Performance of LU decomposition and matrix multiplication parallel methods

Report

Mestrado Integrado em Engenharia Informática e Computação

Parallel Computing

Authors:
David Luís Dias da Silva - up201705373@fe.up.pt
Luís Pedro Pereira Lopes Mascarenhas Cunha - up201706736@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

May, 2021

1 Introduction

Matrix multiplication and LU decomposition are ubiquitous operations in computer science. From machine learning to scientific computing, these operations need to be done often and, sometimes, with time restrictions. With this rises the need to optimize the performance of these algorithms, for example through the use of parallel computing.

In this report, we analyse parallel algorithms for matrix multiplication and LU decomposition, experimenting with different algorithmic approaches. We test this implementations usin different devices such as CPUs and GPUs and using different programming APIs such as OMP, CUDA and Sycl.

We start by giving some background about these operations (section 2), we create a baseline by analysing both operations in a sequential environment (section 4) and then we analyse them under parallelism using different implementations (section 5). Both the sequential and parallel section are divided by operation. In each subsection we start by explaining the different algorithms, followed by a set of experiments and the respective analysis.

2 Background

2.1 Matrix multiplication

Let A be any $m \cdot p$ matrix, and let B be any $p \cdot n$ matrix, say:

$$A = (a_{ij})_{i,j=1}^{m,p} \quad \text{and} \quad B = (b_{ij})_{i,j=1}^{p,n} \tag{1}$$

Then the product $A \cdot B$ is defined to be the $m \cdot n$ matrix $C = (c_{ij})$ whose ij-entry is given by:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{2}$$

2.2 LU Decomposition

The goal of LU decomposition is to solve a linear system $Ax = b$ of n equations in n unknowns by factoring the coefficient matrix A into a product:

$$A = LU \tag{3}$$

where L is lower triangular and U is upper triangular. With this, the system can be solved by:

1. Rewrite $Ax = b$ as $LUx = b$
2. Define matrix y by $Ux = y$
3. Rewrite equation 3 as $Ly = b$ and solve for y
4. Solve $Ux = y$ for x

Notice that solving $Ly = b$ and $Ux = y$ is trivial because L and U are triangular matrices.

3 Experimental Setup

All of the experiments were ran 3 times and then the measurements were averaged in order to eliminate any possible discrepancies.

The OpenMP code was executed with the `-O2` flag as it improved performance in our preliminary experiments.

The Sycl code was compiled using the ComputeCpp implementation. However, since ComputeCpp doesn't support CUDA, we use DPC++ to compile the same code for the GPU experiments.

The CPU used for the experiments was a quad-core, Intel Core i5-6500 Processor running at a 3.20 to 3.60 GHz, with 4x32KB L1 Data Cache and a 4x256KB L2 Unified Cache. For the CUDA and Sycl experiments we also used a Nvidia GTX 1060 6GB. Appendices B and C contain more detailed information about the GPU and CPU respectively.

A description of how the different implementations were compiled can be found in appendix D.

4 Sequential algorithms

4.1 Matrix multiplication

4.1.1 Description

Naive The naive version of matrix multiplication arises from the direct translation of Equation 2, and is presented in Source Code A.1. Note that this version already includes the swap of the k and j loops in order to improve cache-efficiency (as is described in more detail in the previous project).

Our previous project showed that the naive version of this algorithm quickly grows unfeasible with the increase of the matrix size.

Blocks We can further explore cache awareness by using a blocks approach which is presented in Source Code A.2. This approach also enables the application of parallel algorithms which divide the work between threads based on the multiplication of blocks.

To perform a matrix multiplication between two square matrices of size n, $2n^3$ floating-point operations are required.

4.1.2 Experiments

The blocks algorithm was ran for matrices of side length 1024 to 8192 with an increment of 1024 while measuring the time. In addition, the algorithm was measured using block sizes of 64, 128 and 256.

4.1.3 Results

In Figure 1 we can see the execution times of the algorithm for several matrix and block sizes. Figure 2 shows that most block sizes produce a similar performance, around 3 Gflops/s. However, we can see that for large matrix sizes the performance starts to drop for all block sizes.

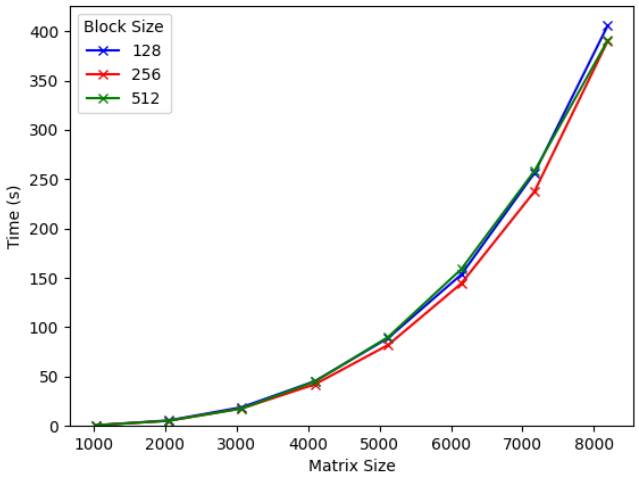

Figure 1: Time in seconds of the sequential matrix multiplication blocks algorithm for different block sizes.

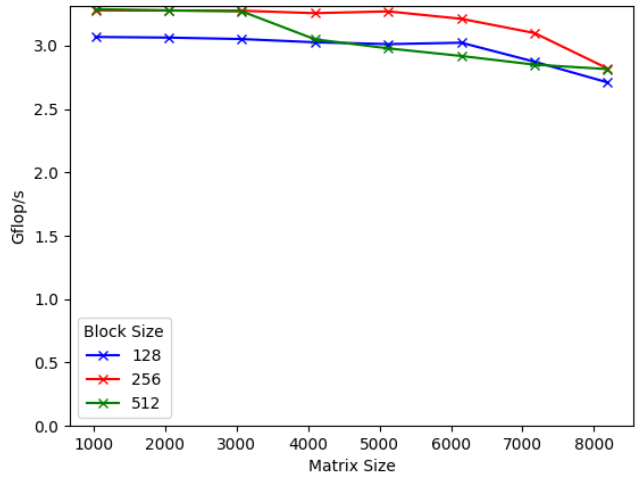

Figure 2: Performance in Gflop/s of the sequential matrix multiplication blocks algorithm for different block sizes.

4.2 LU decomposition

4.2.1 Description

Naive Here we describe a possible algorithm for finding an LU decomposition of a given matrix A of size $N \times N$. For each element of the main diagonal of the matrix, two steps are performed. The first is to update the k^{th} column of the matrix below the main diagonal by dividing its elements by the k^{th} main diagonal element, as in Equation 4, where i ranges from $k+1$ to N. Then, all

elements a_{ij} where i and j range from $k+1$ to N (A' in Figure 3) are updated as in Equation 5.

$$a_{ik} = a_{ik}/a_{kk} \tag{4}$$

$$a_{ij} = a_{ij} - a_{ik}a_{kj} \tag{5}$$

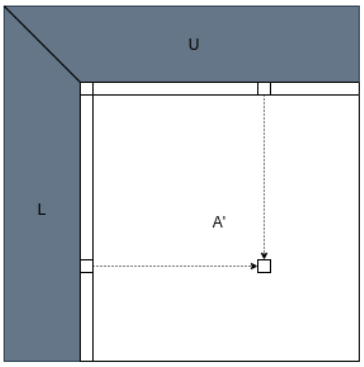

Figure 3: The dark area is the part of the matrix decomposed in previous iterations.

The naive implementation of the algorithm described above consists of performing each operation sequentially. The implementation can be seen in Source Code A.3. Lines 6-8 correspond to the step of updating the k^{th} column, and lines 10-15 correspond to the step of updating the elements to the right and below the k^{th} main diagonal element.

Blocks The implementation of the algorithm can be improved by grouping the operations in blocks. Instead of iterating the main diagonal one by one, always updating submatrix A', we delay its update until $blockSize$ elements of the diagonal are processed. Being $A00$, $A01$, $A10$, and $A11$ the submatrices of A illustrated in Figure 4, the algorithm consists of updating the $A00$ block using the naive algorithm described above. Then, $A01$ and $A10$ are updated using the elements of the diagonal of $A00$ in the same way they are updated in the naive algorithm. $A10$ is updated by, iteratively for each column k of $A10$, dividing the column's elements by the element of the main diagonal of $A00$ and then updating the columns $k+1$ until the end of the block using Equation 5. $A01$ is updated by, for each row of $A01$, updating the rows below using Equation 5.

After $A10$ and $A01$ are updated, $A11$ can be updated by subtracting to it the result of multiplying matrices $A10$ by $A01$. The matrix multiplication of $A10$ and $A01$ can also be blocked to improve cache efficiency. The algorithm is then repeated for the update $A11$. The algorithm can be seen in Source Code A.4, Furthermore, we also block the update of the columns of $A10$, in an attempt to improve the memory access pattern. By accessing each column in its entirety and if the column is large enough, the last elements of the column could displace the cache lines loaded when the first elements were accessed and we would lose the advantage of spatial locality when accessing the elements of the next column.

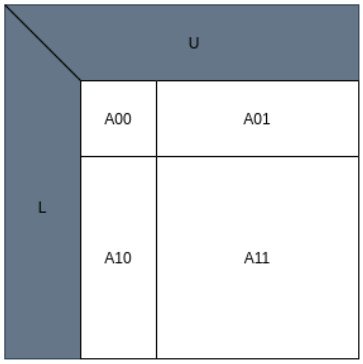

Figure 4: The dark area is the part of the matrix decomposed in previous iterations. $A00$, $A10$, $A01$ and $A11$ represent the submatrix being updated at a given iteration.

Computing the LU factorization of a matrix requires $\frac{2}{3}n^3$ floating point operations.

4.2.2 Experiments

Both algorithms were ran for matrices of side length 1024 to 8192 with an increment of 1024 while measuring the time. In addition, the algorithm was measured using block sizes of 64, 128 and 256.

4.2.3 Results

Figure 5 and Figure 6 shows that a blocking approach significantly outperforms the naive sequential method. In fact, all block sizes achieved a similar performance near 3 Gflop/s, despite a block size of 256 being able to consistently achieve slightly better performance.

We attribute this improved performance to the better utilization of spatial and temporal proximity of the cache. These results were also expected because the update represented by Equation 5 is analogous to a matrix multiplication which benefits from blocking as seen in the previous project.

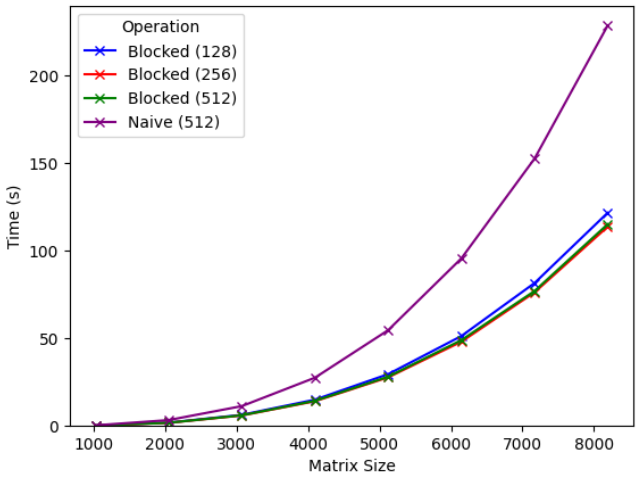

Figure 5: Time in seconds of the Naive (purple) and Block-oriented LU decomposition algorithms.

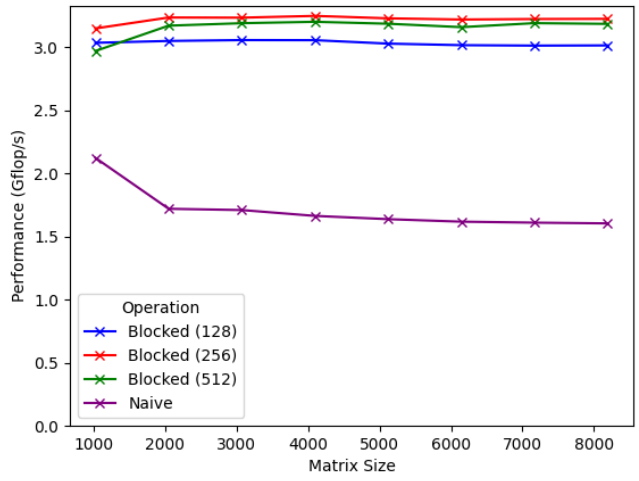

Figure 6: Performance in Gflop/s of the Naive (purple) and Block-oriented LU decomposition algorithms.

5 Parallel algorithms

5.1 Matrix multiplication

5.1.1 Description

Matrix multiplication is an "embarrassingly parallel" problem, which means that it can easily be divided into parallel tasks. In effect, each element of the result matrix can be calculated in parallel. Although this might be feasible

for small matrices, i.e. matrices where the number of elements is inferior to the number of possible threads, for larger matrices we need to employ the parallelism at a block level i.e. where the goal is to calculate each **block** of the result matrix in parallel. What follows are implementations notes of a parallel matrix multiplication algorithm in *OpenMP*, *CUDA* and *Sycl*.

OpenMP Implementing a parallel matrix multiplication algorithm with *OpenMP* is a simple task. By applying the `for` directive to the sequential block algorithm seen in the previous section, we are able do divide the blocks among different threads. The addition of the `collapse(2)` allows this division to be done among the iterations of the two outer loops, which in effect divides the work by the blocks of the resulting matrix. The source code for this implementation is in Source code 9.

CUDA We used CUDA in order to take advantage of an external *Nvidia* GPU. The natural division of the matrices into blocks provided by the algorithm intuitively lead us to divide the thread block grid according to the number of blocks, and the thread blocks according to the size of the matrix blocks (Source code 1).

```
1  dim3 dimGrid(matrixSize/blockSize, matrixSize/blockSize);
2  dim3 dimBlock(blockSize, blockSize);
```

Source code 1: Grid and block dimensions for CUDA kernels

We implemented two algorithms using CUDA, one where we divided the thread blocks according to the result matrix blocks (Source code 10), and one where we did the same but where we threads in the block took an additional step of cooperatively copying the operand blocks from global memory to local thread block memory (Source code 11).

SYCL We implemented 3 different versions of the algorithm using SYCL. One is a naive approach, in which a parallel for is used to execute one kernel for computing each element of the result matrix (source code 12). The other two are similar to the CUDA implementations. Both are block based (source codes 13 and 14), and one uses work group local memory (memory accessible to the threads in the same work group, similar to how shared memory is accessible by threads belonging to the same block).

5.1.2 Experiments

The parallel algorithm was run for matrix side sizes ranging from 1024 to 8192, with 1024 increments while measuring the time. We ran the algorithm in the three mentioned platforms: OMP, CUDA and Sycl.

The OpenMP experiments used block sizes of 128, 256 and 512. We also ran the experiment set using 1 to 4 processors (threads) in order to be able to study speedup, efficiency and scalability.

For CUDA we ran the blocked algorithm and the blocked algorithm with local memory for block sizes 8, 16 and 32. The block sizes are different from the OMP experiments because of limitations of the GPU block sizes.

We ran the Sycl experiments for the CPU and for the GPU. In each of the devices we used the same block sizes as in the CUDA version (due to limitations of the work group sizes). We ran the experiments for the 3 versions of the algorithm mentioned above: naive, blocked and blocked with local memory.

5.1.3 Results

Figure 7 contains the execution time of the OpenMP version of the matrix multiplication algorithm, using 4 processors. As expected, we can see a significant improvement regarding the sequential version. For the different block sizes used, we can see that 128 yields the best results for larger matrix sizes. This can also be seen in Figure 8, block size 128 has higher performance for larger matrix sizes. We can also see that there is some variation of the performance of the algorithm, especially for block size 512.

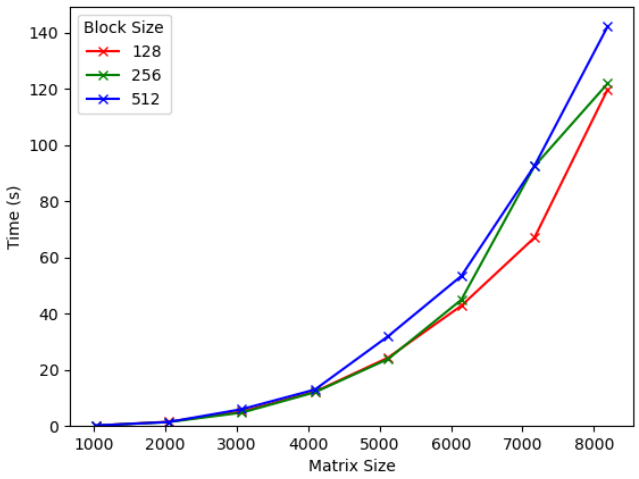

Figure 7: Time in seconds of the OMP the matrix multiplication algorithm for different block and matrix sizes and using 4 processors.

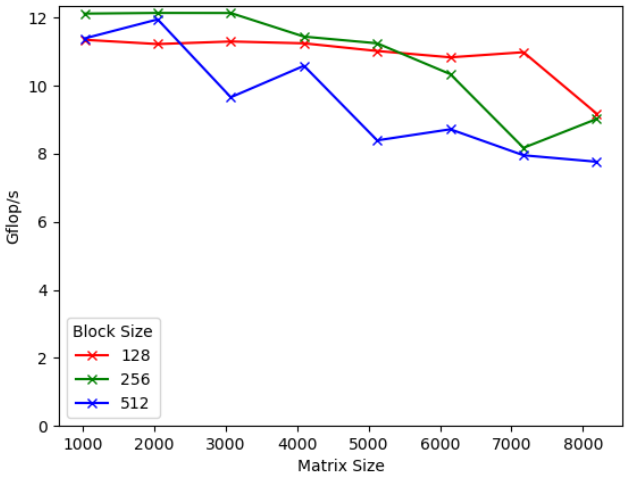

Figure 8: Performance of the OMP the matrix multiplication algorithm for different block and matrix sizes and using 4 processors.

Figure 9 contains the achieved relative speed up, computed by dividing the execution time of the parallel version using only one processor by the execution time of the parallel version using more processors, for different combinations of block size and number of processors. With 4 processors, the achieved speed up ranges from around 3.5, with a block size of 128, to near 2.5 with block sizes of 512 and 256 for large matrix sizes. For 3 processors, the achieved speed up is also best (around 2.7) with block size 128. For 2 processors, the achieved speedup is near 2, for all block sizes. We can also observe that for 2 processors in combination with all tested block sizes, the speed up varies less for different matrix sizes. The same happens for the smaller block size (128) with combination with all tested number of processors, even though the speedup starts to decrease when we reach matrix size 8192 with 4 processors.

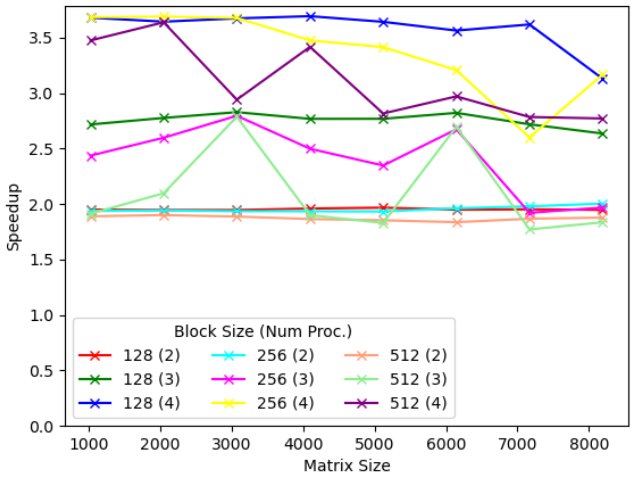

Figure 9: Speedup of OMP matrix multiplication algorithm for different block sizes and number of processors.

Another obtained performance measure is the efficiency, computed by dividing the speedup by the number of processors, whose results are present in Figure 10. We can observe that the highest (and with less variation) efficiency was obtained for 2 processors, with a value very close to 1. Within the measured efficiencies with 2 processors, it is was slightly lower for block size 512. One reason for 2 processors leading to better efficiency than more processors is that the used computer contains only 4 processors, and by using 2 processors for the experiments the other two are still free for other tasks that need to be executed. Using 4 processors for the experiment, the 4 processors are not fully dedicated to the experiments. We can also observe that large block sizes combined with more processors (for example, block size 512 with 4 and 3 processors, block size 256 with 3 processors) result in very unstable efficiency, going as low as 0.6.

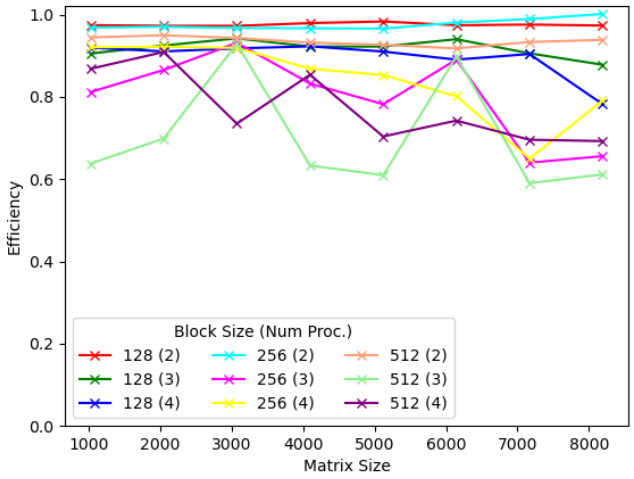

Figure 10: Efficiency of OMP matrix multiplication algorithm for different block sizes and number of processors.

Figure 11 contains the iso-granularity curve of the OMP algorithm for the different block sizes. For one processor, a matrix size of 2048 was used, and the matrix size was increased by 2048 for each additional processor. For all block sizes the performance grows linearly in relation to the number of processors and matrix sizes when the number of processors is between 1 and 3, and grows less (for block size 512 decreases) when the fourth processor is added. A reason for this could be that by using 4 processors in a computer where there are only 4 processors, the processors are being shared with other tasks of the system.

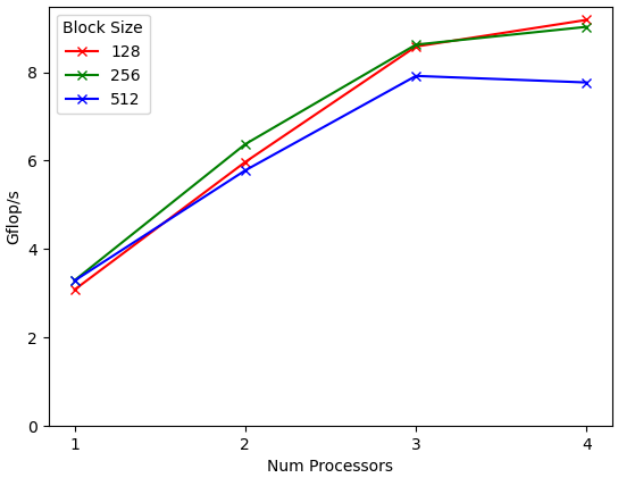

Figure 11: Iso-granularity curve of the OMP algorithm for different block sizes. For one processor, a matrix of size 2048 is used. The matrix size is increased by 2048 for each processor added.

Figures 12 and 13 represent the time and performance of the two CUDA algorithms, in CMY the blocked algorithm without local memory copy (B) and in RGB the algorithm with local memory copy (B_{mem}). We are able to see that algorithm B outperforms B_{mem} except for block size of 8. Every experiment was able to maintain the performance with the increase of the matrix sizes except for B for block sizes of 8. These results go against the ones presented in class for the same problem, we attribute this fact to the more recent and powerful GPU used in our experiments which mitigates the advantages of the local memory transfer. Note that the B experiments with 32 block sizes is able to consistently achieve a performance that's near the theoretical FP64 performance for the GPU (~ 136.7).

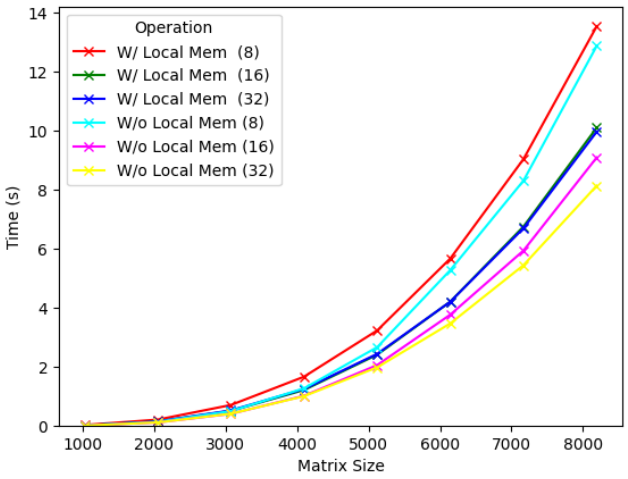

Figure 12: Time in seconds of the CUDA matrix multiplication algorithms for different block sizes. Without local memory in RGB, with local memory in CMY.

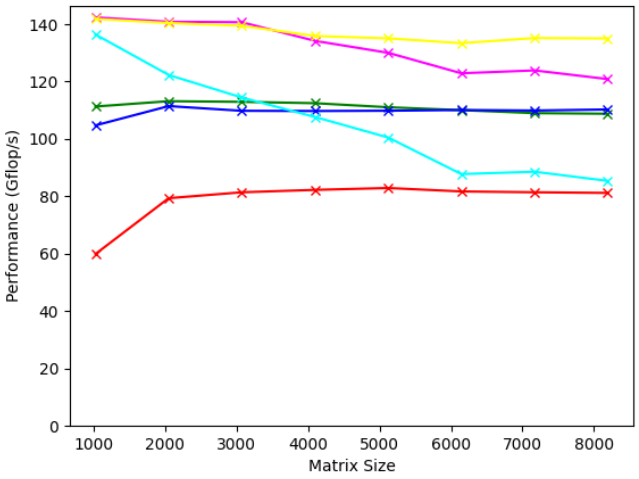

Figure 13: Performance in Gflop/s of the CUDA matrix multiplication algorithms for different block sizes. See legend in figure 12

Figures 14 and 15 represent the time and performance of the three Sycl algorithms in the CPU, in RGB the naive algorithm (N), in CMY the blocked algorithm without local memory copy (B), and the remaining colors the algorithm with local memory copy (B_{mem}). First, we are able to conclude that the naive algorithm, across all block sizes and matrix sizes, has a worse performance than the remaining ones (in the plot we can see that every block size achieves a very similar performance). Regarding the other two algorithms, the B algorithm generally achieves a better performance, except for the runs where a block size of 8 is used. Moreover, the runs with algorithm B for block sizes of 32 have an irregular performance, dropping the performance by about half ($\sim$ 10 Gflop/s) with large matrix sizes ($\geq$ 6144). The OMP implementation is generally better than the Sycl one, being able to consistently get 8 Gflop/s and above for any block size whereas only the algorithm B was able to achieve higher values. Despite this, the better performing of the Sycl implementation was better than the OpenMP one for the largest block size.

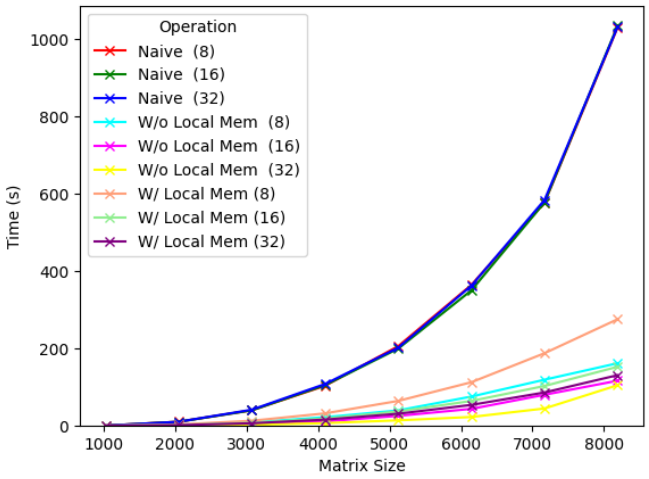

Figure 14: Time in seconds of the Sycl matrix multiplication algorithms for different block sizes in the CPU

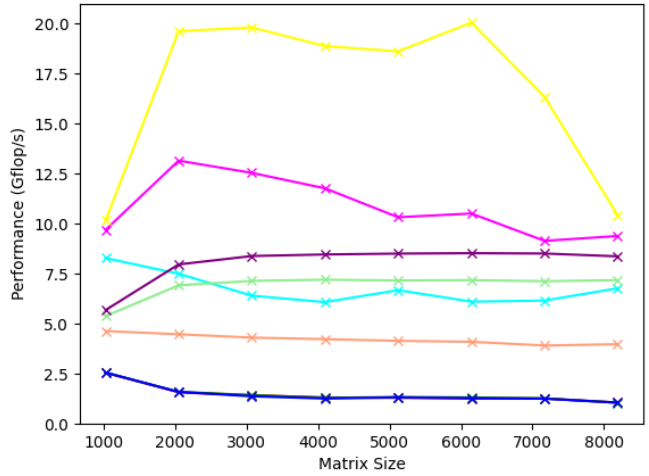

Figure 15: Performance in Gflop/s of the Sycl matrix multiplication algorithms for different block sizes in the CPU. See legend in figure 14. The values for the naive algorithm are all very similar.

Figures 16 and 17 represent the time and performance of the three Sycl algorithms, in RGB the naive algorithm (N), in CMY the blocked algorithm without local memory copy (B), and the remaining colors the algorithm with local memory copy (B_{mem}). We can see that B is the best performing algorithm, particularly for matrices larger than 3072. The three block sizes achieve similar results although the 16 and 32 ones perform slightly better. We are also able to perceive that the naive algorithm incurs in a drop in performance around the 2048-3072 mark, which we suppose is related to the maximum work group size

allowed by the GPU. Comparing with the CUDA results we are able to assess the quality of the Sycl implementation, we can see that the top result using Sycl was about 70 Gflop/s which is less than the worst result in the CUDA experiments about half of the best performing one. In addition, notice that Sycl is not able to produce a constant performance across all matrix sizes, which is generally the case with the CUDA results. This serves as a statement of the cost of the portability offered by Sycl versus programming directly for the chosen device (CUDA).

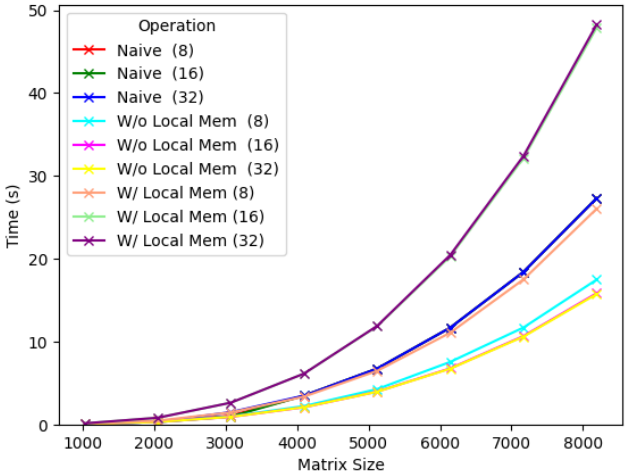

Figure 16: Time in seconds of the Sycl matrix multiplication algorithms for different block sizes in the GPU

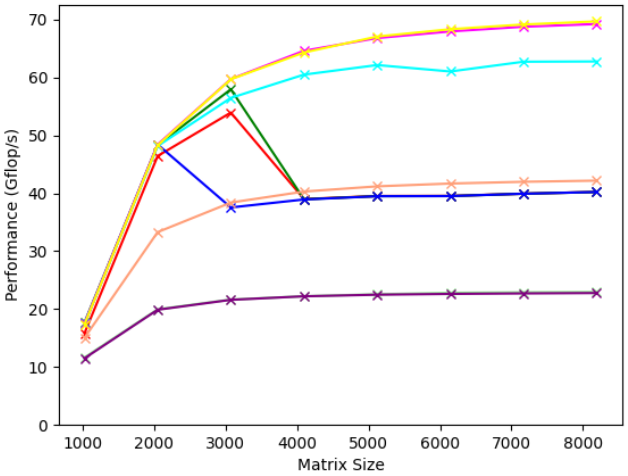

Figure 17: Performance in Gflop/s of the Sycl matrix multiplication algorithms for different block sizes in the GPU. See legend in figure 16. The values for the naive algorithm are all very similar.

5.2 LU decomposition

5.2.1 Description

In contrast to the matrix multiplication algorithm, this algorithm is not embarrassingly parallel, as the update on the elements of the matrix on each iteration depends on the updates done in the previous iterations.

To create a parallel version of LU decomposition we must analyse the dependencies between the several steps of the process in order to find the ones that can run concurrently. In this sense, the blocked algorithm is the prime candidate for parallelization.

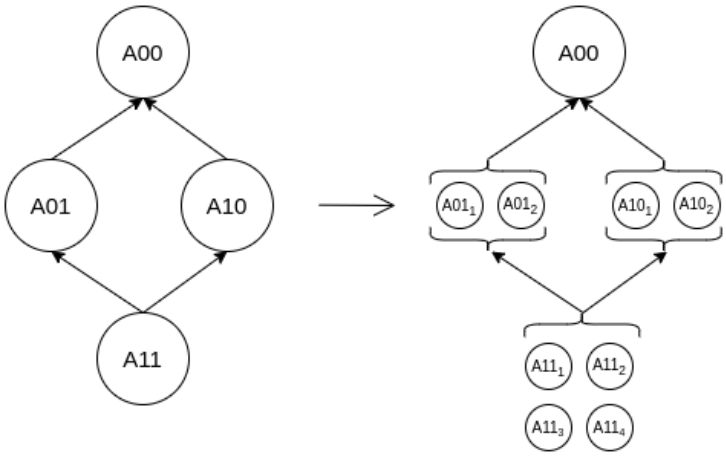

Figure 18: Dependency graph of the different stages of each iteration of the LU decomposition. On the right, the equivalent graph for the blocks algorithm. Note that $A00$ is dependant on $A11$ of the previous iteration.

As we can see in Figure 18, which represents one step of the decomposition (remember that the LU decomposition works iteratively along the diagonal), processing of the $A01$ and $A10$ blocks can be done in parallel, after the $A00$ is finished. In addition, the calculation of the different blocks of $A11$ which correspond to the matrix multiplication between $A01$ and $A10$ can also be done in parallel.

Note that processing $A00$ can only be done after processing $A11$ of the previous iteration (except for the first one).

What follows are implementations notes of a parallel LU decomposition algorithm in *OpenMP* and *Sycl*.

OpenMP We implemented a data parallel and function parallel version of the LU decomposition algorithm. Both attempt to accomplish the parallelism described in the previous section, in particular paralleling the update of $A01$ with the update of $A10$, and the update of the $A11$ matrix.

The **data parallel** version, seen in Source code 15, uses the `omp parallel` pragma to parallelize the `for` loop that iterates through the matrix's diagonal. Note that the `parallel for` directive isn't used because the several iterations are inter dependant. Decomposing $A00$ must be done only by one of the threads hence the use of the `single` pragma. Next we use a `for` directive to parallelize the factorization of $A10$'s blocks (lines 16-19). The same happens for the factorization of $A01$'s blocks (lines 23-26). Note that the `for` pragma of $A10$ is

accompanied by a `nowait` clause in order to allows its parallelization with processing o $A01$. The intent of making these two steps parallel also justifies the use of the `schedule(dynamic)` clause. Finally, the update of the $A11$ (lines 31-34) is analogous to a matrix multiplication, as such, it's parallelization with OMP follows the same scheme presented in section 5.1.1.

To obtain the **functional parallel** version one just needs to make small changes to the previous version. As is seen in Source code 16, in addition to the `parallel` pragma, a `single` directive is added because we want to ensure that the for loop is only ran by one thread which will be tasked with launching the tasks for the remaining threads. The `for` directives are replaced by a `taskgroup` (lines 19-30) which launches several several tasks that process the blocks of $A01$ and $A10$. Parallelizing these two steps is the reason for choosing not to use `taskloop` directives. Finaly, processing the $A11$ block follows the same scheme of the data parallel version but using a `taskloop`.

Sycl This implementation follows the same structure as the previous ones. Lines 12 to 20 handle the sequential factorization of block $A00$, which is why it is executed as a `single_task`. Lines 37 to 52 handle the factorization of the $A01$ block by using a `parallel_for` to divide the work into kernels that execute on its blocks. Similarly, lines 55 to 73 divide the work of processing the $A10$ block. Note that only line 74 contains a `wait_and_throw` instruction, as to parallelize the work of factorizing $A10$ and $A01$. Finally, lines 79 to 94 handle to update of sub matrix $A11$, which uses a kernel that subdivides the work per block of the destination matrix, analogous to the matrix multiplication algorithm.

5.2.2 Experiments

The parallel algorithm was run for matrix side sizes ranging from 1024 to 8192, with 1024 increments while measuring the time. Each experiment was ran 3 times, and we averaged the results. We ran the algorithm for OMP and for Sycl.

The OpenMP experiments used block sizes of 128, 256 and 512. We also ran the experiment set using 1 to 4 processors (threads) in order to be able to study speedup, efficiency and scalability.

We ran the Sycl experiments for the CPU and for the GPU using block sizes of 8, 16 and 32.

5.2.3 Results

Figure 19 contains the measured times of the OMP data parallel version of the LU decomposition algorithm. We can observe that the times are lower for lower block sizes. Figure 20 contains the performance in Gflop/s. Block size 128 yields the best performance, which plateaus at matrix size 2048 until the largest matrix size measured. Block size 256 has a lower performance than block size 128, but matches it for matrix size 4096, then decreasing until stabilizing between matrix sizes 6144 and 8192. Block size 512 the lowest performance, and also decreases, even though less than block size 256, around the 4096 block size and stabilizing between matrix sizes 6144 and 8192.

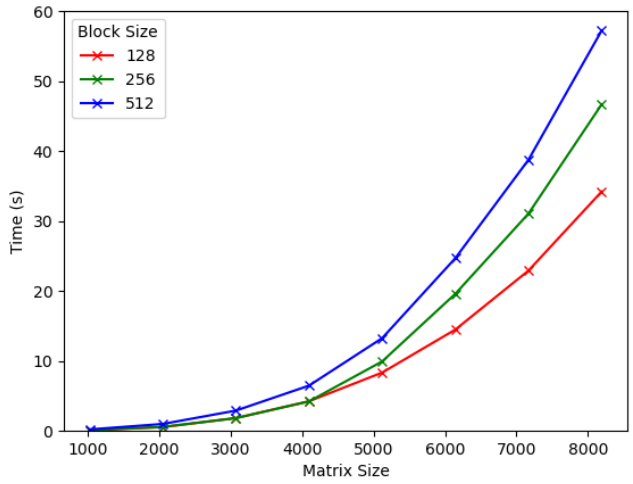

Figure 19: Time in seconds of the OMP data parallel version of the LU decomposition algorithm for different block and matrix sizes and using 4 processors.

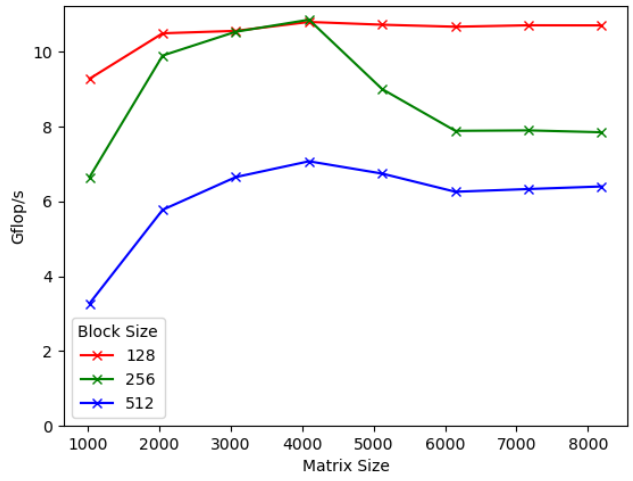

Figure 20: Performance of the OMP data parallel version of the LU decomposition algorithm for different block and matrix sizes and using 4 processors.

Figure 21 contains the relative speedup of the data parallel OMP LU decomposition algorithm. The best speedup is achieved using 128 block size and 4 processors. Unlike in the matrix multiplication algorithm, more processors does not mean better speedup. For instance, block size 512 with 4 processors achieves less speedup than 128 block size with 3 processors. Comparing the speedup for the different number of processors and same block size, the speedup is higher with more processors.

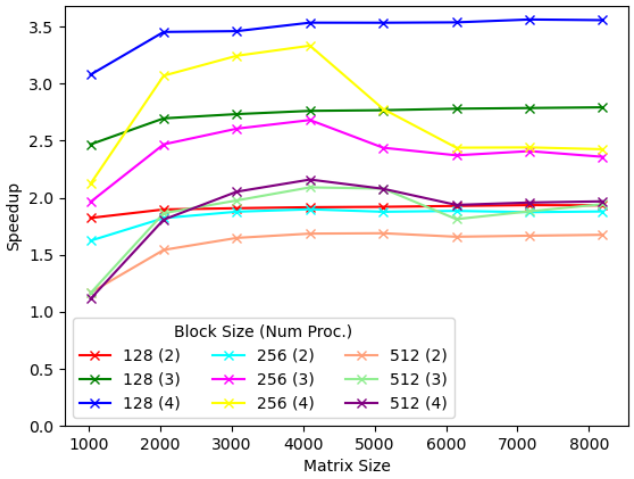

Figure 21: Speedup of the OMP data parallel version of the LU decomposition algorithm for different block sizes and number of processors.

Figure 22 contains the efficiency measure. Again, less processors does not mean better efficiency, without taking into account the block size. Comparing the plots for the same block size, the efficiency is better the lower the processor count. Comparing the plots for the same number of processors, the efficiency is better the lower the block size.

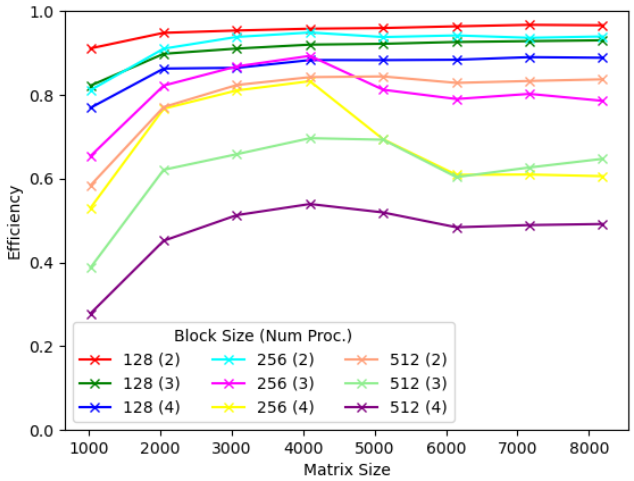

Figure 22: Efficiency the OMP data parallel version of the LU decomposition algorithm for different block sizes and number of processors.

Figure 23 contains the iso-granularity curve, using the same number of processors and block sizes as for the matrix multiplication iso-granularity curve. As in the measures presented above, the 128 block size yields the best results and is more scalable than the other block sizes, as the performance grows linearly with

the number of processors and matrix size. With block size 256 the measured Gflop/s are higher than with block size 128 for a number of processors lesser or equal than 2, but the increase in performance is lower when adding the third processor and the performance even decreases when adding the fourth. Block size 512 has the lowest performance of all tested block sizes.

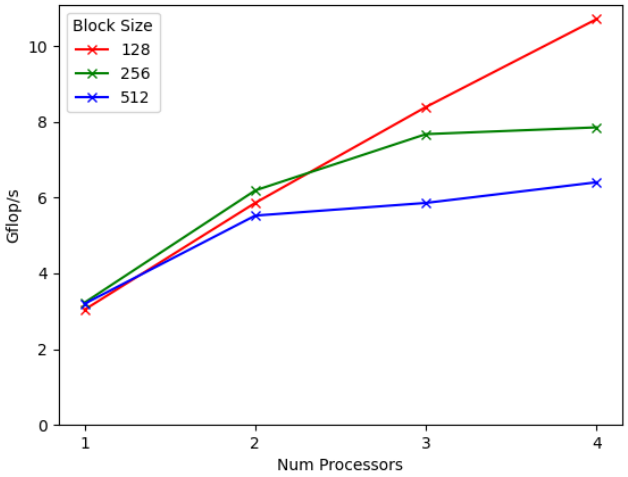

Figure 23: Iso-granularity curve of the OMP data parallel version of the LU decomposition algorithm for different block sizes. For one processor, a matrix of size 2048 is used. The matrix size is increased by 2048 for each processor added.

Figures 24, 25, 26, 27 and 28 represent the time, performance, speedup, efficiency and scalability (iso-granularity) measurements, respectively. The results were almost identical to the data-parallel version, which we attribute to the fact that for this particular problem, the differences between the two versions are almost none.

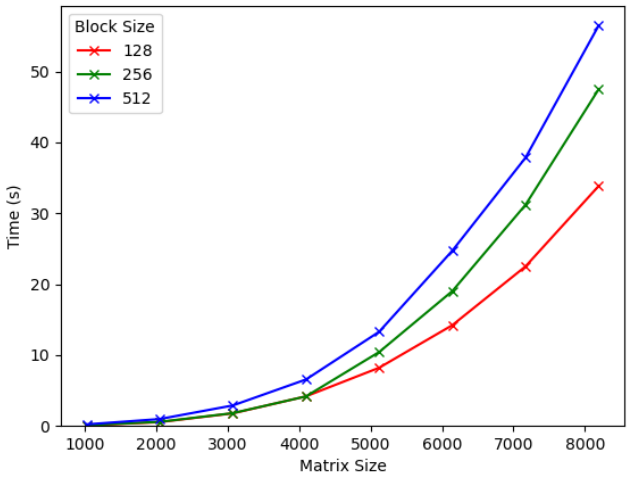

Figure 24: Time in seconds of the OMP functional parallel version of the LU decomposition algorithm for different block and matrix sizes and using 4 processors.

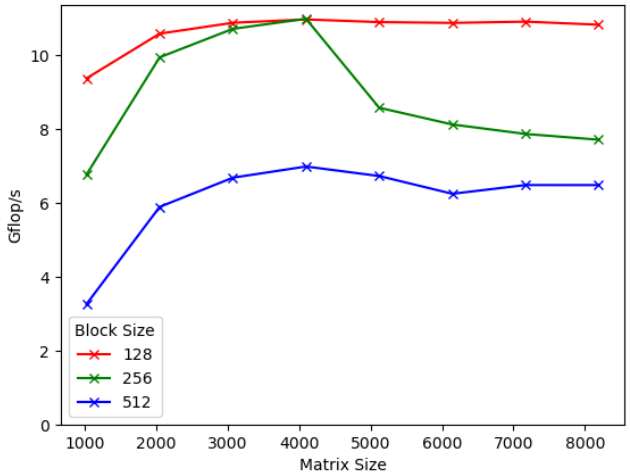

Figure 25: Performance of the OMP functional parallel version of the LU decomposition algorithm for different block and matrix sizes and using 4 processors.

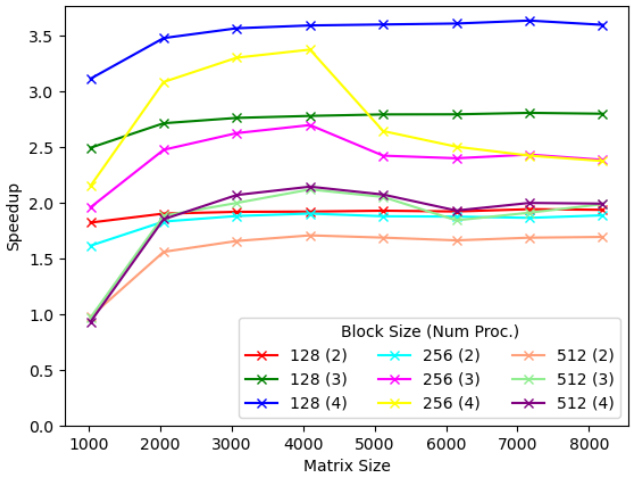

Figure 26: Speedup of the OMP functional parallel version of the LU decomposition algorithm for different block sizes and number of processors.

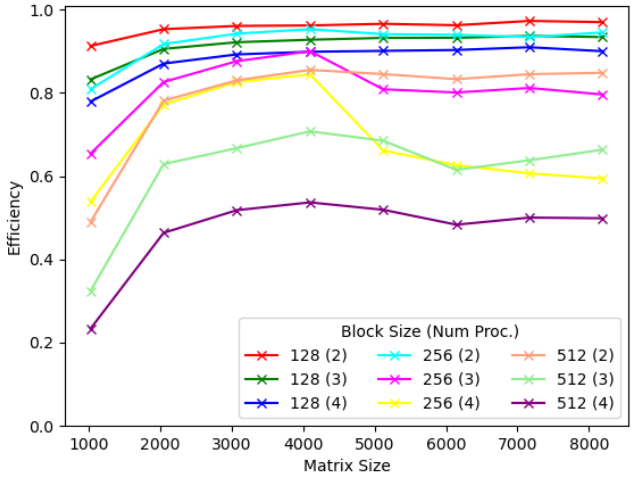

Figure 27: Efficiency the OMP functional parallel version of the LU decomposition algorithm for different block sizes and number of processors.

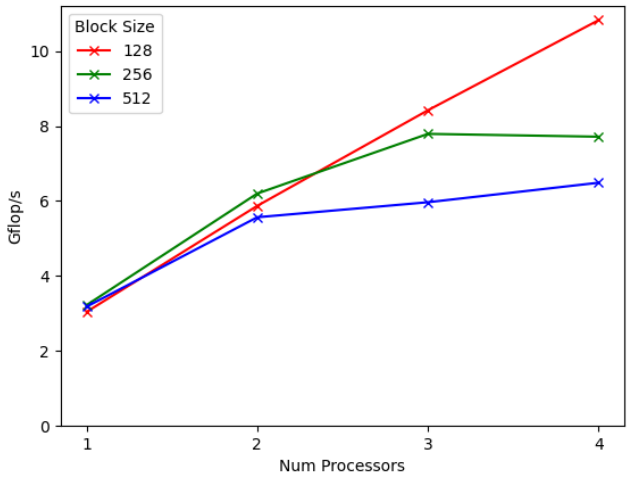

Figure 28: Iso-granularity curve of the OMP functional parallel version of the LU decomposition algorithm for different block sizes. For one processor, a matrix of size 2048 is used. The matrix size is increased by 2048 for each processor added.

Figures 29 and 30 show the results of running the Sycl implementation of the LU decomposition algorithm in the CPU. As we can see, the algorithm is able to maintain about a 7 Gflop/s. Moreover, we are able to see that there isn't much differentiation between the different block sizes used. The performance obtain by this implementation was lower than the average obtained in the OpenMP experiments which was around 10 Gflop/s.

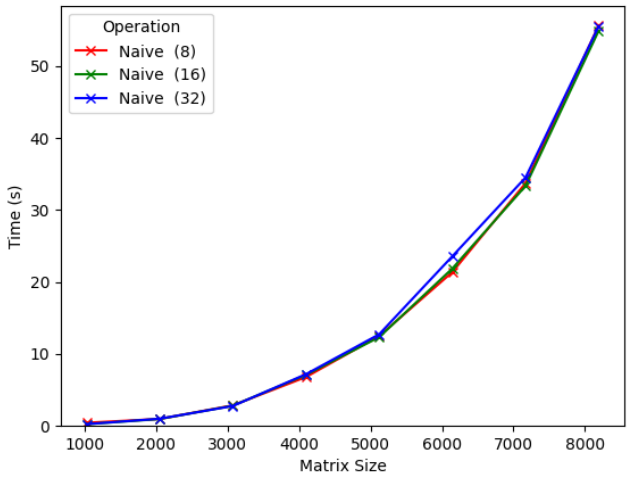

Figure 29: Time in seconds of the Sycl LU decomposition for different block sizes in the CPU.

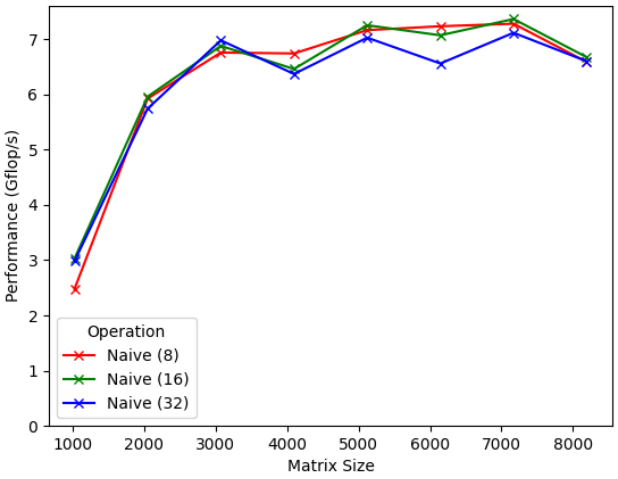

Figure 30: Performance in Gflop/s of the Sycl LU decomposition for different block sizes in the CPU.

Figures 29 and 30 show the results of running the Sycl implementation of the LU decomposition algorithm in the GPU. As expected, using the GPU produces much better performance than using the CPU, however, here we can perceive a greater difference between different block sizes as the runs with block sizes 8 consistently produce a better performance, whereas the runs with block sizes of 32 produce a far worse performance. In addition, contrary to the CPU versions, and similar to what we obtained in the matrix multiplication experiments, the GPU doesn't produce a steady performance. We suppose that this might be equivalent to the rise in performance that we see in CPU experiments, but , due to the higher performance, the stabilization phase comes with larger matrix sizes. However, due to limitations of our setup, in particular the amount of RAM memory, we weren't able to test it any further.

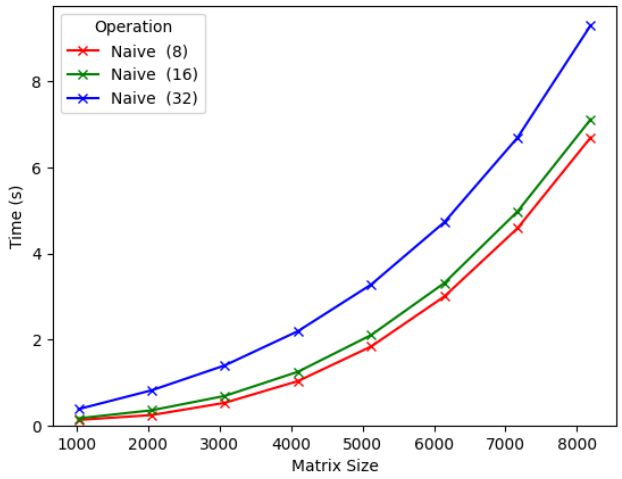

Figure 31: Time in seconds of the Sycl LU decomposition for different block sizes in the GPU.

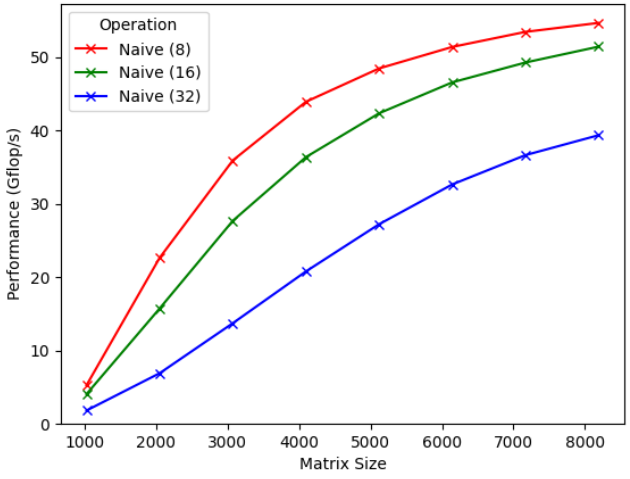

Figure 32: Performance in Gflop/s of the Sycl LU decomposition for different block sizes in the GPU.

6 Conclusion

In this work we implemented and analysed different parallel versions of the matrix multiplication algorithm and LU decomposition. We were able to successfully analyse existing sequential algorithms in order to identify possibles sources of parallelism. In addition, we were also able to demonstrate the use different implementations that allow the implementation of parallel algorithms. These implementations differ on the level of abstraction offered and portability

and we were able to demonstrate how these factors affect the performance of the algorithms. The experiments also allowed us to confirm that the abstraction level, a knowledge of the underlying architecture and features of the target device is paramount to ensure good performance, this knowledge may materialize itself in the creation of different functions that take into account different characteristics of different platforms.

In the future we would like to further explore the algorithms, in particular in the GPU experiments, since we would like to study their behaviour for larger matrices, as we argue that our experiments were bottlenecked by the used system (in particular its RAM).

A Source code

A.1 Naive matrix multiplication

```
void optimCycle(double* op1Matrix, double* op2Matrix, double* resMatrix, int
↪  matrixSize) {
  int i, j, k, rowOffsetI, rowOffsetK;

  for (i = 0; i < matrixSize; i++) {
    rowOffsetI = i * matrixSize;
    for (k = 0; k < matrixSize; k++) {
      rowOffsetK = k * matrixSize;
      for (j = 0; j < matrixSize; j++) {
        resMatrix[rowOffsetI + j] +=
          op1Matrix[rowOffsetI + k] * op2Matrix[rowOffsetK + j];
      }
    }
  }
}
```

Source code 2: Naive matrix multiplication in C

A.2 Block-based multiplication

```
double blockOptimCycle(double* op1Matrix, double* op2Matrix, double* resMatrix,
↪  int matrixSize, int blockSize) {

  int jj, kk, i, j, k, rowOffsetI, rowOffsetK;

  for (jj = 0; jj < matrixSize; jj = jj + blockSize)
    for (kk = 0; kk < matrixSize; kk = kk + blockSize)
      for (i = 0; i < matrixSize; i = i + 1){
        rowOffsetI = i * matrixSize;
        for (k = kk; k < kk + blockSize; k = k + 1) {
          rowOffsetK = k * matrixSize;
          for (j = jj; j < jj + blockSize; j = j + 1)
            resMatrix[rowOffsetI + j] += op1Matrix[rowOffsetI + k] *
            ↪  op2Matrix[rowOffsetK + j];
        };
      }
}
```

Source code 3: Matrix multiplication in C (Block-based)

A.3 Naive LU decomposition

```
void luSequential(double *matrix, size_t nRows, size_t nCols,
                  size_t matrixSize) {
  for (size_t k = 0; k < nCols && matrix[k * matrixSize + k] != 0; k++) {
    size_t offsetK = k * matrixSize;
```

```
5
6       for (size_t i = k + 1; i < nRows; i++) {
7         matrix[i * matrixSize + k] /= matrix[offsetK + k];
8       }
9
10      for (size_t i = k + 1; i < nRows; i++) {
11        size_t offsetI = i * matrixSize;
12        for (size_t j = k + 1; j < nCols; j++) {
13          matrix[offsetI + j] -= matrix[offsetI + k] * matrix[offsetK + j];
14        }
15      }
16    }
17  }
```

Source code 4: Naive LU decomposition in C

A.4 Blocks LU decomposition

```
1   void luBlocks(double *matrix, size_t size, size_t blockSize) {
2     double *diagonalBlock = matrix;
3
4     for (size_t currentDiagonalIdx = 0; currentDiagonalIdx < size;
5          currentDiagonalIdx += blockSize) {
6       luSequential(diagonalBlock, blockSize, blockSize, size);
7
8       if (size - currentDiagonalIdx <= blockSize) break;
9
10      for (size_t ii = 0; ii < size - currentDiagonalIdx - blockSize;
11           ii += blockSize) {
12        factorizeA10(diagonalBlock, size, blockSize, ii);
13      }
14
15      factorizeA01(diagonalBlock, size, blockSize, blockSize,
16                   size - currentDiagonalIdx);
17
18      size_t sizeA11 = size - (blockSize + currentDiagonalIdx);
19      for (size_t ii = 0; ii < sizeA11; ii += blockSize)
20        for (size_t jj = 0; jj < sizeA11; jj += blockSize)
21          factorizeA11(diagonalBlock, size, blockSize, ii, jj);
22
23      diagonalBlock += blockSize * size + blockSize;
24    }
25  }
```

Source code 5: Blocked LU decomposition in C

```
1   void factorizeA10(double *diagonalBlock, size_t size, size_t blockSize,
2                     size_t ii) {
3     double *a10 = diagonalBlock + size * blockSize;
4
5     for (size_t k = 0; k < blockSize; k++) {
```

```
    size_t offsetK = k * size;
    for (size_t i = ii; i < ii + blockSize; i++) {
      a10[i * size + k] /= diagonalBlock[offsetK + k];
    }

    for (size_t i = ii; i < ii + blockSize; i++) {
      size_t offsetI = i * size;
      for (size_t j = k + 1; j < blockSize; j++) {
        a10[offsetI + j] -= a10[offsetI + k] * diagonalBlock[offsetK + j];
      }
    }
  }
}
```

Source code 6: Procedure to factorize $A10$

```
void factorizeA01(double *diagonalBlock, size_t size, size_t blockSize,
                  size_t jj, size_t end) {
  for (size_t k = 0; k < blockSize; k++) {
    size_t offsetK = k * size;
    for (size_t i = k + 1; i < blockSize; i++) {
      size_t rowOffsetI = i * size;
      for (size_t j = jj; j < end; j++) {
        diagonalBlock[rowOffsetI + j] -=
            diagonalBlock[rowOffsetI + k] * diagonalBlock[offsetK + j];
      }
    }
  }
}
```

Source code 7: Procedure to factorize $A01$

```
void factorizeA11(double *diagonalBlock, size_t size, size_t blockSize,
                  size_t ii, size_t jj) {
  double *factorizedColumns = diagonalBlock + blockSize * size;
  double *factorizedRows = diagonalBlock + blockSize;
  double *subMatrix = diagonalBlock + blockSize * size + blockSize;

  for (size_t i = ii; i < ii + blockSize; i++) {
    size_t rowOffsetI = i * size;
    for (size_t k = 0; k < blockSize; k++) {
      size_t rowOffsetK = k * size;
      for (size_t j = jj; j < jj + blockSize; j++) {
        subMatrix[rowOffsetI + j] -=
            factorizedColumns[rowOffsetI + k] * factorizedRows[rowOffsetK + j];
      }
    }
  }
}
```

Source code 8: Procedure to factorize $A11$

A.5 OpenMP Matrix Multiplication

```
void matMulParallelCollapse(double* op1Matrix, double* op2Matrix, double*
↪  resMatrix,
                  int matrixSize, int blockSize) {
  int ii, jj, kk, i, j, k, rowOffsetI, rowOffsetK;
  #pragma omp parallel shared(op1Matrix, op2Matrix, resMatrix) private(ii, jj, kk,
  ↪  i, j, k) num_threads(NUM_THREADS)
  {
    #pragma omp for collapse(2)
    for (ii = 0; ii < matrixSize; ii += blockSize)
      for (jj = 0; jj < matrixSize; jj += blockSize)
        for (kk = 0; kk < matrixSize; kk += blockSize)
          for (i = ii; i < ii + blockSize; i++){
            rowOffsetI = i * matrixSize;
            for (k = kk; k < kk + blockSize; k++){
              rowOffsetK = k * matrixSize;
              for (j = jj; j < jj + blockSize; j++)
                resMatrix[rowOffsetI + j] +=
                    op1Matrix[rowOffsetI + k] *
                    op2Matrix[rowOffsetK + j];
            }
          }
  }

}
```

Source code 9: OpenMP matrix mutiplication

A.6 CUDA blocked matrix multiplication

```
    __global__ void MatrixMulKernelBlock(double* Md, double* Nd, double* Pd, int
    ↪  Width, int blockSize)
    {
      // Calculate the row index of the Pd element and M
      int Row = blockIdx.y*blockSize + threadIdx.y;
      // Calculate the column idenx of Pd and N
      int Col = blockIdx.x*blockSize + threadIdx.x;
      double Pvalue = 0;
      // each thread computes one element of the block sub-matrix
      for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];
      Pd[Row * Width + Col] = Pvalue;
    }
```

Source code 10: CUDA blocked matrix multiplication kernel

A.7 CUDA blocked matrix multiplication with local memory copy

```
__global__ void MatrixMulKernelBlockLocalMem(double *Md, double *Nd, double *Pd, int width, int blockSize) {
  __shared__ double* Mds;
  __shared__ double* Nds;
  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  // Identify the row and column of the Pd element to work on
  int Row = by * blockSize + ty;
  int Col = bx * blockSize + tx;
  double Pvalue = 0;

  if (tx == 0 && ty == 0) {
    Mds = (double*) malloc(blockSize * blockSize * sizeof(double));
    Nds = (double*) malloc(blockSize * blockSize * sizeof(double));
  }
  __syncthreads();
  // Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < width / blockSize; ++m) {
    // Coolaborative loading of Md and Nd tiles into shared memory
    Mds[ty*blockSize + tx] = Md[Row*width + (m*blockSize + tx)];
    Nds[ty*blockSize + tx] = Nd[Col + (m*blockSize + ty)*width];
    __syncthreads();
    for (int k = 0; k < blockSize; ++k) Pvalue +=  Mds[ty*blockSize + k] * Nds[k*blockSize + tx];
    __syncthreads();
  }
  __syncthreads();
  // Only one thread may free the memory!
  if (tx == 0 && ty == 0) {
    free(Mds);
    free(Nds);
  }
  Pd[Row*width+Col] = Pvalue;
}
```

Source code 11: CUDA blocked matrix multiplication kernel with copy to local memory step

A.8 SYCL naive matrix multiplication

```
void matmulNaive(double* MA, double* MB, double* MC, size_t matSize,
                 device dev) {
  queue Q(dev);

  {
    range<2> dimensions(matSize, matSize);
    const property_list props = {property::buffer::use_host_ptr()};
```

```
    buffer<double, 2> bA(MA, dimensions, props);
    buffer<double, 2> bB(MB, dimensions, props);
    buffer<double, 2> bC(MC, dimensions, props);

    Q.submit([&](handler& h) {
      auto a = bA.template get_access<access::mode::read>(h);
      auto b = bB.template get_access<access::mode::read>(h);
      auto c = bC.template get_access<access::mode::write>(h);

      h.parallel_for<matmul_kernel_naive>(range<2>{matSize, matSize},
                                          [=](id<2> idx) {
                                            int j = idx[0];
                                            int i = idx[1];
                                            for (int k = 0; k < matSize; ++k) {
                                              c[j][i] += a[j][k] * b[k][i];
                                            }
                                          });
    });
  }

  Q.wait_and_throw();
}
```

Source code 12: SYCL naive matrix multiplication implementation

A.9 SYCL blocked matrix multiplication

```
void matmulBlocks(double* MA, double* MB, double* MC, size_t matSize, size_t
↪  blockSize,
                  device dev) {
  queue Q(dev);

  {
    range<1> dimensions(matSize * matSize);
    const property_list props = {};
    buffer<T> bA(MA, dimensions, props);
    buffer<T> bB(MB, dimensions, props);
    buffer<T> bC(MC, dimensions, props);

    Q.submit([&](handler& h) {
      auto pA = bA.template get_access<access::mode::read>(h);
      auto pB = bB.template get_access<access::mode::read>(h);
      auto pC = bC.template get_access<access::mode::write>(h);

      h.parallel_for<matmul_kernel_blocks>(
          nd_range<2>{range<2>(matSize, matSize),
                      range<2>(blockSize, blockSize)},
          [=](nd_item<2> item) {
            int blockX = item.get_group(1);
            int blockY = item.get_group(0);

            int localX = item.get_local_id(1);
            int localY = item.get_local_id(0);
```

```
26
27              int a_start = matSize * blockSize * blockY;
28              int a_end = a_start + matSize - 1;
29              int b_start = blockSize * blockX;
30
31              double tmp = 0.0f;
32              for (int a = a_start, b = b_start; a <= a_end;
33                   a += blockSize, b += (blockSize * matSize)) {
34                for (int k = 0; k < blockSize; k++) {
35                  tmp +=
36                      pA[a + matSize * localY + k] * pB[b + matSize * localY + k];
37                }
38              }
39              auto elemIndex =
40                  item.get_global_id(0) * item.get_global_range()[1] +
41                  item.get_global_id(1);
42
43              pC[elemIndex] = tmp;
44            });
45      });
46    }
47    Q.wait_and_throw();
48  }
```

Source code 13: SYCL blocked matrix multiplication implementation

A.10 SYCL blocked matrix multiplication with local memory copy

```
1   void matmulBlocksLocalMem(double* MA, double* MB, double* MC, size_t matSize,
    ↪  size_t blockSize, device dev) {
2     queue Q(dev);
3
4     {
5       range<1> dimensions(matSize * matSize);
6       const property_list props = {property::buffer::use_host_ptr()};
7       buffer<T> bA(MA, dimensions, props);
8       buffer<T> bB(MB, dimensions, props);
9       buffer<T> bC(MC, dimensions, props);
10
11      Q.submit([&](handler& h) {
12        auto pA = bA.template get_access<access::mode::read>(h);
13        auto pB = bB.template get_access<access::mode::read>(h);
14        auto pC = bC.template get_access<access::mode::write>(h);
15        auto localRange = range<1>(blockSize * blockSize);
16
17        accessor<double, 1, access::mode::read_write, access::target::local> pBA(
18            localRange, h);
19        accessor<double, 1, access::mode::read_write, access::target::local> pBB(
20            localRange, h);
21
22        h.parallel_for<matmul_kernel_local_mem>(
```

```
23            nd_range<2>{range<2>(matSize, matSize),
24                        range<2>(blockSize, blockSize)},
25            [=](nd_item<2> item) {
26              int blockX = item.get_group(1);
27              int blockY = item.get_group(0);
28              int localX = item.get_local_id(1);
29              int localY = item.get_local_id(0);
30
31              int Row = blockY * blockSize + localY;
32              int Col = blockX * blockSize + localX;
33
34              // Result for the current C(i,j) element
35              double tmp = 0.0f;
36
37              for (int m = 0; m < matSize / blockSize; ++m) {
38                // Coolaborative loading of blocks into shared memory
39                pBA[localY * blockSize + localX] =
40                    pA[Row * matSize + (m * blockSize + localX)];
41                pBB[localY * blockSize + localX] =
42                    pB[Col + (m * blockSize + localY) * matSize];
43
44                item.barrier(access::fence_space::local_space);
45
46                for (int k = 0; k < blockSize; k++) {
47                  tmp +=
48                      pBA[localY * blockSize + k] * pBB[localX * blockSize + k];
49                }
50                item.barrier(access::fence_space::local_space);
51              }
52
53              auto elemIndex =
54                  item.get_global_id(0) * item.get_global_range()[1] +
55                  item.get_global_id(1);
56
57              pC[elemIndex] = tmp;
58            });
59      });
60    }
61
62    Q.wait_and_throw();
63  }
```

Source code 14: SYCL blocked matrix multiplication with copy to local memory implementation

A.11 OpenMP Data Parallel LU Decomposition

```
1   void luDataParallel(double *matrix, size_t size, size_t blockSize) {
2   #pragma omp parallel num_threads(NUM_THREADS)
3     {
4       double *diagonalBlock = matrix;
5
6       for (size_t currentDiagonalIdx = 0; currentDiagonalIdx < size;
```

```
7           currentDiagonalIdx += blockSize) {
8       // Do LU factorization of block A00
9   #pragma omp single
10      { luSequential(diagonalBlock, blockSize, blockSize, size); }
11
12      if (size - currentDiagonalIdx <= blockSize) break;
13
14        // Do LU factorization for block A10
15  #pragma omp for nowait
16      for (size_t ii = 0; ii < size - currentDiagonalIdx - blockSize;
17           ii += blockSize) {
18        factorizeA10(diagonalBlock, size, blockSize, ii);
19      }
20
21      // Do LU factorization for block A01
22  #pragma omp for schedule(dynamic)
23      for (size_t jj = blockSize; jj < size - currentDiagonalIdx;
24           jj += blockSize) {
25        factorizeA01(diagonalBlock, size, blockSize, jj, jj + blockSize);
26      }
27
28      size_t sizeA11 = size - (blockSize + currentDiagonalIdx);
29
30      // Update A11
31  #pragma omp for collapse(2)
32      for (size_t ii = 0; ii < sizeA11; ii += blockSize)
33        for (size_t jj = 0; jj < sizeA11; jj += blockSize)
34          factorizeA11(diagonalBlock, size, blockSize, ii, jj);
35
36      diagonalBlock += blockSize * size + blockSize;
37    }
38  }
39  }
```

Source code 15: OpenMP blocked LU decomposition using data parallelism

A.12 OpenMP Functional Parallel LU Decomposition

```
1   void luFuncParallel(double *matrix, size_t size, size_t blockSize) {
2   #pragma omp parallel num_threads(NUM_THREADS)
3   #pragma omp single
4     {
5       double *diagonalBlock = matrix;
6
7       for (size_t currentDiagonalIdx = 0; currentDiagonalIdx < size;
8            currentDiagonalIdx += blockSize) {
9         // Do LU factorization of block A00
10        { luSequential(diagonalBlock, blockSize, blockSize, size); }
11
12        if (size - currentDiagonalIdx <= blockSize) break;
13
14          // Do LU factorization for block A10
15  #pragma omp taskgroup
```

```
16        {
17          for (size_t ii = 0; ii < size - currentDiagonalIdx - blockSize;
18               ii += blockSize) {
19  #pragma omp task firstprivate(ii) shared(matrix, diagonalBlock, blockSize, size)
20            factorizeA10(diagonalBlock, size, blockSize, ii);
21          }
22
23          for (size_t jj = blockSize; jj < size - currentDiagonalIdx;
24               jj += blockSize) {
25  #pragma omp task firstprivate(jj) \
26      shared(matrix, blockSize, size, currentDiagonalIdx)
27
28            factorizeA01(diagonalBlock, size, blockSize, jj, jj + blockSize);
29          }
30        }
31
32        size_t sizeA11 = size - (blockSize + currentDiagonalIdx);
33
34        // Update A11
35  #pragma omp taskloop collapse(2)
36        for (size_t ii = 0; ii < sizeA11; ii += blockSize)
37          for (size_t jj = 0; jj < sizeA11; jj += blockSize)
38            factorizeA11(diagonalBlock, size, blockSize, ii, jj);
39
40        diagonalBlock += blockSize * size + blockSize;
41      }
42    }
43  }
```

Source code 16: OpenMP blocked LU decomposition using functional parallelism

A.13 Sycl blocked LU decomposition

```
1   bool luFactorization(double* MA, size_t matSize, size_t blockSize, device dev) {
2     queue Q(dev);
3     double* diagonalBlock = MA;
4     range<2> dimensions((matSize), (matSize));
5     const property_list props = {};
6     buffer<double, 2> matrix(MA, dimensions, props);
7
8     for (size_t currentDiagonalIdx = 0; currentDiagonalIdx < matSize;
9          currentDiagonalIdx += blockSize) {
10
11      // A00 factorization
12      Q.submit([&](handler& h) {
13        auto matrixAcc = matrix.template get_access<access::mode::read_write>(h);
14
15        h.single_task<lu_kernel_0>([=]() {
16          for (size_t k = currentDiagonalIdx;
17               k < currentDiagonalIdx + blockSize && matrixAcc[k][k] != 0; k++) {
18            for (size_t i = k + 1; i < currentDiagonalIdx + blockSize; i++) {
19              matrixAcc[i][k] /= matrixAcc[k][k];
```

```
      }

      for (size_t i = k + 1; i < currentDiagonalIdx + blockSize; i++) {
        for (size_t j = k + 1; j < currentDiagonalIdx + blockSize; j++) {
          matrixAcc[i][j] -= matrixAcc[i][k] * matrixAcc[k][j];
        }
      }
    }
  });
});
Q.wait_and_throw();

if (matSize - currentDiagonalIdx <= blockSize) break;

size_t nBlocks = (matSize - currentDiagonalIdx - blockSize) / blockSize;

// A01 factorization
Q.submit([&](handler& h) {
  auto matrixAcc = matrix.template get_access<access::mode::read_write>(h);

  h.parallel_for<lu_kernel_3>(range<1>{range<1>(nBlocks)}, [=](id<1> id) {
    int jj = id.get(0) * blockSize + blockSize + currentDiagonalIdx;

    for (size_t k = currentDiagonalIdx; k < currentDiagonalIdx + blockSize;
         k++) {
      for (size_t i = k + 1; i < currentDiagonalIdx + blockSize; i++) {
        for (size_t j = jj; j < jj + blockSize; j++) {
          matrixAcc[i][j] -= matrixAcc[i][k] * matrixAcc[k][j];
        }
      }
    }
  });
});

// A10 factorization
Q.submit([&](handler& h) {
  auto matrixAcc = matrix.template get_access<access::mode::read_write>(h);

  h.parallel_for<lu_kernel_2>(range<1>{range<1>(nBlocks)}, [=](id<1> id) {
    int ii = id.get(0) * blockSize + blockSize + currentDiagonalIdx;

    for (size_t k = currentDiagonalIdx; k < currentDiagonalIdx + blockSize;
         k++) {
      for (size_t i = ii; i < ii + blockSize; i++) {
        matrixAcc[i][k] /= matrixAcc[k][k];
      }
      for (size_t i = ii; i < ii + blockSize; i++) {
        for (size_t j = k + 1; j < currentDiagonalIdx + blockSize; j++) {
          matrixAcc[i][j] -= matrixAcc[i][k] * matrixAcc[k][j];
        }
      }
    }
  });
});
Q.wait_and_throw();
```

```
75
76      size_t subMatrixSize = matSize - (blockSize + currentDiagonalIdx);
77
78      // A11 factorization
79      Q.submit([&](handler& h) {
80        auto matrixAcc = matrix.template get_access<access::mode::read_write>(h);
81
82        h.parallel_for<lu_kernel>(
83            range<2>{range<2>(subMatrixSize, subMatrixSize)}, [=](id<2> id) {
84              int j = id.get(1) + blockSize + currentDiagonalIdx;
85              int i = id.get(0) + blockSize + currentDiagonalIdx;
86              double tmp = 0.0;
87
88              for (int k = currentDiagonalIdx; k < currentDiagonalIdx + blockSize;
89                   k++) {
90                tmp += matrixAcc[i][k] * matrixAcc[k][j];
91              }
92              matrixAcc[i][j] -= tmp;
93            });
94      });
95      Q.wait_and_throw();
96
97      diagonalBlock += blockSize * matSize + blockSize;
98    }
99
100   return false;
101 }
```

Source code 17: Sycl blocked LU decomposition

B Nvidia GTX 1060 6GB specification sheet

B.1 Graphics processor

- GPU Name: GP106
- GPU Variant: GP106-400-A1
- Arhitecture: Pascal
- Foundry: TSMC
- Process Size: 16nm
- Transistors: 4,400 million
- Die Size: 200 mm^2

B.2 Clock Speeds

- Base Clock: 1506 MHz
- Boost Clock: 1709 MHz
- Memory Clock: 2002 MHz, 8Gbps effective

B.3 Memory

- Memory Size: 6GB
- Memory Type: GDDR5
- Memory Bus: 192 bit
- Bandwidth: 192.2 GB/s

B.4 Render Config

- Shading Units: 1280
- TMUs (Texture mapping units): 80
- ROPs (Render output units): 48
- Threads per block: 512
- SM Count (Streaming multiprocessors): 10
- L1 Cache: 48 KB (per SM)
- L2 Cache: 1536 KB

B.5 Theoretical Performance

- Pixel Rate: 82.03 GPixel/s
- Texture Rate: 136.7 GTexel/s
- FP16 (half) performance: 68.36 GFLOPS (1:64)
- FP32 (float) performance: 4.375 TFLOPS
- FP64 (double) performance: 136.7 GFLOPS (1:32)

C Intel Core i5 6500 specification sheet

- Architecture: Skylake
- # of Cores: 4
- # of Threads: 4
- Processor Base Frequency: 3.20 GHz
- Max Turbo Frequency: 3.60 GHz
- Cache: 6 MB Intel® Smart Cache
- L1 Data cache: 32KB per core
- L2 Unified cache: 256KB per core
- Bus Speed: 8 GT/s

D Code compilation/running instructions

D.1 OpenMP

To compile the OpenMP code simple use the `g++ -O2 <file_name> -fopenmp` command.

Both the matrix multiplication and the LU decomposition OMP programs receive the following arguments: matrix size, operation, number of runs and block size.

D.2 CUDA

To compile the CUDA version use the `nvcc <file_name>` command.

The CUDA program receives the following arguments: matrix size, operation, number of runs and block size.

D.3 Sycl

To compile the Sycl implementation we may use two possibilities. If the goal is to compile the code to the CPU or to a GPU that is supported by ComputeCpp then we just need to execute the `make <file_name_no_extension>` command on the folder that contains the Makefile (`src/sycl/matmul/` for example).

However, if we need to compile the code for a CUDA GPU we must have the DPC++ implementation of Sycl and we can use the `clang++ -fsycl -fsycl-targets = nvptx64-nvidia-cuda-sycldevice <file_name>` command.

The Sycl program receives the following arguments: matrix size, block size, operation, device (gpu, cpu or manual to interactively choose the device), number of runs and block size.