Comparing different approachs for matrix multiplication omptimization

Eduardo López Fortes

November 10, 2024

Github repo: Second-MatrixMultiplication

Abstract

Throughout this memory, some algorithms for matrix multiplication algorithms will be discussed. To implement these algorithms, we will use the Java programming language because is not only widely used for software development but also because the previous stage showed us that it is the most efficient language for this problem, at least for middle-sized matrices, i.e., matrices with dimensions up to 1024x1024. The algorithms that will be discussed are the following: blocking matrix multiplication, which is a simple algorithm that divides the matrices into blocks and multiplies them; column-row matrix multiplication, which is an algorithm that multiplies the columns of the first matrix by the rows of the second matrix; and Strassen's algorithm, which is a divide-and-conquer algorithm that multiplies the matrices using a recursive approach. The goal of this memory is to analyze the performance of these algorithms and to determine which one is the most efficient for different matrix sizes. Along with those algorithms, a Sparse Matrix Multiplication algorithm will be implemented and analyzed: given two matrices A and B, both of them with a considerable amount of zeros, the algorithm will save the matrices in a way that only the non-zero elements are stored, and then it will multiply them. This algorithm is expected to be more efficient than the others for matrices with a high number of zeros. Once all the algorithms are implemented, they will be tested with matrices of different sizes and the results will be analyzed using Java Microbenchmark Harness (JMH), which is a Java library that allows us to measure the performance of the algorithms. Finally, the results will be discussed, and the most efficient algorithm for each matrix size will be determined by analyzing the performance of the algorithms using graphs and tables.

1 Introduction

First of all, GitHub repo in this link:

Before starting with the memory, the reader would have noticed that the format of the document itself is different from the previous stage. This is beacuse the two-column format was giving me some problems with images, not allowing me to put them in the right place. I hope this format is still clear and easy to read and hopefully it will allow me to put the images in the right place without any complications.

As we stated in the preivous stage, matrix multiplication is a fundamental operation in linear algebra and computer science. It is used in many applications, such as image processing, machine learning, and scientific computing. Furthermore, results have shown that this operation is computationally expensive, and its performance can be improved by using different algorithms. The complexity of the naïve matrix multiplication is not only exponential but also cubic, which makes it inefficient for large matrices. Therefore, it is important to analyze the performance of different algorithms to determine which one is the most efficient for different matrix sizes.

The use of Java in this stage is justified by the results of the previous stage, which showed that Java is the most efficient language for this problem, at least for middle-sized matrices. Whereas Python showed good results for small matrices, and C and C++ showed good results for large matrices, Java showed good results for middle-sized matrices. It is more likely that the possible matrices that will be faced in real-world applications will be of middle size, so Java is the most suitable language for this problem and therefore, it will be the only programming language used in this stage.

There are different approaches when it comes to optimizing the naïve matrix multiplication algorithm. From one hand we can directly optimize the algorithm itself, i.e. focusing on the way that multiplication is done. Other approaches consist in optimizing the way the matrices are stored in memory, i.e. the way that the data is accessed; or use threads to parallelize the multiplication. In this stage, we will focus on the two first approaches: implementing and analyzing different matrix multiplication algorithms and implementing a Sparse Matrix Multiplication algorithm. The third approach, parallelizing the multiplication, goes beyond the scope of this project, but it is a possible future work.

The algorithms that will be implemented are the following:

1. Blocking matrix multiplication: This algorithm divides the matrices into blocks and multiplies them. It is a simple algorithm that can be optimized by changing the block size. The block size is a parameter that can be adjusted to improve the performance of the algorithm. The idea is to reduce the number of cache misses by multiplying blocks that fit in the cache. This algorithm is expected to be more efficient than the naïve matrix multiplication algorithm because it reduces the number of cache misses. However, not for all block sizes the algorithm will be more efficient than the naïve algorithm.

2. Column-row matrix multiplication: This algorithm multiplies the columns of the first matrix by the rows of the second matrix. It is a simple algorithm that can be optimized by changing the order of the loops. The idea is to improve the data locality by accessing the elements of the matrices in the right order. This algorithm is expected to be more efficient than the naïve matrix multiplication algorithm because it improves the data locality.

3. Strassen's algorithm: This algorithm is a divide-and-conquer algorithm that multiplies the matrices using a recursive approach. It is more efficient than the naïve matrix multiplication algorithm because it reduces the number of multiplications. The idea is to divide the matrices into submatrices and multiply them recursively. This algorithm is expected to be more efficient than the naïve matrix multiplication algorithm for large matrices. However, it is not efficient for small matrices because of the overhead of the recursive calls. One approach to improve the performance of this algorithm is to use the naïve algorithm when the matrices are enough small to make the recursive calls inefficient.

   And then, for the sparse matrices, we will implement the following algorithm:

4. Sparse Matrix Multiplication: This algorithm multiplies two sparse matrices by saving them in a way that only the non-zero elements are stored. The idea is to reduce the number of multiplications by avoiding the multiplication of zero elements. Despite the idea of this algorithm is to be more efficient than the other algorithms for matrices with a high number of zeros, we will find out that when the number of zeros is not zero, it is somehow more efficient than the naïve algorithm. The result given by multiplying two non-sparse matrices seems to be correct, so it seems that the algorithm is working properly.

2 Methodology

2.1 Baseline

The first step is to implement the algorithms in Java. The implementation of the algorithms will be done using the Java programming language. The algorithms will be implemented as methods that receive two matrices as input and return the result of the multiplication. The matrices will be represented as two-dimensional arrays of integers. Each implementatio has been developed in a separate project, so that the code is more organized and easier to understand. This approach also helps to avoid conflicts between the different implementations and makes it easier to test them since each implementation is independent of the others and some of them may require different parameters.

2.1.1 Matrix Generation

Before implementing the algorithms, the method in which the matrices will be generated must be defined. The matrices will be generated randomly, and the elements will be double values between 0 and 1. The size of the matrices will be defined as a parameter, so that the algorithms can be tested with different matrix sizes. The matrices will be generated using the Random class from the java.util package. The matrices will be generated in the main method of the project, and then they will be passed as input to the algorithms. The pseudo-code of the matrix generation is as follows:

Algorithm 1 Matrix Generation

```
1: Input: $n$                                          ▷ Size of the matrices
2: Initialize matrices $a[n][n]$ and $b[n][n]$
3: Create random object $random$
4: for $i \leftarrow 0$ to $n-1$ do
5:     for $j \leftarrow 0$ to $n-1$ do
6:         $a[i][j] \leftarrow random.nextDouble()$
7:         $b[i][j] \leftarrow random.nextDouble()$
8:     end for
9: end for
```

Now, how all the algorithms work is going to be explained.

2.1.2 Sparse Matrix Multiplication

In real world we may face situations where the matrices have a considerable amount of zeros. In this case, the Sparse Matrix Multiplication algorithm can be used to reduce the number of multiplications by avoiding the multiplication of zero elements. This method also saves memory by storing only the non-zero elements of the matrices and at the same time it reduces the number of cache misses for the very same reason. There are three different approaches when it comes to store the sparse matrices:

1. Coordinate List (COO)

   This approach stores involves storing the row index, column index and value of each non-zero element in the matrix. This approach is simple and easy to implement, but it is not efficient for matrix multiplication because it requires searching for the elements in the list. There it is an example of how the COO approach works:

   Given the matrix
$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 3 & 0 & 0 \end{bmatrix}$$
   the COO representation of the matrix would be as follows:
$$\text{row index} = [0, 2, 1]$$
$$\text{column index} = [0, 0, 2]$$
$$\text{values} = [1, 3, 2]$$
   That means that the non-zero elements of the matrix are $A[0][0] = 1$, $A[2][0] = 3$, and $A[1][2] = 2$.

2. Compressed Sparse Row (CSR)

   This approach stores the column index, row pointer, and value of each non-zero element in the matrix, but it also stores the index of the first element of each row and the number of non-zero elements in each row. This approach is more efficient than the COO approach because it allows to access the elements of the matrix in a more efficient way.

   Given the following matrix
$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 2 \\ 3 & 0 & 3 \end{bmatrix}$$
   the CSR representation of the matrix would be as follows:
$$\text{column index} = [0, 1, 2, 0, 2]$$

$$\text{row pointer} = [0, 1, 3, 5]$$

$$\text{values} = [1, 2, 2, 3, 3]$$

Where the row pointer indicates the index of the first element of each row in the column index array. The first row starts at index 0, the second row starts at index 1, and the third row starts at index 3.

3. Compressed Sparse Column (CSC)

   This approach is similar to the CSR approach, but it stores the row index instead of the column index. This approach is also more efficient than the COO approach.

   Given the following matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 2 \\ 3 & 0 & 3 \end{bmatrix}$$

   the CSC representation of the matrix would be as follows:

$$\text{row index} = [0, 2, 1, 1, 2]$$

$$\text{column pointer} = [0, 2, 3, 5]$$

$$\text{values} = [1, 3, 2, 2, 3]$$

   Here, the column pointer indicates the index of the first element of each column in the row index array. The first column starts at index 0, the second column starts at index 2, and the third column starts at index 3.

In this project, we will use the CSC approach to store the sparse matrices because it is more efficient than the COO approach. The Sparse Matrix Multiplication algorithm will be implemented as a method that receives two matrices in CSR format as input and returns the result of the multiplication.

In this case, matrix generation is slightly different from the previous cases. The matrices will be generated randomly, but the number of non-zero elements will be limited to a certain percentage of the total number of elements. The percentage of non-zero elements will be defined as a parameter, so that the algorithm can be tested with different percentages. The matrices will be generated using the Random class from the java.util package. The pseudo-code of the matrix generation is as follows:

Algorithm 2 GenerateRandomMatrices(n, *percentage*)

```
 1: Initialize matrix $a$ of size $n \times n$
 2: Initialize matrix $b$ of size $n \times n$
 3: Initialize random number generator
 4: for $i = 0$ to $n - 1$ do
 5:     for $j = 0$ to $n - 1$ do
 6:         $n1 \leftarrow$ random number between 0 and 1
 7:         $n2 \leftarrow$ random number between 0 and 1
 8:         if $n1 >$ percentage then
 9:             $a[i][j] \leftarrow n1$
10:         else
11:             $a[i][j] \leftarrow 0$
12:         end if
13:         if $n2 >$ percentage then
14:             $b[i][j] \leftarrow n2$
15:         else
16:             $b[i][j] \leftarrow 0$
17:         end if
18:     end for
19: end for
```

Then, the implementation of the matrix multiplication by using this method is the following (after, of course, the conversion of the initial matrix to the CSC format):

Algorithm 3 Multiply(A, B)

```
Initialize empty lists: $resultValues$, $resultRowIndices$, and $resultColPointers$
Add 0 to $resultColPointers$
Initialize temporary array $colResult$ of size $this.rows$
for $jB = 0$ to $B.cols - 1$ do
    Clear $colResult$
    for $k = B.colPointers[jB]$ to $B.colPointers[jB + 1] - 1$ do
        $rowB \leftarrow B.rowIndices[k]$
        $valB \leftarrow B.values[k]$
        for $i = this.colPointers[rowB]$ to $this.colPointers[rowB + 1] - 1$ do
            $rowA \leftarrow this.rowIndices[i]$
            $valA \leftarrow this.values[i]$
            $colResult[rowA] \leftarrow colResult[rowA] + valA \times valB$
        end for
    end for
    $nonZeroCount \leftarrow 0$
    for $i = 0$ to $this.rows - 1$ do
        if $colResult[i] \neq 0$ then
            Add $colResult[i]$ to $resultValues$
            Add $i$ to $resultRowIndices$
            $nonZeroCount \leftarrow nonZeroCount + 1$
        end if
    end for
    Add $nonZeroCount$ to $resultColPointers$
end for
Convert $resultValues$, $resultRowIndices$, $resultColPointers$ to arrays
return new $CSCMatrix(resultValues, resultRowIndices, resultColPointers, this.rows, B.cols)$
```

2.1.3 Blocking Matrix Multiplication

The first algorithm to be implemented is the blocking matrix multiplication algorithm. As stated previously, this algorithm divides the matrices into blocks and multiplies them. The block size is a parameter that can be adjusted to improve the performance of the algorithm. The idea is to reduce the number of cache misses by multiplying blocks that fit in the cache. The algorithm will be implemented as a method that receives two matrices and the block size as input and returns the result of the multiplication. We have to ensure that the block size is a divisor of the matrix size, so that the matrices can be divided into blocks of the same size and that the block size is smaller than the matrix size, so that the algorithm can take advantage of the cache. The pseudo-code of the blocking matrix multiplication algorithm is as follows:

Algorithm 4 blockMatrixMultiplication(a, b, c, n, $block\_size$)

```
for $i \leftarrow 0$ to $n - 1$ by $block\_size$ do
    for $j \leftarrow 0$ to $n - 1$ by $block\_size$ do
        for $k \leftarrow 0$ to $n - 1$ by $block\_size$ do
            multiplyBlock($a$, $b$, $c$, $i$, $j$, $k$, $n$, $block\_size$)
        end for
    end for
end for
```

In this algorithm, the matrices are divided into blocks, and each block is multiplied individually. The algorithm iterates over the blocks of the matrices and multiplies them using the multiplyBlock method. The multiplyBlock method multiplies the individual blocks of the matrices and updates the result matrix. The algorithm is expected to be more efficient than the naïve matrix multiplication algorithm because it reduces the number of cache misses. However, not for all block sizes the algorithm will be more efficient than the naïve algorithm. The performance of the algorithm will be analyzed by measuring the execution time for different block sizes.

Algorithm 5 multiplyBlock(a, b, c, $rowBlock$, $colBlock$, $kBlock$, n, $block\_size$)

```
1: for $i \leftarrow rowBlock$ to $\min(rowBlock + block\_size, n) - 1$ do
2:     for $j \leftarrow colBlock$ to $\min(colBlock + block\_size, n) - 1$ do
3:         $sum \leftarrow 0$
4:         for $k \leftarrow kBlock$ to $\min(kBlock + block\_size, n) - 1$ do
5:             $sum \leftarrow sum + a[i][k] \times b[k][j]$
6:         end for
7:         $c[i][j] \leftarrow c[i][j] + sum$
8:     end for
9: end for
```

2.1.4 Strassen's Matrix Multiplication

If we move to the Strassen's algorithm, it is a divide-and-conquer algorithm that multiplies the matrices using a recursive approach. It is more efficient than the naïve matrix multiplication algorithm because it reduces the number of multiplications. The idea is to divide the matrices into submatrices and multiply them recursively. The algorithm will be implemented as a method that receives two matrices as input and returns the result of the multiplication. The algorithm will use a recursive approach to multiply the matrices. More precisely:

We have two matrices A and B of size n x n that we want to multiply, which can be written as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Where A11, A12, A21, A22, B11, B12, B21, and B22 are submatrices of size n/2 x n/2.

Using the classical matrix multiplication approach, we compute the product $C = A \times B$ as:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

However, Strassen's algorithm reduces the number of multiplications to 7 by calculating the following intermediate matrices:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

The final result is computed using these intermediate matrices:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Thus, the product matrix C is given by:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Strassen's algorithm reduces the number of scalar multiplications from 8 to 7, making it more efficient than the traditional approach for larger matrices. That way we now have a complexity of O($n^{2.81}$) instead of O(n^3)

The pseudo-code looks similir to what it has just been exposed, but with the addition of the recursive calls:

Algorithm 6 StrassenMultiply(A, B)

```
1: $n \leftarrow$ size of $A$
2: if $n == 1$ then
3:     return $A[0][0] \times B[0][0]$
4: end if
5: Divide $A$ into $A_{11}, A_{12}, A_{21}, A_{22}$
6: Divide $B$ into $B_{11}, B_{12}, B_{21}, B_{22}$
7: $M_1 \leftarrow$ StrassenMultiply$(A_{11} + A_{22}, B_{11} + B_{22})$
8: $M_2 \leftarrow$ StrassenMultiply$(A_{21} + A_{22}, B_{11})$
9: $M_3 \leftarrow$ StrassenMultiply$(A_{11}, B_{12} - B_{22})$
10: $M_4 \leftarrow$ StrassenMultiply$(A_{22}, B_{21} - B_{11})$
11: $M_5 \leftarrow$ StrassenMultiply$(A_{11} + A_{12}, B_{22})$
12: $M_6 \leftarrow$ StrassenMultiply$(A_{21} - A_{11}, B_{11} + B_{12})$
13: $M_7 \leftarrow$ StrassenMultiply$(A_{12} - A_{22}, B_{21} + B_{22})$
14: Compute $C_{11}, C_{12}, C_{21}, C_{22}$ using $M_1$ to $M_7$
15: Combine $C_{11}, C_{12}, C_{21}, C_{22}$ into result matrix $C$
16: return $C$
```

2.1.5 Column-Major Matrix Miltiplication

The last algorithm to be implemented is the column-row major algorithm. This algorithm multiplies the columns of the first matrix by the rows of the second matrix. It is a simple algorithm that can be optimized by changing the order of the loops. The idea is to improve the data locality by accessing the elements of the matrices in the right order. The algorithm will be implemented as a method that receives two matrices as input and returns the result of the multiplication. The pseudo-code of the column-row major algorithm is as follows:

Algorithm 7 Matrix Multiplication

```
1: Input: $x$                                                    ▷ Size of the matrices
2: Initialize $A$ and $B$ as $x \times x$ matrices with random values
3: Initialize $C$ as a $x \times x$ matrix with zeros
4: for $j = 1$ to $x$ do
5:     for $i = 1$ to $x$ do
6:         for $k = 1$ to $x$ do
7:             $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
8:         end for
9:     end for
10: end for
11: return $C$
```

Note that besides the loop order have changed (now we iterate first over j and then over i) the way we access the matrix does not change: we still access $A[i][k]$ and $A[k][j]$ despite we iterate the other way around.

2.2 Test Developing

With all the algorithms implemented, the next step is to test them with matrices of different sizes and analyze the performance of the algorithms. The performance of the algorithms will be analyzed by measuring the execution time for different matrix sizes. The execution time will be measured using Java Microbenchmark Harness (JMH), which is a Java library that allows us to measure the performance of the algorithms. The results will be analyzed by comparing the execution times of the algorithms for different matrix sizes. The goal is to determine which algorithm is the most efficient for different matrix sizes. The results will be discussed, and the most efficient algorithm for each matrix size will be determined by analyzing the performance of the algorithms using graphs and tables.

The experiment is more precise if we consider the process of creation is inherited from the matrix multiplication. As a result, when benchmarking, it was not treated as setup but as part of the process itself.

For this experiment we tested different sizes of matrix: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 (2048 and 4096 were removed from the test set as it toakes to much time to compute them).

For the column-row major algorithm and the strassen methods, no additional parameter are needed (only the size of the matrix), but for the blocking matrix multiplication algorithm, the block size will be a parameter that can be adjusted to improve the performance of the algorithm. The block size is a divisor of the matrix size, so that the matrices can be divided into blocks of the same size, and it is smaller than the matrix size, so that the algorithm can take advantage of the cache. The performance of the algorithm will be analyzed by measuring the execution time for different block sizes. The results will be analyzed by comparing the execution times of the algorithms for different block sizes. It must be taken into account that the matrix size have to be divisible by the chosen block size, so for the testing purposes, the electable block sizes would be the previous powers fot two of the matriz size (e.g. for matrix size 128 the following block sizes will be tested: 1,2,4,8,16,32,64)

The sparse matrix multiplication also needs an additional parameter, which is the percentage of non-zero elements. The percentage of non-zero elements will be defined as a parameter, so that the algorithm can be tested with different percentages. The performance of the algorithm will be analyzed by measuring the execution time for different percentages of non-zero elements. The results will be analyzed by comparing the execution times of the algorithms for different percentages of non-zero elements (also 0% of zero-elements or 100% non-zero elements).

For every size the performance would be sized by measuring the time it took to the program to be executed (including the process of creating the matrix) in milliseconds.

In addition, in order to have consistent and reliable results, the following procedure was followed:

1. An arbitrary number of warm-ups rounds would be set to run before testing the program
2. The test would be ran several times (5)
3. This two steps would form a complete round. This process would be completed 3 times before moving to testing the next size

2.3 Software used

Intellij Idea along with a maven project were chosen for running and editing Java code, and then Java Microbenchmarking Harness for the testing purposes - added as a dependency in the pom.xml file:

```
- Java 11
- Java Microbenchmarking Harness 1.35
```

2.4 Machine Specifications

Hardware specifications influences notably on the testing results since many factors could change the way in which the program is executed: amount of RAM, type of processor - including number of cores -, graphic card (if applicable), etc.

For that reason, the specifications of the machine in which the test were ran are the following:

```
- Machine: AMD64
- Processor: AMD Ryzen 7 5700U with 8 cores, 1.80 GHz
- RAM: 8GB
- SSD storage: 512GB
- Operating System: Windows 10 Home
- Version: 22H2
```

3 Experiments

3.1 Blocking Matrix Multiplication

Common sense tells us that this method will be more efficient for larger matrix sizes and for that reson, the first matrix sized to be tested is 1024.

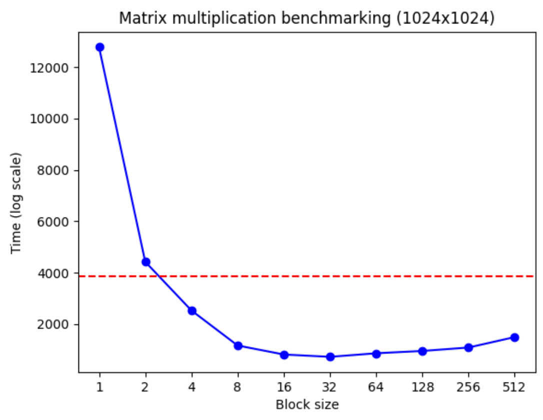

Figure 1: Time of execution for different block sizes for matrix multiplication (1024x1024)

We can appreciate in Figure 1 that not all block sizes actually improve the naïve solution. For example, for block sizes 1 and 2, the solution is even worse than the naïve one. This is because the overhead of the block size is greater than the benefit of granularity. We can also appreciate that the relationship between block size and performance is not linear but rather follows a curve. This is because the overhead of the block size is not constant but rather increases with the block size. This is why we need to find the optimal block size that minimizes the overhead and maximizes the benefit of granularity, which in this case is 32.

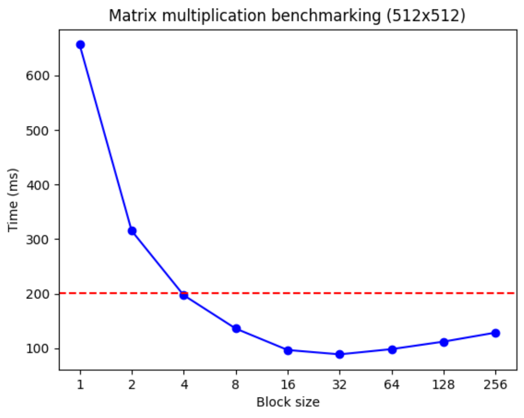

Figure 2: Time of execution for different block sizes for matrix multiplication (512x512)

Figure 2 shows us that for the matrix size 512, the results remain the same: not all block sizes actually improve the naïve solution and the optimal block size that minimizes the overhead and maximizes the benefit of granularity is 32.

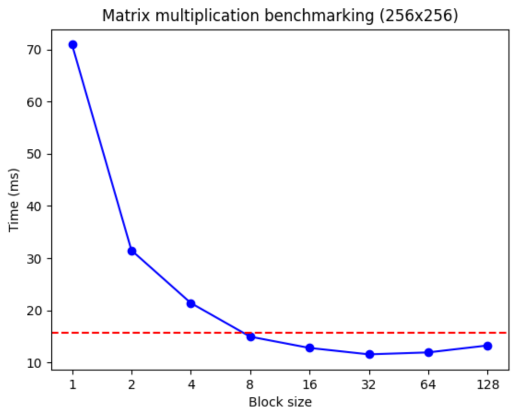

Figure 3: Time of execution for different block sizes for matrix multiplication (256x256)

Figure 3 exposes that for the matrix size 1024, the results remain the same and again, the optimal block size is 32, which may be the optimal block size for all matrix sizes, but we need to test it for the rest of the matrix sizes to confirm this.

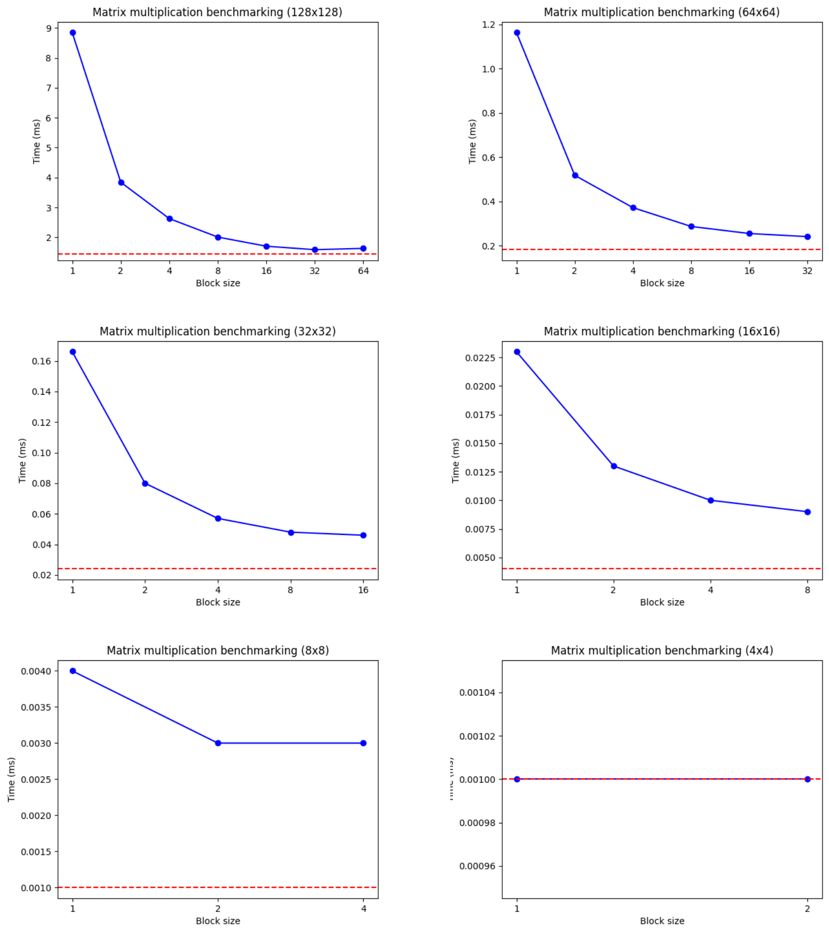

Figure 4: Time of execution for different block sizes for matrix multiplication (sizes 4,8,16,32,64 and 128)

However, Figure 4 help us to reaalize that none of the block sizes actually improve the naïve solution for matrix sizes 4,8,32,64 and 128. This occurs because the overhead of the block size is greater than the benefit of granularity.

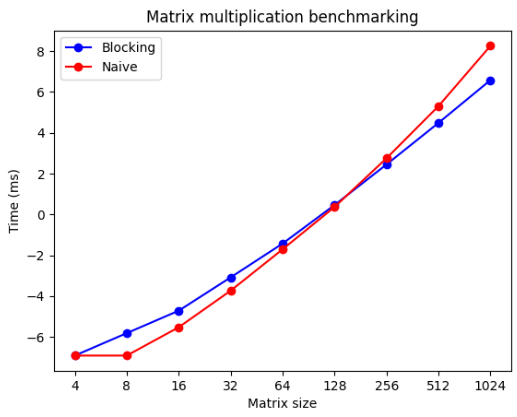

Figure 5: Comparison of the naïve algorithm and the blocking matrix multiplication algorithm

Comparing the naïve approach with the blocking algorithm as Figure 5 suggests, there is evidence that this method works better for matrices over 128 and worse for smaller matrices.

3.2 Sparse Matrix Multiplication

For this test we took 4 levels of sparsing: 0,0.2,0.4,0.6 and 0.8 for every size of matrix. And these are the results:

Figure 6 shows that for matrix sizes: 4,8,16,32 and 64, the process of converting the initial matrix to the sparse matrix format is not worth the cost, since the naïve algorithm seems to perform better for these matrix sizes.

I consider that taking into account the process of converting the matrix into the sparse matrix should be part of the multiplication process, since in the real world, anyone would rarely face the problem of multiplying sparse formatted matrices.

Following this procedure, the time it takes to convert the sparse matrix to the usual format should also be taken into account in the overall time, but this time I consider that process is out of our scope

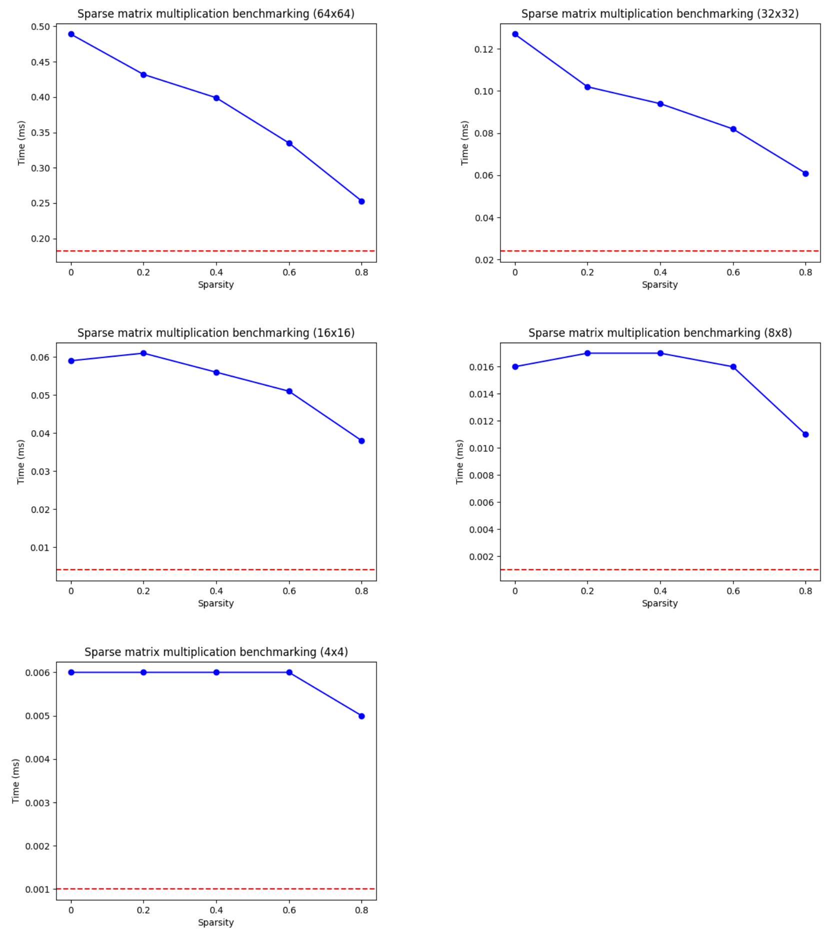

Figure 6: Time of execution for different sparsing for matrix multiplication (sizes 4,8,16,32 and 64)

However, for matrix sizes 128 and 256, there are certains levels od sparsity for which this algorithm is better than the naïve algorithm, and in that cases, this process of formatting the matrices is worth it. Figure 7

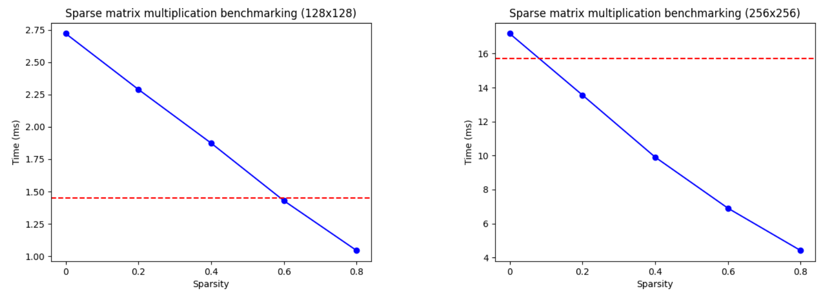

Figure 7: Time of execution for different sparsing for matrix multiplication (128 and 256)

And for some reason, when it comes to multiplying matrices of size 512 or 1024, even if the sparsity level is 0 (that means that the mean of zeros is 0) this algorithm perfomrs better than the naïve algorithm. I suspect that this is because of the way matrices are being stored and process of searching non-zeros values, which somehow reduces the number of cache misses. It has been checked taht the results returned in these cases are correct, so that is the only explanation for why this method is faster for larger matrices. Figure 8

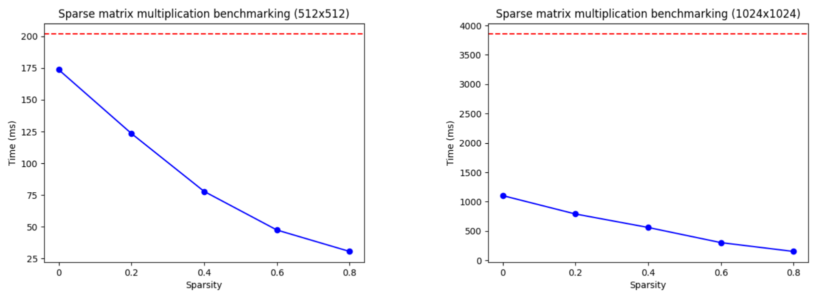

Figure 8: Time of execution for different sparsing for matrix multiplication (128 and 256)

In Figure 9 we can check the above exposed results and tell that for small matrices is not worth it, but for medium-sized and large matrices is.

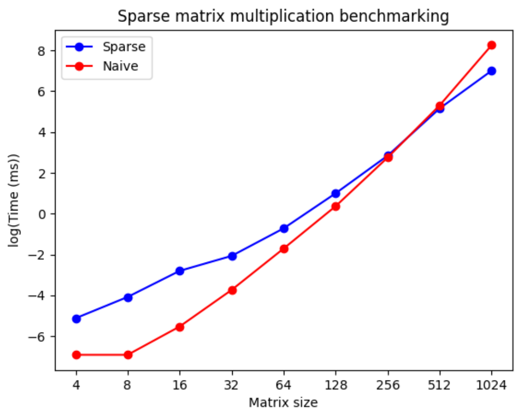

Figure 9: Comparison of naïve and sparse methods. Log scale

3.3 Strassen Matrix Multiplication

Figure 10 shows that for all cases that the strassen method is way slower than the naïve method. This is because of the overhead of the recursive calls and the extra operations that the strassen method has to do. Despite the number of multiplications being reduced from 8 to 7, the overhead of the recursive calls and the extra operations make the strassen method slower than the naïve method.

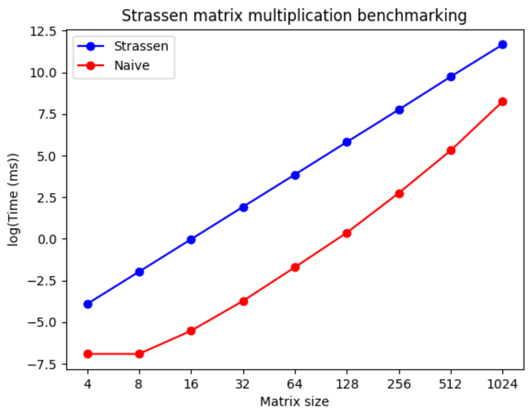

Figure 10: Comparison of naïve and sparse methods. Log scale

3.4 Column-Major Matrix Multiplication

If we reverse the order of the loops, we can see that the cloumn major order is faster than the row major order for small matrix sizes but for larger matrix sizes, the row major order is slightly faster. The difference between the two is not significant, but it is worth to mention that the difference between them for small matrices is bigger than the difference between them for larger matrices. So we could say that if you had to choose between the two, you should choose the column major order for small matrices and the row major order for larger matrices. Figure 11

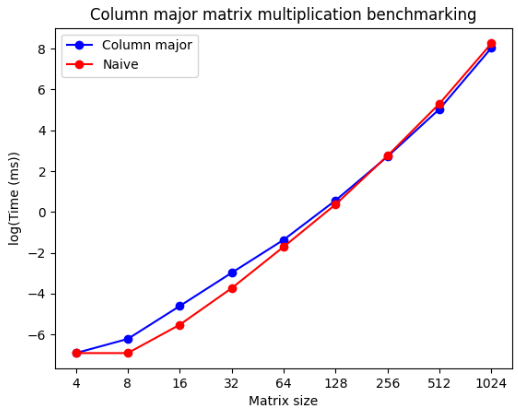

Figure 11: Comparison of naïve and column major order methods. Log scale

4 Conclusions

If the reader take a look on the following graphic (Figure 12) it can be concluded that the Strassen algorithm is the worst over all. And apart from that, we can conclude that:

1. Sometimes the simplest thing is also the best one. For matriz sizes up to 64 or even 128, the naïve method was the one that gave better results. The reason for this is because of the underlying process of each methods, which are extremely expensive and not worth it for small matrices; but not because the methods are not good enoguh.
2. For matrix sizes over 128 the winner is the sparse method (even if the number of zeros tends to 0) but I admit that it may be complicated to implement, so the blocking method would be a better option and still gives a better performance than the näive approach
3. The way the Strassen method is implemented needs an improvement if we want it to work, since in its actual state is useless. We may consider adding a point from which it made the multipliction using the naïve method or maybe try to combine it with threads and take advantage of parallelization

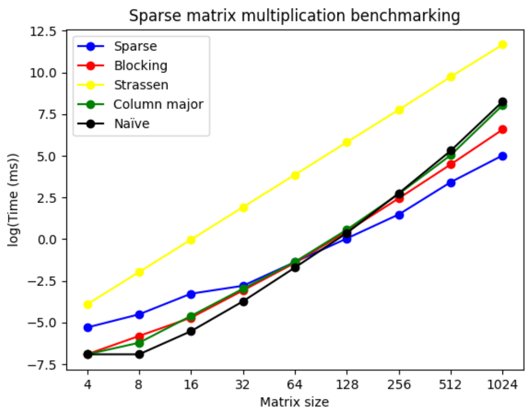

Figure 12: Comparison of all the methods for different matrix sizes. Log scale

5 Future Work

In this stage we have implemented an tested several algorithms that could improve the naïve matrix multiplitation algorithm. However, some of them may be more prone to be used in the real world than others.

For that reason, this work could be expanded by looking into another fields such as parallelization, i.e. using threads to take advantages of the different logical cores residing on our CPU, or vectorization, among other things.

I will never get tired of saying that the world of matrix multiplication is extremely wide and the options to optimize this problem sometimes seems to be endless since new ideas and new approaches arise constantly.

However, vectorization and parallelization seems a good approach for the next stage, in order to keep learning in this fascinating universe of the mtrix multiplication.