Performance evaluation of LU factorization through hardware counter measurements

Simplice DONFACK* Stanimire Tomov † Jack Dongarra ‡

October 1, 2012

Abstract

The growing demand for scalable and effective scientific and numerical libraries on multicore architectures forces hardware manufacturers to design solutions that improve both the processor speed and transfer rates between their memory hierarchies. Several studies show that these improvement factors are disproportionate and may vary widely from one architecture to another and then have a strong impact on the tuning and the performance prediction of numerical libraries. In this paper, we analyze the communication and performance of some routines in well known libraries on different architectures and we establish a relation model between hardware parameters and performance. We focus on the LU factorization, which is one the most popular algorithms in the scientific field, therefore also used as a benchmark, e.g., the HPL benchmark to rank the TOP500 supercomputers. Our experiments in terms of hardware counter measurements allow us to predict the performance behavior of numerical algorithms (LU in particular) on different architectures.

1 Introduction

One of the hardest goals in linear algebra is to implement routines that are efficient and achieve high performance on a variety of platforms. This goal sounds as a contradiction because, on the first hand, routines should be optimized to exploit the possibilities of the underlying architecture and, on the other side, they should be quite independent of the architecture to be portable. Then it is obvious that a routine may be efficient on a specific architecture but leads to significant lost of performance on another architecture. The scalability of such routine is no more guaranteed if it does not take into account some important hardware factors that may impact performances. These factors are: the speed of the processor and the speed of data transfer between several computations units. Unfortunately, these two factors are increasingly disproportionate on different architectures. For example, several studies show that the yearly improvements in processor speed is 4 times faster than the improvements in data transfer speed. By the same way, in order to improve the concurrent

*Innovative Computing Laboratory, University of Tennessee
†Innovative Computing Laboratory, University of Tennessee
‡Innovative Computing Laboratory, University of Tennessee

data access in memory while continuing to increase the number of processors, architectures are involved to become more complex. One of the common point in different architectures is the notion of caches; which denotes small memory units associated to processors aiming at reducing the time and the number of slow memory access. Data manipulated are first searched in top level caches (cache hit) before being sough in lower level of memories that are larger but slower (cache misses). Reducing data transfer time implies a quick data access and then an increasing performance.

The effective use of caches in order to achieve high performance in a parallel program lies on the ability to reuse data already in the cache of the processors. However, for complex routines, this objective usually leads to serious trade-off between the load balancing between processors and the data locality. In fact, an available task during the execution of a routine should be executed by any available processor or by the processor which already has the data associated to the task. On architectures having a fast data transfer between different memory components (such as Intel Nehalem for example) one will tend to opt for the first solution in order to execute as fast as possible critical tasks; while on architectures having remote and slow memory accesses (such as AMD Opteron for example) one will tend to opt for the second one in order to avoid paying the high cost of remote memory access.

PLASMA [4] and MAGMA[3] are two libraries which implement several routines in linear algebra. The purpose of these libraries is to design simple, efficient, and portable functions derived from LAPACK such as to achieve high performance on parallel machines. To avoid loss of performance due to the impact of hardware factors, several approaches and options have been implemented in PLASMA and are actively evaluated in MAGMA. These options include using LAPACK and tile layout for data. The advantage of tile layout is that data associated to a task is stored in a continuous way in memory so, unlike the LAPACK layout, data associated to that task reside fully in the cache of the processors executing it and then do not generate extra cache misses (excepted at the extreme border of the whole block of data). Different scheduling approaches have been developed such as static and dynamic scheduling. Also, solutions of type cache oblivious algorithms[9] have been recently successfully implemented (see parallel recursive panel[8]). The advantage of such class of algorithms is to use effectively caches without knowing or pre-evaluating their sizes. Last but not least, a recent approach based on a new class of algorithms referred to as communication avoiding algorithms [10, 5] has been designed for QR (PLASMA CAQR [13]). The idea behind communication avoiding algorithms is to reduce communications and memory transfers by doing some redundant computations.

Although PLASMA shows high performance on underlined architectures, no studies have been conducted in depth to determine the percentage of performance brought by each optimization separately on the overall performance, as well as its behavior on completely different architecture. Our experiments show that some factors may appear to be neglected or hidden by other optimizations on the same platform. By the same way, no studies in the literature offer a reliable model to guide the trade-off between software optimizations and the impact of hardware factors. In the work of Donfack et al.[6], the authors pointed out the danger of implementing algorithm without taking into account dynamique change in the system as well as system characteristics. In their approach, the author propose a new scheduling strategy that self-adapt itself to these change in the system but it requires several change in existing code in order to be used.

In the first part of this report, we evaluate several hardware impacts on performance in linear algebra routines. To do so, we focus on the LU factorization, which is one of the most solicited algorithms for this type of evaluation because of its high rate of synchronization and data transfer caused by the pivoting. In the second part, we propose a theoretical model to guide the design of routines for futures architectures. In another words, our model should be able to predict the behavior of an algorithm on parallel machines where architecture type, characteristics of the computations units, and memory transfer parameters are known in advance.

The factors that we have selected for this study are classified in the following table.

Pivoting technique	incremental pivoting, recursive parallel panel, TSLU
Cache level and look-aside translation	L1, L2, L3, TLB
Data Layout	LAPACK, Tile, 2D bloc cyclic LAPACK, 2-level block layout
Scheduling strategy	static, dynamic

In the next section, we present a background of PLASMA and communication avoiding LU (CALU). Then, we show the results of the evaluation of impact factors in these libraries. Then we present a theoretical approach to link the performance obtained by these routines to these factors. And finally we give a conclusion.

2 Background

In this section we briefly recall the communication avoiding LU factorization and the LU factorization implementations in PLASMA.

2.1 Data layout

Data layout is important for performance in linear algebra as how the data is stored in memory influences its access and manipulation by the processors. Linear algebra introduced the column major layout (CM) which stores the matrix in the memory column by column as shown on Figure 1 a. This data layout is currently used in LAPACK and SCALAPACK. In LAPACK, the initial matrix is fully represented using a column majour layout while in SCALAPACK, it is cyclically distributed to the different processors, and each processor stores its part in his local memory using the column major data layout. The disadvantage of this layout is that it does not match most of the routines in numerical algebra libraries. For example, the parallel product of matrices (DGEMM) decomposes matrices into small square blocks and performs several products and additions on those small blocks. The square blocks are sub-matrices of the initial matrix and their columns are not continuous in memory, which may cause cache and TLB misses since consecutive columns of the blocks are not stored continuously in the memory. To reduce this problem in shared memory machines, several data layouts have been developed such as: tile data layout [4], 2D Block cyclic data layout, two-level block cyclic data layout [6], and many more. We present in this report the ones actually implemented in CALU [7] and PLASMA [4].

2.1.1 2D Block cyclic data layout

2D Block cyclic data layout (BCL) stores the matrix as in SCALAPACK. Initially, the matrix is cyclically distributed to all threads participating in the computation. Then each thread stores its part of the matrix in its local memory using a column major data layout as shown at the left in Figure 2. In order to physically store a part of the matrix in local memory, each thread allocates a separate memory space which will be physically committed in its local memory by the operating system. This step aims at keeping data as close as possible to the thread that manipulates it in order to avoid latency penalties and make access fast. On NUMA systems, some operating systems do not allocate memory space in the local memory of the processor that requests the allocation (by calling malloc in C). Instead, the operation system reorganizes requested memory into pages of data that are committed into the local memory of the thread which is the first to touch at least one element on that page. This refers to as first touch policy. The idea behind is to avoid the case where only one processor allocates memory that gets to be manipulated by other processors. To ensure that this is not the case, the original matrix can be allocated by the main process, but the blocks of the matrix belonging to a thread be initialized by that thread (so that the corresponding page is committed in its local memory).

2.1.2 Tile data layout

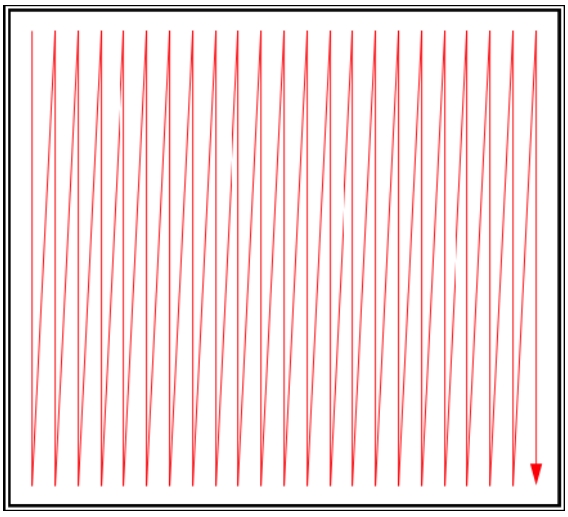
The tile data layout stores the blocks of the matrix into a continuous space in memory. Initially, the matrix is partitioned into square blocks, and each block is stored column by column in memory. From an algorithmic point of view, the matrix is represented as in the column major data layout, but physically, the columns of the same block are stored in continuous memory space as illustrated by the red arrows in Figure 1b. This format allows to minimize cache and TLB misses. For optimal performance, the size of the blocks can be taken so that they fully fit in the cache of the threads that perform operations on them in order to avoid hitting the next level of the cache hierarchy. This data layout is currently implemented in PLASMA and has shown to achieve good performance.

2.1.3 Two-level block cyclic data layout (2l-BL)

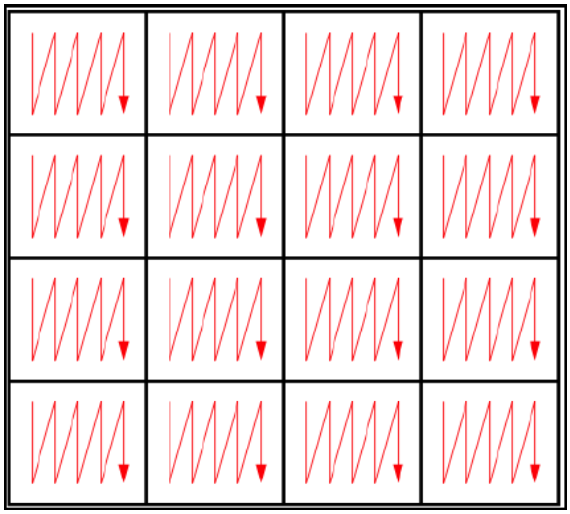
The two level block cyclic data layout is a combination of the 2D block cyclic data layout and the tile data layout. Initially each thread allocates a memory space to store its corresponding part of the input matrix, then each block of the local matrix is stored using the same principle as in the tile data layout. This presents two advantages. First, the part of the initial matrix that a thread owns is stored in its local memory which reduces remote memory accesses. Second, each block of the local matrix is stored in a tile data layout which reduces the accesses to the next cache level (that is shared with the others threads of the same socket). Another good property of the multiple blocking is that it matches perfectly architectures with several level of hierarchical memory. For example, if a part of the block manipulated by a thread is not present in L1 cache, it can be found in the L2 cache, L3 cache and so on.

2.2 PLASMA

In this section, we briefly introduce PLASMA, which is a software that implements efficiently various routines derived from LAPACK. The main goal of PLASMA is to remove the fork and



a. Column major data layout (CM). The matrix is stored column by column in memory.



b. Tile data layout. Blocks of the matrix are stored column by column in memory.

Figure 1: Example of two data layout formats. The figure shows how the matrix is represented in an algorithm point of view and red arrows show how elements are stored physically in the memory for each data layout.

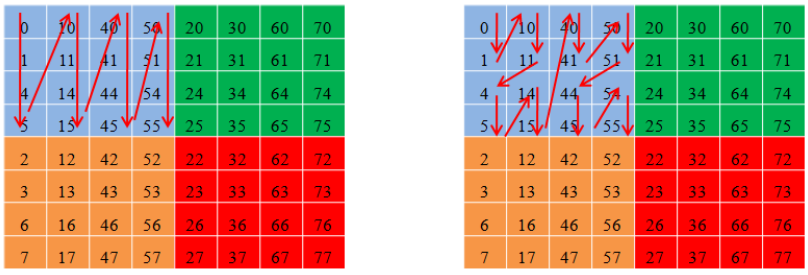


Figure 2: Data layout. The figure on the left displays a matrix partitioning into four blocks using a block cyclic layout (BCL) based on blocks of size $b \times b$. Each of the four blocks is stored contiguously in memory. The figure on the right illustrates the two level block layout (2l-BL) layout, which stores contiguously in memory blocks of size $b \times b$ for each of the four blocks.

join approach actually used in order to bring multithreaded parallelism in LAPACK routines. In principle, the naive parallelization of LAPACK routines calls multithreaded BLAS inside each routines. This solution presents the disadvantage that it requires synchronizations of all the threads at the beginning and at the end of each call of parallel BLAS, which could be damaging for the performance. PLASMA removes this bottleneck by decomposing the computation into tasks that are represented using a DAG. Each task operates on a small part of the matrix and is considered as an atomic operation that can be executed by a thread using sequential BLAS.

In order to reduce the bottleneck introduced by the partial pivoting, PLASMA has recently introduced a new approach, so called parallel recursive panel [8] that works well in practice. The algorithm is based on an approach referred to as cache oblivious [9] that operates on blocks of data without determining the cache size. The principle behind the parallel recursive panel is to use as much as possible the data in the thread cache and then to introduce more Level 3 BLAS operations when updating the small portion of the trailing panel. The process is done recursively.

The previous implementation of LU in PLASMA was based on incremental pivoting where the first block of a panel is factorized and then used to annihilate the off-diagonal blocks. Due to stability issue, this algorithm is no longer used by default in PLASMA. But for information, as it leads to great parallelism at the price of the stability, we evaluate it in this report.

2.3 Communication avoiding algorithms

In this section, we introduce the communication avoiding LU algorithm. The growing communication cost compared to the time to performs arithmetic floating point operations motivates the search for new algorithms that reduce communications. Communication avoiding algorithms are a new class of algorithms that reduce communication by doing some redundant computations. In the LU factorization, it aims at reducing the number of message exchanged during the panel factorization. Contrary to SCALAPACK, where each column of the matrix needs synchronization of processors to place the maximum element on the diagonal during the panel factorization, CALU needs a synchronization only for each columns block of the panel. For a matrix of size n, SCALAPACK will therefore require $O(n \log P)$ messages while CALU will only require $O(\frac{n}{b} \log P)$ messages, where b is the block size. So the larger b is, the lower the number of messages is. For example, for $b = n$ or for only one panel of size n, CALU will requires $O(\log P)$ messages while SCALAPACK will require $O(n \log P)$ messages.

The major difference between CALU and the classic LU factorization lies on the panel factorization. CALU partitions the input matrix into block columns of size b. Then each block column is again partitioned into P blocks, where P is the number of processors participating in the panel factorization. The panel factorization using CALU is illustrated in Figure 3. At the first step of the panel factorization, each thread performs Gauss elimination with partial pivoting on its block, applies the resulting permutation vector on its original block, and then keeps the first b rows of its permuted local blocks as the pivot candidates for the next level of the computation. This step is represented by the dash arrows in 3. The next step of the panel factorization is a reduction operation depending on the underlying reduction tree. For a binary tree, the pivot candidates are merged one on top of another at each node of the tree, then Gauss elimination with partial pivoting is applied again as in the first step. The resulting permutation vector is applied on the original merged blocks and then the pivot candidates are selected for the next level of the reduction. This requires $O(\log P)$ steps. After the reduction operation, the pivot candidates are moved on the top

of the panel being factorized and an LU without pivoting of the entire panel is computed.

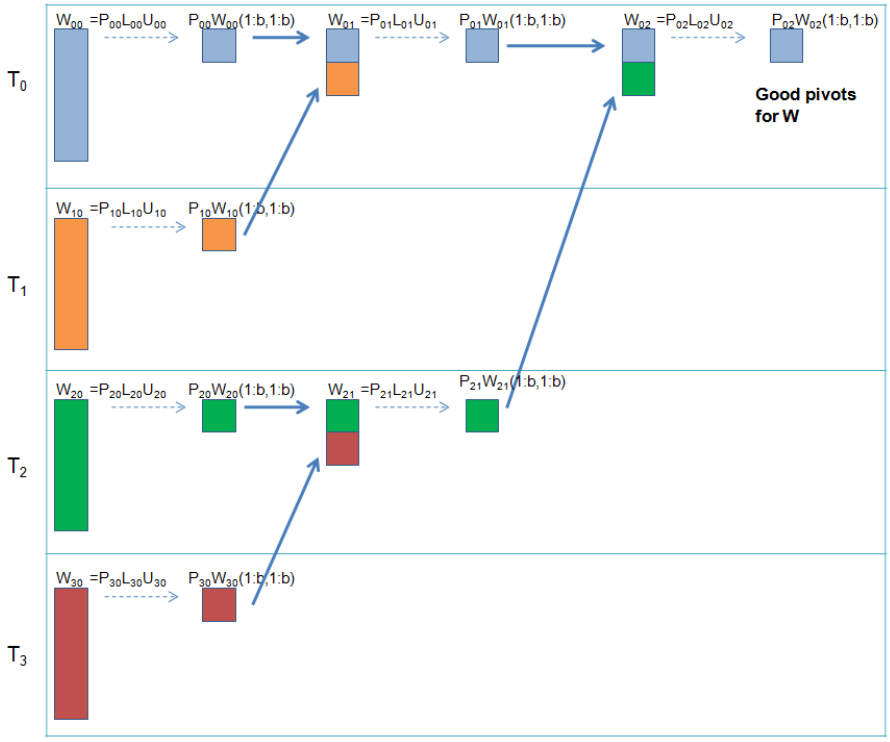


Figure 3: Example of a panel factorizing using CALU with 4 processors on the panel. Every thread is represented by a color. The dashed arrows represent the computations at the first step of the binary and the solid ones represent communication between threads (the transfert of a $b \times b$ block into a merging operation at the node of the reduction tree).

The stability of the algorithm has been proven on a large set of special and random matrices [10]. CALU is less stable than Gauss elimination with partial pivoting, but in practice it leads to a stable algorithm. We present in the next section its implementation for shared memory machines.

2.3.1 Communication avoiding algorithm for multicore

Donfack et al. [7] have adapted and implemented communication avoiding algorithms for multicore architectures. In their approach, the input matrix is partitioned into block columns of size b and each block column is factorized iteratively. At each iteration, a block column referred to as panel is factored, and then the trailing submatrix is updated. In the column major version of CALU (CM), each panel is partitioned into P parts where P is the number of threads participating in the operation while in both BCL and 2l-BL version, the panel is partitioned into blocks of size b. For all versions, the partitioning of the block column in the trailing submatrix follows the same partitioning as the panel. So, the partitioning leads to rectangular blocks for the column major data layout version or for square blocks for the BCL or 2l-BL data layout. Each computation on a block (rectangular or square) is associated to a task; these tasks are represented using a DAG and can be executed in any order as soon as their dependencies are not violated. By using CALU, the panel

factorization is broken into several tasks that can be executed simultaneously and asynchronously following the dependencies dictated by the underlying reduction tree. When the panel is factored, the corresponding update is applied on the different tasks of the trailing submatrix.

3 Experimental section

In this section we evaluate the performance of the different variants of CALU and PLASMA on a two-socket, sixteen core machine based on Intel Xeon X5660 processors and on a four-socket, twelve-core machine based on AMD Opteron processors running Linux.

Each core of the Intel machine has a frequency of 2.8GHz, a private L1 cache of size 32 Kbytes, a private L2 cache of size 256 Kbytes, and a L3 cache of size 12,288 Kbytes shared with the other cores of the same socket. Each core of the AMD machine has a frequency of 2.1 GHz, a private L1 cache of size 64 Kbytes, a private L2 cache of size 512 Kbytes, and a L3 cache of size 5,118 Kbytes shared with the other cores of the same socket.

We first present measurements of hardware counters such as the L1, L2, and L3 cache misses, TLB misses, and the performance for CALU and PLASMA on the systems described above. Then we discuss the impact of the data layout on these metrics for CALU. Finally, we discuss the impact of the scheduling strategy for a chosen data layout.

For these experiments, we use PLASMA 2.4.5 and all routines are linked with the BLAS version of MKL 11.1.069 [12] vendor library.

CALU static refers to the version of CALU that uses a static scheduling, while CALU dynamic refers to the version that uses a dynamic scheduling. In the static approach, tasks are assigned to threads during the compilation while in the dynamic approach the assignment is during the runtime. The 2l-BL, BL, CM, and Tile notations given in brackets refer correspondingly to the version of CALU or PLASMA that use two-level block data layout, block cyclic data Layout, column major data layout or tile data layout. PLASMA recLU refers to the new parallel recursive panel in PLASMA [8], and PLASMA incpiv refers to the incremental pivoting algorithm implemented in PLASMA. The incremental pivoting algorithm is no longer used as default algorithm for LU factorization in PLASMA because of the stability issue.

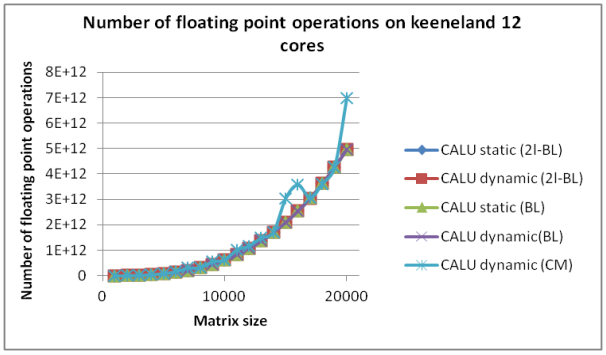
3.1 Hardware counter measurements and performance

3.1.1 Number of flops

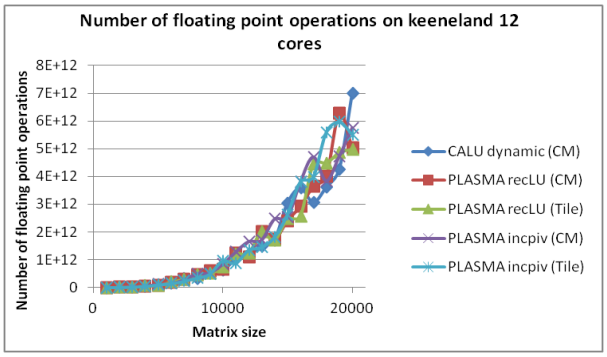
Figure 4 shows the number of floating point operations performed by CALU and PLASMA measured by the PAPI counter FP_OPS. As shown in Figure 4a, the variations in terms of flops between various implementations of CALU is very low. While for PLASMA, as shown in Figure 4b, these variations are more important. We observe a difference of up to 15% when we compare PLASMA incpiv (Tile) and PLASMA incpiv (CM), or when we compare PLASMA recLU (Tile) and PLASMA recLU (CM). On average, all implementations in PLASMA using column major data layout perform slightly more flops than those using tile data layout.

Surprisingly PLASMA performs more floating point operations than CALU in our experiments. In practice, this difference can be explained on the first hand, by what PAPI considers as floating point operation. The measurements of the routine with PAPI include the number of flops

done by the scheduler and by the same way some internal computations not directly related to the factorization; on the other hand it can be explained by the optimizations introduced in PLASMA in order to increase performance. At this point it is difficult to know how these additional operations impact the measurements. However, CALU (2l-BL) and CALU (BL) have the same implementation but only the data layout changes, and no significative additional flops are observed for these two implementations.



a. Versions of CALU

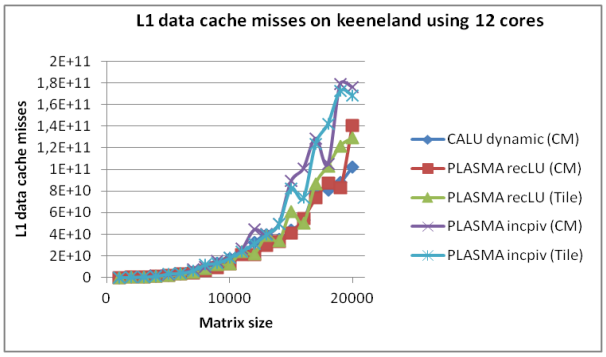


b. Versions of PLASMA and CALU (CM)

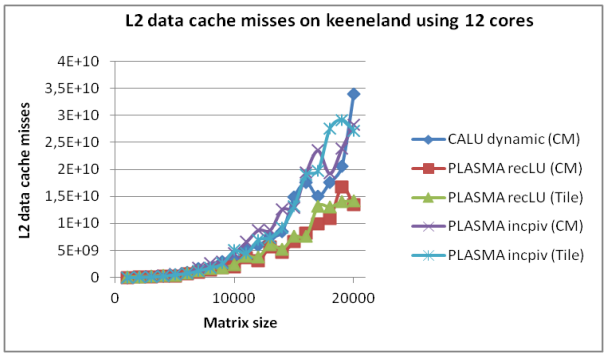
Figure 4: Number of floating point operations of CALU and PLASMA

3.1.2 L1, L2, L3 cache misses

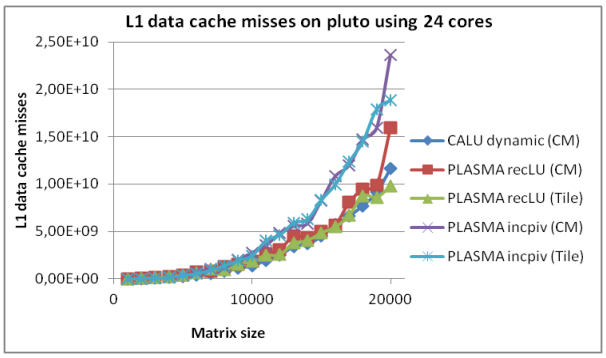
Figure 5 shows L1 and L2 data cache misses for CALU and PLASMA on the Intel and AMD machines. We observed that on these systems, PLASMA incpiv has the highest L1 data caches misses while PLASMA and CALU do slightly the same number of L1 caches misses. As shown on Figure 5, PLASMA recLU has the lowest L2 data cache misses on the Intel machine but the highest one on the AMD one. We recall that CALU aims at minimizing the global number of communications, that is, at reducing the number of data moved from main memory to core cache, while recursive LU implemented in PLASMA aims at reusing as soon as possible the local data in the cache of each core in order to reduce bandwidth usage.



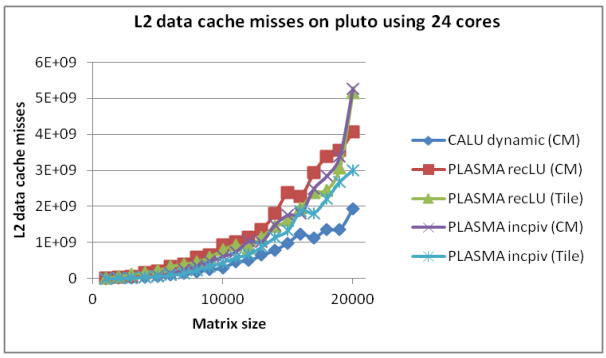
a. L1 data cache misses on keeneland



b. L2 data cache misses on keeneland



c. L1 data caches misses on pluto



d. L2 data cache misses on pluto

Figure 5: L1 and L2 data cache misses of CALU and PLASMA

Figure 6 shows that CALU and PLASMA recLU are competitive in terms of L3 cache misses on the Intel machine. Although there is a significative difference in terms of L2 data cache misses between PLASMA and CALU, the two implementations generate almost the same number of L3 cache misses on that machine. PLASMA incpiv leads to few L3 data cache misses compared to PLASMA and CALU. Due to hardware limitations, it was not possible to measure L3 cache misses on AMD machine.

One notion that may affect the behavior of cache policy is the false sharing. It occurs when two threads access different data that reside on the same cache line and one of them performs an update of that data. In that situation, the cache line is invalided and then the other thread is forced to reload its data from the main memory. Since on both of our systems, each core has a private L1 and L2 cache, and shared L3 cache, false sharing is likely to occur for L3 cache. Detecting false

sharing is difficult in practice, so its impact of L3 measurements is also difficult to predict.

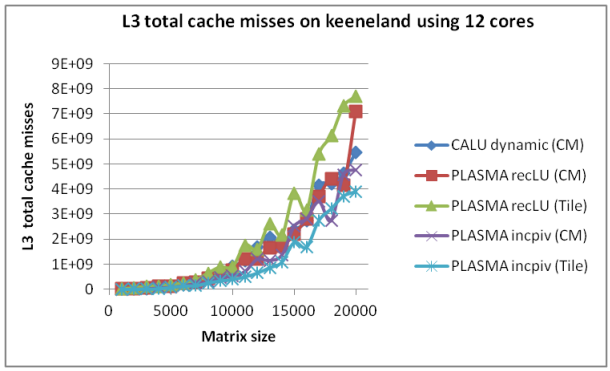
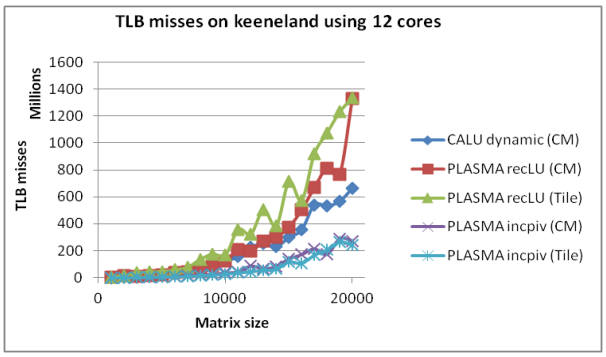


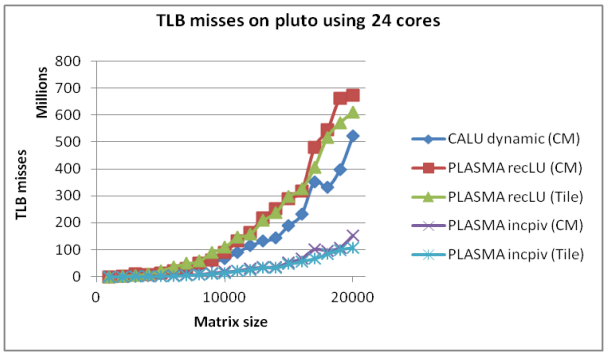
Figure 6: L3 total cache misses on keeneland machine using 12 cores.

3.1.3 TLB counter

As we have presented in the previous section, one advantage of using a tile data layout is to minimize TLB misses. Figure 7 shows that PLASMA incpiv leads to lowest TLB misses on both Intel and AMD machines. We observe that, CALU generates up to 30% less TLB misses than PLASMA. Surprisingly, PLASMA using tile data layout, does not provide less TLB misses as expected and at this point, it is difficult to know exactly why.



a. TLB misses on keeneland



b. TLB misses on pluto

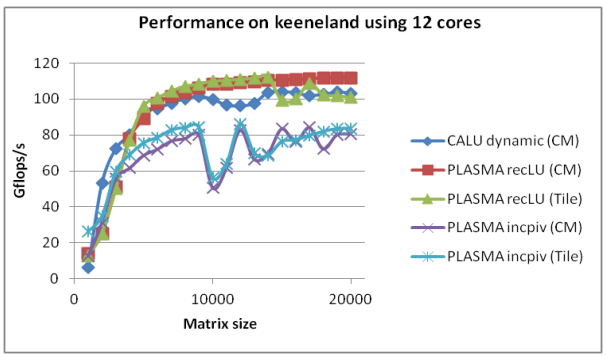
Figure 7: TLB misses

3.1.4 Performance

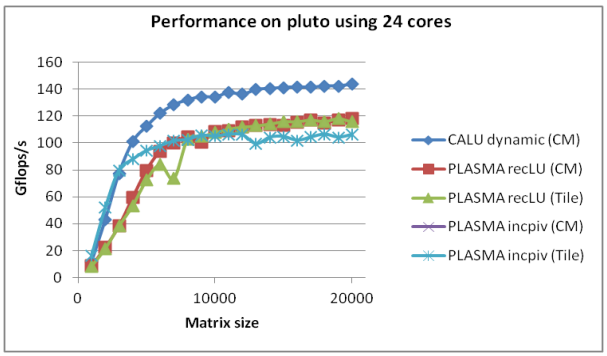
Figure 8 shows the performance of CALU and PLASMA on both systems. On the Intel machine, PLASMA recLU is up to 10% faster than CALU, while on the AMD machine, it is up to 20% slower than CALU.

On the NUMA AMD system, the remote memory access is relatively high compared to the processor speed. Thus, the additional flops that CALU performs help to latency overheads and to achieve better performance. On the Intel machine the additional flops introduced by CALU do

not lead to better performance because of the fast memory access compared to processor speed. In that case, the addition computations (in the reduction of the panel) bring more cache misses. The volume of cache misses can be expressed as $\frac{N}{b}O(b^2)\log P$ where N is the matrix size, b the block size, and P the number of threads working on the panel. In fact, at each step of the reduction operation in the panel factorization using a binary tree, at least a block of size b^2 is moved from one thread to another.



a. Performance on keeneland



b. Performance on pluto

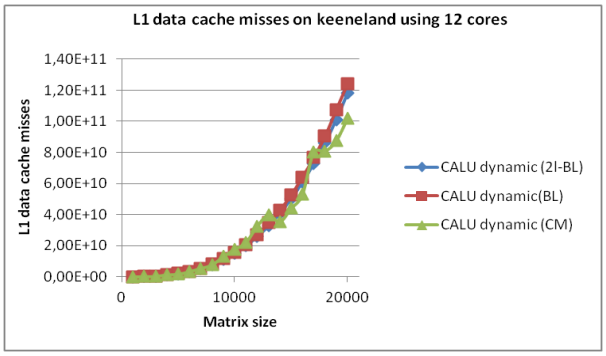
Figure 8: Performance of CALU and PLASMA

3.2 Impact of data layout

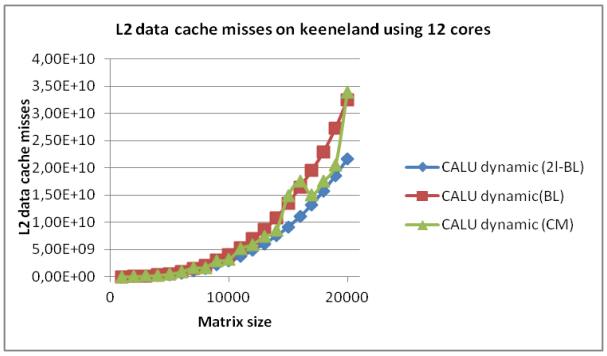
This section studies the impact of the data layout on hardware counters and performance when using dynamic scheduling. We choose CALU here because its implementation has less impact on performance when the data layout changes. In terms of L2 data cache misses, CALU dynamic (2l-BL) is often the best, but sometimes CALU dynamic (CM) is better. This obviously shows that using a dynamic scheduling may annihilate the effort brought by the data layout in order to minimize data movement. The same behavior is observed for L3 cache misses as shown in Figure 10.

3.2.1 L1, L2, and L3 counters

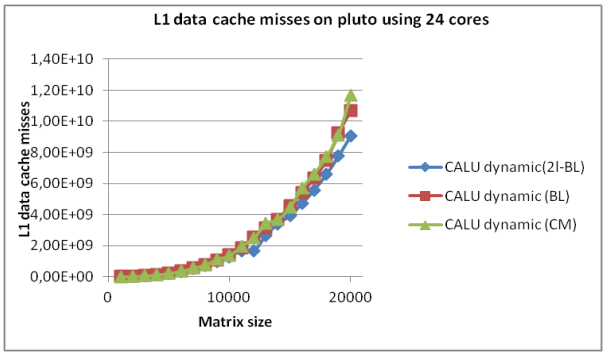
Figure 9 shows that by using a dynamic scheduling, the different implementations of CALU generate almost the same number of L1 cache misses



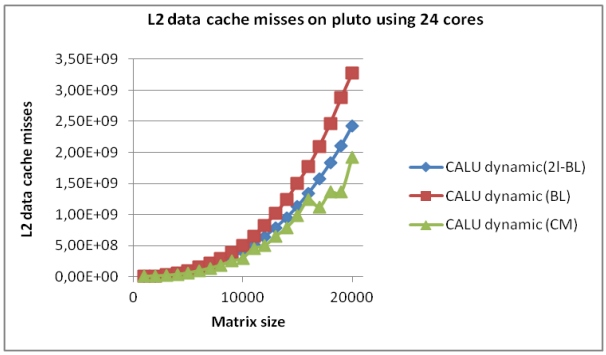
a. L1 data cache misses on keeneland



b. L2 data cache misses on keeneland



c. L1 data caches misses on pluto



d. L2 data cache misses on pluto

Figure 9: L1 and L2 data cache misses of CALU using dynamic scheduling with different data layouts

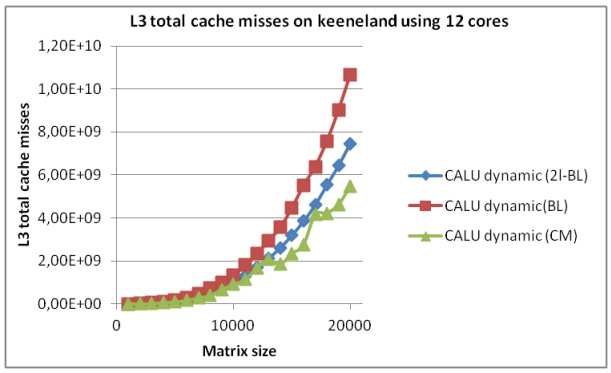
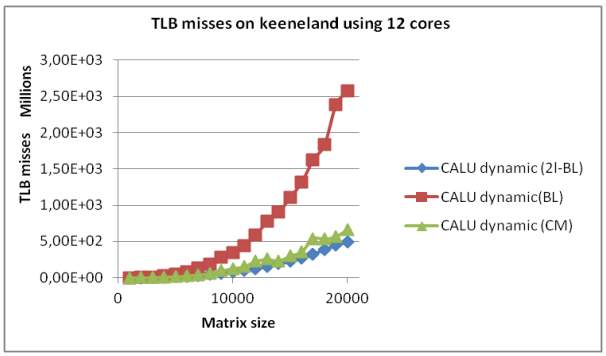


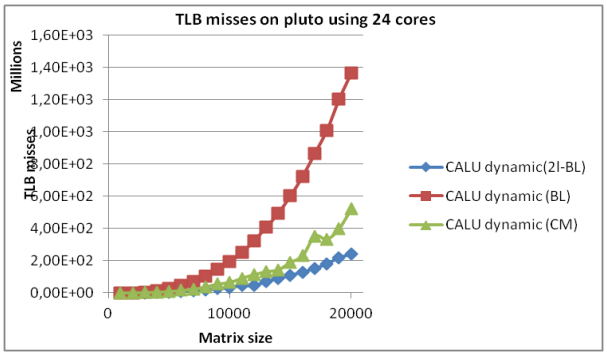
Figure 10: L3 total cache misses of CALU using dynamic scheduling with different data layouts on keeneland using 12 cores.

3.2.2 TLB counter

Figure 11 shows that, on both system, CALU (2l-BL) leads to less TLB misses than using the other data layouts. We observe that using block cyclic data layout (BL) leads to growing TLB misses.



a. TLB misses on keeneland



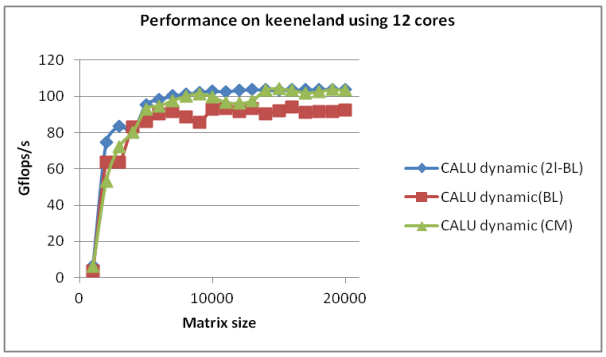
b. TLB misses on pluto

Figure 11: TLB misses of CALU using dynamic scheduling with different data layouts

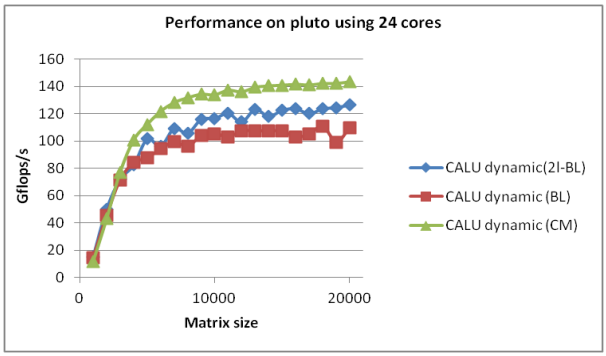
3.2.3 Performance

Figure 12 shows that CALU using block data layout is less efficient than CALU using the other data layouts. On Intel machine, CALU (2l-BL) is faster than CALU(CM) while on AMD machine, CALU (CM) shows an important speedup compared to CALU(2l-BL). Although CALU(2l-BL) may lead to less TLB and cache misses, it can not achieve better performance on all systems. This shows that reducing communication is not enough to guarantee performance.

It is well known that performance of an algorithm depends not only on the memory movement but also on the scheduling time. One advantage of column major layout is that tasks are performed on coarse grain. This implementation takes full advantage of the matrix-matrix product (dgemm) kernel and pays less time on scheduling overhead. On contrary, for CALU(2l-BL), the number of tiles grows faster with the matrix size. Increasing the number of tiles increases the scheduling overhead and leads to a negative impact on performance.



a. Performance on keeneland



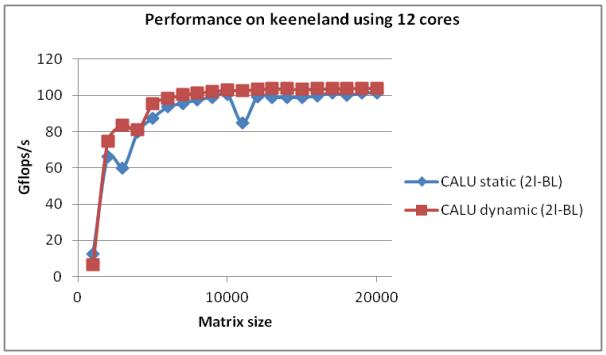
b. Performance on pluto

Figure 12: Performance of CALU using dynamic scheduling with different data layouts

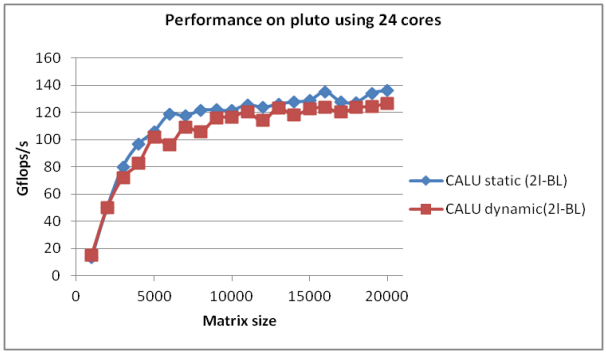
3.3 Impact of scheduling

As we have seen in the previous section, scheduling plays an important role on performance. Using static scheduling is usually recommended to keep the advantage of the data layout. Figure 13 shows that CALU static (2l-BL) is faster than CALU dynamic (2l-BL) on the NUMA AMD machine but slower than CALU dynamic (2l-BL) on Intel machine. This shows that, by using the same data layout, the best scheduling strategy depends on the hardware.

3.3.1 Performance



a. Performance on keeneland



b. Performance on pluto

Figure 13: Performance of CALU static and CALU dynamic using 2l-BL data layout

4 Conclusion

This study evaluates the different metrics of the current implementations of CALU and LU in PLASMA. We target two objectives: first, predict the behaviour of these libraries on more complex architectures, and second, improve and optimize the performance of these implementations. The various measurements show that CALU and PLASMA recursive LU are very competitive and they minimizes sometimes the L2 and L3 caches depending on the system.

CALU has shown to reduce global communication while recursive LU has shown to be effective on reducing local communication or bandwidth. In perspective, it could be interesting to design an algorithm that reduces communication at the global point of view by using communication avoiding technique, and then reducing bandwidth at the local level by using parallel recursive approaches.

Acknowledgments

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

References

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

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

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

[4] Plasma. http://icl.cs.utk.edu/plasma/.

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

[6] S. Donfack, L. Grigori, W.D. Gropp, and V. Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. In *Parallel & Distributed Processing (IPDPS), 2012 IEEE International Symposium on*, to appear.

[7] 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.

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

[9] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[10] 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.

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

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

[13] F. Song, H. Ltaief, B. Hadri, and J. Dongarra. Scalable tile communication-avoiding qr factorization on multicore cluster systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

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