

A wide comparison between different linear algebra libraries using the well-known LU factorization problem

Sergio Rivas-Gómez

sergio.rivas@um.es

DIS - University of Murcia
Spain

Domingo Giménez

domingo@um.es

DIS - University of Murcia
Spain

Javier Cuenca

javiercm@dittec.um.es

DITEC - University of Murcia
Spain

Abstract

This work focuses in the comparison between different linear algebra libraries solving the LU decomposition problem. This is done by developing a sequential version of the outer Gaussian decomposition, which will be optimized following a block-division approach and evaluated by including different vector-vector / matrix-vector / matrix-matrix operations contained within several linear algebra libraries based on the BLAS specification. The work also compares the benefits of using an already implemented LU factorization (e.g., calling directly to LAPACK's method), combine different levels of parallelism, explore out-of-core implementations and will finish analyzing the benefits of using the GPU as an alternative for general linear algebra computation.

1 INTRODUCTION

During these years, linear algebra libraries have become critical mainly due the increasing requirements in the field of computer vision and computer graphics (e.g., CGI rendering), among others. But many other fields like chemistry or medicine, are interested in the possibilities of parallelism to solve some of the most important research problems.

In this way and especially during the last decade, many computer libraries and techniques have emerged to take advantage of the resources of the new digital era that has been established this century. The processing power of personal computers and supercomputers are, today, an opportunity for many computer science researchers.

This paper aims to establish a basis for future research by evaluating some of the most important linear algebra libraries available in the market, which include the popular Math Kernel Library [1] from Intel. All of the libraries are based on Basic Linear Algebra Subprograms (BLAS) [2] specification.

The next subsection is defined to help the reader understand some of the key concepts that will be used within this document.

1.1 BACKGROUND

Although the first specification and implementation came in 1979 [3], BLAS is nowadays the “de facto” standard in linear algebra subroutines. It is a set of low-level kernel functions that perform common linear algebra operations such as copying, vector inner product and matrix multiplication.

The reference implementation, which was originally coded in FORTRAN, has been improved these years by more hardware depending optimized versions. Manufacturers like AMD are continuously releasing new improved releases of their BLAS implementation (i.e., APPML [4]) to take advantage of most of the features included in their CPU's and now newest APU's.

But unfortunately for AMD, historically Intel has taken advantage in the professional sector and many of the best supercomputers

worldwide use their processors. What is more and as well as we introduced earlier, Intel's MKL is one of the best BLAS implementations available. Its performance, as we will discover in the coming sections, is very good.

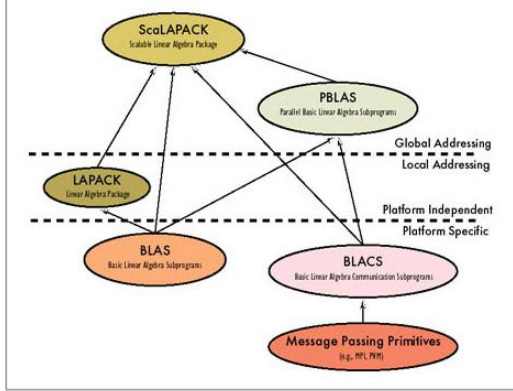


Figure 1 - Example diagram of linear algebra solutions available.

In spite of the fact that BLAS offers low-level linear algebra functions, there are other API specifications that has grown from this one with the purpose of providing a one-call optimized function to solve many well-known problems, as for example the LU factorization. This means a huge step in usability, as non-mathematics users can easily solve complex problems in an efficient and simple way.

As the reader may appreciate in Figure 1, LAPACK [5] is one of these BLAS-based package and, in fact, is one of the most popular. Furthermore, modified versions which integrate MPI have emerged from it and enables communication between different nodes solving a common problem.

This and other concepts will be addressed later in the proper section.

1.2 STATE-OF-THE-ART

The meaning of this section is to introduce the system where all the test will be executed, as well as to describe briefly many of the test that will be launched to evaluate the performance of each case we are interested.

All the test will be executed in a NUMA workstation located at the Scientific Computing and Parallel Programming Group laboratory, located at the University of Murcia, Spain. This system offers the following specification:

Component	Description
CPU mfr.	Intel Corporation
# CPU's	4 x Xeon E7530 @ 1.8GHz
# cores	4 x 6 cores => 24 cores
Cache / core	L1=32KB / L2=256KB
RAM / node	8GB (32GB in total)
HT enabled?	No

Table 1 - Specification of the workstation used

Although the system uses an old Intel CPU and one could think that the results will seem useless, it is more likely for most researchers to use similar workstations than newer ones to launch their parallel algorithms. In fact, today's CPUs have a reduced power consumption, but not a huge performance improvement in comparison.

The general idea is to launch multiple test-cases that could use all the resources available and that, at the same time, could give us an idea of how well is the performance of the implementation used. Some of these tests, which will always be based on the LU decomposition problem, include:

- **Matrix size evaluation:** All the tests will be evaluated modifying the size of the matrix itself. As the objective is to evaluate real-life like problems, sizes will vary from 1000 by 1000 matrices to 10000 by 10000.
- **Block size evaluation:** As the reader may have appreciated in the abstract of this document, there will be some implementations based on block optimization. In this way, this kind of tests will include a block variation with values power of 2 (i.e., 2, 4, 8, 16 and so on).
- **Number of threads evaluation:** Some of the implementations will make use of parallelism with different number of threads in different levels. The idea is to modify the number of concurrent threads solving the problem, which will obviously vary from 1 to 24 (i.e., maximum number of cores available).

For simplicity, we will only test square matrices and not rectangular ones (which could affect dramatically to the final results).

1.3 LU FACTORIZATION

DEFINITION

Prior to the performance analysis, the reader might be interested in understanding the meaning of the LU decomposition.

Basically, given a square matrix A whose terms are the values of a system of linear equations, the LU factorization refers to the process where A is split into a lower triangular matrix L and an upper triangular matrix U , so that this is true if there is a solution:

$$A = LU = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

This factorization is made by applying Gaussian elimination to the original matrix. The idea is that U contains the final result of the elimination process, and L contains the changes made to the original matrix with sign changed (i.e., we could convert U back to A , which is the original matrix).

The main application of this LU decomposition is to easily solve several linear equations given different b values:

$$Ax = b$$

If we already have the LU factorization of A , solving the linear equation is easy. First by applying forward substitution and then by doing back substitution:

First step: $Lc = b$

Second step: $Ux = c$

The computational cost is way lower, because the number of operations to solve a linear equation given LU will be n^2 for the first step and also n^2 for the second step. So this means an $O(2n^2)$ in cost.

The key factor here is that the elimination process is expensive, as it requires $n^3/3$ multiplications and $n^3/3$ subtractions. But this is only made once, and further right-hand side b 's can be directly solved by the process explained earlier.

The reader can obtain more information about how the complete process works by reading [6], including other concepts which involve row permutations (i.e., pivoting). There

is also an explanation of the computational cost in the cited reference.

2 SINGLE AND MULTI-CORE EVALUATION

This section's objective is to provide an analysis of the performance of several implementations solving the LU decomposition problem using a single core of the NUMA workstation and, as well, many of the 24 cores available.

We have already detailed how the test-cases will be designed, so let's begin directly with the following subsections.

2.1 CLASSIC IMPLEMENTATION

The first implementation evaluated will act as the reference one in terms of time comparison. It is basically the algorithmic version of the Gaussian elimination process, which will transform a matrix A into an LU factorization matrix.

However, Gauss elimination is row-guided and we know this could lead into a huge performance decrement due to the cache's temporal and spatial locality feature. So the idea is to change the process to a different column-based one called outer product Gaussian elimination. The result will be equivalent, but with the main difference that L will no longer contain 1's in its diagonal and will be U the one who receive these values instead.

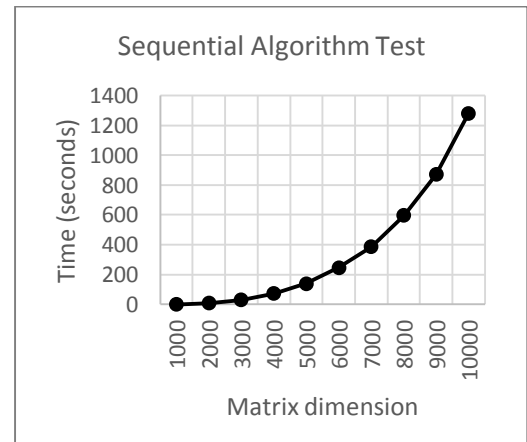


Figure 2 - Sequential algorithm results of the LU decomposition modifying matrix size with a time limit of 20 minutes.

Figure 2 shows the performance of this algorithm by varying the matrix size from 1000 by 1000 up to 10000 by 10000. As we expected, for smaller matrices the results are good and even less than 10 seconds are needed to factorize a 2000 by 2000 matrix.

But as we increment the size, it is clearly visible the $O(n^2)$ estimated cost of the operations involved in the process. What is more, the time limit defined of 20 minutes is reached with the last test-case, and this lead us to the next subsection as we try to provide a better solution.

2.1.1 Block optimization

Almost any algorithm involving matrix operations can benefit from block optimization. In the past, many researches like [7] has proved the benefits of this technique, as it takes advantage of cache's locality feature.

In this way, the LU decomposition problem can be split into several subproblems that can make a good performance improvement. The key idea is to define a block size, which will be a square matrix whose values will be extracted from the diagonal. The algorithm will calculate the LU factorization of this smaller matrix, and then "extend" the results to the upper-right rectangular matrix, the lower-left one and finally the rest of the matrix as a combination of the latest two.

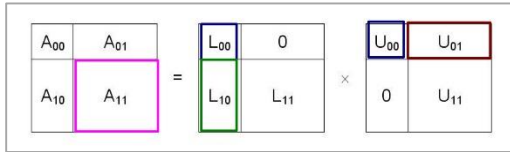


Figure 3 - Example of how the block optimization works.

In other words, following Figure 3, the steps of the process are defined as:

$$\text{Step 1: } A_{00} = L_{00}U_{00}$$

$$\text{Step 2: } A_{01} = L_{00}U_{01}$$

$$\text{Step 3: } A_{10} = L_{10}U_{00}$$

$$\text{Step 4: } A'_{11} = A_{11} - L_{10}U_{01}$$

The process continues, taking now the block A₀₀ from the top-left corner of A'₁₁. This will be repeated until the lower-right corner is reached. At this moment, the algorithm

has finished and the LU decomposition of A is made.

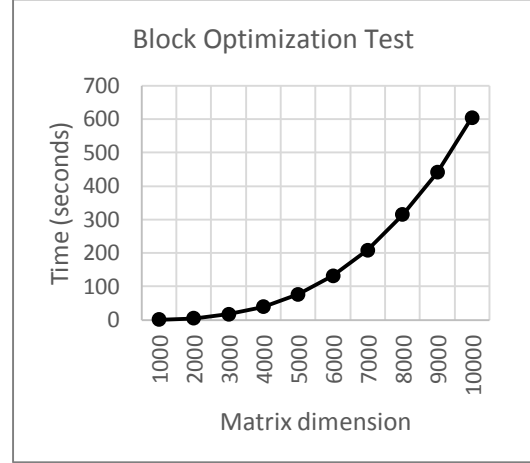


Figure 4 - Block optimization results of the LU decomposition modifying matrix size with a time limit of 20 minutes.

The idea sounds apparently fine, but does this change really improve the performance of the original algorithm? Well, according to Figure 4 it does. This graph reflects the best times obtained with a specific block size given a matrix dimension of 1000 by 1000 up to 10000 by 10000. Now we appreciate that even with the maximum size, the 20 minutes time limit is not exceeded and about 600 seconds are needed (i.e., 10 minutes).

In terms of SpeedUp, which reflects the performance improvement comparing to a reference test (i.e., the sequential), the algorithm seems to be nearly as twice as faster compared to the same 5000 by 5000 result:

$$SpeedUp = \frac{Result_{seq}}{Result_{block}} = \frac{140.11}{76.49} = 1.83$$

But these results have not come randomly. The truth is that the test-case launched here was designed to evaluate, for each matrix size, each block size power of 2 smaller than the size itself. So for a 9000 dimension matrix, the block sizes tested were 2, 4, 8, ..., 4096 and 8192.

What we tried to evaluate is how the performance could vary modifying the block size and according to the resources available (i.e., each core has an L1 of 32KB and an L2 of 256KB). Unfortunately and as expected, the results are problem-dependent and also are related to many other external aspects like how

the CPU brings data from the main memory to the cache. This means that defining a 32KB block isn't necessarily the best decision [7].

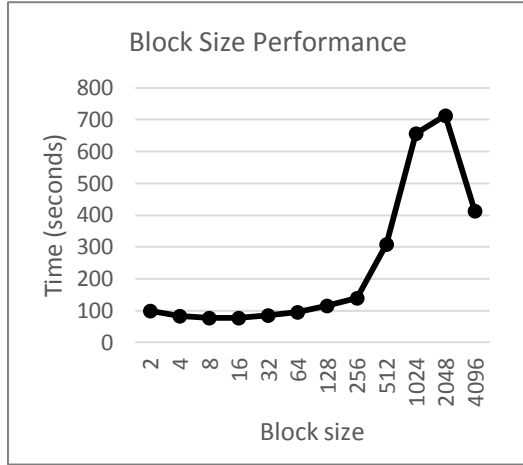


Figure 5 - Block size performance comparison of the LU decomposition with a square 5000×5000 matrix.

Anyway, if we analyze the results, sizes of 8 and 16 give always the best performance for any size (as seen in Figure 5 for a dimension of 5000). Curiously, a block size of 4096 offers better performance compared to a size of 1024 or 2048. The reason might be related to the fact that the page size is established to 4KB or because the L1 size is 32KB, which is exactly the size of 4096 **double** values.

We may check later if this behaviour is repeated when we integrate the linear algebra libraries within the implementation.

2.2 USING BLAS FUNCTIONS

The next step defined is to modify the prior code to include calls to many of the BLAS subroutines available. As these methods are optimized (and implemented) in low-level, it is supposed to offer a good performance improvement over the previous results.

We will start first using the Math Kernel Library implementation of the BLAS API, then we will make use of the ATLAS version and finally we will test the performance of the popular OpenBLAS (i.e., GotoBLAS).

2.2.0 Algorithm changes and calling if.

Although the implemented code will be exactly the same as the block optimized version, we should adapt it to make use of the BLAS subroutines wherever possible.

In this way, there are three different locations which could lead into a huge performance improvement:

1. **LU auxiliary method:** The complete algorithm computes the LU decomposition, but we must also compute the LU for the current block. In this way, the code is using the original sequential version of the whole algorithm and we can easily change two different loops within this method.

The first loop divides the row elements by the pivot position. As there's a vector operation called **dscal()** [8] which scales a double precision vector (i.e., our row) by a factor alpha (i.e., our pivot inversed), we can remove this loop by calling it.

And the second loop is the result of a row-column multiplication whose values will be subtracted to the lower-right matrix available. So, it can be translated as a matrix-matrix multiplication whose sizes are $N \times 1$ and $1 \times N$ respectively. The method **dgemm()** [9] can be easily used here.

2. **Upper/Lower-triangular:** The second and third step of the algorithm defined in 2.1.1 involves the use of the already calculated LU within the current block. These values are used to extend the result to the upper-right / lower-left corner of the matrix.

The process can be defined as an equation solver, and in this way the method **dtrsm()** [10] is very useful.

3. **Matrix multiplication:** The latest step is to multiply the upper-right matrix with the lower-left corner matrix. It is basically a matrix multiplication, and although the result will be subtracted to the original values, the method **dgemm()** [9] can be used again here.

That said, we should tackle another last point. There are two different ways to call the

mentioned functions: using the FORTRAN interface or using the C interface. The first one is, in our opinion, the best compatible way to call BLAS subroutines, as almost all the libraries implement this interface. The problem is that FORTRAN stores matrices in a column-major order and this could lead into many unexpected results if we are using other languages like C, whose matrices are stored in a row-major order.

For simplicity and as all the libraries that will be evaluated implement both interfaces, after some tests we decided to use CBLAS instead. The main reason was that although **dgemm()** lets you specify that the input matrices are in row-major order, the result will always be produced in column-major order, which is equivalent to the transpose of the resultant matrix (i.e., it is not what we are really expecting).

Let's now begin with the analysis of the different alternatives.

2.2.1 BLAS / MKL implementation

Intel's Math Kernel Library is one of the BLAS reference implementations in terms of good performance (at least, of course, using Intel CPUs). The manufacturer ensures that each of their CPU models are compatible with this implementation and that each subroutine has been manually optimized to offer the best performance possible [11].

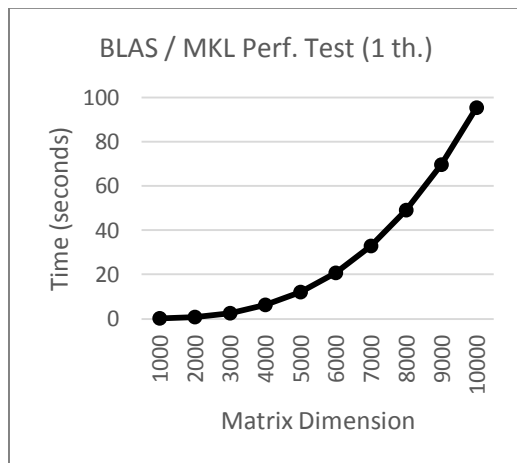


Figure 6 - BLAS / MKL best results (using 1 thread) of the LU decomposition modifying matrix size.

The library also offer multi-threading capabilities in almost all of the BLAS subrou-

tines [12]. What is more, one could easily modify the number of concurrent threads [13] to improve the performance of the execution by setting several environment variables like **OMP_NUM_THREADS** or **MKL_NUM_THREADS**.

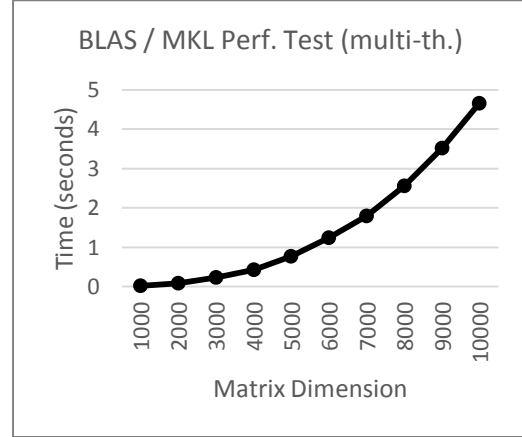


Figure 7 - BLAS / MKL best results (using multiple threads) of the LU decomposition modifying matrix size.

This seems to be the key evaluation for this implementation. We cannot only test the behaviour of the algorithm by modifying the block size and substituting part of the code with BLAS' methods, but also we can analyze the impact in performance as we increment the number of concurrent threads in each test case. So, the test-case was organized to launch different matrix sizes, different block sizes for each matrix size and finally different number of threads for each matrix and block size.

Figure 7 shows the best result obtained for each matrix size (e.g., MatrixDim=5000, BlockSize=128 and NumberOfThreads=20). The performance is, by far, better than we initially expected. For a 10000 by 10000 matrix, the LU decomposition was obtained in less than 5 seconds. If we compare this result with the one obtained in the block optimized code, we can clearly see the difference:

$$SpeedUp = \frac{Result_{block}}{Result_{block_{BLAS}}} = \frac{604.20}{4.64} \approx 130$$

But we might be very careful comparing these results, because that 4.64 seconds time was obtained by using multi-threading (24 threads exactly, which corresponds to the maximum number of cores in the workstation).

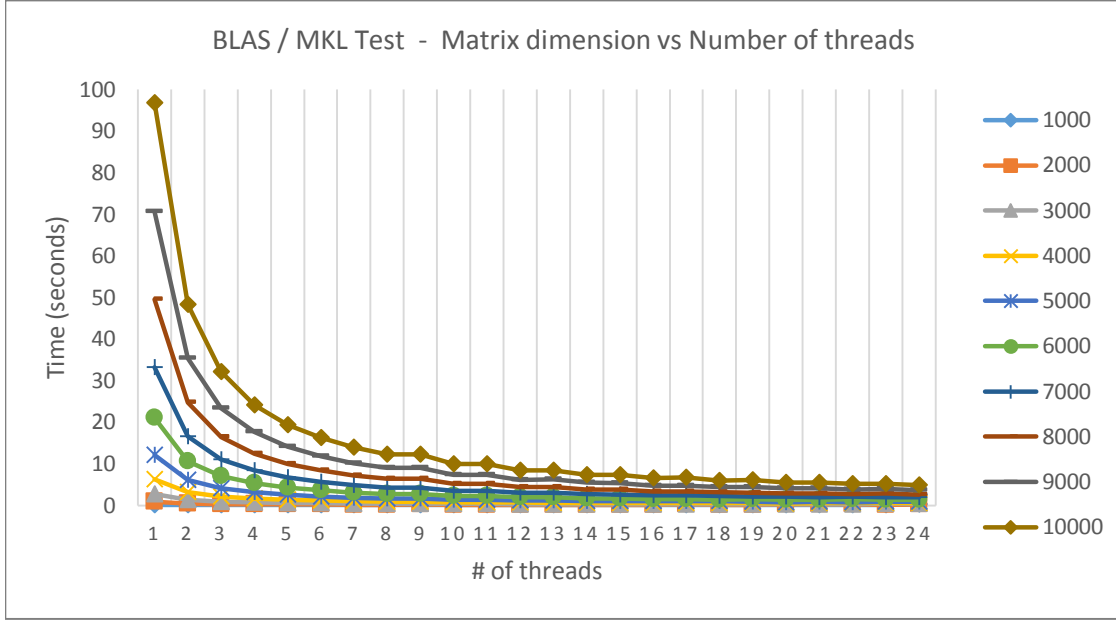


Figure 8 - Results comparing how the number of concurrent threads improves the performance given a matrix size (right).

Figure 6 shows the best results available for one single thread. The performance is still impressive, and the SpeedUp obtained is also very good but not equivalent:

$$SpeedUp = \frac{Result_{block}}{Result_{blockBLAS}} = \frac{604.20}{95.20} = 6.33$$

One interesting thing about the new designed algorithm is to check its scalability. It is known that if we are introducing parallelism with 24 concurrent threads, the result in theory will be 24 times faster. Well, this is not always true as many factors could affect the final result, but we can have an idea of how it responds by evaluating Figure 8. This graph shows a comparison, based in time, between the matrix size and the number of simultaneous threads. And, as expected, there are two clear points involved:

- The first one is that there is no huge improvement after 9 threads. This means that we are not going to have an LU decomposition 24 faster, but slightly 10-20 times faster¹.
- And the second one, related to the first one, is that the performance with higher number of threads depends on the size of the problem.

One last note prior to continuing with the next library, is that the best block size right now is bigger than 16. Specifically, 128 and 256 are the sizes where the performance is clearly better if the number of threads is high.

On the one hand, dividing 128 or 256 by 24 gives a value near 8, which we know offers a good performance for one single thread. But although this could be an explanation, on the other hand the performance is also good when we use only one thread. So, we may think it is all about how Intel implemented BLAS sub-routines or about a possible calling overload.

2.2.2 BLAS / ATLAS implementation

Automatically Tuned Linear Algebra Software (ATLAS) is a generic implementation of the BLAS API subroutines. It is not hardware dependent, which means that it is not optimized for a specific CPU manufacturer as MKL was.

As its name, ATLAS' purpose is to provide an optimal linear algebra software which automatically tune its parameters according to the hardware available [14]. This implies that after installation, the library has a fixed configuration which is supposed to be nearly as optimal as if a manual optimization were made.

¹ We will try to solve this later with different levels of parallelism.

This have some kind of disadvantage, as we cannot modify this auto-configuration as we would like to. For example, you cannot modify the number of threads per execution, as ATLAS decide in compilation time how many threads could potentially use in the given functions [15] (it could use less threads than expected if the problem size is small).

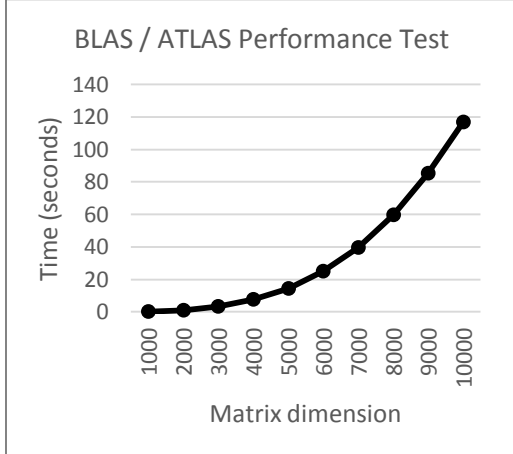


Figure 9 - BLAS / ATLAS best results of the LU decomposition modifying matrix size.

As it implements the FORTRAN and CBLAS interface, no specific changes must be done to the already implemented code and it is all about how the code is compiled.

Graph 7 shows how well this implementation performed for each matrix size tested. For smaller sizes, ATLAS times are impressive and pretty similar to the MKL results. But the more the size of the matrices is increased, the more the time spent to solve the LU decomposition problem.

One thing is clear: the results are better than the original block-optimized version. If we look at the 5000 square dimension test, the time spent was 14.5 seconds, which is a complete minute faster than the result obtained in the original optimized code.

One strange behaviour was, again, how block size has affected the final results. For some reason, 256 is a good block size while using ATLAS to solve the LU factorization, which could mean that maybe both Intel and ATLAS' team used similar optimizations (the reader can appreciate this behaviour by looking at Figure 10).

But this coincidence has open up the door of the overload justification, as we introduced in the last subsection. For a small block size, could the overload to call an ATLAS or MKL subroutine be so high than the performance is decreased? The answer is no. We tested again the 5000 dimension matrix case restoring the original LU sequential algorithm (i.e., without using BLAS within the first step of the block-optimized algorithm) and there is only an improvement of less than 2 seconds in comparison.

2.2.3 OpenBLAS implementation

GotoBLAS is another high-performance implementation of the BLAS specification [16]. The library uses improved algorithms and memory techniques for optimal performance of the BLAS methods and, unlike ATLAS, it is optimized for different Intel and AMD processors architectures. Right now is maintained

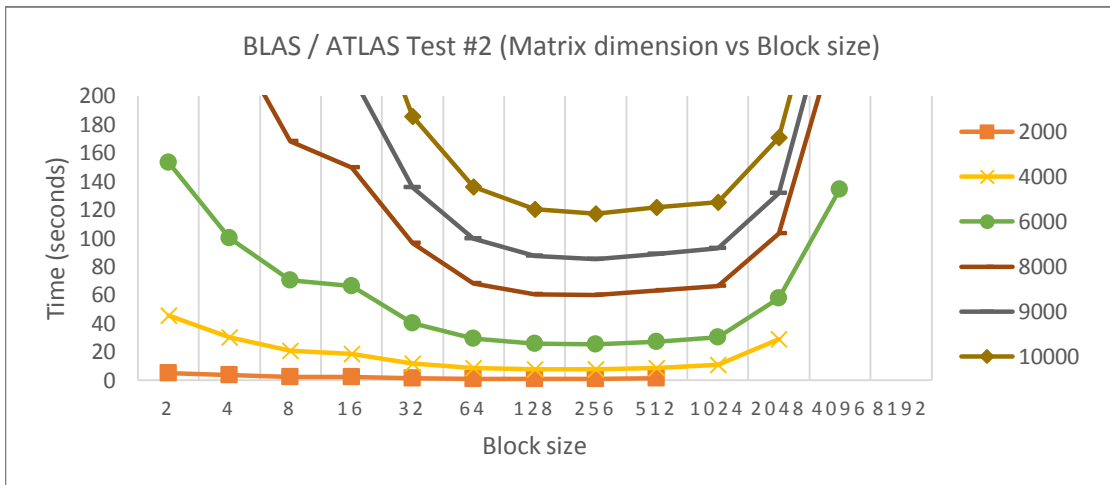


Figure 10 - BLAS / ATLAS results comparing how the block size improves / decreases the performance given a matrix dimension (right).

by the open-source community with the name of OpenBLAS [17].

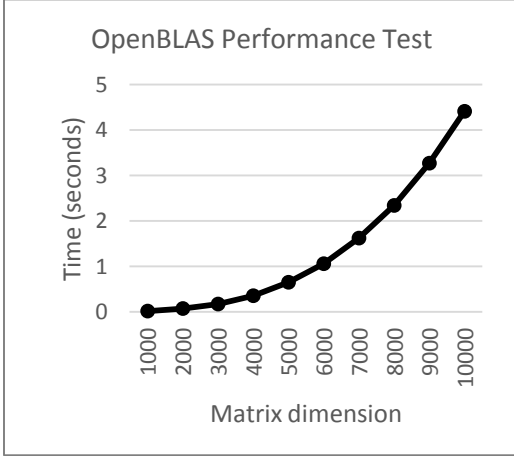


Figure 11 - OpenBLAS best results of the LU decomposition modifying matrix size.

The library is similar to ATLAS in the way it decides how many concurrent threads will launch for a given problem size. At least and according to the official documentation, it is possible to modify this parameter if we like to. We finally decided not to change this setting, as we would like to evaluate this library in the same way we did with ATLAS.

One of the clear advantage of this library is its popular matrix multiplication method. Many papers have talked about how well this library handles this operation [18] [19], as it offers a very good performance.

Our results are not different from what we anticipated, and for all sizes the algorithm finishes in less than 5 seconds. Figure 11 reflects the performance of the LU decomposition using OpenBLAS, which is slightly better than the optimized MKL and way faster than ATLAS for a 10000 square matrix:

$$SpeedUp_{MKL} = \frac{Result_{block_{MKL}}}{Result_{block_{GOTO}}} = \frac{4.64}{4.40} = 1.05$$

$$SpeedUp_{ATLAS} = \frac{Result_{block_{ATLAS}}}{Result_{block_{GOTO}}} = \frac{117.02}{4.40} \approx 26$$

There is no doubt this is a very good library in terms of good performance and, we admit, even more impressive than MKL. It also offers a main advantage: it is not strictly hardware or manufacturer dependent and right now is open source, which implies no license costs.

In other terms, Graph 10 shows how the block size improves or decreases the performance of the algorithm. It is clear that something has changed compared to the last two libraries, as we can get nearly the same results using 8 and 16 as size. But comparing the different results, clearly 64 and 128 are always slightly better.

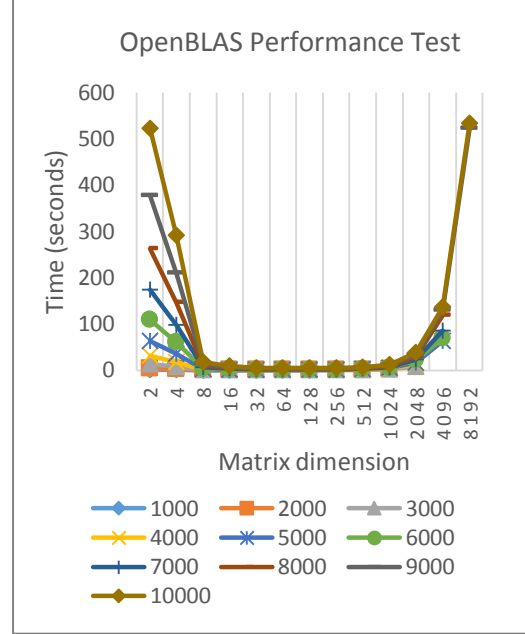


Figure 12 - OpenBLAS results comparing how the block size improves / decreases the performance given a matrix size (bottom).

2.3 USING AN IMPLEMENTED LU METHOD

We have recently analyzed how the LU factorization of an A matrix can be implemented and optimized comparing different linear algebra solutions based on BLAS specification.

In this section, our objective is to move to a higher level inside the hierarchy shown in Figure 1 and make use of an already implemented LU factorization method. We will mainly focus on MKL's LAPACK implementation and a new library called PLASMA [20].

2.3.1 LAPACK / MKL implementation

MKL not only offers an optimized implementation of the BLAS subroutines, but also includes an implementation of LAPACK specification. This implies that we have access to many other functions that can solve systems of linear equations, eigenvalue problems

and matrix factorizations such as QR or the one we are interested in: the LU factorization. What is more, Intel included multi-threading capabilities in these function too.

So the idea now is to forget all the code we have already implemented and make a single call to LAPACK's LU factorization method, which is called **dgetrf()** [21]. Our plan is to evaluate the performance differences between a hand-coded method and an already implemented one.

Two related tests will then be made: matrix dimension variation and different number of threads per matrix size. Figure 14 shows the best results obtained using MKL's LAPACK implementation. There is a clear performance decrement compared to the BLAS / MKL version or even the OpenBLAS one, but we are talking in terms of seconds of difference.

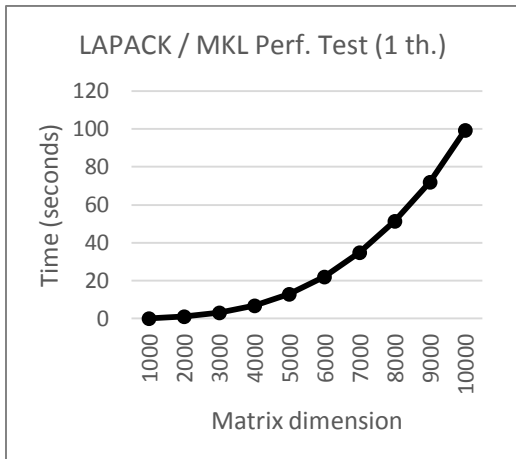


Figure 13 - LAPACK / MKL best results (using 1 thread) of the LU decomposition modifying matrix dimension.

The last case, the 10000 square dimension matrix, took 8.18 seconds to complete, which is 7000% faster than the original block-optimized code:

$$SpeedUp = \frac{Result_{block}}{Result_{block_{BLAS}}} = \frac{604.20}{8.18} \approx 74$$

This means that it is even better than coding a complete version of the algorithm using ATLAS, as we tested earlier. But keep in mind that these results have made use of multiple threads to solve the factorization.

In this way, Figure 13 shows how well the algorithm performs by using one single thread. It is slightly worse, but still better than the

original block-optimized algorithm and also better than the one which made use of ATLAS. For example, for a 10000 square dimension matrix, ATLAS spent 117.01 seconds versus LAPACK / MKL's 99.42 seconds (17 seconds faster).

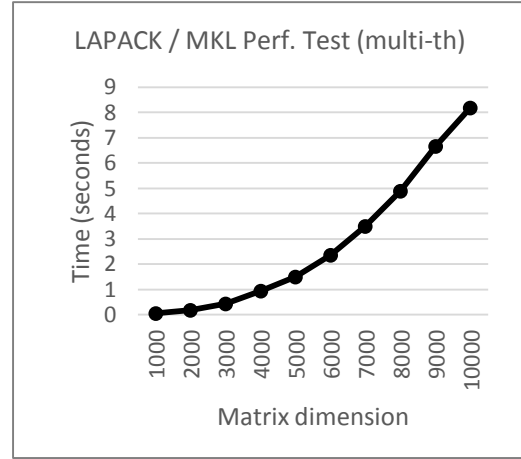


Figure 14 - LAPACK / MKL best results (using multi-threading) of the LU decomposition modifying matrix dimension.

The performance is not always equivalent, as Figure 15 reflects, and later we will try to solve this issue with different levels of parallelism. As you may appreciate, the behaviour is similar to the one found in **Error! No se encuentra el origen de la referencia.** 8 using BLAS/MKL and different number of threads.

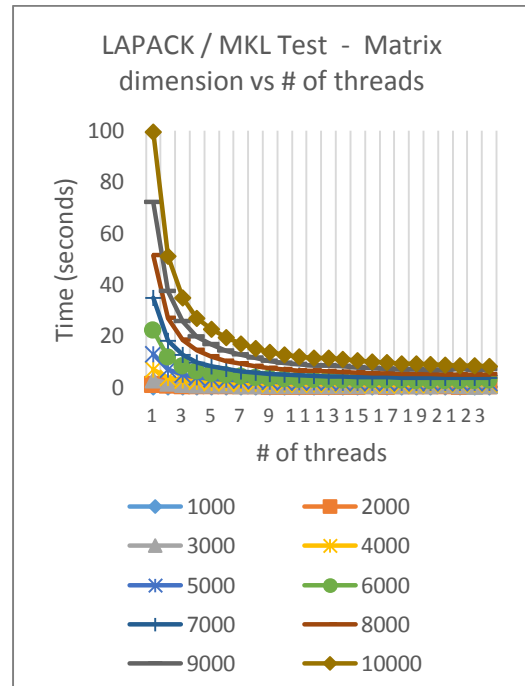


Figure 15 - LAPACK / MKL results comparing how the number of simultaneous threads improves the performance given different matrix dimensions (bottom).

2.3.2 PLASMA implementation

PLASMA, whose name comes from “Parallel Linear Algebra for Scalable Multi-core Architectures”, aims to address the disruptive situation that is facing linear algebra and high performance computing libraries since the introduction of the multi-core architectures [20]. The goal of this implementation is to offer portability by the development of programming models that enforce out of order asynchronous operations.

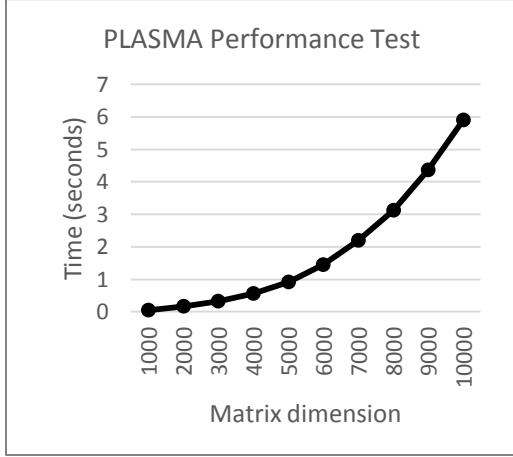


Figure 16 - PLASMA best results (using MKL) of the LU decomposition modifying matrix dimension.

Inside, it is using any BLAS implementation introduced while installation (e.g., MKL), so it is all about how this team designed their algorithms. In some way, these algorithm decides how many threads will launch given the problem size and the resources available. It is equivalent to ATLAS or GotoBLAS, but focused on algorithm scalability.

So the only test we can actually make here is to evaluate the performance of the library

when we modify the dimension of the A matrix and maybe the maximum number of PLASMA threads at launch. No other results can be tested, as we expected to make use of MKL’s multi-threading capabilities and we were finally unable (although the documentation says it is possible since version 2.5.1 [22]).

Figure 16 shows that PLASMA offers a really good performance without any special configuration (the last case was completed in 5.90 seconds). These results were achieved selecting 1 launch thread and letting PLASMA’s algorithms decide where to use parallelism.

2.4 DIFFERENT LEVELS OF PARALLELISM

We have discovered some issues related with the scalability of the algorithm when incrementing the number of simultaneous threads. In theory, if we have 24 cores, launching 24 threads to solve a problem could make the algorithm run 24 times faster. But this behaviour may not always be true, because many factors can potentially affect the final results obtained.

Our objective now is to add another level of parallelism using the well-known OpenMP library to exploit the resources available. Please note that, although we increased the level of parallelism within all the regions in the source code, initial tests demonstrated the existence of an overload cost within some lightweight methods. We finally decided to focus our attention in the Step 3 of the algorithm (see 2.1.1 for further information), which is the most computational cost expensive.

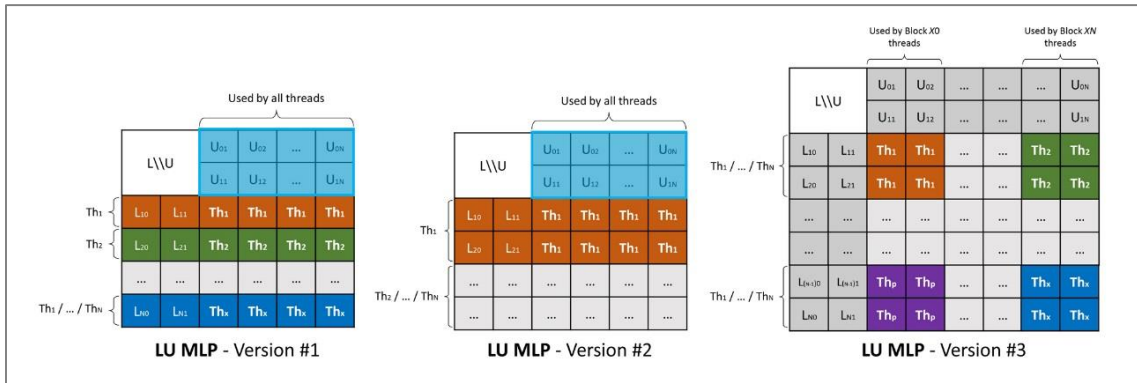


Figure 17 - Example of how the different versions with multiple levels of parallelism have been implemented in our tests.

Our new tests have used as a basis the BLAS / MKL implementation, which gave very good results. We implemented three different versions, each one with a specific purpose (as seen in Figure 17). The parallelism will be introduced in the multiplication itself:

1. **Version #1:** The initial version splits the L_{10} matrix into different rows times the full matrix U_{01} , which will produce as an output each row of the A'_{11} submatrix. In this way, we can easily assign each thread a different row and let it solve the multiplication.
2. **Version #2:** This version is more complex but designed in the same way as the prior one. The idea here is to equally assign the same number of simultaneous rows to each thread, so that they solve a full matrix multiplication and produce multiple rows of the A'_{11} submatrix.
3. **Version #3:** The latest one divides the submatrix A_{11} in different blocks. Each thread has assigned multiple blocks, based on its unique identifier. This implies that the first block will be assigned to the first thread (i.e., $Th_{id}=0$), the second to the second thread (i.e., $Th_{id}=1$) and so on.

At the same time, each thread will use BLAS to solve the assigned product, which introduces the second level of parallelism by using Intel's MKL. So, in our test, we have checked how the performance was by setting 1 thread within MKL and 24 threads in OMP (i.e., the maximum number of cores available), 2 threads within MKL and 12 threads in OMP, and so on until 24 threads in MKL and 1 in OMP was assigned¹.

We have also tested different block sizes, as we are still using the block algorithm implemented originally with BLAS. Please note that only block sizes powers of 2 between 16 and 256 dimension were tested, which are actually the ones that offered better results in our prior tests cases.

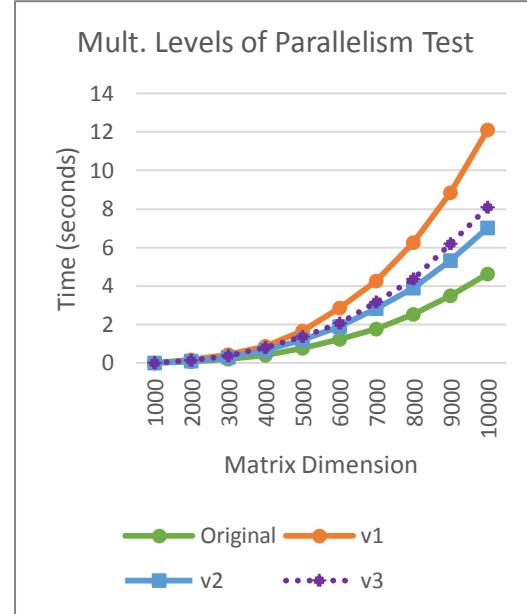


Figure 18 - Best results of the LU decomposition using multiple levels of parallelism. The graph shows the performance of the different implemented versions.

As we expected, the overload introduced by handling these two levels of parallelism is noticeable and there is no improvement compared to the original version. Figure 18 clearly reflects how the performance decreases as we increment the matrix dimension compared to the original implementation using only BLAS without OpenMP.

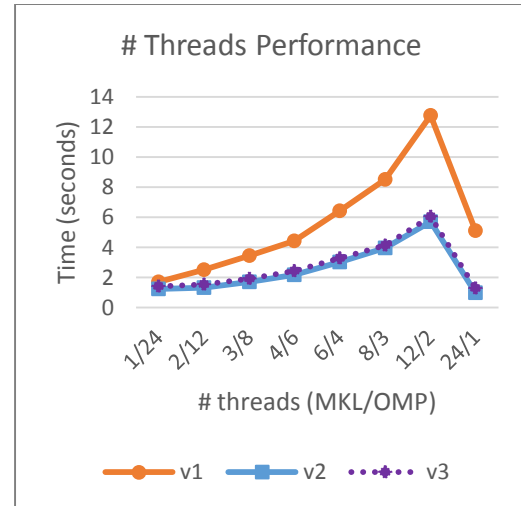


Figure 19 - Performance results obtained modifying the number of threads for a 5000 square matrix.

What is more, the best results have been obtained using 1 MKL thread and 24 OMP threads, 2 MKL threads and 12 OMP threads,

¹ Note: These latest results will be discarded, as it is equivalent as launching the test case without OpenMP.

and 24 MKL threads and 1 OMP (although we discarded this test case). We can appreciate this behaviour looking at Figure 19, where the best results obtained for a 5000 square matrix using different number of threads are shown.

In terms of SpeedUp and comparing to the original version, the new implementations demonstrate that do not offer any benefits, as we already know. Each version offers a good performance for small cases, as seen in Figure 20, but increasing the matrix dimension clearly decreases the performance.

One interesting behaviour about the second and third version is that, after a 4000 by 4000 dimension, the performance stabilizes compared to the original version. It is still worse, but stops decreasing as the first version does. Figure 18 also reflected this observance, where the first version spent 12.11 seconds to solve the latest test case and the second and third one took 7.04 and 8.07 seconds, respectively. The original version needed, instead, 4.64 seconds.

In any way, we can potentially conclude that increasing the levels of parallelism does not necessary increases the performance, at least in the LU decomposition problem for the matrix sizes tested. However, this could change for larger matrices, as the multi-level algorithm might have better scalability; something we discovered by analyzing Figure 20 earlier.

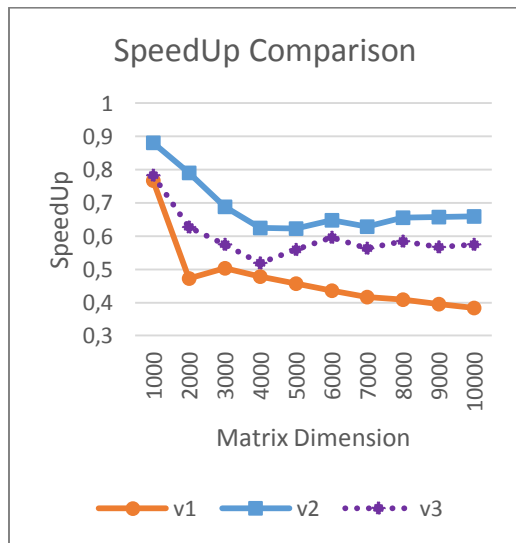


Figure 20 - SpeedUp comparison between the different implementations designed.

2.5 WORKING WITH HIGHER DIMENSION MATRICES

Until now, we have tested the proposed implementations by defining relatively large matrices. Our 10000 by 10000 matrix can be considered big, no objection, but scientist usually work with problems with a slightly higher dimension.

This introduces a new and, until now, unexpected problem: memory limit. As an example, a 10000 square matrix of **double** values implies nearly 800 Megabytes to be stored in RAM memory, while a 20000 square matrix means 3 Gigabytes and a 30000 square matrix implies nearly 7 Gigabytes of RAM. This cannot be easily handled in every computer and the problem must be split in different sub-problems of less size.

The next subsection will analyze a technique called Out-Of-Core that could help in this process.

2.5.1 Out-Of-Core LU factorization

Out-Of-Core is a technique that offers a clear benefit in cases where large amount of data must be processed to solve a concrete problem. This technique aims to store all the input / output data in secondary storage, loading only smaller parts of the problem into the RAM memory. So, for example, if we would like to use a 30000 square matrix as an input for the LU decomposition, we can split this matrix into 4 submatrices of dimension 15000 by 15000, offering a real and effective alternative to compute the factorization.

The main inconvenience of this technique is that we must handle thousands (maybe millions) of I/O operations to and from secondary memory. This introduces a clear bottleneck, as we are not able to read and write data faster than in RAM memory.

Our idea is to analyze a general algorithm similar to the third version seen in section 2.4, whose objective is to divide in blocks the original matrix. In this way, the process is exactly the same as the block-based LU factorization, but with higher dimensions and keeping in mind that each block is stored in secondary memory (Figure 21).

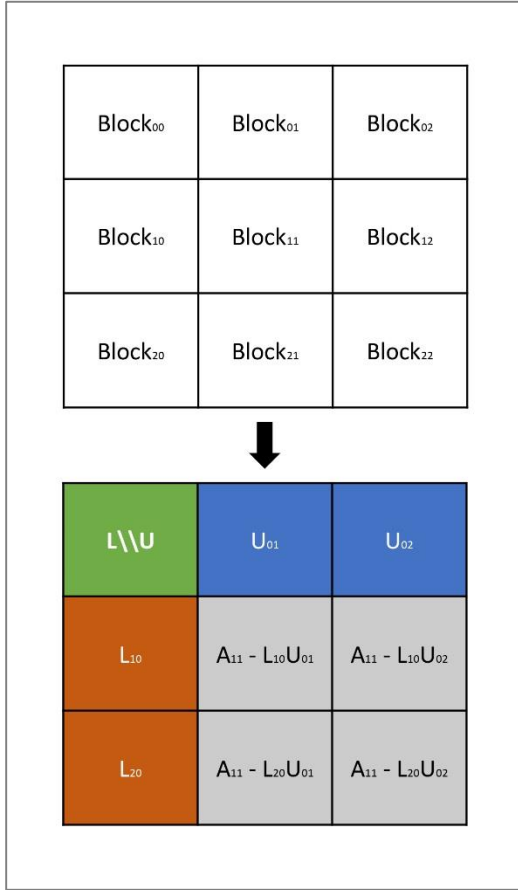


Figure 21 - Example of how the Out-Of-Core algorithm works. Above is represented how the data is stored in secondary memory, while below is how the algorithm manipulates this data.

The tests defined will start with a 10000 square matrix divided in 5000 square blocks, and will end with a 100000 square matrix divided by 25000 square blocks. The idea is to use, when possible, a half of the full size or the first multiple available within the memory limits of the workstation used (e.g., 25000 multiplied by 4 gives 100000, which means we will need to split the original matrix into $4 \times 4 = 16$ matrices of 25000 square dimension).

This last point is very important, because we cannot use more than 32 Gigabytes of RAM memory (the total available). In fact, we may consider 29 Gigabytes as our limit because many services are currently running, including the operating system itself.

So, given 64 bits for each double, we are able to store a total of:

$$MAX_{doubles} = \frac{29 \cdot 2^{30}}{(64/8)} \approx 38.92 \cdot 10^8 \text{ doubles}$$

And as we know we are working with square matrices, we can actually discover what the matrix size limit is:

$$MAX_{dimension} = \sqrt{MAX_{doubles}} \approx 62388$$

But this is not completely correct, because we may actually need to load into RAM memory up to 3 simultaneous matrices. For example, when we want to calculate the sub-matrix $A'_{11} = A_{11} - L_{10}U_{01}$, we need to load A_{11} , L_{10} and U_{01} , which implies that the dimension limit is the square root of a third of the maximum number of doubles:

$$MAX_{dimension_{final}} = \sqrt{\frac{MAX_{doubles}}{3}} \approx 36019$$

For simplicity, we will always use multiples of the full dimension. In this way, for a 60000 square matrix we will split its data into 4 smaller 30000 square matrices, while for a 70000 square matrix we will divide it by 4 smaller 35000 square matrices (i.e., near the limit). This means that, for an 80000 square matrix, we cannot use a 40000 square matrix and instead the highest multiple below the memory limit is 20000.

On the other hand and as we know the I/O operations are expensive, the algorithm designed follows a top-down $\leftrightarrow$ bottom-up perspective to avoid repeated reads from secondary memory. For example, following Figure 21, L_{10} is used with U_{01} and U_{02} to calculate A'_{21} and A'_{22} . So the next iteration of the algorithm will not start in the first row/second column but in the last row/second column, avoiding an additional read from disk. The more the block operations, the more the benefit of this design.

One last point prior to the analysis of the results obtained is to advice that there is not only a limit in terms of RAM memory, but a limit in secondary storage to save all the data that will be processed during the LU decomposition. In other words, a 64 bit double stored as plain text implies multiple characters of 1 byte in size, including additional ones like the sign of the number or the dot to represent the decimal part. As a reference, for the 50000 square matrix test case, one of the four low-

precision 25000x25000 matrices used during execution occupied near 5 GB (20GB in total).

If we increment the precision by storing more decimal numbers, we can easily reach about 100GB, and it will be a lot worse for higher dimension test cases like the 100000 square one (where, using low-precision, the total amount of data occupied about 70GB).

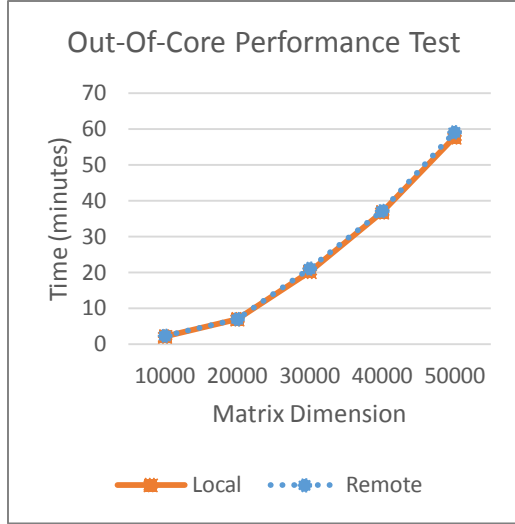


Figure 22 - Results achieved for the Out-Of-Core implementation of the LU factorization. The graph shows two different results, based on the fact of where the data was stored (i.e., local or remote).

This fact gave us the opportunity to think about a possibility of remote storage instead local storage. If the network is considerably fast, we could avoid the local bottleneck and store the information in a remote computer.

Figure 22 shows the results obtained for matrix sizes starting at 10000 by 10000 up to 50000 by 50000, storing data both locally and remote. As we can see, there is no difference between the tests (about seconds) because even for the initial case the time needed was more than 2 minutes, which clearly reflects how the I/O operations are affecting to the process. As 4.64 were needed in the BLAS / MKL implementation, there is a huge performance decrement in comparison:

$$SpeedUp = \frac{Result_{block_{BLAS}}}{Result_{block_{OOC}}} = \frac{4.64}{129.02} \approx 0.04$$

But our interest is focused in higher dimension matrices, something we could not have been able to handle using directly the original BLAS / MKL algorithm. The reader can appreciate in Figure 22 that the algorithm

finishes the LU factorization in less than 1 hour for the 50000 square matrix, which can be considered as a good result given the fact that we need about 20 Gigabytes of RAM memory to handle such a problem in non Out-of-Core implementations. Furthermore, the original sequential algorithm was unable to compute the 10000 square matrix test in less than 30 minutes, where the Out-Of-Core can compute a 40000 by 40000 matrix in about the same time.

In other terms, Figure 23 shows the SpeedUp of the local storage version compared to the remote one, which is only slightly better (something, we admit, was initially unexpected). What is more, an improvement in the connection between the workstation and the remote storage could benefit the whole process - right now it is located in the same laboratory and connected thru the same network using 10/100 Mbps Ethernet, but optical fiber cable replacement could improve the bandwidth available.

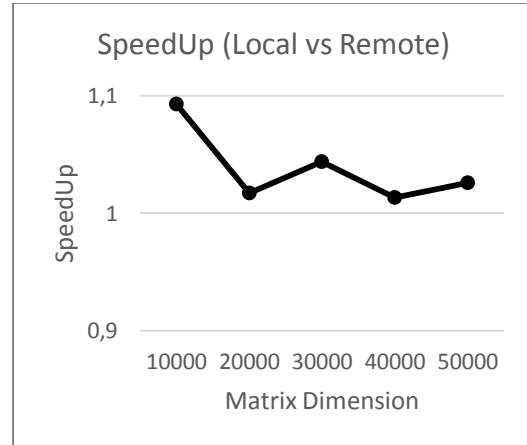


Figure 23 - SpeedUp of the local storage version compared to the remote storage one.

The next subsection will provide a better solution of the algorithm that could be potentially combined with this latest fact in future works.

2.5.1.1 Trying to improve the I/O operations performance

There is no objection to agree that I/O operations must be improved to increase the general performance of the algorithm. We decided to develop an additional version, where two different changes were introduced in comparison with the previous algorithm:

1. **Multi-threaded I/O:** Until now, a single thread was assigned to read and store the data into the secondary memory. The new approach assigns this job to multiple simultaneous threads working with the same file.

This implies that a fixed length size per number must be established, because, otherwise, a variable size could produce conflicts between different threads. This fact increases the size per file and the amount of data to be handled within secondary memory, but we expect a benefit in chance.

2. **Different I/O operations:** Using a fixed length per number offers a huge gain while using I/O operations. It is known that `stdio` library for C [23] includes many operations that help to efficiently handle read and write operations, and although we have been using these methods until now, we can now directly change the mode we read /write the data as we actually know directly the length per number.

These modifications have drastically change the performance of the implementation. The first test case of a 10000 dimension matrix finished in 11.13 seconds, improving noticeably the original Out-Of-Core implementation by a 1000%:

$$SpeedUp = \frac{Result_{blockOoC}}{Result_{blockOoCv2}} = \frac{109.02}{11.13} = 11.59$$

Figure 24 reflects the performance comparison between the original version and the multi-threaded I/O one. The results obtained have been more than satisfactory, spending only 15 minutes to solve the LU decomposition of a 50000 by 50000 matrix. This is even better than many of the results obtained for smaller dimensions in the previous sections.

Nevertheless, there is a test case whose result is interesting: the 70000 dimension one. The algorithm not only spent more time than the preceding 60000 square matrix test (which is completely normal), but a way more than the next two following test cases (which is less

usual). This behaviour coincides with the fact that the block size is nearly the RAM memory limit and this may have caused this performance decrement. It could also signify that now it is more efficient to store smaller blocks handled with multi-threaded I/O operations than storing huge blocks as earlier.

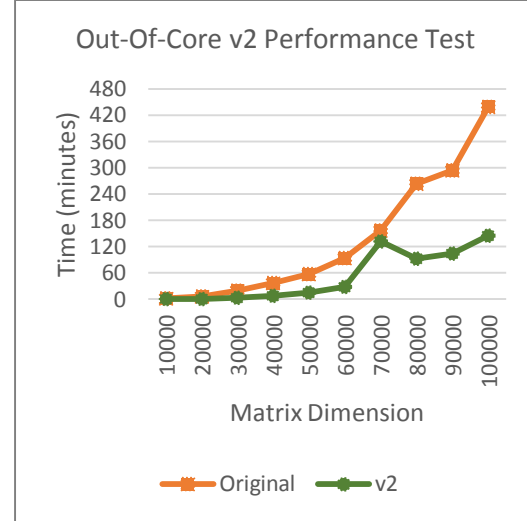


Figure 24 - Performance comparison between the original Out-Of-Core implementation and the new multi-threaded I/O one.

To clarify what really happened, we tested again the same case with a smaller block size multiple of 75000 (i.e., 14000 by 14000). The result was positive and the time was reduced by 40 minutes, which possibly means that the operating system was using swapping when the RAM memory limit was reached. This is something we did not appreciate earlier when using single-threaded I/O operations, so it could be related by this change.

We also pointed earlier that using smaller block sizes could be more efficient, but this fact cannot always be true due to the fact that more I/O operations time and less computation time is established. In other words, if we reduce the block size, the algorithm spends less time computing data and more time writing values to secondary memory.

To conclude, we would like to add that the fixed length per number increased the size per file needed between 4 to 5 times the size of the variable length version. For example, a 30000 square matrix using low-precision now needs 18GB per file, while a 35000 square matrix occupies about 25GB per file.

2.5.1.2 Using different storage methods

Until now, we have evaluated the performance of the Out-Of-Core algorithm by storing the values of the whole matrix in different blocks but using a plain text mode. We have seen that this is obviously a problem, because each character occupies 1 byte and more precision means more storage. Furthermore, this also implies that more I/O operations must be made and this might cause a notably performance decrement.

An alternative is to store this data in binary mode. By doing so, we have a fixed 64 bits (8 bytes) length per number with the most available precision for a double value and also less secondary storage requirements. For example, a 30000 square matrix occupies around 7GB, which is far below the 18GB requirement of the original text mode.

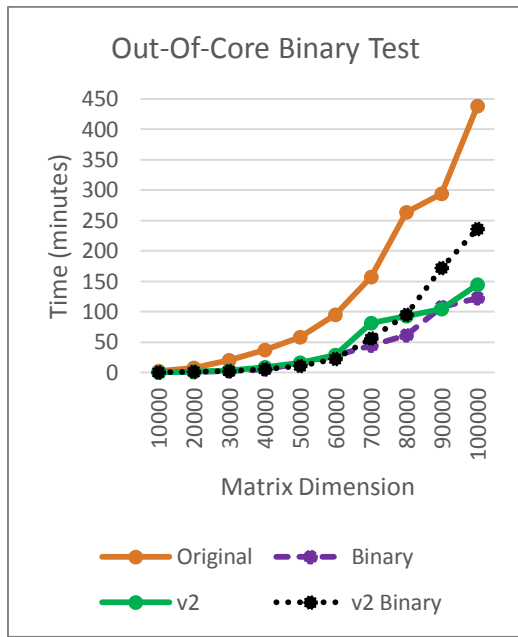


Figure 25 - Performance comparison between the original Out-Of-Core implementation, the multi-threaded I/O one and both using binary storage mode.

For a square matrix size up to 100000, Figure 25 reflects the performance obtained by the different implementations of the same algorithm. The graphs with a “v2” tag implies that the I/O operations have been made by multiple threads, while the ones with “Binary” implies the data was stored in this mode. We can appreciate different interesting behaviors:

- Although the cited figure is not completely clear, it is better to split the

I/O operations in multiple light-weight operations managed by different threads simultaneously. A slightly performance difference can be appreciated up to dimension 60000, with a 21% performance increment (around 5 minutes of difference) in this case. After that size, it is better to use single-threaded I/O.

- Secondly and related with the latest fact, it is not always better to use binary reads and stores, except for the fact that it always occupies less secondary storage. For example, following the multi-threaded / text mode approach, we can get a similar performance to the single-threaded / binary mode version. Furthermore, a slightly 3% performance increment is appreciated in the 90000 square test case, and we might observe better results for higher dimension matrices.

In the end, we have to agree that the results briefly explained gave us a good idea of how convincing is this technique for larger matrices. The following sections aims to solve the LU decomposition by using computational alternatives like the GPU.

3 GPU PERFORMANCE EVALUATION

We have already seen that using multiple CPU threads even while storing the data is translated into a performance gain. Over the last few years, a different technique has emerged using GPUs for general computing usage instead of graphics usage. The advantage is found in the fact that graphics operations are commonly based on matrices and the hardware is ready to this kind of operations by using multiple tiny processors that can handle thousands of simultaneous mathematical operations like sums.

Using the graphic card to compute matrix operations is, in theory, a benefit. This section’s objective is to evaluate the performance obtained by solving the LU decomposition using NVIDIA CUDA [24]. More specifically,

we will use NVIDIA's BLAS implementation based on CUDA (called cuBLAS [25]) and also an already implemented LU factorization by calling directly to MAGMA library [26].

3.1 LU DECOMPOSITION USING CUDA (cuBLAS)

NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) [25] is a GPU version of the BLAS standard library implemented by NVIDIA. According to the manufacturer, it is said to deliver better performance than MKL BLAS.

As we already implemented an LU decomposition using MKL BLAS, translating the same code to cuBLAS can be done in a few steps. The only restriction is that this library expects the data to be stored only in a column-based format, something we have already seen in MKL BLAS but which offered a row-based alternative that we used.

There is also another important issue. Our block-based algorithm computes smaller LU decompositions as a first step for each iteration (please check 2.1.1 for further information), and although we have translated these operations to the GPU cuBLAS equivalent, one still must divide each element of each row by its pivot. Until now we have been computing the pivot's inverse and calling a BLAS level 2 operation with the result as a constant, but the problem is that in this case we do not have the value and **it must be retrieved from the GPU**.

This forces thousands of little I/O just to retrieve this value, so we decided to implement several versions to evaluate its performance:

1. **cuBLAS version:** This is exactly the same algorithm using cuBLAS instead of BLAS and with the pivot problem. We also measured the time spent to copy and retrieve the full matrix to and from the GPU.
2. **cuBLAS version without data transfer measurement:** Like the previous one, but we excluded the measurement of the time spent to copy and retrieve the data from the GPU.
3. **cuBLAS version using streams:** An interesting technique is to use CUDA streams [27] to parallelize several workloads in the GPU. We integrated two streams within step 2 and 3 of the algorithm.
4. **cuBLAS version using MKL:** With the idea to avoid the performance decrement of the pivot, we tested out a different version where we use MKL BLAS to solve the smaller LU decomposition block. This means that only two I/O operations from GPU will be made per iteration (i.e., one to retrieve the block to RAM memory and another one to store it back to the GPU memory).

We evaluated again the usage of different block sizes per matrix dimension, as well as the usage of different number of concurrent MKL threads in the latest version. These tests have been conducted in the same workstation (see Table 2 for information about Saturno).

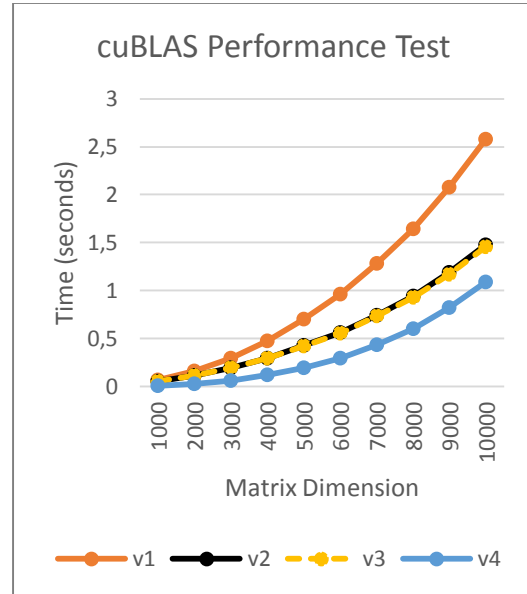


Figure 26 - Performance comparison between different implementations using cuBLAS library (i.e., GPU).

The performance has increased notably using the GPU, as seen in Figure 26. Compared to MKL BLAS using only the CPU, we can appreciate that its scalability for higher dimension matrices is clearly better. For example, a 10000 square matrix LU decomposition can be computed in less than 2 seconds, and combining MKL BLAS with cuBLAS (i.e., tag

v4 in the graph) the factorization is completed in 1.09 seconds. This means that we have even improved the original MKL BLAS algorithm:

$$SpeedUp = \frac{Result_{blockBLAS}}{Result_{blockcuBLASv4}} = \frac{4.64}{1.09} = 4.26$$

Figure 27 confirms this behaviour, where the SpeedUp for each matrix dimension in comparison with MKL BLAS is shown.

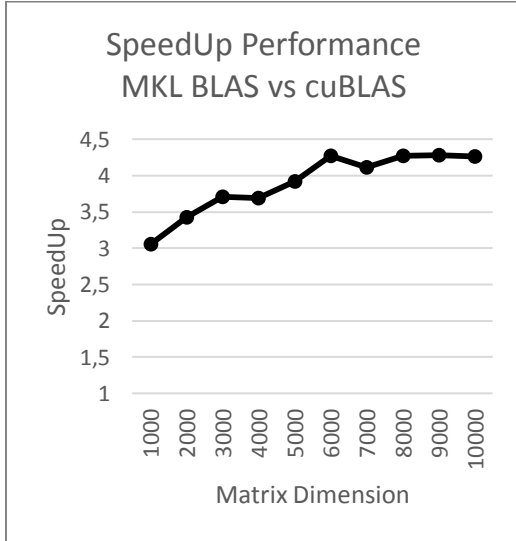


Figure 27 - SpeedUp comparison between the LU decomposition implementation using BLAS (MKL) and cuBLAS.

If we measure the time dedicated to copy and retrieve the whole matrix from the GPU (i.e., tag v1 in Figure 26), we still get a very good performance with 2.56 seconds of computing time for a 10000 by 10000 matrix. Also, we have again noticed that the best block sizes are 64, 128 and 256, something we already discovered in the previous sections (please note that only the best results have been shown).

Increasing the number of simultaneous threads does not offer a direct benefit, as expected. The reason is that there is a little overhead while computing such a tiny matrix with multiple threads. Or in other words, it is more suitable to handle the operations with one single thread in comparison (e.g., 1.09 seconds of 1 thread compared to 1.14 seconds of 24 threads).

The results are, then, very interesting. We have not only increased the performance of the original algorithm, but also we have evaluated that even the time spent to copy and retrieve data from the GPU is not huge.

3.2 LU DECOMPOSITION USING MAGMA

Similarly to what we have seen with LAPACK, MAGMA [26] is a library that offers a linear algebra library for heterogeneous / hybrid architectures. It is based on the idea of addressing different challenges using hybrid solutions that combine both CPU and GPU computing.

In this way, the library offers an LU factorization method that can directly solve the decomposition. The algorithm is similar to the fourth cuBLAS version that we implemented in the previous section, but with a slightly different (and expected efficiently) design.

Please note that, as we are not using pivoting, to precisely evaluate the performance we will make a call directly to an alternative LU decomposition method that the library actually offers [28].

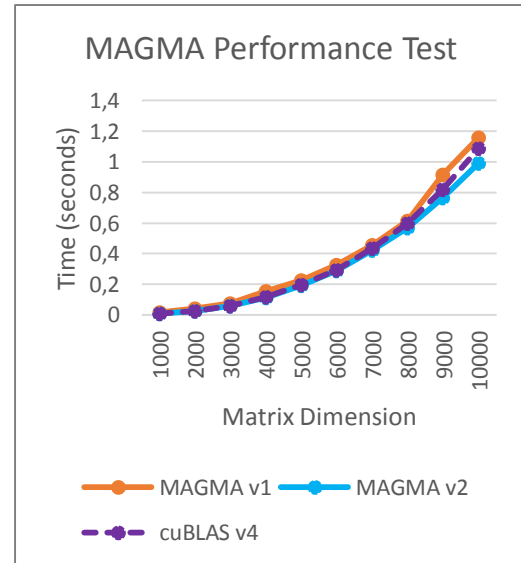


Figure 28 - Performance comparison between two different implementations that use MAGMA and the best cuBLAS implementation of the previous section.

We have implemented two different versions of the algorithm:

1. **MAGMA method:** The first version will directly call to MAGMA's LU factorization method.
2. **MAGMA + cuBLAS:** This is a hybrid version that combines the same algorithm implemented in the previous section but that uses MAGMA to solve the LU block of each iteration.

For a matrix dimension up to 10000 square, we have tested the performance by using different block sizes (i.e., MAGMA + cuBLAS version) and, as well, the usage of different number of threads due to the fact that its method also uses BLAS (MKL).

Figure 28 reflects the performance of the described implementations compared to the fourth cuBLAS algorithm previously implemented, which offered the best performance. The times obtained are very similar to the latter version, and in fact MAGMA's LU decomposition offered nearly exact times with only a slightly performance increment.

But combining cuBLAS with the usage of MAGMA is different. We can reach times below 1 second even for a 10000 square matrix, which is better than what we have already achieved using only cuBLAS:

$$SpeedUp = \frac{Result_{blockBLAS}}{Result_{blockcuBLASv4}} = \frac{4.64}{0.99} = 4.69$$

If we now evaluate how block sizes affect the performance of the second implementation (Figure 29), we can confirm that larger sizes are better. For example, the 0.99 seconds of the 10000 square matrix has been obtained by using a 1024 dimension block. The reason is obvious, as we are now directly using MAGMA's LU for each iteration and higher dimensions implies higher performance (up to a limit between 256, 512 and 1024).

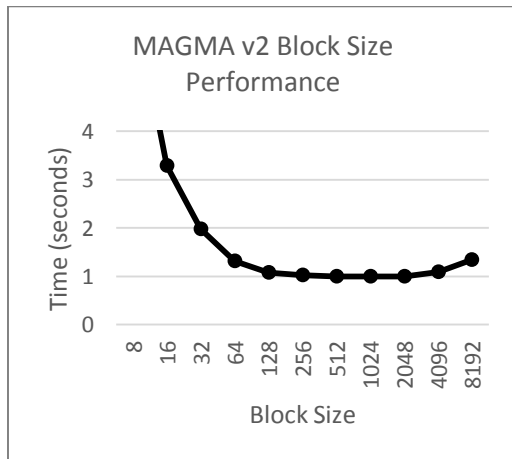


Figure 29 - Behaviour of the second algorithm by varying the block size for a 10000 dimension matrix.

Note also that varying the number of threads has in this case a direct benefit.

4 HETEROGENEOUS LU EVALUATION

Until now, we have seen many different approaches to solve the LU decomposition. We also introduced the reader into a technique called Out-of-Core, useful for larger matrix dimensions, and we also evaluated that storing remotely each block of the whole matrix to be factorized is equivalent as storing data locally, with a small performance impact.

In this section we propose an alternative Out-of-Core algorithm that uses MPI to distribute the data between different heterogeneous nodes. Each node will previously have assigned a specific workload and only the essential data will be sent for each iteration of the algorithm.

Although the processing power of these nodes is very different, the GPU power while solving the LU decomposition is very similar and only differs by some seconds according to our tests (check Table 2 for further information about the processing capabilities of these nodes). In this way, we can consider each node to have the same processing power, as the main cost will be mainly located in the communications.

Node	GPU card available
Saturno (Master)	NVIDIA Tesla K20c
Jupiter	NVIDIA Tesla C2075
Marte	NVIDIA GeForce GTX 480
Mercurio	NVIDIA GeForce GTX 480

Table 2 - Specification of the workstation used

The idea of the algorithm is to split the whole matrix into multiple blocks that are stored in secondary memory, as we did with the original Out-of-Core algorithm. Each row of blocks is pre-assigned to a node, so that only that node has to store and manipulate the data and just a final synchronization to the master node must be made (i.e., to Saturno).

The nodes know exactly the rows they have assigned because of their unique MPI node identifier. This means that using the module operation with the number of row we

can easily assign different rows to different nodes.

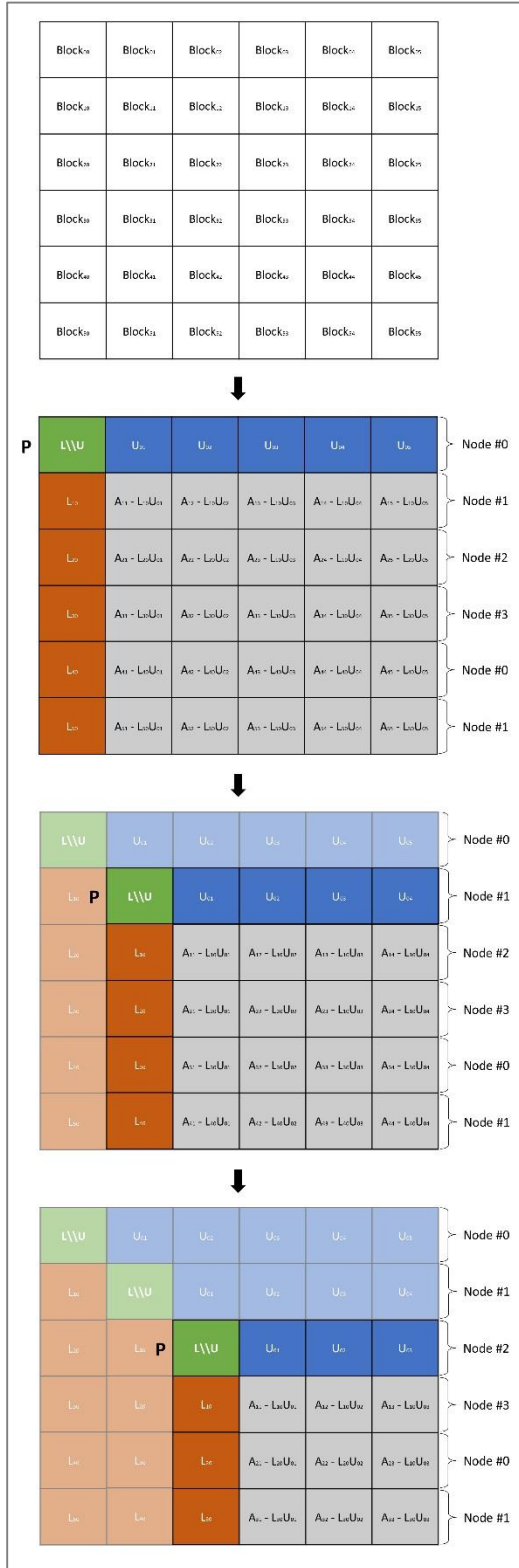


Figure 30 - Example of how the Heterogeneous Out-of-Core algorithm designed works for the first three iterations using 4 nodes and a matrix split into 6 by 6 secondary storage blocks. The "P" character means that the node is the current pivot for the specified row, which has to be synchronized with the rest of the nodes. Note that some computing is overlapped to avoid idle times.

We then consider an LU block algorithm as we did earlier in section 2.1.1, but with a slightly change. For each iteration, one node acts as the “pivot node”, which means that it is the one that will locally solve its LU block (i.e., the first one) and extend the results to the rest of the blocks of its row according to the step 2 of the followed algorithm. The rest of the nodes will synchronize remotely with the current pivot, overlapping communications with computation time to avoid possible idle times (i.e., combining step 3 and 4 with communications with the pivot).

This also means that, in the following iteration of the algorithm, the “pivot node” will be the one who previously worked with its own row, repeating the whole process of calculating the LU and synchronizing with the rest of the nodes. Figure 30 shows an example of the algorithm for a 6 by 6 secondary memory block split of a relatively large matrix (e.g., 30000 square LU decomposition using 5000 dimension OoC blocks).

Note that the algorithm reuses some of the reads to reduce the overhead of the I/O operations to secondary memory. Also, note that there is not any explicit synchronization barrier after each iteration, which implies that the next “pivot node” can continue computing its LU even while the previous nodes are still working with their rows.

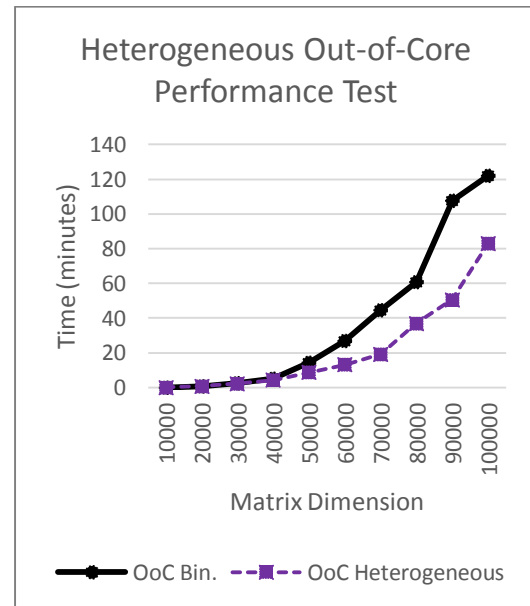


Figure 31 - Performance comparison between one of the OoC algorithms and the Heterogeneous version varying the matrix size.

The algorithm has been implemented using the Out-of-Core algorithm whose I/O operations were made in binary mode. Although we discovered that using a multi-threaded text mode offered some benefits, the average performance is more stable in this case and the key is actually how the algorithm manages the communications between the nodes.

Figure 31 reflects the performance obtained by this heterogeneous algorithm compared to the original Out-Of-Core algorithm that was evaluated exclusively within the same workstation. The heterogeneous algorithm is clearly better even for smaller matrix sizes, and the difference is noticeably starting from a 50000 by 50000 dimension.

In fact, for a 90000 square matrix there is a time difference of nearly 1 hour, which is an 89% of performance gain in comparison. This is exactly what is depicted in Figure 32, where the absolute time differences are shown for different matrix dimensions.

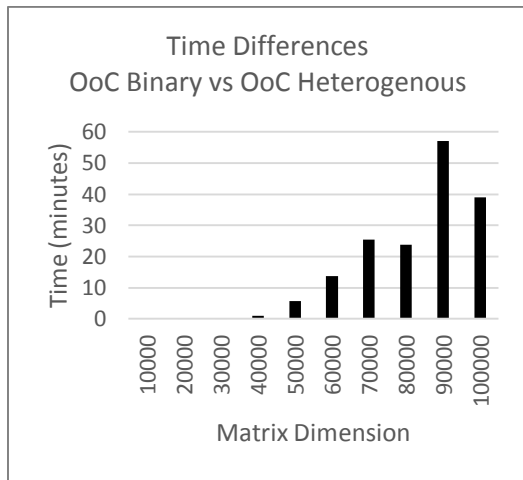


Figure 32 - Absolute time differences between the Out-Of-Core Binary algorithm and the Heterogeneous described in this section. Higher bars indicate better performance of the latter.

The results might not sound as surprising due to the fact that we are using multiple nodes, but the truth is that the algorithm's scalability is notably better and larger matrices could reflect that the usage of multiple simultaneous nodes is clearly an advantage.

Future work could be focused on reducing the communications between the nodes or increasing the computation parallelism by, for example, combining CPU+GPU or more than one GPU computing data simultaneously.

5 CONCLUSION

In this work we have analyzed multiple implementations of the LU decomposition by first introducing some of the concepts needed to understand the whole document, including how the basis algorithm work.

We latter analyzed and compared a block-based approach by alternatively using libraries based on the BLAS standard, which included MKL BLAS, ATLAS and GotoBLAS, and also those based on LAPACK by calling directly an LU factorization method (e.g., PLASMA library). The results obtained were positive, with an average of less than 5 seconds to solve an LU decomposition of a 10000 by 10000 matrix.

Then, different levels of parallelism were introduced with the idea of solving some performance issues, and also four Out-Of-Core algorithms were implemented to analyze the performance of the LU decomposition for relatively large matrices up to 100000 square, storing the data both locally and remotely. Only 2 hours of computation time was needed to solve the larger test case and even less than 30 minutes for the 50000 by 50000 matrix.

We finally introduced GPU computing by using cuBLAS and MAGMA, reaching a 0.99 seconds mark for a 10000 square matrix. Furthermore, these results were later translated into a Heterogeneous algorithm, where we improved the original Out-of-Core version by a 47% and reduced in 39 minutes the 100000 square matrix test case. In fact, the original optimized classical Gaussian elimination algorithm reached the 20 minutes time limit for a 10000 square matrix, which is more than the time spent by the last algorithm to solve a 70000 by 70000 square matrix.

In conclusion, we can clearly determine that the usage of different techniques to solve mathematical problems like the LU factorization is a direct benefit for science in general. Our work has been very positive and we expect that future research using, for example, hybrid computing could notably increase the performance gain and clearly improve the obtained results.

6 REFERENCES

- [1] "Intel® Math Kernel Library," Intel Corporation, [Online]. Available: <http://software.intel.com/en-us/intel-mkl>.
- [2] "BLAS (Basic Linear Algebra Subprograms)," [Online]. Available: <http://www.netlib.org/blas/>.
- [3] C. L. Lawson, R. J. Hanson, D. Kincaid and F. T. Krogh, "Basic Linear Algebra Subprograms for FORTRAN usage," *ACM Trans. Math. Software* 5, p. 308–323, 1979.
- [4] "Accelerated Parallel Processing Math Libraries - Official web site," Advanced Micro Devices, Inc., 2013. [Online]. Available: <http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-math-libraries/>.
- [5] "Linear Algebra PACKage," [Online]. Available: <http://www.netlib.org/lapack/>.
- [6] G. Strang, "Elimination = Factorization: $A=LU$," in *Introduction to Linear Algebra 4th edition*, Wellesley-Cambridge Press, 2009, pp. 95-98.
- [7] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimization of Blocked Algorithms," *ACM SIGARCH Computer Architecture News*, vol. Vol. 19, no. 2, pp. 63-74, 1991.
- [8] "DSCAL - Scales a double precision vector," [Online]. Available: <http://www.mathkeisan.com/UsersGuide/man/dscal.html>.
- [9] "DGEMM - Perform matrix-matrix operations," [Online]. Available: <http://www.math.utah.edu/software/lapack/lapack-blas/dgemm.html>.
- [10] "DTRSM - Solves one matrix equation," [Online]. Available: <http://www.math.utah.edu/software/lapack/lapack-blas/dtrsm.html>.
- [11] "Intel® MKL Product Brief," Intel Corporation, 2013. [Online]. Available: http://software.intel.com/intel_mkl_product_brief_en.
- [12] "Using the Intel® MKL Parallelism," Intel Corporation, [Online]. Available: http://software.intel.com/sites/products/documentation/hpc/mkl/lin/MKL_UG_managing_performance/Using_the_Intel_MKL_Parallelism.htm.
- [13] "Intel® MKL - Techniques to Set the Number of Threads," Intel Corporation, [Online]. Available: http://software.intel.com/sites/products/documentation/hpc/mkl/lin/MKL_UG_managing_performance/Techniques_to_Set_the_Number_of_Threads.htm#XREF_Techniques_to_Specify.
- [14] "What is ATLAS?," [Online]. Available: <http://math-atlas.sourceforge.net/faq.html#what>.
- [15] "Can I vary the number of threads ATLAS uses dynamically?," [Online]. Available: <http://math-atlas.sourceforge.net/faq.html#tnum>.
- [16] "GotoBLAS2 official website," Texas Advanced Computing Center, 2011. [Online]. Available: <http://www.tacc.utexas.edu/tacc-projects/gotoblas2>.
- [17] "OpenBLAS official website," [Online]. Available: <http://www.openblas.net/>.
- [18] K. Goto and R. A. van de Geijn, "Anatomy of High-Performance Matrix Multiplication," *ACM Transactions on Mathematical Software* 34, no. 12, 2008.
- [19] J. Meng-qi, Z. Yun-quan, S. Gang and L. Yu-cheng, "Research on High Performance Implementation Mechanism of GOTOBLAS General

- Matrix-matrix Multiplication," Academy of Sciences, Beijing, 07 2008. [Online]. Available: http://en.cnki.com.cn/Article_en/CJFDTOTAL-JSJC200807030.htm.
- [20] "PLASMA official website," Innovative Computing Laboratory, University of Tennessee, 2013. [Online]. Available: <http://icl.cs.utk.edu/plasma/>.
- [21] "DGETRF - Computes an LU factorization of a general M-by-N matrix," [Online]. Available: <http://www.math.utah.edu/software/lapack/lapack-d/dgetrf.html>.
- [22] "PLASMA can exploits multithreaded BLAS libraries to improve performance," Innovative Computing Laboratory, University of Tennessee, [Online]. Available: http://icl.cs.utk.edu/projectsfiles/plasma/html/InstallationGuide.html#_installing_blas.
- [23] "C library to perform Input/Output operations," cplusplus.com, [Online]. Available: <http://www.cplusplus.com/reference/cstdio/>.
- [24] "CUDA Parallel Computing Platform," NVIDIA Corporation, [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html.
- [25] "NVIDIA CUDA Basic Linear Algebra Subroutines," NVIDIA Corporation, 2014. [Online]. Available: <https://developer.nvidia.com/cuBLAS>.
- [26] "Matrix Algebra on GPU and Multicore Architectures," University of Tennessee Knoxville, [Online]. Available: <http://icl.utk.edu/magma/>.
- [27] "How to Overlap Data Transfers in CUDA C/C++," NVIDIA Corporation, 2012. [Online]. Available: <http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/>.
- [28] "MAGMA - dgetrf_nopiv_gpu," University of Tennessee Knoxville, [Online]. Available: http://icl.cs.utk.edu/magma/docs/magma__d_8h.html#ade6195acb1aa52aaa493a865c4e9e17f.