Optimization matrix multiplication and inversion

Academic year 2020/2021
Domenico Ragusa
Danilo Modica

1. Matrix multiplication

The matrix-matrix multiplication is an algebraic operation where, given two matrices, A whose dimensions are *m x n* and B whose dimensions are *n x p,* returns a matrix C with dimensions *m x p*.

This operation is widely used in numerous scientific applications such as systems of equations, image processing, pattern recognition or also in developing neural networks.

1.1. Analysis of the serial algorithm

Considers two matrices $A = \begin{pmatrix} 2 & 1 & 3 \\ -1 & 0 & -2 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 5 \\ -3 & -4 \\ 2 & 1 \end{pmatrix}$, in particular first row of A,

$R_{1A} = (2 \quad 1 \quad 3)$, and first column of B, $C_{1B} = \begin{pmatrix} 1 \\ -3 \\ 2 \end{pmatrix}$.

The algorithm consists of multiplying the first element of R_{1A} with the first of C_{1B} and so on for all the elements of the A rows, which are the same in number as the considered elements of the column of B. At this point it is possible to sum one another the products obtaining the element (1,1) of the product matrix C (also called result matrix).

$$C_{1,1} = 2 \times 1 + 1 \times (-3) + 3 \times 2 = 5$$

The same steps are repeated considering the same row of A for all the other columns of B obtaining all the elements of the first row of C. Repeating it with the other rows of A allows to obtain the other remaining elements, as figure 1 illustrates.

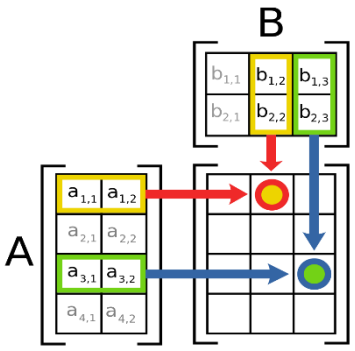

Figure 1. Graphical explanation of matrix-matrix multiplication

More in general, the element ij[th] of the matrix C is obtained with this summation:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

This can be implemented in C by using three nested loops with the number of iterations which depends on the matrix dimensions, as shown in Figure 2.1 where i is the index for rows, j for columns and k an auxiliary index. The complexity of the algorithm is approximately $\Theta(n^3)$.

```
for (i = 0; i < m; i++) {
        for (j = 0; j < p; j++) {
                for (k = 0; k < p; k++) {
                        m3[i][j] += m1[i][k] * m2[k][j];
                }
        }
}
```

Figure 2.1. Serial multiplication in C

The heaviest part of the algorithm is the computation done by considering one element of the product matrix at a time, it means one summation at a time which could be very long if large matrices are considered. It should be parallelized in a way that more elements can be calculated at the same time.
Problems of this algorithm are memory accesses and cache misses depending on whether the matrices are stored in row-major order or in column-major order. Using the above sequence for the three loops we exploit the row-major order to access elements in memory and so here we are correctly making use of the locality principle.

Considering these problems, we also tried another solution, to better exploit the locality principle and CPU caches, which in turn subdivides the matrices in blocks/tiles. It can be implemented in C using three additional for loops to manage each block/tile of the matrices, as you can see in Figure 2.2. In particular the first three nested loops iterate among tiles and the other three nested loops, inside the former, iterate among elements in each tile.

```
for(i=0; i<m; i+=BS){
        for(j=0; j<p; j+=BS){
                for(k=0; k<p; k+=BS){
                        for (ii = i; ii < i+BS; ii++) {
                                for (jj = j; jj < j+BS; jj++) {
                                        for (kk = k; kk < k+BS; kk++) {
                                                m3[ii][jj] += m1[ii][kk] * m2[kk][jj];
                                        }
                                }
                        }
                }
        }
}
```

Figure 2.2. Serial multiplication using tiles

We directly wrote the first serial code from scratch, using the simplest algorithms, while the second one was inspired by the laboratory activities.

1.2. A-priori study of available parallelism and parallel implementations

The algorithm can be divided into four main parts:

1. Initialization with memory allocation
2. Reading of two matrices
3. Multiplication
4. Writing the resulting matrix

Using the Gprof profiler on the serial codes, we noticed that the only part that takes more time is the one associated to the matrix multiplication itself (*matrixMul* function), as can be seen for example in Figure 3 (profiling was done with a *1024x1024* matrix using the first serial code). So, we have concentrated only on it leaving out the part related to reading and writing (I/O operations) from being time measured and/or parallelized. Practically same results were obtained for the tiled version of the algorithm.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.60      7.43     7.43        1     7.43     7.43  matrixMul
  0.00      7.43     0.00        2     0.00     0.00  readMatrix
  0.00      7.43     0.00        1     0.00     0.00  printMatrix
```

Figure 3. Gprof profile on the serial algorithm

To try to speedup this function, which is the one that takes more or less time depending on the matrix dimensions, we used three different approaches:

1) Parallelization per rows.
2) Parallelization per tiles.
3) Parallelization with concurrent elements (CUDA version).

To implement them we used both OpenMP, due to its simplicity in exploiting the multitude of CPU threads and for the mastery obtained from the lessons in the lab, and CUDA for the high number of cores in a graphic card that could drastically improve performance.

1.2.1. Parallelization per rows

In the first approach used, the simplest and immediate one, since each element in the result matrix can be calculated independently from the others, we assigned to each thread a row of that matrix (graphic description in Figure 4.2) through the *#pragma omp for* directive on the outermost loop. Threads share the three matrices and each one has its private loop variables, as shown in Figure 4.1, so they do not try to write in the same memory location and there is no need for synchronization techniques.

```
void matrixMul(double** m1, double** m2, double** m3, int m, int p) {
        int i, j, k;

        //Allocating and setting to zero rows of result matrix
        for(i = 0; i < m; i++) {
                m3[i] = (double*)malloc(p * sizeof(double));
                memset(m3[i], 0,  p * sizeof(double));
        }

        #pragma omp parallel shared(m1, m2, m3) private(i, j, k)
        {
                #pragma omp for schedule (dynamic)
                for (i = 0; i < m; i++) {
                        for (j = 0; j < p; j++) {
                                for (k = 0; k < p; k++) {
                                        m3[i][j] += m1[i][k] * m2[k][j];
                                }
                        }
                }
        }
}
```

Figure 4.1. Parallelization per rows implementation with OpenMP

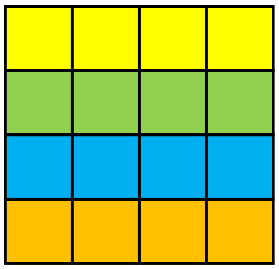

Figure 4.2. Result matrix subdivision per rows

We also tried to collapse together the for loops using the collapse() clause specifying two or more loops but this caused wrong results.

1.2.2. Parallelization per tiles

With the second approach we tried to speed up the second serial code, the one which makes the most of memory cache and locality principle. Cache misses are now minimum making the most of L1 and L2 CPU caches. In this parallelization we have division of the matrices into tiles and each thread computes a single tile (for example, as depicted in Figure 5.1). We chose a fixed 64x64 dimension for the tiles because it is a power of 2 and it allows to have at least one block per thread with matrices greater than 512x512; with smaller matrices the computation time is always small and acceptable moreover we tested different block sizes and this choice resulted to be the fastest one.

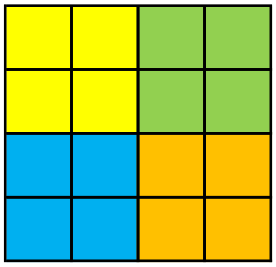

Figure 5.1. Result matrix subdivision per blocks

The last three nested loops which iterate inside tiles can be considered a "work package" and through the *#pragma omp task* we associated a tile to a single thread. Thanks to *depend* clauses (especially the *inout* one) we are sure that threads are synchronized and do not write in the same memory locations of the result matrix. The OpenMP implementation in Figure 5.2 was also inspired by the slides seen in laboratory sessions.

```
void matrixMul(int BS, double** m1, double** m2, double** m3, int m, int p) {
        int i, j, k;
        int ii, jj, kk;

        for(i=0; i<m; i+=BS){
                for(j=0; j<p; j+=BS){
                        for(k=0; k<p; k+=BS){
                                #pragma omp task depend(in: m1[i:BS][k:BS], m2[k:BS][j:BS]) depend(inout: m3[i:BS][j:BS])
                                for (ii = i; ii < i+BS; ii++) {
                                        for (jj = j; jj < j+BS; jj++) {
                                                for (kk = k; kk < k+BS; kk++) {
                                                        m3[ii][jj] += m1[ii][kk] * m2[kk][jj];
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Figure 5.2. Parallelization per tiles implementation with OpenMP

1.2.3. Parallelization with concurrent elements

In the third and last parallelization, based on the first serial code, we exploit the huge number of computation units of a GPU to speed up the algorithm. To each CUDA core we assigned the computation of an element (and so a summation) which occurs concurrently with the others (graphic explanation in Figure 6.1). This is not a big problem because a GPU has got hundreds of computation units.

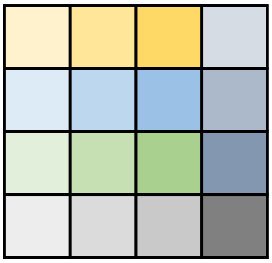

Figure 6.1. Result matrix subdivision per elements

In Figure 6.2 you can see the parallelization where the two outer loops where practically substituted by the GPU grid of threads. Here since it is not possible to store complex data structures (e.g. multi-dimensional arrays) in the GPU memory, we used vectors to linearize the matrices (one-

dimensional array) and each thread univocally access to one element depending on its coordinates in the threads grid.

The latter was a two-dimension grid divided in such a way each block was 32x32, a multiple of 32 which is the maximum dimension of a warp, so that you always have all threads working on.

```
__global__ void matrixMul(double *d_m1, double *d_m2, double *d_m3, int row1, int row2, int col1, int col2){
        int i = blockIdx.y*blockDim.y+threadIdx.y;
        int j = blockIdx.x*blockDim.x+threadIdx.x;

        double sum = 0;
        int k;

        //the two previous for cycle are substituted by the matrix of threads
        if ((i < row1) && (j < col2)){
                for(k = 0; k<col1; k++){
                        sum += d_m1[i*col1+k]*d_m2[k*col2+j];
                }
                d_m3[i*col2+j]=sum;
  }
}
```

Figure 6.2. CUDA parallel implementation

1.2.4. Theoretical speedup with Amdahl's law

In order to state the maximum speedup that we can achieve called also "a-priori theoretical speedup" we used the Amdahl's law:

$$Speedup_{overall} = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{speedup_{enhanced}}}$$

Since we divided the number of rows or tiles/blocks almost equally among threads, we thought that using for example 2 threads the times could be cut by a half, so for the law we used as $speedup_{enhanced}$ the number of threads, in particular the maximum which was 24.

Instead, the $fraction_{enhanced}$ was calculated depending on the times obtained by Gprof:

$$fraction_{enhanced} = \frac{time\ for\ matrixMul}{total\ time}$$

This one resulted to be very close to 1 because matrixMul could be thought the only parallelizable function and so it led us to obtain practically a $speedup_{overall}$ equal to the number of threads in use. So the maximum speedup achievable is 24 for OpenMP and 640 for CUDA.

1.3. Testing and debugging

To test the algorithms and evaluate the possible speedup of the parallel ones we needed matrices of different dimensions. Since using small matrices results in measuring no time, we decided to develop a Python script which through the NumPy module builds automatically random matrices given the desired dimensions. They were random in order to test all possible cases and not sparse because multiplications by zero would mean reducing computation times in some cases rather than others, and so we preferred to keep the variance of the times low.

The multiplication algorithm supports also non-squared matrices but for simplicity we decided to use squared matrices, with the following dimensions:

256x256	512x512	1024x1024	2048x2048	3072x3072

We did not use greater matrices because the multiplication required a lot of time to be done just with the serial code and this means too much credit wasted using the Google Cloud Platform which was used to scale the applications usage and the possibility of missing the deadline we set for the project.

To test the correctness of the serial algorithm we developed a Python script to calculate the same matrix-matrix multiplication, using a predefined module, and then using the *diff* Linux command we compared the outputs. The results of the parallel algorithm instead were compared with the output of the serial ones.

Tests were conducted keeping opened the system monitor to see that threads were busy and to track the memory consumption.

The virtual machine we created to run tests on GCP is an Ubuntu 20.04 machine, powered by a N1 series CPU platform, with 24vCPUs and 22GB of RAM.

1.4. Performance analysis

To make a performance analysis we used, as we said, the Google Cloud Platform by measuring speedups using a virtual machine exploiting 2, 4, 8, 16 and 24 vCPU. On GCP, for each matrix dimension and number of threads, we repeated three times the tests, using Linux shell scripts, and we computed the average execution times.

Instead, to analyze the performance for CUDA, we had not to use GCP, due to organization restrictions, so we used our laptop with a GTX 1050.

All the measurements of the speedup consider the wall clock time relative to the only multiplication function.

The results obtained are presented in the following paragraphs. For synthesis reasons we did not decide to report here all the data we collected, which are available at the following link on Microsoft 365: https://bit.ly/3hmlRjW

1.4.1 Parallelization by row results

OpenMP – **Parallelization by row**					
Matrix size	vCPU				
	2	**4**	**8**	**16**	**24**
256x256	1.88	3.67	6.59	8.80	8.95
512x512	1.99	3.93	7.56	11.30	12.27
1024x1024	2.01	4.03	7.90	11.80	12.31
2048x2048	2.00	4.03	8.05	12.36	12.65
3072x3072	2.00	4.02	8.03	12.29	12.68

Table 1. Speedups with parallelization by rows

In general, as shown in table 1, there has been an increase in speedup both by increasing the number of threads, as expected, due to the greater number of processing units working simultaneously, and by increasing the size of the matrices as you can better exploit the parallelism considering the greater complexity and the greater number of operations required.

Nevertheless, using 16 threads or more you can see that there is an asymptotic behavior (as depicted in the Figure 7), because of the limits in the parallelism scalability.

In all cases we got a speedup of about 12, half of that expected with the Amdahl's law, mainly due to the multitude of memory accesses to read/write matrices.

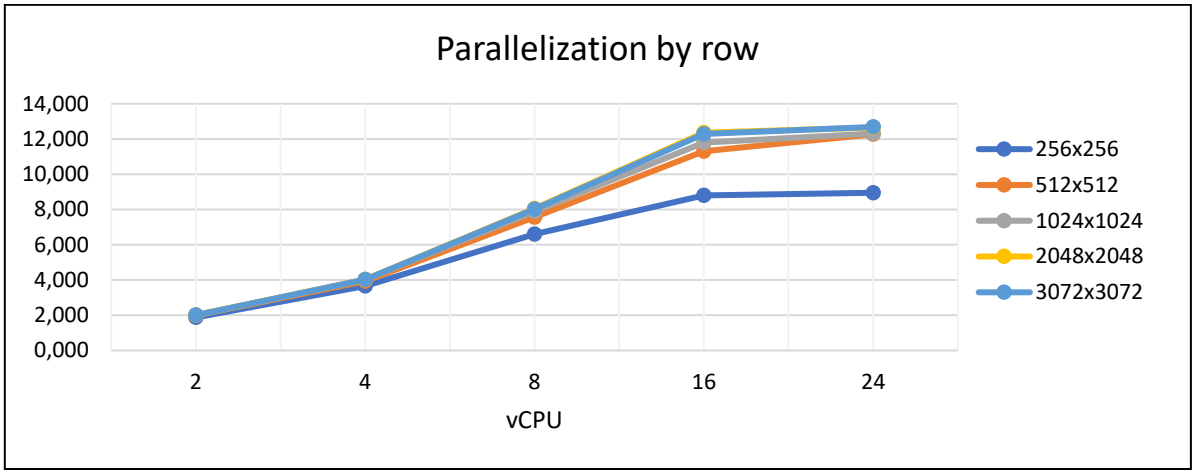

Figure 7. Graphic relative to the parallelization by rows

1.4.2 Parallelization with tiles results

OpenMP - **Tiles** approach					
Matrix size	vCPU				
	2	**4**	**8**	**16**	**24**
256x256	1.81	3.45	6.17	6.78	6.56
512x512	1.89	3.71	7.14	11.46	13.17
1024x1024	1.92	3.83	7.54	12.44	15.13
2048x2048	1.91	3.81	7.61	12.80	15.46
3072x3072	1.89	3.79	7.57	12.80	15.65

Table 2.1 Speedups with tiles approach

Matrix size	**Serial - rows**	**Serial - tiles**	**OpenMP - rows - 24 threads**	**OpenMP - tiles - 24 threads**
256x256	0.023	0.019	0.003	0.003
512x512	0.205	0.163	0.017	0.012
1024x1024	1.695	1.322	0.138	0.087
2048x2048	73.468	10.592	5.807	0.685
3072x3072	263.583	35.891	20.779	2.294

Table 2.2 Comparisons between times of rows and tiles algorithms

If compared with the previous ones, in table 2.1 there seems to be no additional gain, however this is not exactly true because here we are making a comparison between two different serial codes. In fact, looking at the execution times in table 2.2 we can see that moving from one algorithm to another we get much shorter times. If we consider the Amdahl's law the speedups obtained continue to be lower than the theoretical one, in fact the maximum speedup obtained is just below 16. Figure 8 describes graphically what happens.

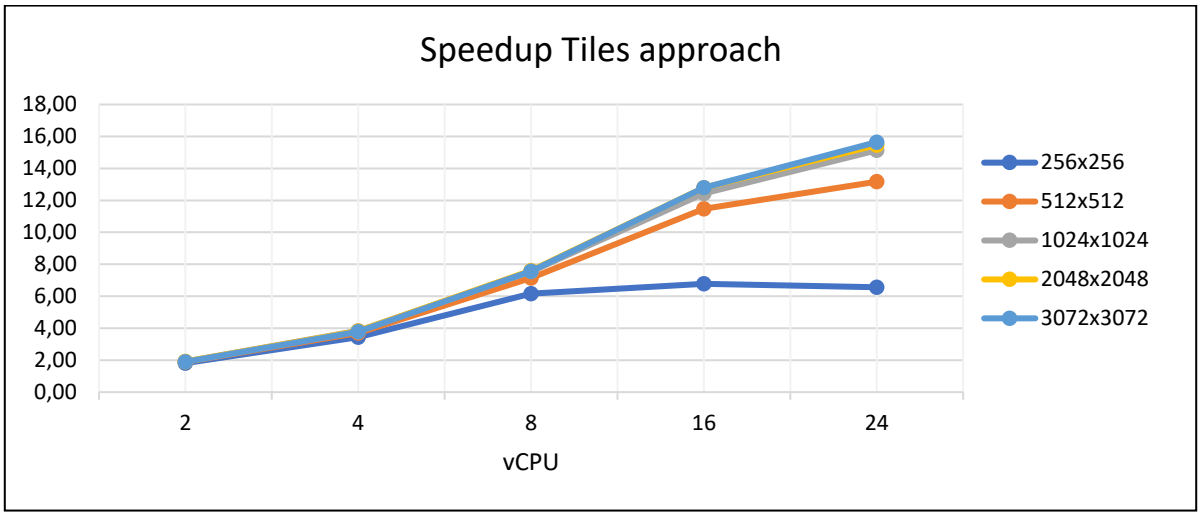

Figure 8. Graphic relative to the tiles approach

1.4.3 Parallelization with concurrent elements results

CUDA – GTX 1050 (640 cores)	
Matrix Size	**Speedup**
256x256	23.12
512x512	34.16
1024x1024	41.34
2048x2048	166.60
3072x3072	174.33

Table 3. CUDA multiplication speedups

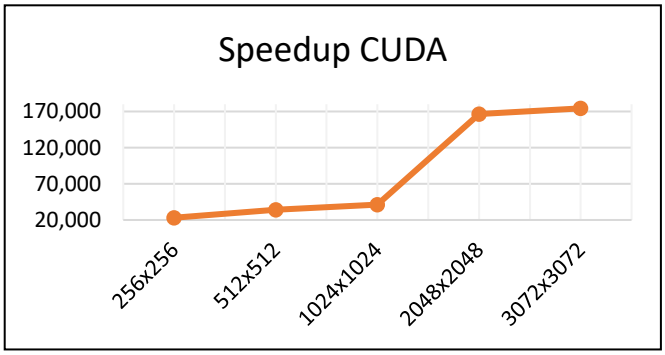

Figure 9. Graphic relative to speedups with CUDA

As expected with CUDA, significant performance improvements have been achieved due to the high number of threads and the strong parallelization of the algorithm that has been done. This leads to speedups that are up to 12 times better than those obtained with OpenMP, as shown in Table 3 and Figure 9. The results however remain quite inferior to the expectations of the Amdahl's law due to the kernel calls overhead and memory transfers between host and device.

2. Matrix inversion

As with the multiplication, the inversion of a matrix is an algebraic operation widely used in Computer and Data Science (e.g., estimation of statistical models). However, it also requires an increasing amount of time depending on the size of the matrix to be inverted, and that is why performing a parallelization would help reducting calculation times.

2.1. Analysis of the serial algorithm

The explanation of the LU decomposition method follows. Let us start with $Ax = b$ where A is a *n x n* squared matrix and b is a *n x 1* column vector, the aim is to solve the system to obtain x.

It is important that the matrix is squared and that it is not a singular matrix. In order to do these checks, we compute the determinant of the matrix A and even if it is equal to zero, we use the pivoting process to avoid divisions by zero.

Then, the core idea is to break up A into two matrices L (lower triangular) and U (upper triangular) through the LU decomposition, whose code is in Figure 10. In this way the original equation is replaced with $(LU)x = b$:

$$\left(\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}\right)\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Note that A in reality is PA because we have done partial pivoting before and P is the permutation matrix.

```
void decomposition(double **l, double **u, int n) {
        int i, j, k;

        for (k = 0; k < n; k++) {
                for (i = k + 1; i < n; i++) {
                        l[i][k] = u[i][k] / u[k][k];
                        for (j = k; j < n; j++) {
                                u[i][j] = u[i][j] - l[i][k] * u[k][j];
                        }
                }
        }
}
```

Figure 10. Serial LU decomposition

Shifting the parentheses, the previous system can be divided into two systems:

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

(here *y* is another column vector similarly to b). Now, using forward and backward substitution techniques, we firstly solve $Ly = b$ (with forward substitution, Figure 11) getting *y* and then we substitute it into $Ux = y$ to solve the system getting *x* (with backward substitution, Figure 12).
In our codes, the vector *b* is equivalent, as explained below, to the columns of the identity matrix while *x* to the values of the columns of the inverse matrix (*a_inv*).

```
void forwardSubstitution(double **l, double **p, double *y, int column, int n) {
        int i, j;
        double sum = 0;

        for (i = 0; i < n; i++) {
                for (j = 0; j < i; j++) {
                        sum = sum + l[i][j] * y[j];
                }
                y[i] = (p[i][column] - sum) / l[i][i];
                sum = 0;
        }
}
```

$$Y_n = \frac{I_n - \sum_{j=1}^{n-1} L_{n,j} * Y_j}{L_{n,n}}$$

Figure 11. Serial forward substitution

```
void backwardSubstitution(double **u, double *y, double **a_inv, int column, int n) {
        int i, j;
        double sum;

        a_inv[n-1][column] = y[n-1] / u[n-1][n-1];
        for (i = n - 2; i >= 0; i--) {
                sum = y[i];
                for (j = n - 1; j > i; j--) {
                        sum = sum - u[i][j] * a_inv[j][column];
        }
        a_inv[i][column] = sum / u[i][i];
        sum = 0;
    }
}
```

$$A^{-1} = \frac{Y_n}{U_{n,n}}$$

Figure 12. Serial backward substitution

This LU decomposition can be used to compute the inverse of a matrix A which is defined as $AA^{-1} = I$ where I is the identity matrix. This, for example, looks like:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} a_{11}' & a_{12}' & a_{13}' \\ a_{21}' & a_{22}' & a_{23}' \\ a_{31}' & a_{32}' & a_{33}' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The problem now can be set up like *n* "$Ax = b$" problems with different *b* vectors and the *x* vector becoming the n^{th} column of the inverse matrix, computed with backward and forward substitutions as descripted above, by using a single for loop (as shown in Figure 13).

```
for (i = 0; i < n; i++) {
        forwardSubstitution(l, p, y, i, n);
        backwardSubstitution(u, y, a_inv, i, n);
 }
```

Figure 13. Forward and Backward substitutions calls

The overall complexity of the LU algorithm is about $\Theta(\frac{2}{3}n^3)$.

We wrote the serial code from scratch, by translating the MATLAB code, that we have studied in "Numerical Methods for Engineering Sciences" course (https://bit.ly/3Apm4ux), to the C language.

2.2. A-priori study of available parallelism and parallel implementations

The program can be divided into the following main parts:

- Memory allocations with malloc
- Reading matrix
- Computing inversion
    - Pivoting
    - Computing determinant and checks
    - LU decomposition
    - Forward and Backward substitutions
- Writing the resulting matrix

As with multiplication, we decided to ignore memory allocations and I/O operations since they only take a very small portion of the whole execution time of the program. So, we have focused on the

different functions in the *computing inversion* phase, and with Gprof we stated that the most part of the time was taken by backward and forward substitutions as well as decomposition. The flat profile, obtained using a 1024x1024 matrix, can be seen in Figure 14 below.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 35.08      3.38     3.38     1024     0.00     0.00  backwardSubstitution
 32.58      6.52     3.14     1024     0.00     0.00  forwardSubstitution
 31.39      9.54     3.02        1     3.02     3.02  decomposition
  0.10      9.55     0.01        1     0.01     0.01  pivoting
  0.00      9.55     0.00        1     0.00     0.00  determinant
  0.00      9.55     0.00        1     0.00     0.00  printMatrix
  0.00      9.55     0.00        1     0.00     0.00  readMatrix
```

Figure 14. Gprof profile on serial inversion

Again, for the same reasons given for the matrix multiplication, both OpenMP and CUDA were used in search of the best possible speedup.

2.2.1. OpenMP parallelization

For what concern the decomposition, we assigned to each thread the computation of the elements of each row in L and U matrices applying the *#pragma omp for* to the second loop, as shown in Figure 15, not on the first because otherwise there would be data races within U matrix. We used a static scheduling since each row has got the same number of elements and we do not work with sparse matrices, so there would not be a significant difference using dynamic scheduling.

```
void decomposition(double **l, double **u, int n) {
    int i, j, k;

        #pragma omp parallel private (i,j,k)
        {
                for (k = 0; k < n; k++) {
                        #pragma omp for schedule(static)
                        for (i = k + 1; i < n; i++) {
                                l[i][k] = u[i][k] / u[k][k];
                                for (j = k; j < n; j++) {
                                        u[i][j] = u[i][j] - l[i][k] * u[k][j];
                                 }
                        }
                 }
         }
}
```

Figure 15. OpenMP LU decomposition

Regarding the two substitutions, we tried two approaches:

1. In the first one we parallelized the summation on each iteration using the reduction clause, but it resulted in a small speedup compared to the other way.
2. In the second one, as we explained in the serial algorithm discussion, we gave to each thread the computation of an equations system to obtain a column of the inverse matrix, it was enough to parallelize the for loop in Figure 13 using the *#pragma omp for* with a dynamic scheduling and this gave us better improvements compared to the first approach.

```
#pragma omp parallel shared(a_inv) private(i)
{
        #pragma omp for schedule(dynamic)
        for (i = 0; i < n; i++) {
                double *y = (double*)malloc(n * sizeof(double));
                forwardSubstitution(l, p, y, i, n);
                backwardSubstitution(u, y, a_inv, i, n);
                free(y);
        }
}
```

Figure 16. OMP backward and forward substitution calls

Minor improvements in speedup were obtained parallelizing the determinant function which compute the determinant of the matrix in order to check if it is possible to compute the inversion, here we used only a simple reduction clause on the multiplication of L and U diagonal elements (Figure 17).

```
double determinant(double **l, double **u, int n, int *perm) {
        int i;
        double det = 1;

        #pragma omp parallel
        {
                #pragma omp for reduction(*: det)
                for(i = 0; i < n; i++)
                        det *= l[i][i] * u[i][i];
        }

        return pow(-1, perm[0]) * det;
}
```

$$det(\mathrm{A}) = det(LU) = det(L) * det(U)$$

(where det(L) and det(U) are computed multiplying elements on their diagonal)

Figure 17. OMP determinant function

The last function that we have considered was the pivoting one, but we did not parallelize it because there were some *memcpy* used to swap the rows of A and a parallelization could cause overlapping in memory accesses and writings. Moreover, the parallelization will create potential data conflicts with shared variables which should indicate the row that contains the maximum value. Furthermore, this function, as shown in Gprof results, occupies only a minor fraction of the whole computation time because it is used only in particular cases.

2.2.2. CUDA parallelization

With CUDA we practically exploited only the power of a GPU modifying the previous functions into device functions since the reasoning was the same compared with the OpenMP parallelization. The functions were adapted to be computed using CUDA cores, which are divided in a grid and blocks only in one direction (x). Only the decomposition was not touched since it was hard to use the blocks subdivision with the inner loops, even if we tried to use a for loop, calling the kernel several times but this resulted in overhead addition.

2.2.3. Theoretical speedup with Amdahl's law

In order to compute the a-priori speedup we need again the Amdahl's law calculating again the $speedup_{enhanced}$ which is equal to the number of threads used, in particular the maximum which was 24 and the $fraction_{enhanced}$ which we calculated as:

$$fraction_{enhanced} = \frac{backward\ time\ +\ forward\ time\ + decomposition\ time\ +\ determinant\ time}{total\ time}$$

Since backward, forward, decomposition and determinant were the only functions that could be parallelizable, the resulting theoretical speedup is about 19.51 for OpenMP and 86.60 for CUDA.

2.3. Testing and debugging

To test the algorithm and evaluate speedups we used the same scripts and methodologies used for multiplication tests.
The inversion operation can only be done with squared matrices, and it requires generally less time than multiplication, so it is possible to use matrices whose order is up to 4096, as shown in the following table:

512x512	1024x1024	2048x2048	3072x3072	4096x4096

As with the multiplication, a Python script was used to calculate the inversion via a predefined library (NumPy), then comparing the outputs via the Linux diff command. Instead, the results of the parallel versions were compared to the serial ones.

In addition, small matrices were initially used, and their inverse was calculated either by hand or through online tools. To validate the results, we also used the same previous multiplication program to obtain the identity matrix since $AA^{-1} = I$. Instead to test the pivoting, some matrices that required it have been considered and then the output was validated.

The configuration of the virtual machine on GCP is the same as the one used for the multiplication.

2.4. Performance analysis

All the measurements of the speedup, again, consider the wall clock time relative to the inversion functions without considering the initial memory allocations and I/O operations. The execution times were obtained in the same way of the multiplication results (i.e., average times of three executions).

The results obtained are reported in the following paragraphs, also here, for synthesis reasons, the complete results have been added to the same datasheet for the multiplication results: https://bit.ly/3hmlRjW

2.4.1 OpenMP results

OpenMP – **Inversion**					
Matrix size	vCPU				
	2	**4**	**8**	**16**	**24**
512x512	1.94	3.80	6.91	6.06	4.97
1024x1024	2.06	3.61	5.33	5.71	5.00
2048x2048	2.03	3.85	7.01	9.64	9.45
3072x3072	2.00	4.05	8.00	11.62	12.29
4096x4096	1.98	4.03	7.97	12.10	13.11

Table 4. Speedups with parallelized inversion

With OpenMP we have a speedup that initially is linear and then, with the smallest matrices, an upper limit with 8 threads is reached due to the limits of parallelization. With larger matrices, instead, you get higher speedups as the sizes and number of threads increases up to a maximum of 13. This is well below the theoretical speedup since not all the code has been fully parallelized and there is still an overhead due to concurrency and memory accesses, which is visible even with small matrices (see 512x512 in figure 18).

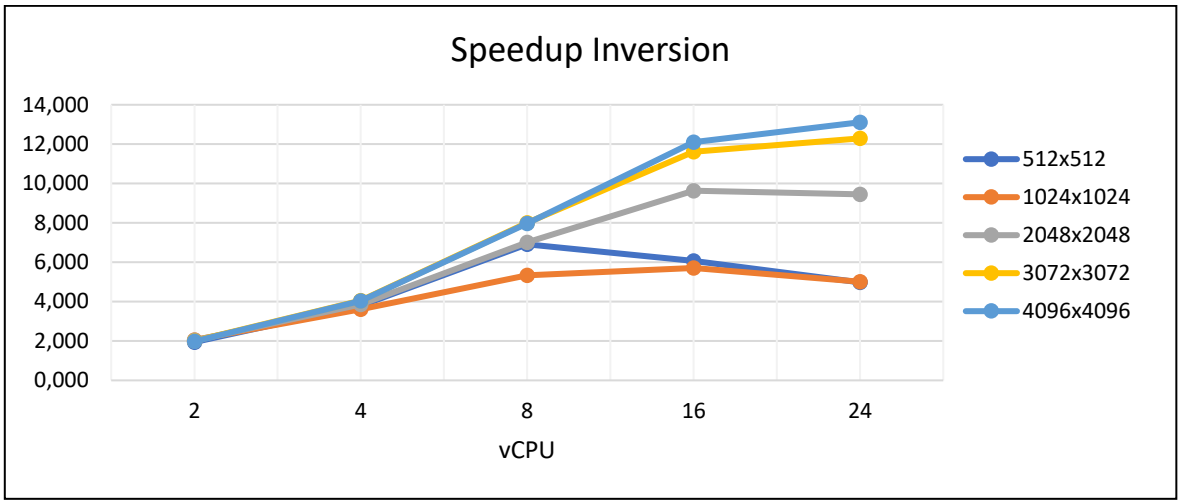

Figure 18. Graphic relative to the inversion speedups

2.4.2 CUDA results

With CUDA, the speedup increases by increasing the size of the matrices, until it reaches about 13x (as reported in Table 5 and Figure 19), similarly to what happen with OpenMP and much lower than a-priori speedup. These results could be explained with the data transfers (from host to device and vice-versa) and the overhead caused by kernel calls which affect only a small portion of the whole execution time in larger matrices.

Also, it should be considered that, given the complexity, in this implementation the decomposition has not been parallelized.

CUDA – GTX 1050 (640 cores)	
Matrix Size	**Speedup**
512x512	3.78
1024x1024	4.70
2048x2048	9.33
3072x3072	13.21
4096x4096	13.36

Table 5. CUDA Inversion speedups

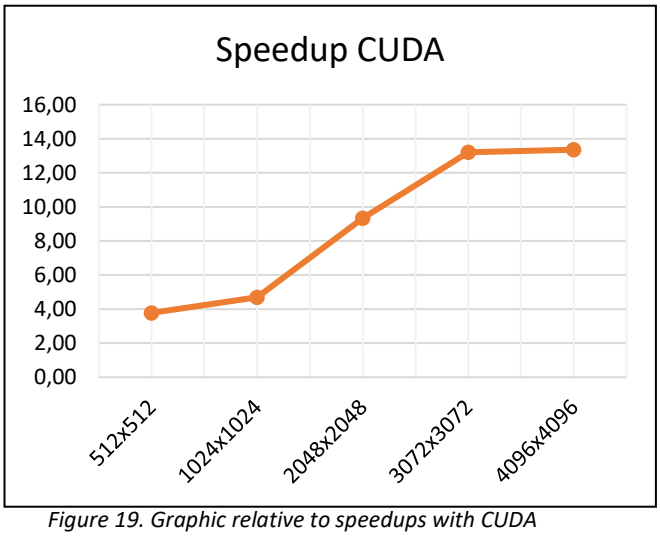

Figure 19. Graphic relative to speedups with CUDA

3. Individual contributions of the group members

We performed most of the work in video-calls, by analyzing together the different approaches and solutions to the two problems. We exploited pair-programming especially for the CUDA implementations, since only one of us has got an NVIDIA GPU and one of us has worked before with CUDA framework. Few tasks were performed alone:

- Domenico Ragusa
  - Generation of bash scripts for the tests on GCP and Python scripts for generating matrices of different dimensions.
  - Launch of tests on GCP for matrix-multiplication.
  - Debugging of matrix-multiplication codes and checking outputs correctness.
- Danilo Modica
  - Creation of Python scripts to check the correctness of the two algorithms.
  - Launch of tests on GCP for matrix-inversion.
  - Debugging of matrix-inversion codes and checking outputs correctness.

Anyway, test tasks were designed together and only the execution of them was accomplished alone.

4. References

- Our GitHub repository with codes: https://github.com/domenico-rgs/ACA-Project
- Datasheets with our results: https://universitadipaviait-my.sharepoint.com/:x:/g/personal/domenico_ragusa01_universitadipavia_it/EWlSBCGzpbFJhKOkYKtLeGkBFhk78OeajVflqJybmk7NqA?e=YCR6pq
- Matrix Inversion with LU from *Numerical Methods* course: https://www-dimat.unipv.it/sangalli/numerical_methods_eng_sciences/1%20-%20Linear%20Systems%20(part%20I).pdf
- Matrix Multiplication algorithm: https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
- Matrix Multiplication with blocks/tiles from *Advanced Computer Architecture* course: http://www.unipv.it/mferretti/cdol/aca/Charts/Parallel%20programming%20models/OpenMP/esercizi-openmp-3.pdf