Performance Evaluation Of A Single Core

Final Report
CPD 2022/23

T04G13

Diogo Babo (up202004950@up.pt)

Gustavo Costa (up202004187@up.pt)

João Oliveira (up202004407@up.pt)

José Araújo (up202007921@up.pt)

Index

1 - Introduction

The aim of this project is to implement different algorithms for multiplying matrices and evaluate the performance of a single CPU core. The programs were developed in **three** different programming languages: **C++**, **Go** and **JavaScript**.

This report will, not only, focus on the specification of the implemented algorithms but also on the analysis and explanation of the results.

2 - Problem Description

The matrix multiplication problem is an ideal case study since it deals with large amounts of data, making it optimal for us to study the effect of memory hierarchy on the processor performance.

In order to study this we were proposed to implement, in two different languages, three algorithms: Standard matrix multiplication (a.k.a ijk algorithm), Line multiplication (a.k.a ikj algorithm) and Block multiplication. Since our group is composed of 4 members, we decided to implement them in three different languages: C++, Go and JavaScript

Throughout the development and execution of the algorithms, different measures were taken. While for the C++ we used different **PAPI** metrics to evaluate the CPU performance, on the other two languages we just measured the time taken for the execution of different algorithms and matrix sizes.

The project was divided into 3 parts (we extended all parts to all languages):

1. Implementation of the standard matrix multiplication algorithm and measuring the desired metrics for matrices ranging from 600x600 to 3000x3000 in size, with increments of 400x400.
2. Implementation of the line multiplication algorithm and measuring the desired metrics for matrices ranging from 600x600 to 3000x3000 in size, with increments of 400x400, and from 4096x4096 to 10240x10240, with increments of 2048x2048.
3. Implementation of the block multiplication algorithm with line multiplication and measuring the desired metrics for matrices ranging from 4096x4096 to 10240x10240, with increments of 2048x2048 for block sizes of 128, 256, 512 and 1024.

3 - Algorithms Explanation

3.1 - Standard Multiplication

This algorithm was initially provided in C++ along with the project description, and simply consists of multiplying each row of matrix A by each column of matrix B. It was also implemented in Go and JavaScript.

Time Complexity: $O(N^3)$, where N is the line/column size of the square matrix
Space Complexity: $O(N^2)$

3.2 - Line Multiplication

This algorithm is a more efficient implementation of the matrix multiplication algorithm presented above. Despite appearing similar to the previous one, now we multiply one single element from the first matrix by the respective line in the second matrix. We achieve this by changing the loops' variant order.

It's more efficient because it takes advantage of how the values are stored in memory. While on the first algorithm the values are stored in column-major order, on this algorithm the values are stored in a row-major order, meaning that the elements in the same row will be stored consecutively in memory.

Also, since one of the array lookups the ijk algorithm does no longer varies in value in the third loop, the compiler can optimize the ikj algorithm by storing the value that is looked up in a temporary variable, saving time.

Time Complexity: $O(N^3)$
Space Complexity: $O(N^2)$

3.3 - Block Multiplication

The idea of this algorithm consists of splitting a matrix into blocks (smaller matrices) and using them to compute the matrix multiplication. By subdividing the matrices into smaller blocks, we can reduce the time needed to multiply them, as smaller blocks can be moved into the fast local memory and their elements can then be repeatedly used. (To multiply the submatrices the line algorithm is used).

This results in a great reuse of data, thus increasing performance and reducing runtime when compared to the other matrix multiplication algorithms specified above (3.1 and 3.2).

In the previous algorithm if the line is large enough it might not fit in the processor's cache, therefore resulting in multiple cache misses. By splitting the matrix into smaller portions of it we can decrease the cache misses and consequently improve the execution time.

Time Complexity: $O(N^3 / B^3 \times B^3) = O(N^3)$, where B is the block size
Space Complexity: $O(N^2)$

4 - Performance Metrics

The performance API (PAPI) has several performance metrics that can be used to measure CPU and Cache performance. It is important to note that each CPU has support for different metrics and, in our case, the metrics supported for the Ryzen 9 5900HS that would fit the theme of this project better, since it's heavily related with cache operations were:

- PAPI_L1_DCM: L1 data cache misses
- PAPI_L1_DCA: L1 data cache accesses
- PAPI_L2_DCM: L2 data cache misses
- PAPI_L2_DCH: L2 data cache hits
- PAPI_FP_OPS: Floating point operations

Sadly, even though PAPI_FP_OPS appeared as an available and supported metric, we could not get it to work, so we stuck with the other four.

5 - Results & Analysis

5.1 - Algorithm Comparison

To adequately compare these algorithms, the execution time and GFlops of each one was plotted over an increasing matrix size, for all of them, and, in order to avoid inconsistencies, every measurement was taken exactly three times. Consequently, the plotting will take in consideration the average of these three values.

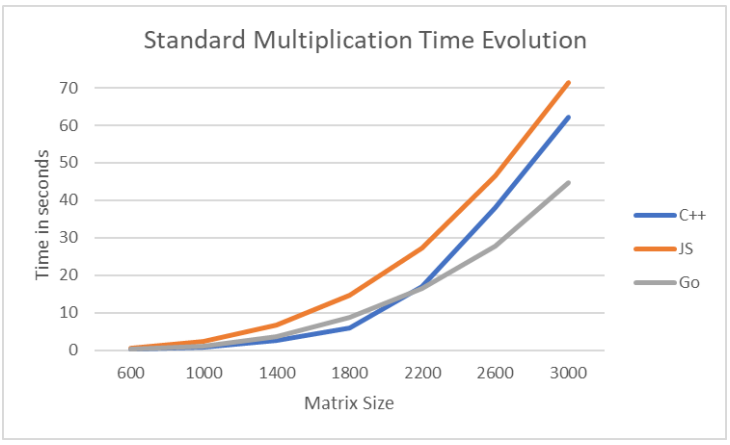

Figure 1.1 - Exec. time (s) for Standard Multiplication

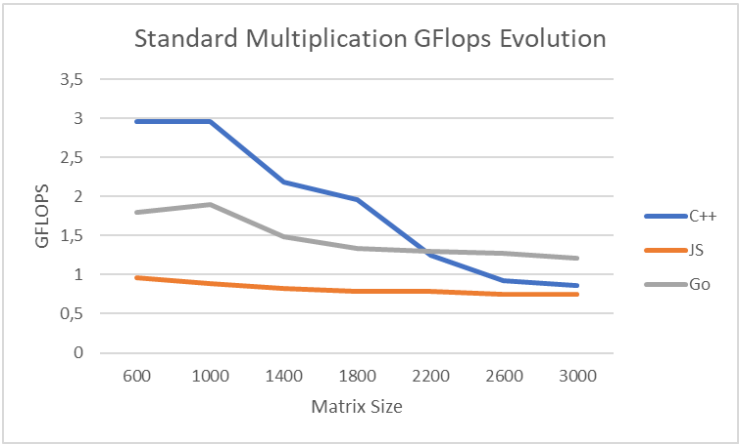

Figure 1.2 - GFlops for Standard Multiplication

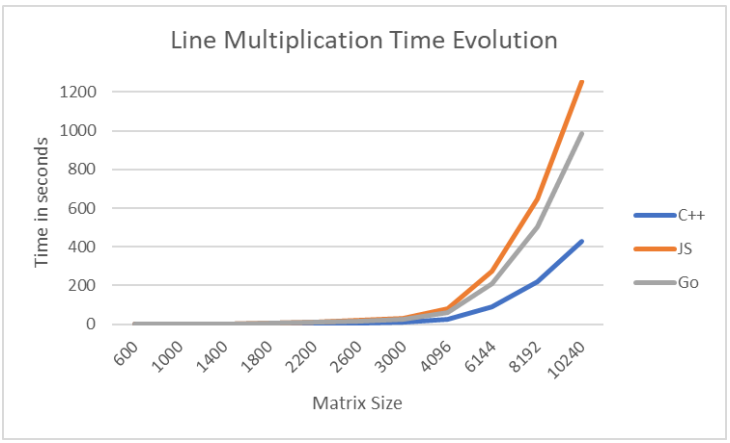

Figure 2.1 - Exec. time (s) for Line Multiplication

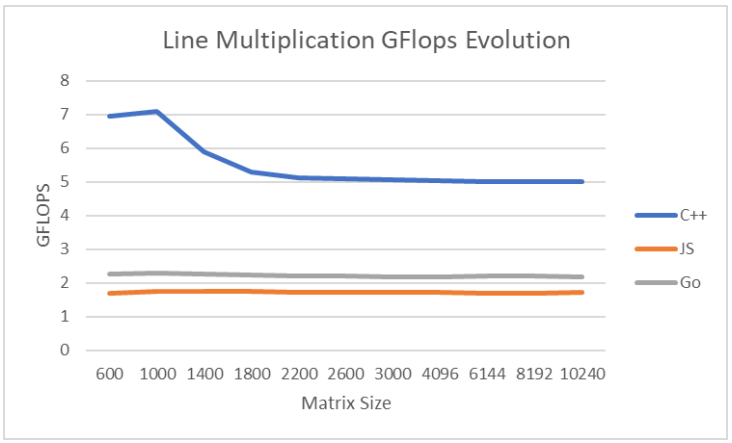

Figure 2.2 - GFlops for Line Multiplication

As the results in Figure 1.1, 1.2, 2.1 and 2.2 show, we can conclude that the Standard Multiplication algorithm has worse performance than the Line Multiplication as it's expected, since the latter is reading from contiguous positions in memory, leading to a lower amount of cache misses.

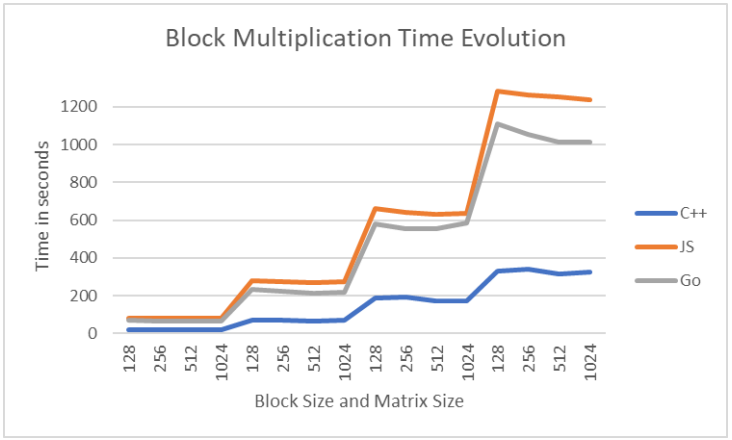

Figure 3.1 - Exec. time (s) for Block Multiplication

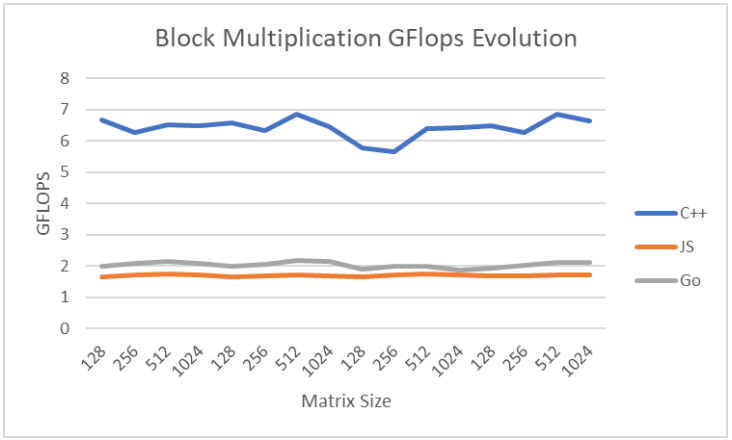

Figure 3.2 - GFlops for Block Multiplication

Also, when comparing Line Multiplication with Block Multiplication, we observe 2 different results: in **C++**, despite the block size, the execution time is lower for the same matrix size, but for both **JavaScript** and **GO**, the block algorithm execution time is either higher or very similar, which seems to contradict the theory.

By analyzing the Block Multiplication algorithm time evolution plot (Figure 3.1), it is noticeable that, for a given matrix size, there's an optimal block size that minimizes execution time. This has to do with the fact that, as block size grows, it becomes closer to the matrix line size being utilized in the Line Multiplication algorithm, which should be slower than the block one. As a result, what we can conclude is that there is an optimal value for block size, a value between 128 and the matrix line size, where the Block Multiplication algorithm excels.

5.2 - PAPI Metrics Comparison

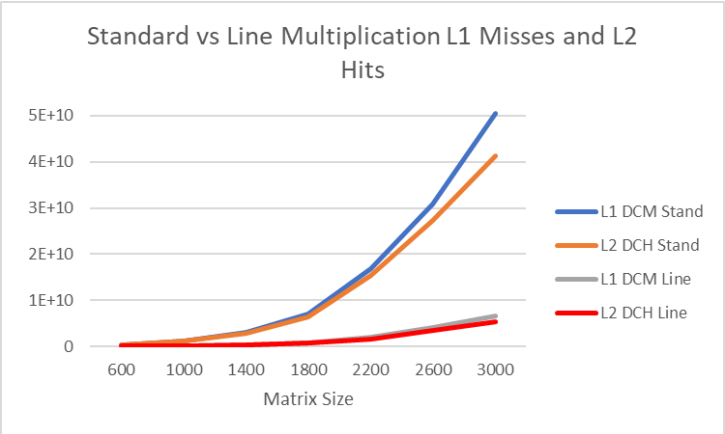

Figure 4 - L1 Misses and L2 Hits Standard/Line Multiplication

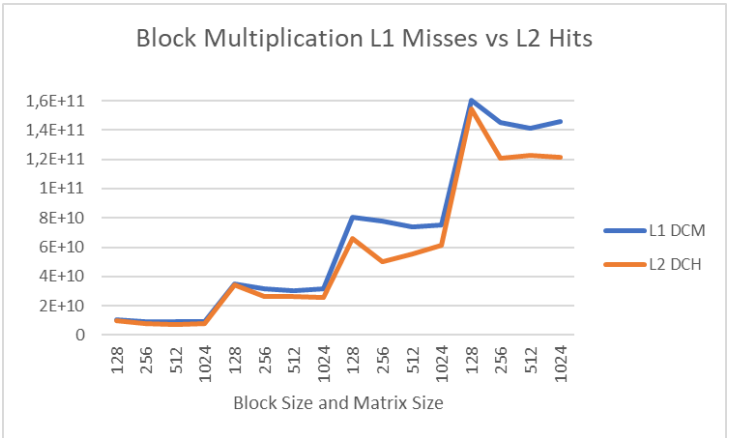

Figure 5 - L1 Misses and L2 Hits Block Multiplication

Based on the graphs above, it appears that the line multiplication method outperforms the standard one in terms of cache performance. This may be gathered from the fact that the line multiplication algorithm has less L1 misses and L2 hits for the same matrix size, indicating that it can use the cache more efficiently than the regular approach.

As for the block multiplication method, we can observe that, for matrix sizes between 4096 and 6144, despite the block size, the L1 misses and L2 hits are pretty similar. However, for matrix sizes higher than 6144 ,and lower block sizes, the L1 misses and consequently L2 hits are higher when compared to higher block sizes for the same matrix size.

6 - Installation & Compilation

Installation and compilation information can be found in the README of this project's GitLab repository.

7 - Conclusions

The results of testing the three languages with matrices of varying sizes, consistently showed that C++ performed better than Go or JS. By analyzing the results we could also prove what we thought in theory, that the Line Algorithm and the Block algorithm would obviously be a more efficient implementation of the standard multiplication algorithm.

Additionally, we were able to fully study the importance of cache, concluding that making small changes to an algorithm can greatly improve its performance.

8- Resources

Apart from the resources mentioned above, we used the following:
PAPI preset events:
https://icl.utk.edu/projects/papi/files/html_man3/papi_presets.html