layout in presence of pivoting, tuning the kernels for two different machines composed of multiple recent NVIDIA Tesla S1070 (four GPUs total) and Fermi-based S2050 GPUs (three GPUs total), respectively. The hybrid tile LU asymptotically achieves 1 Tflop/s in single precision on both hardwares. The performance in double precision arithmetic reaches 500 Gflop/s on the Fermi-based system, twice faster than the old GPU generation of Tesla S1070. We also discuss the impact of the number of tiles on the numerical stability. We show that the numerical results of the tile LU factorization will be accurate enough for most applications as long as the computations are performed in double precision arithmetic.**

***Keywords*-High Performance Computing; Dense Linear Algebra; LU Factorization; Hybrid Architecture; Multiple GPU Accelerators; Multicore; Tile Algorithm; Numerical Accuracy**

I. Introduction

The LU factorization (or decomposition) of a matrix A consists of writing that matrix as a matrix product $A = LU$ where L is lower triangular and U is upper triangular. It is a central kernel in linear algebra because it is commonly used in many important operations such as solving a non-symmetric linear system, inverting a matrix, computing a determinant or an approximation of a condition number. In most cases, the decomposition is the computationally dominant step. For instance, solving a linear system ($Ax = b$) of n equations can be performed as an LU factorization ($A = LU$), requiring $\theta(n^3)$ operations, and two triangular substitutions ($Ly = b$ and $Ux = y$), requiring $\theta(n^2)$ operations. The performance of the decomposition is thus critical and needs to be optimized especially for the emergent platforms. This is why a large effort was recently dedicated to accelerate it on multicore platforms [1], [2]. In this paper, we propose to accelerate the LU factorization on a multicore node enhanced with multiple GPU accelerators. We follow a methodology previously employed in the context of the Cholesky factorization [3] and QR factorization [4] that we apply to the tile LU decomposition algorithm [1]. We bring four contributions. First, we present the design of new CPU/GPU hybrid kernels for performing an LU factorization on a GPU associated to a CPU core. Second, we study the impact on performance of the looking variants as well as the storage layout (row or column major) in presence of pivoting. Third, we adapt and tune our kernels for the recent NVIDIA Tesla S1070 and S2050 (Fermi) Computing Processors and we present experimental results on two machines composed of such multiple GPUs. And last but not least, we present the impact of the number of tiles on the numerical accuracy.

The reminder of the paper is organized as follows. In Section II, we present the LU factorization, some related work and our experimental environment. The LU algorithm we rely on, so-called tile LU factorization, splits the computational routines into tasks of fine granularity that can be run concurrently on different GPUs. We describe the individual tasks, the kernels, and their optimized hybrid CPU/GPU implementation in Section III. In sections IV, V and VI, we show how we schedule those operations through two different runtime environments and discuss the subsequent performance. We briefly discuss the numerical accuracy of tile LU factorization in Section VII to motivate the use of an efficient double precision factorization. We conclude and present our future work in Section VIII.

II. Background

A. Standard LU Algorithm

LAPACK and ScaLAPACK are the current de facto standard libraries for performing advanced dense linear algebra operations. The LU factorization implemented in those packages is designed as a high-level algorithm relying on basic block routines from the Basic Linear Algebra Subprograms (BLAS) and the Basic Linear Algebra

Communication Subprograms (BLACS) for LAPACK and SCALAPACK, respectively. The matrix is conceptually split in blocks of columns, so called *panels*. At each column of the panel being factorized, a partial pivoting scheme is used. It consists of finding the element of highest magnitude (so-called *pivot*) within the column, swapping the corresponding rows and then dividing the other column elements by that value. This process is repeated for each column within the panel. The factorization then proceeds by an update procedure of the trailing submatrix at each step of the factorization until all the subsequent panels are processed. This panel-update sequence, called *blocking*, enables the use of the level-3 BLAS matrix multiplication (`gemm`) leading to a better data reuse, which is critical on cache-based architectures. However, a synchronization point is required in-between, similar to a fork-join paradigm. Furthermore, the parallelism only occurs within each phase (panel or update) and is expressed at the level of BLAS and BLACS, which ultimately leads to a limited performance.

B. Tile LU Algorithm

To alleviate this bottleneck, so-called *tile algorithms* permit to break the panel factorization and the update of the trailing submatrix into tasks of fine granularity. The update phase can now be initiated while the corresponding panel is still being factorized. Initially proposed for *updating factorizations* [5], the idea of tile algorithms consists of splitting the panels in square submatrices (so-called *tiles*). Using updating techniques to obtain tasks of fine granularity was first exploited in the context of out-of-memory (often called *out-of-core*) factorizations [6]. Two implementations of the tile LU factorization for multicore architectures were more recently proposed [1], [2]. Tile LU algorithm annihilates matrix elements by tiles instead of rectangular panels as in LAPACK. In this paper, we use the high-level algorithm presented in [1], which relies on four kernels and whose first panel-update sequence is unrolled in Figure 1.

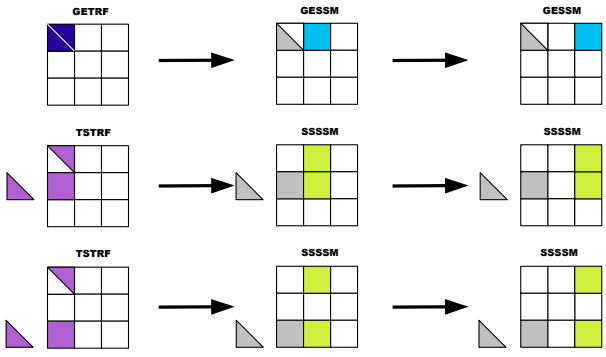


Figure 1. Tile LU Factorization: first panel factorization and corresponding updates.

C. Related work

Various projects provide GPU kernels for linear algebra. The CULA library implements many BLAS and LAPACK kernels [7], but those are limited to a single GPU and problems that fit into the memory. Kurzak *et al.* accelerated the QR decomposition of the PLASMA library with a single GPU by designing non standard BLAS kernels. The PLAPACK [8] and the DPLASMA [9] linear algebra libraries both target accelerator-based clusters. Both of them rely on their own runtime systems that provide intra-node dynamic scheduling capabilities.

Different runtime systems were designed to support accelerator-based platforms. The StarSs [10] language is an annotation-based language executed either on CPUs, GPUs or Cell processors, respectively, using SMPSs, GPUSs or CellSs. Diamos and Yalamanchili also propose to use features that are similar to those available in StarPU in the Harmony runtime system [11]. Sequoia [12] statically maps hierarchical applications on top of clusters of hybrid machines, and the Charm++ runtime system was extended to support GPUs [13].

D. Experimental environment

The experiments have been conducted on two different hybrid platforms in single and double precision arithmetics. The first machine is a quad-socket quad-core host machine based on an AMD Opteron(tm) Processor operating at 2.4 GHz. The cache size per core is 512 KB and the size of the main memory is 32 GB. A NVIDIA Tesla S1070 graphical card is connected to the host via PCI Express 16x adapter cards. It is composed of four GPUs C1060 with two PCI Express connectors driving two GPUs each. Each GPU has 4 GB GDDR-3 of memory and 30 processors (240 cores), operating at 1.44 GHz. The computational time ratio between single and double precision arithmetics on this Tesla S1070 is 6. We will refer to this machine as **Tesla-C1060**. The second machine is a dual-socket hexa-core host machine based on INTEL Nehalem Processors operating at 2.67 GHz. Each core has 256 KB of L2 cache and each socket has 12 MB of L3 cache. The size of the main memory is 48 GB. Three NVIDIA Fermi C2050 cars are connected to the host with a 16x PCI bus. Each GPU has 3 GB of GDDR-5 of memory and 14 processors (448 cores), operating at 1.15 GHz. ECC was disabled on these cards to have as much memory as possible. The computation in double precision arithmetic has been fixed in this last GPU generation and the ratio compared to single precision arithmetic is now two. We will refer to this machine as **Fermi-C2050**.

III. Kernel Implementation

A. Design

The tile LU algorithm of interest is based on four computational kernels, described as follows:

GETRF This kernel performs the standard LU factorization of a tile as in LAPACK and generates a permutation vector according to the pivot selection;

GESSM This kernel applies to a tile from the left the permutations followed by a triangular solve using the permutation vector and the lower triangular tile computed by the GETRF kernel;

TSTRF This kernel performs the block LU factorization on a pair of tiles, one on top of the other, where the upper tile (also the diagonal tile) is already upper triangular. A permutation vector is also produced as well as a unit, lower triangular tile as a result of swapping rows in order to prevent overwriting of existing data;

SSSSM This kernel applies the permutations and the transformations computed from the TSTRF kernel to two stacked tiles from the left. A triangular solve is applied to update the top tile using the unit, lower triangular tile from TSTRF. A matrix-matrix multiplication is then applied to update the bottom tile using the full tile computed by TSTRF.

Critical for the performance of the overall algorithm, SSSSM is the most compute intensive kernel and asymptotically dominates all three other kernels in terms of flops. We integrated MAGMA BLAS matrix-matrix multiplication (`gemm`) and triangular matrix solver (`trsm`), specifically tuned for the tile LU algorithm. Moreover, we developed a hybrid TSTRF kernel which takes advantage of look-ahead technics, similar to the one-sided hybrid LU factorizations in MAGMA for single GPU. The other two kernels, GETRF and GESSM, were adapted and derived from the MAGMA library. Those kernels have been implemented in single and double precision arithmetics.

B. Tuning

Two parameters must be selected to tune the tile-LU algorithm: the size of the tiles, and the internal blocking used by the different kernels. Empirical tuning was performed on the most compute intensive kernel (SSSSM). While it is crucial to optimize the most time consuming kernel, large internal blocking results in a fast SSSSM kernel but slows down the overall performance due to the overhead brought by the extra flops [14]. Several sets of parameters were thus tested to find the combination that not only maximizes the performance of the kernel, but also that of the overall algorithm.

Once tuned for a specific hardware, an efficient runtime environment to ensure a proper scheduling across the host (multicore system) and the device (GPU accelerators) becomes paramount.

IV. Static scheduling

A. Design

Originally implemented to schedule the Cholesky and QR factorizations on the Cell processor [15], the hand-coded static scheduler has been extended to handle the dispatch of the hybrid kernels (from Section III) across all CPU-GPU pairs available on the system in one dimensional cyclic fashion. Figure 2 shows how the *column-wise* partitioning is achieved. This static runtime imposes a linear scheduling order on all the kernels during the factorization. This order enforces the execution of a predetermined subset of kernels on a particular CPU-GPU pair.

Two distinct global progress table are required to first ensure numerical correctness and second, to get high performance. The first progress table keeps track of dependencies among the different tasks at different steps of the factorization. A dependency check is performed before executing each kernel by examining the local copy of the progress table. The hosting CPUs stall with busy waiting on volatile variables until the corresponding dependencies are satisfied, which simultaneously triggers the release of the kernel to the designated CPU-GPU pair.

The second progress table is required to avoid unnecessary copies from/to CPU to/from device GPU and therefore, optimize the performance of the overall algorithm. In fact, at any specific time of the computation, a same data tile can reside both on the host's and/or device's main memory (MM). The issue is to determine the consistency of the data. This problem is well-know in cache coherency protocols. The second progress table helps to keep track of:

- which data tile is considered as final output to ensure data consistency with the other processing units (PUs). If the tile is located on the device's MM, it is copied back to the host's MM for a true shared access with the other PUs available on the system.
- which data tile is still transient and has not yet reached its final state. This tile cannot be shared among other PUs (false-sharing).

This progress table is incremented at each step of the factorization. Each CPU thread hosting a GPU probes the progress table about the state of the data tile it is about to access. If the data tile state is final, no transfer is necessary and the computation proceeds. If its state is transient, the CPU thread will initiate a transfer to its GPU in order to acquire the up-to-date data tile prior to execute its scheduled task.

This turns out to considerably decrease the number of communications involved between a CPU-GPU pair, which is critical given that the PCI bus is two orders of magnitude less efficient than the computational power of the accelerators.

V. Algorithmic Optimizations

We introduce four incremental optimizations integrated into the application to improve the overall performance. First, we manually adjust the order of the loops, and then we optimize the compute kernels so that they fit to the specificities of GPUs.

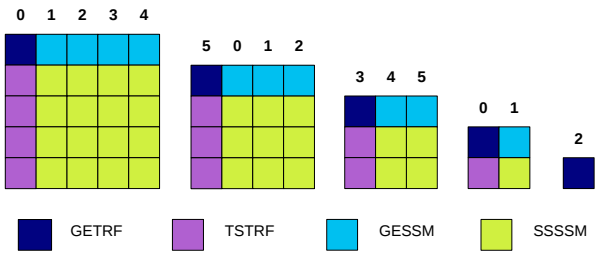


Figure 2. Task partitioning across six processing units.

A. Looking Variants

We study in this section the effect of three looking variants, which have an influence on the overall scheduling.

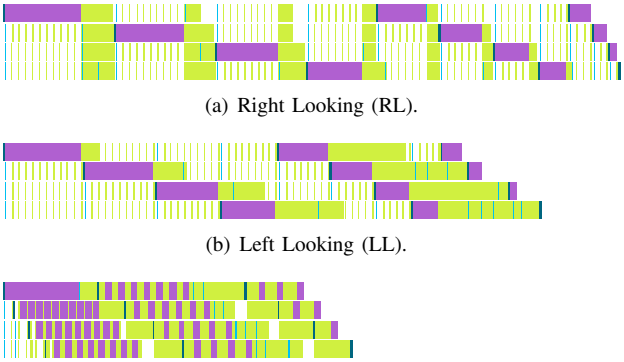
The right-looking variant (RL) consists in updating first the trailing submatrix, located on the right side of the current panel, before the execution of the next panel starts. This looking variant generates many concurrent tasks which can potentially run in parallel.

The left-looking variant (LL), also called the *lazy* variant, applies all subsequent updates generated from the left side to the current panel before proceeding with the next panel. Since the updates are restricted to the panel only, data reuse is maximized while at the same time parallelism can get limited.

The methodology of this static scheduling allows for pipelined execution of factorizations steps, which usually provides similar benefits to dynamic scheduling to some extent. For example, the execution of the inefficient Level 2 BLAS operations can be overlapped by the efficient Level 3 BLAS operations. This phenomenon has been successfully demonstrated for the scheduling of one-sided factorizations in the context of homogeneous multicore [16]. However, when tackling hybrid system components, i.e., multicore associated with GPU accelerators, a tremendous gap in terms of sequential performance may exist between the hybrid CPU-GPU kernels and the GPU kernels as seen in Section III-B. Therefore, one of the main disadvantages of the static scheduling is a potential suboptimal scheduling, i.e., stalling in situations where work is available.

Figures 3(a) and 3(b) clearly describe this drawback of the static scheduling on four CPU-GPU pairs. The dark purple colors represent the panel tasks (GETRF and TSTRF) and the light green colors are the update kernels (GESSM and SSSSM). The panel tasks are hybrid and the GPU needs the CPU to perform the Level 2 BLAS operations while the update kernels are highly efficient Level 3 BLAS operations performed on the GPU only. Figure 3(a) shows the RL variant with lots of stalls represented by white empty spaces. The panel tasks indeed become a bottleneck and the updates tasks cannot proceed until the completeness of the panel tasks. Figure 3(b) presents the LL variant. The scheduling contains less gaps but still suffers from the lack of parallelism, especially in the beginning. And this inefficiency is even more exacerbated by the slow panel hybrid kernels.

A new looking variant has then been implemented to alleviate this bottleneck combining the previous LL version with a breadth-first search task execution (BF-LL). Each CPU-GPU pair applies all subsequent transformations, once for all, (update and panel tasks) on a particular tile on the current panel before proceeding with the tile below it. The obtained trace as seen in Figure 3(c) is very compact and dense as shown in the bottom trace. On Figure 4, the BF-LL variant therefore clearly outperforms the other variants for all problem sizes. Noteworthy to mention that some natural load imbalance starts to appear toward the end of the traces. This explains the irregularities observed on the different curves of Figure 4.



(a) Right Looking (RL).

(b) Left Looking (LL).

(c) Breadth-First (BF-LL).

Figure 3. Trace of the different looking variants.

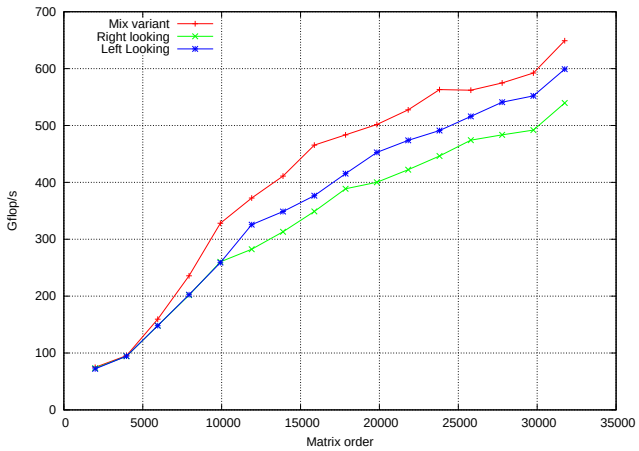


Figure 4. Looking variants for static SGETRF on **Tesla-C1060**

B. Block Pivoting

LAPACK and ScaLAPACK apply permutations sequen-

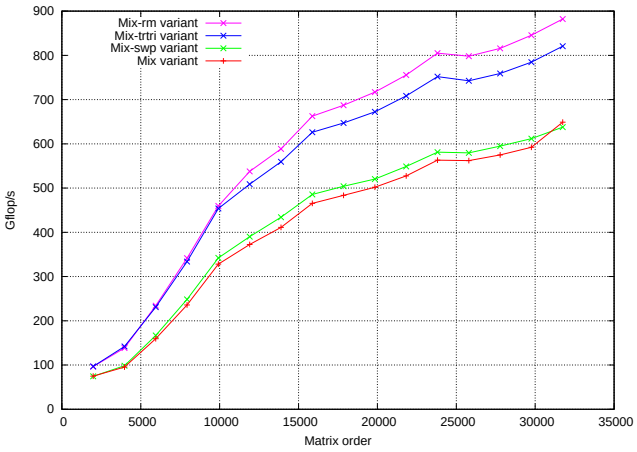


Figure 5. Algorithmic optimizations for static SGETRF on **Tesla-C1060**

tially i.e., they swap rows one after another. This operation can be extremely slow on GPUs. For that reason, we have implemented a single GPU kernel that applies all permutations within a block at once. Applying this technique to the BF-LL variant improves the performance by a few percents on Figure 5.

C. Triangular Solve

The second optimization consists in splitting the call to triangular solve (`trsm`) on the GPU by a triangular matrix inversion (`trtri`) on the CPU followed by (1) a triangular matrix-matrix multiplication (`trmm`) and (2) a matrix-matrix multiplication (`gemm`) both on the GPU to further improve the performance of the most compute intensive GPU kernel. The same modification has been proposed by Volkov and Demmel [17]. This affects the overall stability analysis of the algorithm but, as long as a reasonable pivoting strategy is used (so that the norm of L is not too large), we can guarantee that the scheme stays normwise stable. The benefit in terms of performance is however very significant: on Figure 5, the asymptotic speed is now 821 Gflop/s compared to 638 Gflop/s before.

D. Storage Layout

One critical part of the standard LU factorization is the pivoting scheme. The LU algorithm as implemented in LAPACK uses partial pivoting which generates swapping between rows. The tile LU factorization uses a pairwise pivoting scheme which implies pivoting between a pair of tiles. This generates more pivoting than the standard LU factorization and also presents some stability issues which are explained in Section VII. The storage layout optimization enables to improve the memory accesses on the GPU when pivoting by transposing the matrix accordingly. The swapping of rows, which are now contiguous in memory, can use coalescent memory accesses to substantially improve the overall performance of the application.

Implementation and experimentation were therefore carried out with tiles in two storage layouts – column and row major correspondingly. On Figure 5, we obtain about 882 Gflop/s by converting the matrix into a Row-Major layout. Previous work for example, has shown that the computational cost for pivoting can be prohibitively high for performance.

E. Scalability

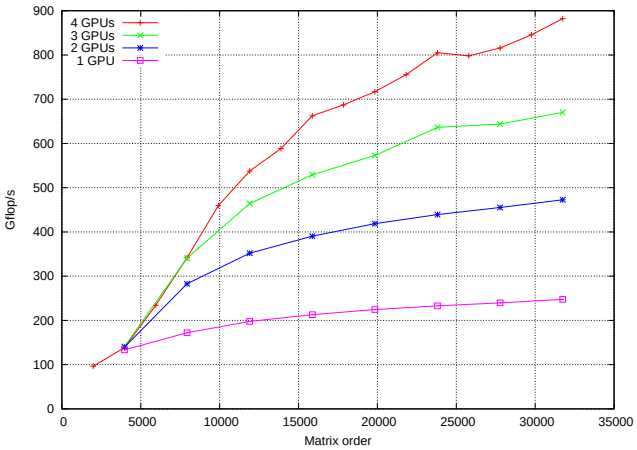


Figure 6. Scalability of static SGETRF on **Tesla-C1060**

Figure 6 shows the strong scalability of the SGETRF algorithm with respect to the number of GPU-CPU pairs. We obtain a 1.9 speedup on 2 GPUs and a 3.6 speedup on 4 GPUs. While the successive optimizations ensure that enough parallelism is created, the scalability is limited by the contention on the bus. As shown on the trace on Figure 3(c), this statically scheduled algorithm also suffers from some load imbalance which affects its scalability.

VI. Dynamic Scheduling

The dynamic scheduler described in this section proposes a more productive and portable approach to schedule the tile LU factorization across a hybrid system.

A. Design

The application to be scheduled can be split into multiple tasks. We use the StarPU [18] runtime system to schedule these tasks dynamically simultaneously on CPU cores and on CUDA devices. StarPU is a library that schedules tasks on accelerator-based platforms. It automates data management and provides a flexible framework to design scheduling policies. The two main steps required to port an application on StarPU are first to register all data to StarPU, and then to split the computation into tasks that access registered data. StarPU provides performance portability by making it possible to concentrate on purely algorithmic concerns and on kernel optimization rather than dealing with low-level and non portable problems such as data management.

1) Data management: Once a piece of data has been registered, StarPU ensures that it is available for the different tasks that need to access it, either from a CPU or from a GPU. StarPU keeps track of the location of the different data replicates and make sure that they remain coherent: when a task modifies a piece of data, all other replicates are automatically invalidated.

2) Task scheduling: The task structure proposed by StarPU contains a multi-versioned kernel, named *codelet*, and the list of data which should be accessed by the task, as well as the access mode (e.g., read or read/write). The codelet structure encapsulates the different implementations of the task on the different types of processing units: when StarPU assigns the task to a certain type of processing unit (e.g., CUDA or OpenCL), the appropriate implementation is called. StarPU will not assign a task to a processing unit for which the codelet is not implemented. It is therefore not required to give an implementation of the codelet for each type of processing unit.

Task dependencies are either explicit or implicitly derived from data dependencies. In the tile-LU algorithm, StarPU automatically detects dependencies so that we simply unfold the DAG describing the tile-LU algorithm in a sequential fashion.

StarPU provides various scheduling strategies to address a wide scope of applications. For the tile-LU algorithm, we used the HEFT-TMDP-PR policy. This strategy relies on performance predictions based on previous executions, which make it possible to accurately distribute the load between the heterogeneous processing units. This strategy also minimizes data transfer overhead by initiating asynchronous data transfers as soon as a task is attributed to a processing unit prior to its actual execution. The total amount of data transfers is also reduced by predicting data transfer time, and therefore avoiding to schedule tasks on processing units that do not have a local copy of the input data. More details on the HEFT-TMDP-PR policy are available in a previous study [18].

B. Performance results

Figure 7 present performance results obtained with either the static or the dynamic schedulers. The three optimizations of sections V-D, V-B and V-C are applied again. Dynamically scheduled tasks are submitted following the Right-Looking ordering. We therefore show the speed of both the Right-Looking (RL) and the Breadth-First Looking (BF-LL) statically scheduled algorithms. It is worth noting that data transfer time is not accounted in the static algorithms. Contrary to the static algorithm that was manually tuned to minimize the amount of data transfers and to balance the load, the dynamic scheduler has to take decisions even before all tasks have been submitted. For small problems, static algorithms thus perform better than the dynamic one. For larger problems, the dynamic scheduler is able

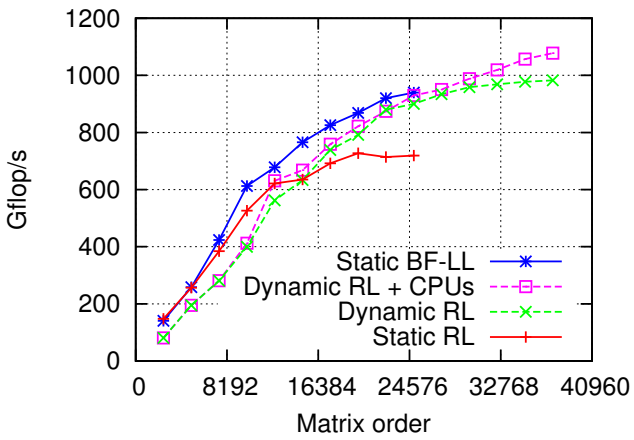


Figure 7. Static vs. Dynamic scheduling on the 3 GPUs of **Fermi-C2050**

to distribute the load better than the RL algorithm even though tasks are submitted in the same order in both cases. The static algorithms are unable to handle problems larger than the size of the GPUs, while StarPU transparently handles larger problems. In the future, we plan to unfold the dynamically scheduled DAG following the BF-LL order in order to guide the dynamic scheduler. This would make it possible to benefits from the expertise introduced within the static algorithm while preserving code maintainability. Performance portability is also illustrated on the curve obtained when using all CPUs in conjunction with the GPUs on Figure 7.

VII. Numerical Accuracy

We have mentioned above that the use of double precision may be critical when relying on the Tile LU factorization. We now illustrate this claim. The purpose of this section is not to accomplish a full study of the numerical stability and accuracy of the Tile LU algorithm; this would deserve an entire manuscript. Our goal is to justify that Tile LU algorithm can be used in practice on the matrices we usually consider and to highlight its limits.

Pivoting is necessary during the LU factorization process in order to ensure the stability of the algorithm. The standard way to perform pivoting is *partial pivoting*: for each column, one pivot is used to eliminate all the others elements of this column. The pivot is chosen to be an element in the column with the largest magnitude. This scheme is, for example, implemented in LAPACK and ScaLAPACK and it is *practically stable*; in the sense that pathological matrices exist for which it is not stable; but, these matrices have not been observed in practical applications. During the Tile LU algorithms, the partial pivoting scheme is altered in order to enable a pipeline in the update of the columns. Indeed, for a single column, several pivots can be used during the elimination process. The pivoting scheme for Tile

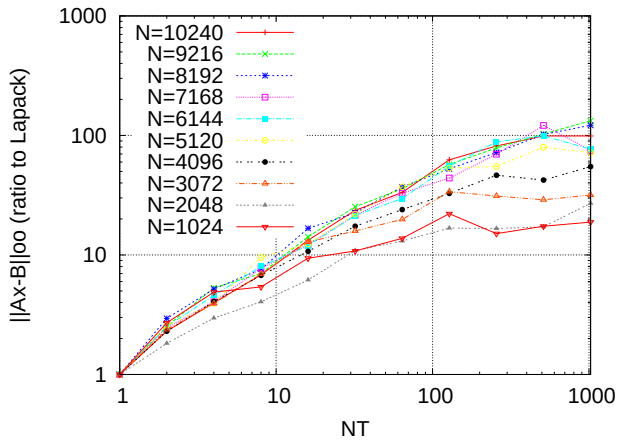


Figure 8. Numerical accuracy of the residual ($\|Ax - b\|\infty$, logarithmic y-axis) when the number of tiles per row (t, logarithmic x-axis) varies, for random matrices A of various orders N in double precision. The residual is normalized to the residual obtained with LAPACK.

LU is a block variant of *pairwise pivoting* and we call it *block pairwise pivoting*. Stability of pairwise pivoting has been studied in [19]. The authors experimentally showed that pairwise pivoting is less stable than partial pivoting. This motivates this experimental section on the stability of block pairwise pivoting. A preliminary study has been proposed in [20] but this only addresses the case of two tiles. In practical applications, our algorithms will have tens to hundreds of tiles.

Our main result can be stated as follows. We experimentally observe that, when compared to partial pivoting, the stability of block pairwise pivoting is independent of the size of the tiles and solely depends on the number of tiles in a column. As the number of tiles increases, the accuracy of the factorization deteriorates. We observe that, on random matrices tile LU performed on $1,000$-by-$1,000$ tile matrix loses at most 2-digit of accuracy with respect to partial pivoting.

Here, we consider ten matrices of order ranging from 1024 to 10240 that were randomly generated with a uniform distribution. For each matrix, Figure 8 shows the infinity norm of the residual ($\|Ax - b\|\infty$) obtained for Tile LU relatively to the residual obtained with LAPACK. LAPACK LU algorithm implements a standard partial pivoting scheme consisting of choosing the element of highest magnitude as pivot for the current column. When using one single tile, Tile LU exactly performs a partial pivoting scheme. Therefore, Tile LU and LAPACK lead to the exact same residual in that case ($t = 1$, leftmost point of Figure 8). When the number of tiles per row (the matrix is composed of $t \times t$ tiles if there are t tiles per row) increases, Figure 8 shows that the numerical accuracy decreases. One decimal digit of accuracy is lost (ratio equal to 10) when 10 to 30 tiles are used, depending on the matrices considered. On the right-most part of the graph (the scale is logarithmic), we study the loss of accuracy when considering up to an improbable number of $1,000$ tiles that would not fit in the memory of our target machines if we were considering a tile size optimized for performance. Only two matrices out of ten loose a second decimal digit of accuracy (ratio equal to 100). One of our main observation is that the stability is influenced by the number of tiles used in the algorithm, and is not influenced by the size of the tiles. If this cannot be considered as a statistical result, it still shows that the degradation of Tile LU on numerical accuracy is not necessarily dramatic. Indeed, an iterative refinement procedure often recovers several decimal digit per iteration. Therefore, a Tile LU factorization in double precision is likely to lead to a very acceptable numerical accuracy. On the contrary, in single precision, the numerical accuracy of the standard LU factorization with partial factorization is often barely high enough to yield to the solution. Therefore, the computation of the solution of a linear system may fail because of one or two lost decimal digits.

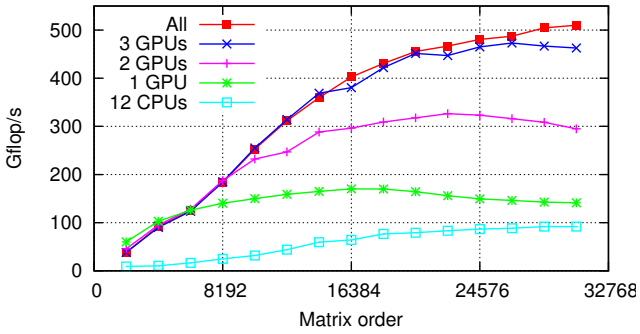


Figure 9. Scalability of dynamic DGETRF on **Fermi-C2050**

As a partial conclusion, for the matrix sizes and tile sizes we consider, we have shown that the obtained accuracy is critical in single precision but is likely to be acceptable in double precision, emphasizing the importance of efficient numerical solvers in double precision. On Figure 9, the dynamically scheduled DGETRF kernel reaches 500 Gflop/s on our Fermi-based system, which is half of the speed obtained by the single precision algorithm on the same platform. The different results that we have presented in single precision are also directly applicable to the double precision implementation.

VIII. Conclusion

We have presented the design and implementation of a new hybrid algorithm for performing the tile LU factorization on a multicore node enhanced with multiple GPUs. This work completes our study of one-sided factorizations on such platforms. We have shown that we could reach very high performance thanks to new CPU/GPU hybrid kernels. These kernels have been specifically tuned to fully exploit

the potential of the NVIDIA Tesla S1070 and the new generation Fermi-based S2050 GPU accelerators. Tile LU achieves 1 Tflop/s in single precision arithmetic and 500 Gflop/s in double precision on the Fermi-based system. The numerical accuracy of the LU factorization is very sensitive to the underlying algorithm. From a high-level point view, we rely on the Tile LU factorization algorithm. We have exhibited that the numerical accuracy decreases when the number of tiles increases. For the matrix sizes and tile sizes we consider, we have shown that this degradation is critical in single precision but acceptable in double precision, highlighting the importance of using efficient double precision hardware and software.

This work aims at unifying PLASMA (initially designed for homogeneous multicore architectures) and MAGMA (initially designed for single-GPU architectures) and the resulting software will be incorporated in those libraries. The purpose is to constitute a library equivalent in functionality to LAPACK. We plan to pursue this work with two-sided factorizations (Hessenberg reduction, tridiagonalization and bidiagonalization). We are also investigating memory bound algorithms such as the solution step that follows a one-sided factorization. For those algorithms, minimizing data movement is critical. This is still a complex and open problem on heterogeneous machines.

REFERENCES

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

[2] E. Chan, E. S. Quintana-Ortí, G. Gregorio Quintana-Ortí, and R. van de Geijn, "Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures," in *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, June 2007, pp. 116–125.

[3] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, *GPU Computing Gems*. NVIDIA, 2010, vol. 2, ch. Faster, Cheaper, Better - a Hybridization Methodology to Develop Linear Algebra Software for GPUs, accepted, to appear.

[4] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators," *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*.

[5] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 2nd ed. Baltimore, MD, USA: The Johns Hopkins University Press, 1989.

[6] E. L. Yip, "Fortran subroutines for out-of-core solutions of large complex linear systems." *Technical Report CR-159142, NASA*, November 1979.

[7] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 7705, Apr. 2010.

[8] M. Fogué, F. D. Igual, E. S. Quintana-ortí, and R. V. D. Geijn, "Retargeting plapack to clusters with hardware accelerators flame working note #42," 2010.

[9] G. Bosilca, A. Bouteiller, T. Herault, P. Lemarinier, N. Saengpatsa, S. Tomov, and J. Dongarra, "A unified HPC environment for hybrid manycore/GPU distributed systems," LAPACK Working Note, Tech. Rep. 234, Oct. 2010.

[10] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.

[11] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 197–200.

[12] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE SC'06 Conference*, 2006.

[13] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling Hierarchical N-body Simulations on GPU Clusters," in *Proceedings of the ACM/IEEE Supercomputing Conference 2010 (to appear)*, 2010.

[14] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra, "Comparative study of one-sided factorizations with multiple software packages on multi-core hardware," *2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '09)*, 2009.

[15] J. Kurzak and J. J. Dongarra, "QR factorization for the CELL processor," *Scientific Programming, Special Issue: High Performance Computing with the Cell Broadband Engine*, vol. 17, no. 1-2, pp. 31–42, 2009.

[16] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurrency Computat.: Pract. Exper.*, vol. 21, no. 1, pp. 15–44, 2009, DOI: 10.1002/cpe.1467.

[17] V. Volkov and J. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," Tech. Rep. UCB/EECS-2008-49, 2008.

[18] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," INRIA, Technical Report 7240, Mar. 2010. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00467677

[19] L. N. Trefethen and R. S. Schreiber, "Average-case stability of Gaussian elimination," *SIAM J. Matrix Anal. Appl.*, vol. 11, pp. 335–360, May 1990.

[20] E. S. Quintana-Ortí and R. A. van de Geijn, "Updating an LU factorization with pivoting," *ACM Trans. Math. Softw.*, vol. 35, no. 2, pp. 1–16, 2008.