Parallel and Distributed Computing

1st Project Report

Guilherme Almeida
João Malva
Nuno Pereira

L.EIC
Faculdade de Engenharia da Universidade do Porto
March 10, 2023

Contents

1 Introduction **1**

2 Setup **1**

3 Algorithm Analysis **1**

3.1 Normal Matrix Multiplication 1

3.2 "Line by Line" Matrix Multiplication 2

3.3 Block Matrix Multiplication 2

4 Performance Analysis **3**

4.1 Data graphics 3

4.1.1 Execution time(s) 3

4.1.2 GFlop/s 4

4.1.3 Level 1 Data Cache Misses 5

4.1.4 Level 2 Data Cache Misses 6

5 Conclusions **7**

A Appendix **8**

A.1 Code 8

1 Introduction

Since matrices represent equation systems, they can encode a solution a lot simpler than performing the calculations one by one. Matrix multiplications build on top of this in that they can represent multiple systems interacting. Due to this fact, the matrix multiplication algorithms used by developers should be fast enough as to not introduce a bottleneck on programs that make use of them, such as graphics simulations or artificial intelligence algorithms.

With this in mind, the proposal of this project is to develop and analyze 3 variants of the matrix multiplication algorithm, with the purpose of measuring their performance against each other and draw valuable conclusions.

The algorithms explored are:

- Normal Matrix Multiplication;
- "Line by Line" Matrix Multiplication;
- Block Matrix Multiplication;

2 Setup

All the code was tested and ran on a Aorus Gigabyte computer hosting EndeavorOS, with version 6.2.2 of the Linux kernel. The CPU used was a Intel®Core™i7-10750H CPU @ 2.60GHz.

The L1 data cache has a size of 192 KiB, whilst the L2 data cache has a size of 1.5 MiB.

3 Algorithm Analysis

The algorithms described were implemented using the C++ programming language, the Java programming language and the Performance API, PAPI for short.

PAPI is used in conjunction with the C++ code because it allows us to directly collect several metrics related to the performance of the algorithms developed. It is not used with the Java code since it runs on the Java Virtual Machine, and therefore the performance registers are virtualised, which inhibits us from collecting their values.

3.1 Normal Matrix Multiplication

This algorithm is the *naïve* implementation of the normal matrix multiplication procedure.

It iterates through each line of the first matrix and each column of the second matrix to perform the dot product between each vector, accumulating the results in the destination matrix.

The algorithm can be (grossly) coded as the following:

Algorithm 1 Naïve Matrix Multiplication Algorithm

```
function OnMult($a, b$)
    $mx\_size \leftarrow Length(a)$                 ▷ $a$ should have the same side length as $b$
    $c \leftarrow matrix[mx\_size][mx\_size]$      ▷ $c$ is initialized with 0s
    for $i \leftarrow 0$ to $mx\_size$ do
        for $j \leftarrow 0$ to $mx\_size$ do
            for $k \leftarrow 0$ to $mx\_size$ do
                $c_{i,j} \leftarrow c_{i,j} + a_{i,k} * b_{k,j}$
            end for
        end for
    end for
    return $c$
end function
```

3.2 "Line by Line" Matrix Multiplication

This algorithm is an improvement over the normal matrix multiplication algorithm.
It exploits the properties of cache locality to speed up the matrix product computations.
The algorithm can be (grossly) coded as the following:

Algorithm 2 Line Matrix Multiplication Algorithm

```
1: function OnMultLine($a, b$)
2:     $mx\_size \leftarrow Length(a)$                          ▷ $a$ should have the same side length as $b$
3:     $c \leftarrow matrix[mx\_size][mx\_size]$                 ▷ $c$ is initialized with 0s
4:     for $i \leftarrow 0$ to $mx\_size$ do
5:         for $k \leftarrow 0$ to $mx\_size$ do
6:             for $j \leftarrow 0$ to $mx\_size$ do
7:                 $c_{i,j} \leftarrow c_{i,j} + a_{i,k} * b_{k,j}$
8:             end for
9:         end for
10:    end for
11:    return $c$
12: end function
```

The main difference between this algorithm and the previous one is that, by changing the order of iteration of `j` and `k`, the processor does not need to load a new set of data into cache due to the multiplication order following the second matrix's lines, thus reducing drastically the number of cache misses resulting from the *naïve* implementation.

3.3 Block Matrix Multiplication

This reasoning behind this algorithm is to "divide and conquer": by dividing the original matrix into many sub-matrices and performing the multiplication on these, the process is sped up.

The algorithm used with each sub-matrix is the "Line by Line" one since it offers better performance over the *naïve* implementation. The algorithm can be coded like the following:

Algorithm 3 Block x Block Matrix Multiplication Algorithm

```
1: function OnMultBlock($a, b, bkSize$)
2:     $mx_{size} \leftarrow Length(a)$                          ▷ $a$ should have the same side length as $b$
3:     $c \leftarrow matrix[mx_{size}][mx_{size}]$                 ▷ $c$ is initialized with 0s
4:     $sideLenBlocks \leftarrow mx_{size}/bkSize$
5:     $block_a \leftarrow matrix[bkSize][bkSize]$
6:     $block_b \leftarrow matrix[bkSize][bkSize]$
7:     for $ii \leftarrow 0$ to $sideLenBlocks$ do
8:         for $jj \leftarrow 0$ to $sideLenBlocks$ do
9:             for $kk \leftarrow 0$ to $sideLenBlocks$ do
10:                for $i \leftarrow ii \cdot bkSize$ to $(ii + 1) \cdot bkSize$ do
11:                    for $k \leftarrow kk \cdot bkSize$ to $(kk + 1) \cdot bkSize$ do
12:                        for $j \leftarrow jj \cdot bkSize$ to $(jj + 1) \cdot bkSize$ do
13:                            $c_{i,j} \leftarrow c_{i,j} + a_{i,k} \cdot b_{k,j}$
14:                        end for
15:                    end for
16:                end for
17:            end for
18:        end for
19:    end for
20:    return $c$
21: end function
```

Note: We have implemented this in a different way, by first computing the blocks being multiplied and then calculating the resulting block, which might have a little impact on the overall performance of the algorithm, but we decided to do it for readability purposes.

This algorithm also has another input parameter: the block size. This value controls the size of the resulting sub-matrices. The impact this parameter has on the overall performance of the algorithm is discussed bellow.

Contrary to the other two, this algorithm was developed only in the C++ programming language.

4 Performance Analysis

The performance metrics collected to analyze the developed algorithms were:

- The execution time;
- The Level 1 Data Cache misses;
- The Level 2 Data Cache misses;

These metrics give us an overview of the behavior of the algorithms and allow us to derive another important metric: Gflops/s. This value indicates how many floating point operations were performed per second: (typically) the higher the value the more operations were performed and the faster the algorithm was.

4.1 Data graphics

The following graphics depict the differences between the two languages used across the various metrics measured:

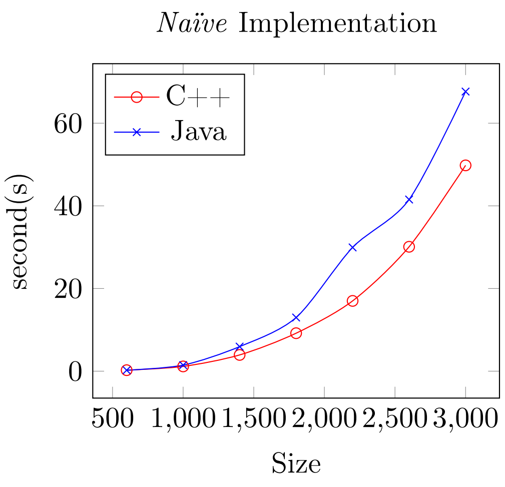
4.1.1 Execution time(s)

Since C++ compiles to native machine code, it was expected that the C++ code would run faster than the Java code: that hypothesis is confirmed by our empirical results.

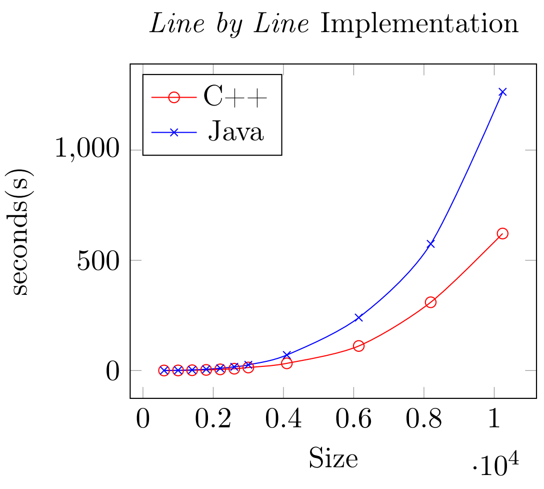
It is also worth noting the exponential increase in the execution time of the algorithms with the increase of the matrix size : this is related to the complexity of these algorithms ($O(n^3)$).

Other than that, as expected, the *Line by Line* algorithm presents a much flatter curve with respect to the *naïve* one.

Surprisingly, however, are the results of the Block multiplication algorithm: the runtimes for different block sizes are similar, which indicates that this metric does not have a big impact in the overall performance of the algorithm (at least in the way it was coded by us).



(a) Run time for the *naïve* algorithm (C++/Java)



(b) Run time for the *line by line* algorithm (C++/Java)

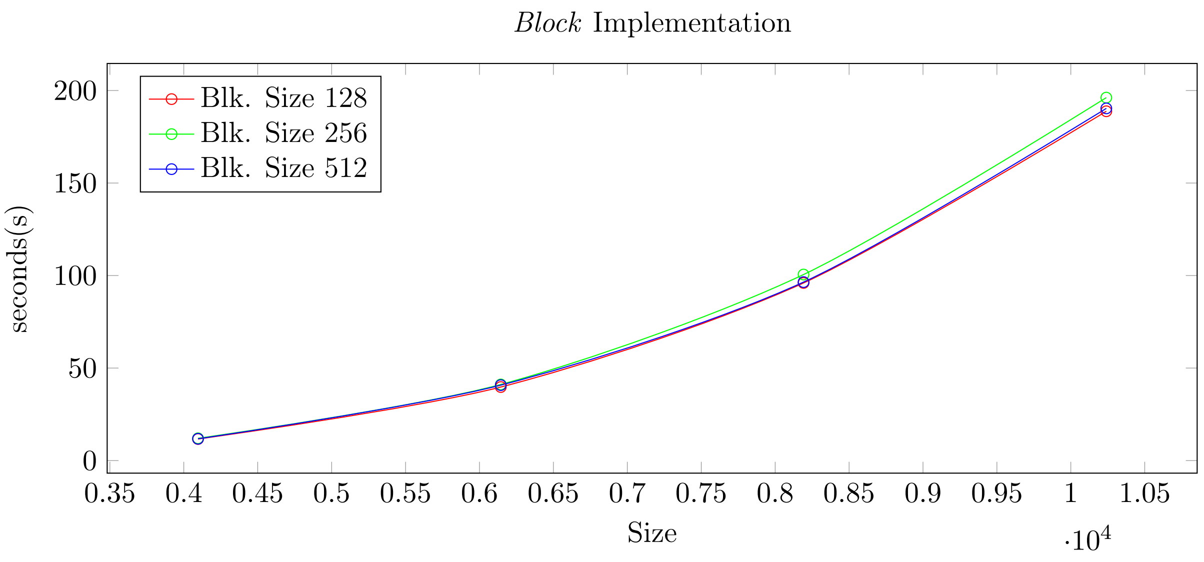
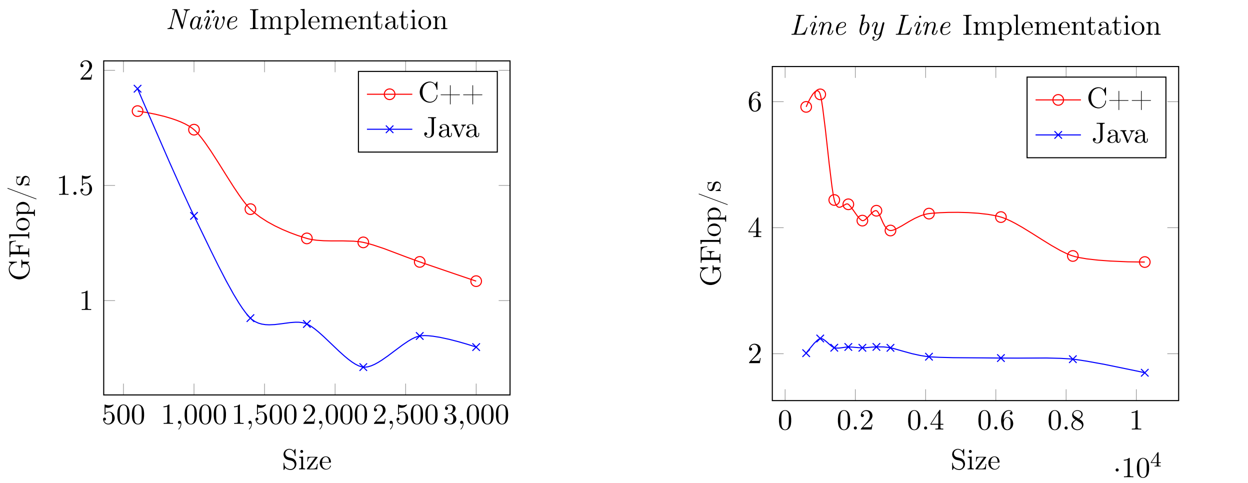


Figure 2: Run time for the *block* algorithm (C++)

4.1.2 GFlop/s

As expected, the C++ code shows an increased amount of floating point operations per second when compared to the Java one (directly related to the difference in execution times). However, it is interesting to note that, for a small enough matrix size, for the *naïve* implementation, the Java implementation outperforms the C++ one (this could not be noticed in the execution time graph).

Additionally, even though the various block sizes produced little variation in the execution times of the algorithms, the same cannot be said for the GFLOP/s, with the bigger amount being produced by the smaller block size.



(a) GFlop/s for the *naïve* algorithm (C++/Java)

(b) GFlop/s for the *line by line* algorithm (C++/Java)

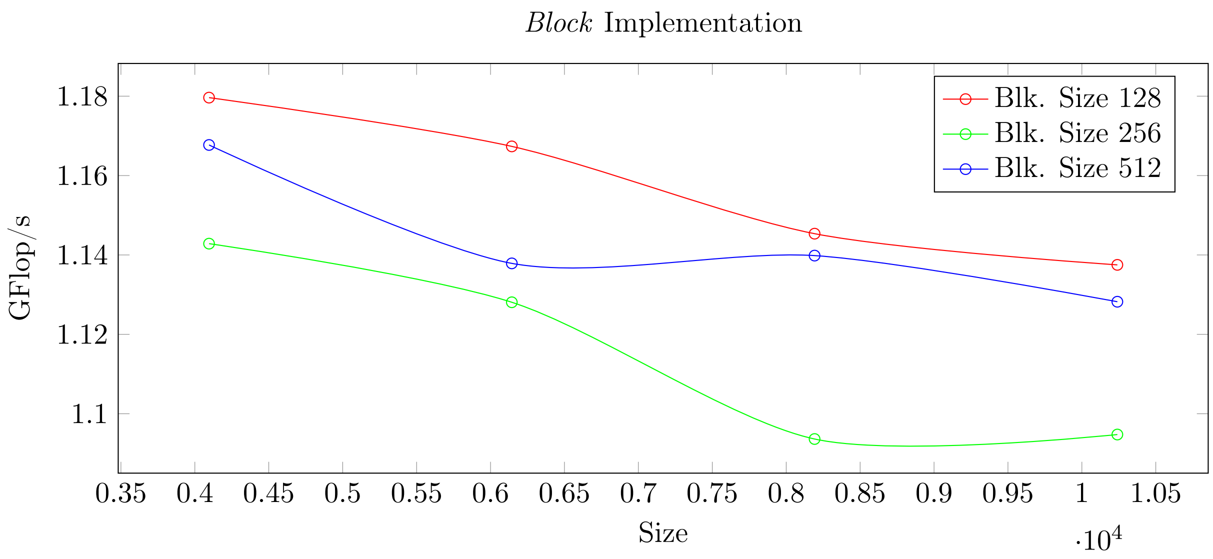
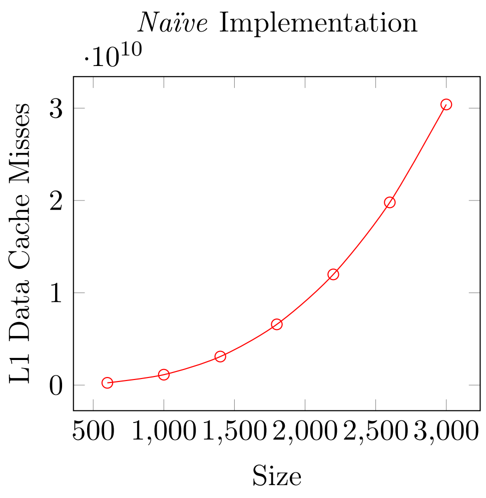


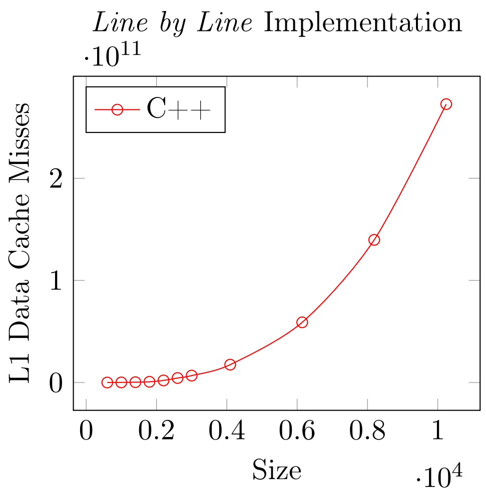
Figure 4: GFlop/s for the *block* algorithm (C++)

4.1.3 Level 1 Data Cache Misses

The 3 different block sizes produced little to no variation in the amount of cache misses detected, which indicates that the use of the L1 Data cache cannot accommodate for the frequency of data accesses.



(a) L1 Data Cache Misses for the *naïve* algorithm



(b) L1 Data Cache Misses for the *line by line* algorithm

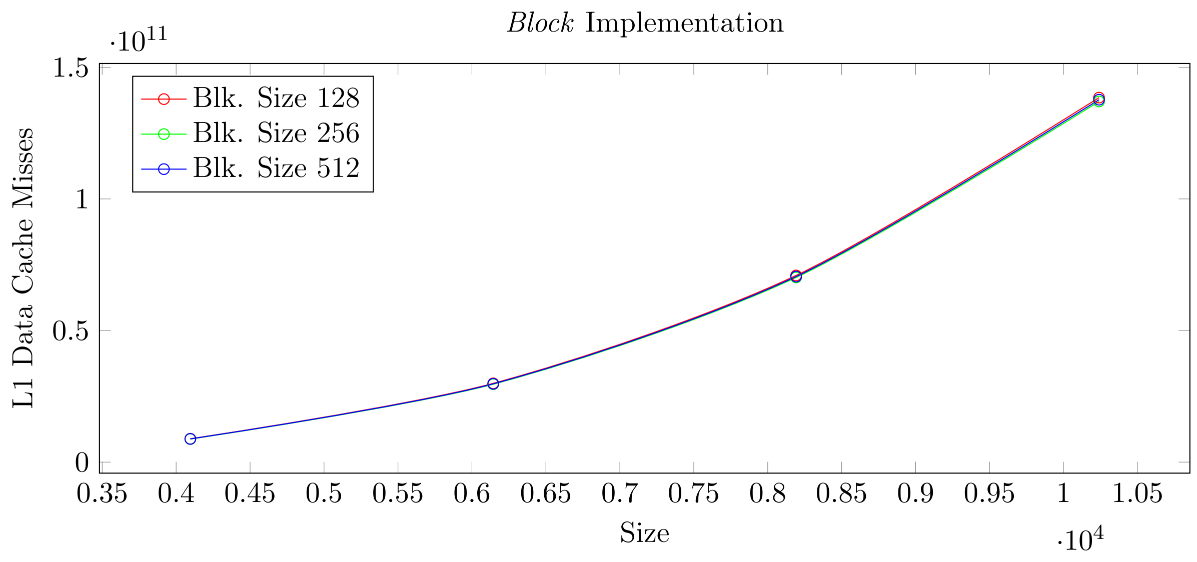
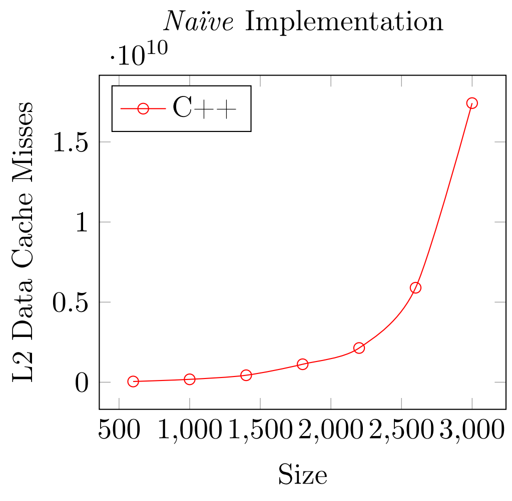


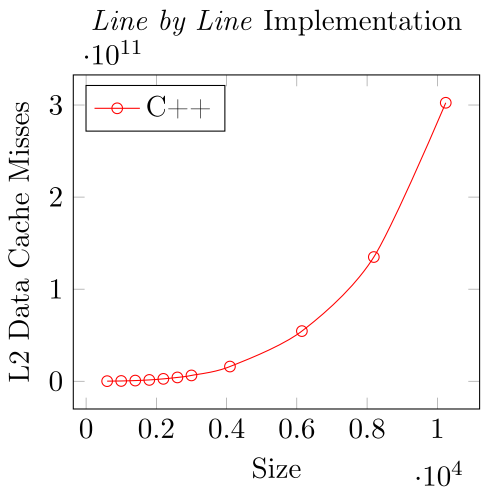
Figure 6: L1 Data Cache Misses for the *block* algorithm

4.1.4 Level 2 Data Cache Misses

Contrary to the previous section, we see a near $2x$ improvement between the biggest and the smallest block size tested. This makes sense since the bigger block size loads more data into the L2 Data cache, which is more readily available when it is needed than with smaller sized blocks.



(a) L2 Data Cache Misses for the *naïve* algorithm



(b) L2 Data Cache Misses for the *line by line* algorithm

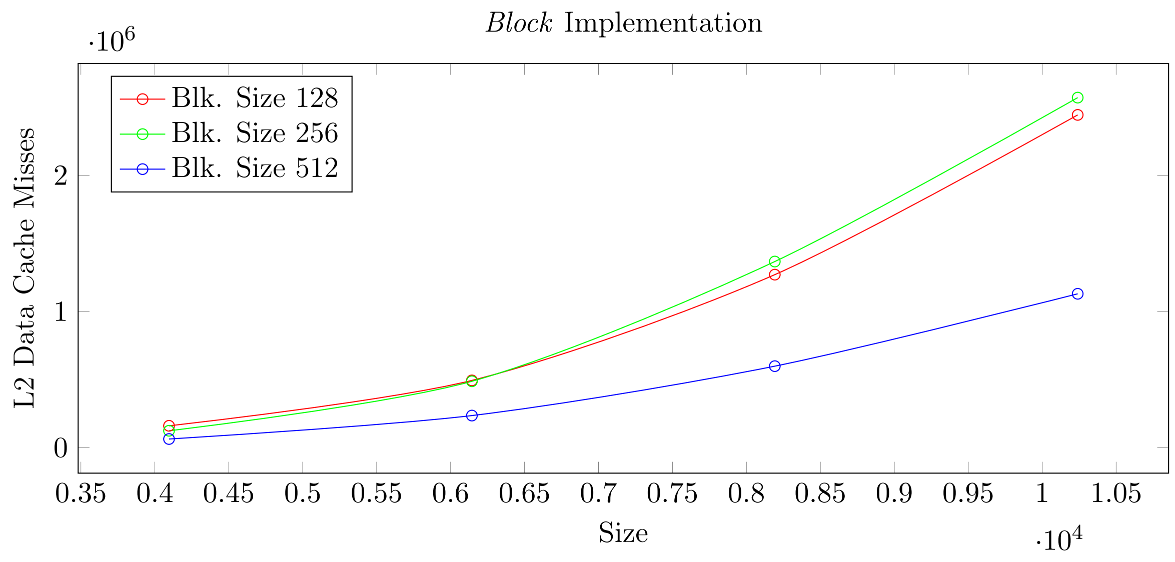


Figure 8: L2 Data Cache Misses for the *block* algorithm

5 Conclusions

In this project, we implemented and analyzed three different algorithms for matrix multiplication: Normal Matrix Multiplication, Line-by-Line Matrix Multiplication, and Block Matrix Multiplication. We used C++ and Java programming languages and the PAPI Performance API for C++ code to collect performance data.

The performance data collected from PAPI Performance API shows that Block Matrix Multiplication is the fastest algorithm among the three, followed by Line-by-Line Matrix Multiplication, and lastly, Normal Matrix Multiplication. This is consistent with our expectations as Block Matrix Multiplication is optimized for cache utilization, and cache hits are faster than memory accesses, which are utilized by the other algorithms.

In conclusion, the Block Matrix Multiplication algorithm is the most efficient method for matrix multiplication, and it can be used to optimize various applications that require matrix multiplication.

Additionally, we successfully collaborated on this project, with each member contributing equally to the project's success.

The following table shows the participation percentage of each team member:

Member	Participation
Guilherme Almeida	33.3 %
João Malva	33.3 %
Nuno Pereira	33.3 %

A Appendix

A.1 Code

Versions of the algorithms in C++/Java are available in folder `src/`.