U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Bachelor in Informatics and Computing Engineering
Information

Performance evaluation of a single core

Parallel and Distributed Computing

David Preda - up201904726
Fernando Rego - up201905951
Miguel Amorim - up201907756

March 28, 2022

Contents

Chapter 1

Problem Description

The memory system is a hierarchy of storage devices with different capacities, costs, and access times. This hierarchy can be abstracted as a triangle, where the bottom represents the cheaper storage devices with a larger but slower amount of memory. On the other hand, the higher levels of the triangle represent storage devices with small capacities but with a much faster access time.

Memory hierarchies perform as well as they due to caching. The faster levels of memory keep recently used values in order to enable faster access to them, which drastically decreases the time spent fetching data from memory. On top of this, this organization pattern allows to diminish costs, as the larger part of memory can keep being slower with much less impact on performance.

This project aims to study the effect of the memory hierarchy on the processor performance when accessing large amounts of data. Three different algorithms for the product of two matrices will be approached, given that this problem usually deals with many memory accesses.

Finally, during the execution of each of the algorithms, several metrics will be collected, so conclusions can be made at the end of this report.

Chapter 2

Algorithms Explanation

The following algorithms all solve the problem of multiplying two matrices. However, each algorithm presented is intended to have memory access have a lesser effect on performance than the previous ones.

2.1 Dot Product Algorithm

The simplest algorith comes directly from the math definition of matrix multiplication, $C = A \cdot B$. Each line of the first matrix is multiplied by each column of the second matrix. Extending the math definition, we can obtain:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$$

From the aforementioned, a simple algorithm can be developed using nested loops over the indices i, j and k to attain the product matrix:

Algorithm 1 Matrix Multiplication

```
1: $A \leftarrow$ First Matrix; $B \leftarrow$ Second Matrix;
2: for $i = 0, 1, \ldots, size(A)$ do
3:     for $j = 0, 1, \ldots, size(B)$ do
4:         $temp \leftarrow 0$;
5:         for $k = 0, 1, \ldots, size(A)$ do
6:             $temp += A_{ik} \cdot B_{kj}$
7:         end for
8:         $C_{ij} += temp$
9:     end for
10: end for
11: return $C$;
```

2.2 Line Matrix Multiplication

The line matrix multiplication algorithm is very similar to the first algorithm. The only difference is that, instead of multiply one line of the first matrix by each column of the second matrix, this version multiplies one single element from the first matrix by the correspondent line of the second matrix.

To program this algorithm, it is possible to depart from the previous one and change the order of the nested for loops. The following pseudocode makes this more explicit:

Algorithm 2 Line Matrix Multiplication

```
1: $A \leftarrow$ First Matrix; $B \leftarrow$ Second Matrix;
2: $C \leftarrow$ Matrix initialized with 0's;
3: for $i = 0, 1, \ldots, size(A)$ do
4:     for $k = 0, 1, \ldots, size(A)$ do
5:         for $j = 0, 1, \ldots, size(B)$ do
6:             $C_{ij} += A_{ik} \cdot B_{kj}$
7:         end for
8:     end for
9: end for
10: return $C$;
```

2.3 Block Matrix Multiplication

The block matrix multiplication algorithm consists in partitioning the initial matrix into blocks or sub-matrices of a given size. Finally with the partitioned matrix, we can perform the multiplication of the two initial matrix by multiplying the sub-matrices of smaller size.

After dividing the matrices in blocks there is two possible approaches. The first approach consist in the dot product, used in the matrix multiplication algorithm, and the second in the line product, used in the line matrix multiplication algorithm. Our team opted to use the line product since this approach have substantial gains when compared to the first. The algorithm can be represented with the following pseudocode:

Algorithm 3 Line Matrix Multiplication

```
1: $A \leftarrow$ First Matrix; $B \leftarrow$ Second Matrix;
2: $C \leftarrow$ Matrix initialized with 0's;
3: $N \leftarrow size(A)$;
4: $b \leftarrow$ Block size;
5: for $ii = 0, b, \ldots, N$ do
6:     for $jj = 0, b, \ldots, N$ do
7:         for $kk = 0, b, \ldots, N$ do
8:             for $i = ii, ii + 1, \ldots, ii + b$ do
9:                 for $j = jj, jj + 1, \ldots, jj + b$ do
10:                    for $k = kk, kk + 1, \ldots, kk + b$ do
11:                        $C_{ik} += A_{ij} \cdot B_{jk}$
12:                    end for
13:                end for
14:            end for
15:        end for
16:    end for
17: end for
18: return $C$;
```

Chapter 3

Results and Analysis

As requested for this project, the dot product and the line multiplication algorithms were implemented in two languages - C++ and Rust. The block multiplication algorithm was only implemented in C++.

It is worth noting that the Rust version is using a slightly less efficient implementation than in the C++ one. However, given that the aim of the project is to analyse the effects of memory access on the performance of a single core and not the direct comparison of both languages, the authors believe that this is not a major issue.

3.1 Performance Metrics

In order to correctly evaluate performance and the correlation between memory access and execution time, the *Performance Application Programming Interface* (or *PAPI*, for short) was used to collect the number of data cache misses on the L1 and L2 cache level.

The authors original intention was to measure the percentage of data cache misses relative to the total of data cache accesses. However, due to hardware limitations, *PAPI* wasn't able to register these values. Although this metric could have been of more value to the study, but, sadly, it was not possible to present them.

Regardless, the execution time for each of the algorithms was also registered, without the use of any external API.

3.2 Testing Hardware

CPU - Intel i5-9300H (8) @ 4.100GHz

RAM - 20Gb DDR4 Synchronous 2667 MHz

CACHE L1 - 256KiB

CACHE L2 - 1MiB

3.3 Algorithm Comparison

As a way of avoiding any possible misunderstanding, the reader should consider that whenever *cache* is mentioned, the authors mean, specifically, the data cache. Whenever another type of cache, such as the instructions cache, is being mentioned, it should be explicitly clarified.

3.3.1 Dot Product vs Line Multiplication

Let us start our analysis by comparing the cache misses at both the L1 and L2 levels.

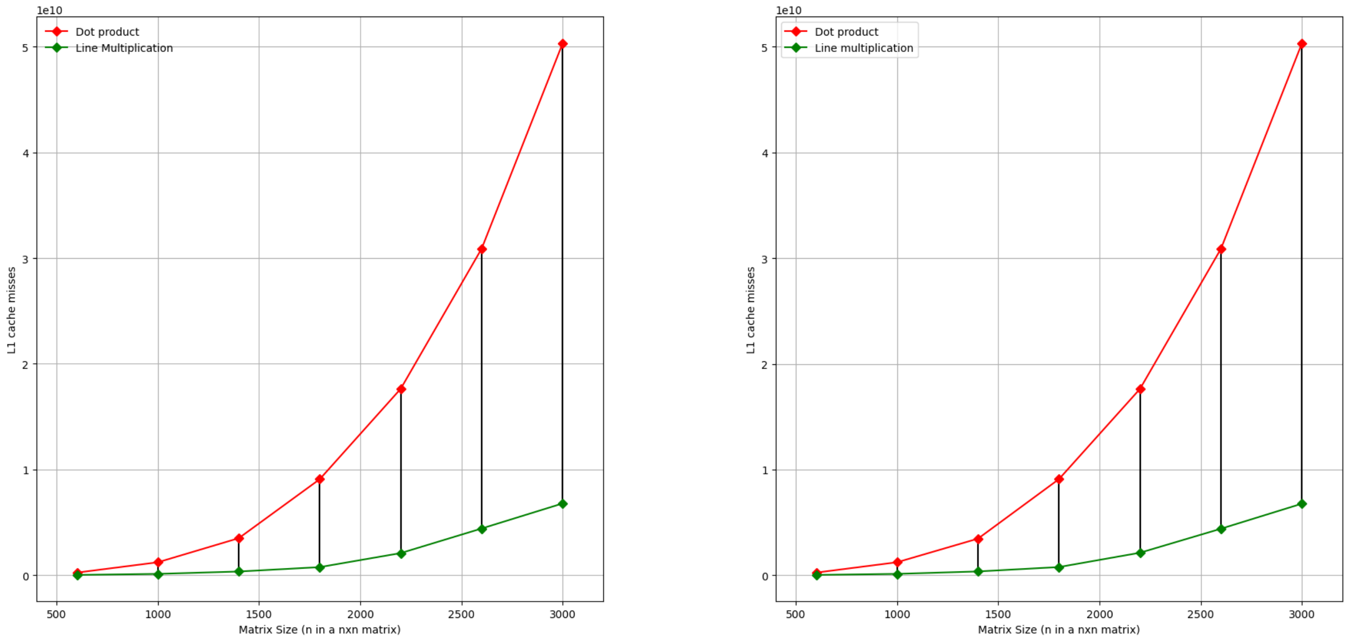


Figure 3.1: Comparison of the number of L1 data cache misses of the dot product and line multiplication algorithms for matrices of increasing size. To the left: C++ implementation; to the right: Rust implementation.

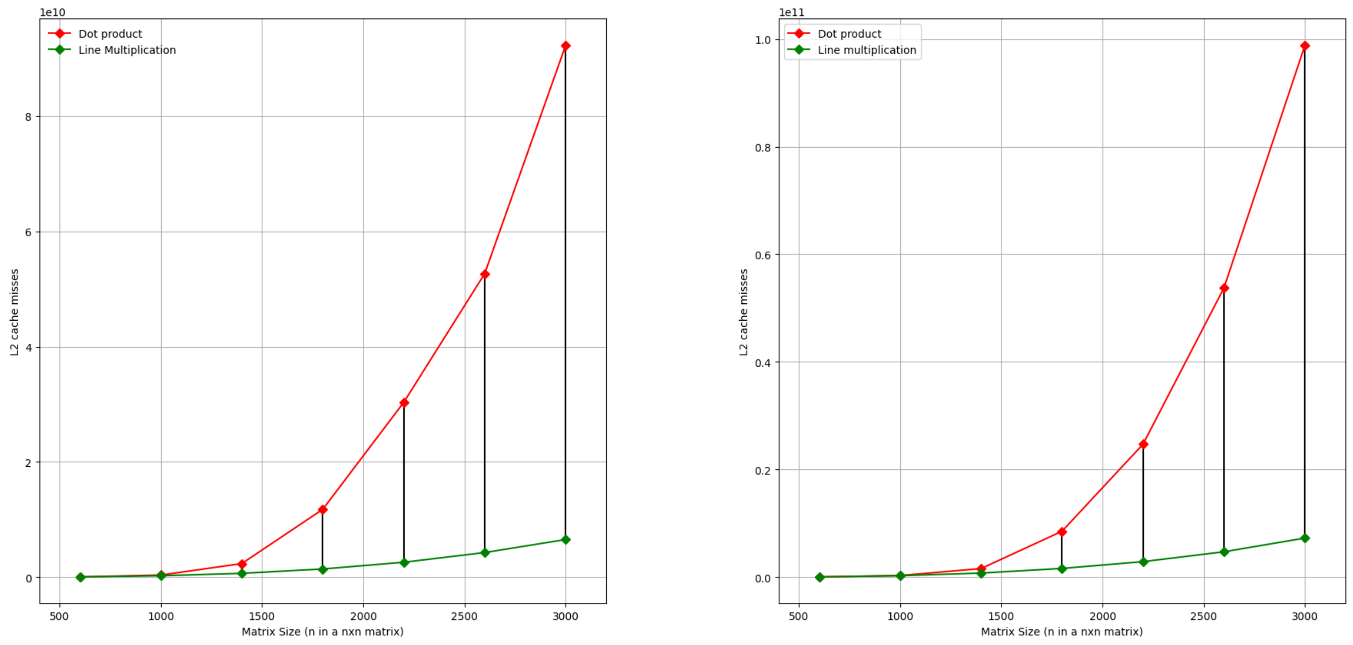


Figure 3.2: Comparison of the number of L2 data cache misses of the dot product and line multiplication algorithms for matrices of increasing size. To the left: C++ implementation; to the right: Rust implementation.

As it is pretty straightforward to see, at both level of caches there is a significant reduction on the number of cache misses from the dot product to the line multiplication algorithm.

In theory, this should mean a reduced execution time for the line multiplication algorithm, given that it does not have to go through the overhead of higher-level memory access as often as the dot product algorithm.

Let us now compare the execution time of both algorithms:

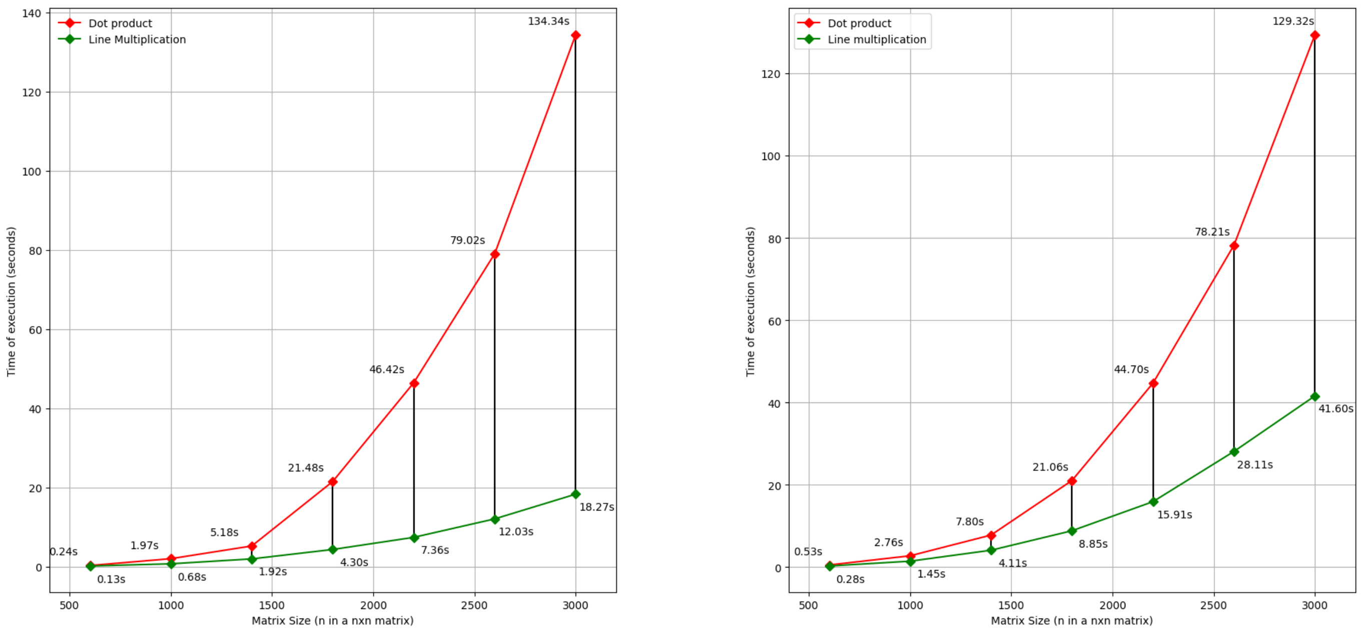


Figure 3.3: Comparison of execution time of the dot product and line multiplication algorithms for matrices of increasing size. To the left: C++ implementation; to the right: Rust implementation.

As expected, the line multiplication algorithm is able to perform all of its operations in a much smaller time, going from 134.34 seconds to only 18.27 seconds for 3000x3000 matrices.

3.3.2 Block Multiplication with Different Sizes

For the block multiplication algorithm, let us now analyse the metrics inversely - starting with the execution time:

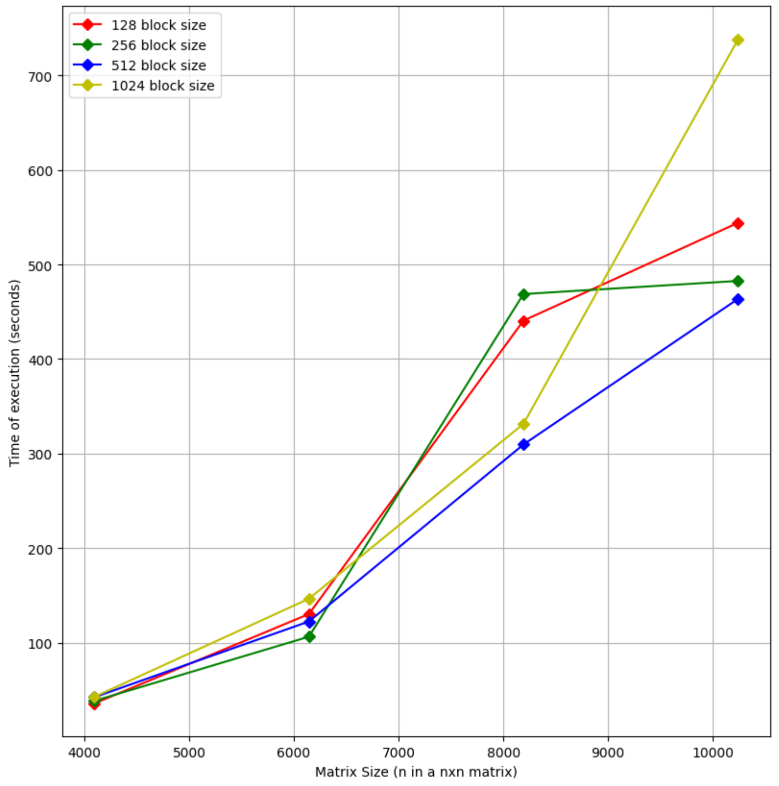


Figure 3.4: Execution time of the block multiplication algorithm for increasing block and matrix sizes.

From the figure, one can conclude that, on average, the 512 block size version of the algorithm performs better than the others.

With this in mind, let us now analyse the cache misses per millisecond:

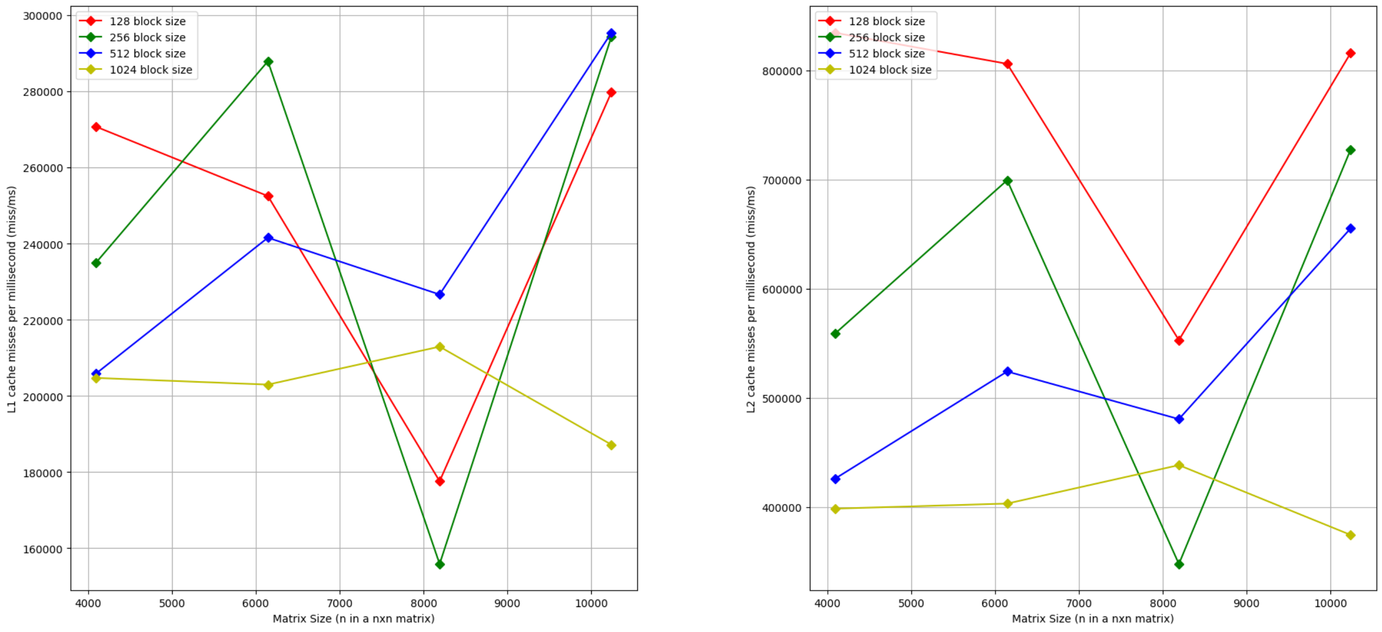


Figure 3.5: Comparison of data cache misses of the block multiplication for increasing block and matrix sizes. To the left: L1 cache misses per millisecond; to the right: L2 cache misses per millisecond.

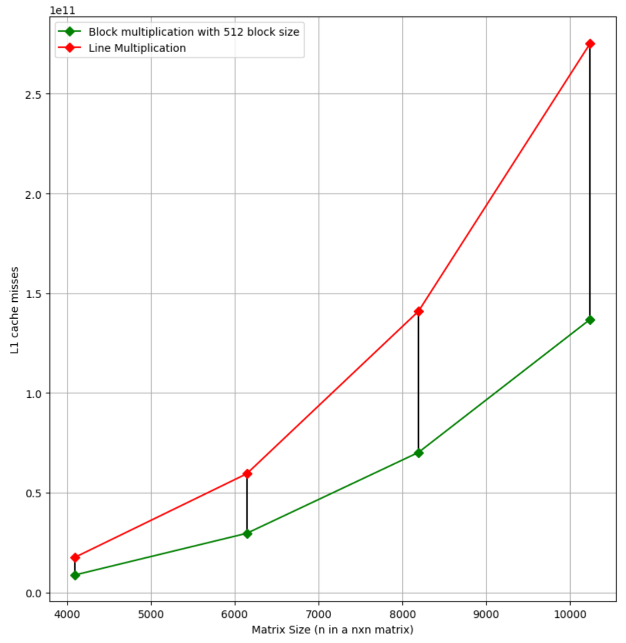
Contrary to the expected, the 512 block size version does not have a lower cache miss per millisecond rate. Therefore, it is not possible to relate the better performance with the accesses to memory.

However, after some research, the authors concluded that the better performance could be related to computational intensity. Bundling together arithmetic operations can lead to faster performance, depending on the cache size of the underlying hardware and the load of the operations. [1]

3.3.3 Line Multiplication vs Block Multiplication

Finally, let us compare the line multiplication algorithm with block multiplication algorithm. As it performed better, the 512 block size version will be the one used to represent the block approach.

As in the first comparison, let's first verify how both algorithms behave cache-wise:



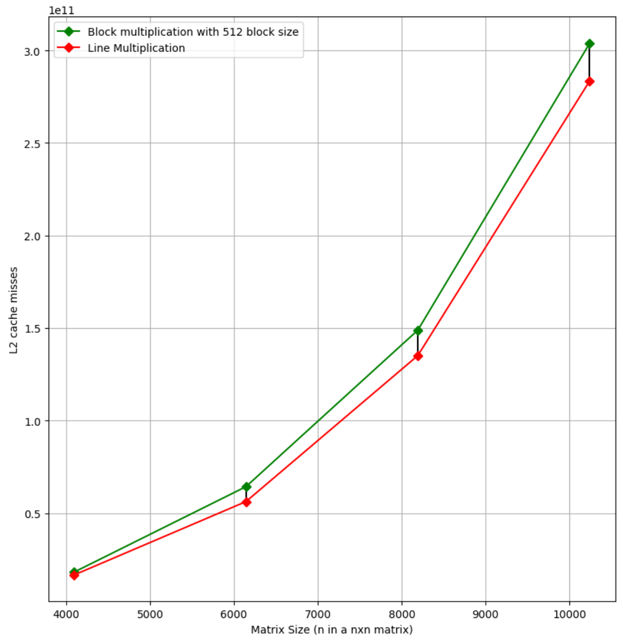


Figure 3.6: Comparison of data cache misses between the line multiplication algorithm and the block multiplication algorithm for increasing matrices. To the left: Total L1 data cache misses; to the right: Total L2 data cache misses.

The total L1 data cache misses are drastically smaller for the block multiplication algorithm. However, they are almost on par at the L2 cache level, with the line multiplication having less misses.

Let us now see how if there is any correlation with the execution time:

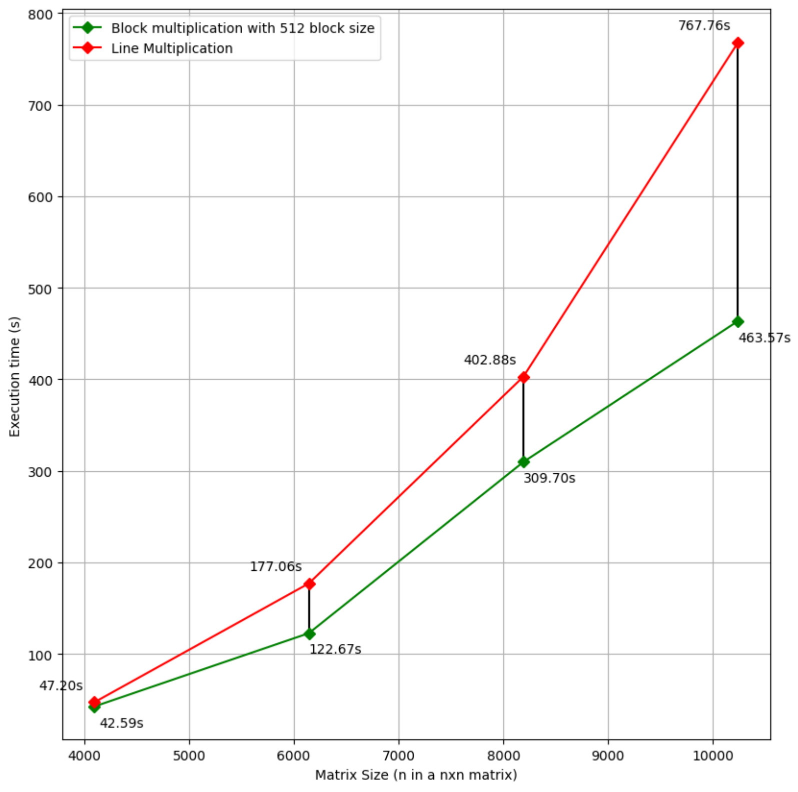


Figure 3.7: Comparison of execution time between the line multiplication algorithm and block multiplication algorithm for increasing matrices.

The block multiplication algorithm, as the matrices increase in size, starts to take less and less time, when comparing it with the line multiplication algorithm. This difference is analogous to what is seen on the graph comparing the L1 data cache misses between the two algorithms.

We, therefore, conclude that the block approach leads to a better performance due to being less heavy on higher-level memory access.

3.4 Language Comparison

Finally, a comparison between both languages is now due. The line multiplication algorithm will be used as the basis for this comparison, as it is the best performing one that is implemented in both language.

Let us start by comparing data cache misses:

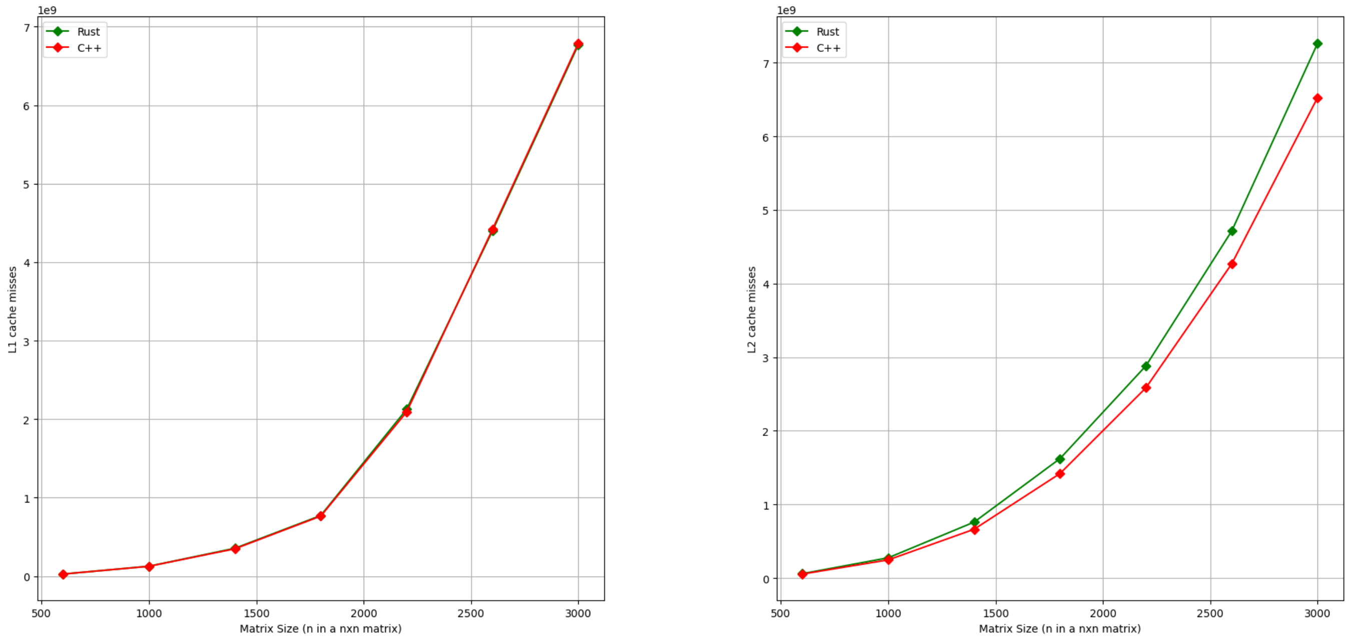


Figure 3.8: Comparison of data cache misses of the line multiplication algorithm between the Rust and C++ implementations for increasing matrices. To the left: Total L1 data cache misses; to the right: Total L2 data cache misses.

For the L1 level, the languages are essentially tied. However, Rust performs slightly worse at the L2 level.

The graph below shows that this slight difference translates into a better performance for the C++ implementation. However, it should also be taken into consideration that the implementations are not entirely similar, with structures with more overhead being used on the Rust implementation.

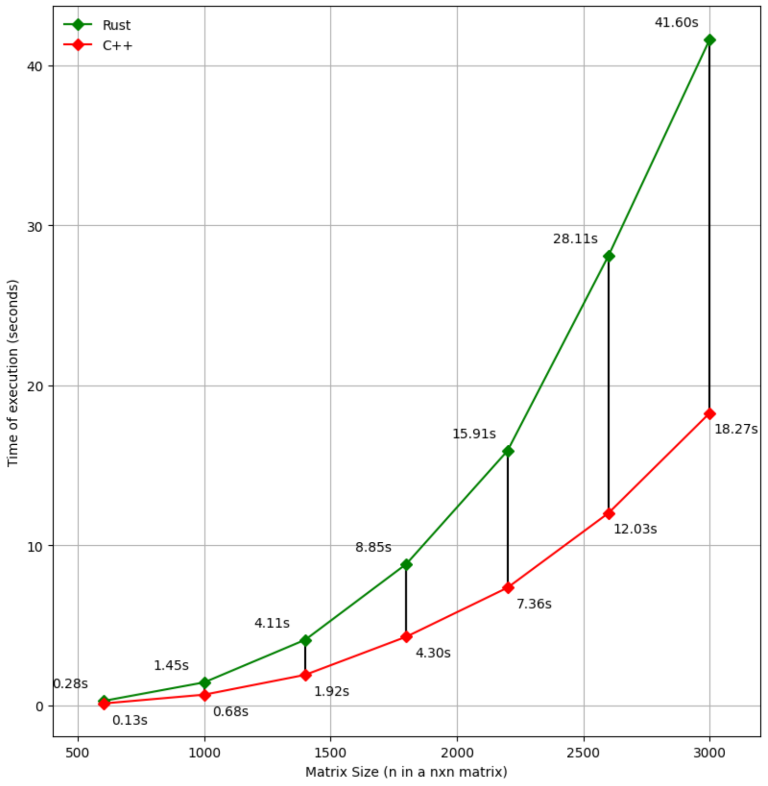


Figure 3.9: Comparison of the execution time of the line multiplication algorithm between the Rust and C++ implementations for increasing matrices.

It can be concluded that C++ performed better, even though this wasn't the fairest of comparisons.

Chapter 4

Conclusions

This project allowed us to deeply understand some computer concepts as memory hierarchy and its performance with a lot of memory accesses.

The creation and application of the three different algorithms for the product of two matrices made us acknowledge the effect of the memory hierarchy on the processor performance and the importance of being mindful about memory access when developing software.

Bibliography

[1] JAYAWEERA, Malith *Blocked Matrix Multiplication*
`https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/`