U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Performance and Evaluation of a Single Core Using Matrix Multiplication

Parallel and Distributed Computing 2022/2023

Gabriel Amorim Carvalho - up201904508
Miguel Curval - up201105191
Jorge Costa - up201706518
Diogo Pinto - up201906067

# Contents

# Problem Description

This project aims to analyze the performance of the memory hierarchy of the processor when accessing large amounts of data. In this study, we created three distinct methods to simulate a memory-intensive problem by computing the product of two matrices. Finally, for a better understanding of the impact of the memory hierarchy on the program's performance, we used Performance API (PAPI) indicators for different matrix sizes.

# Algorithms Explained

For this chapter we assume a multiplication of 2 matrices A and B and a result matrix C.

## Basic Multiplication

This algorithm consists of row-by-column multiplication, where the entries of the rows of matrix A are multiplied by they're corresponding entries of the columns in matrix B. The resulting products are summed to obtain the value of each element in the resulting matrix C. This algorithm was already provided by the structure of the project in C/C++, so we implemented it in python. It has a time complexity of O(n^3).

**Pseudo-Code:**

```
for line_i in matrix_a:
	for col_j in matrix_b:
		for line_j in line_i:
			matrix_c[i][j] = elem * col_j
```

## Line Multiplication

Line multiplication is a more efficient algorithm for multiplying matrices compared to basic multiplication. This algorithm involves computing the product of each row in the matrix A with each column in the matrix B accumulating the products in the respective position of the matrix C.

**Pseudo-Code:**

```
for line_i in matrix_a:
	for elem in line_i:
		for line_j in matrix_b:
			matrix_c[i][j] = elem * line_j
```

## Block Multiplication

Block multiplication is the most efficient algorithm of all 3. It consists in dividing the matrices A and B into blocks and then performing the previous algorithm (Line Multiplication) to calculate the result matrix C.

**Pseudo-Code:**

```
for (ii = 0; ii < numRows; ii += blockSize) {
  for (jj = 0; jj < numRows; jj += blockSize) {
    for (kk = 0; kk < numRows; kk += blockSize) {
      for (i = ii; i < ii + blockSize; ++i) {
        for (j = jj; j < jj + blockSize; ++j) {
          for (k = kk; k < kk + blockSize; ++k) {
            matC[i*numRows + j] += matA[i * numRows + k] * matB[k*numRows+j];
```

All 3 algorithms have time complexity O(n^3). For the first 2 algorithms it is easily understandable since there are 3 for cycles chained. For the last algorithm, the way the first 3 for cycles and the 3 for cycles included are iterated reduce the time complexity to O(n^3).

# Performance Metrics

Since the goal of this project was to compare the performance of the processor in this particular problem of multiplying 2 matrices, we compared the results using 2 programming languages and different sizes of matrices.

In order to analyze and compare the performance of the programs in C++ we used the performance API (PAPI) to measure the number of **Data Cache Misses (DCM) on level 1 and level 2 caches.** We also collected the **time(in seconds)** it takes to execute the algorithm.

Finally, we decided that we would use another metric to compare the different algorithms using different programming languages and different sizes, which is the ***GFLOP/s.*** It is the number of Floating Point Operations per second, calculated by dividing the number of Floating Point Operations by the time it takes to execute the algorithm.

# Results and Analysis

Before going into detail about the performance analysis, it's crucial to remember that all tests were carried out on the same machine, running an Intel(R) Core(™) i7-9750H CPU 2.60 GHz, 16GB of RAM, and running the Windows operating system.
Every result in the graphics that we will discuss in this chapter can be found at the '*src'* folder in our repository.

## Basic Multiplication

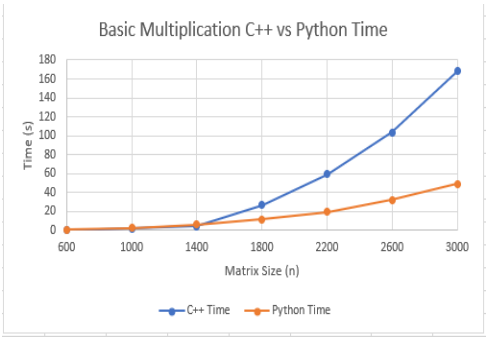



**Figure 1:** Time performance C++ vs Python

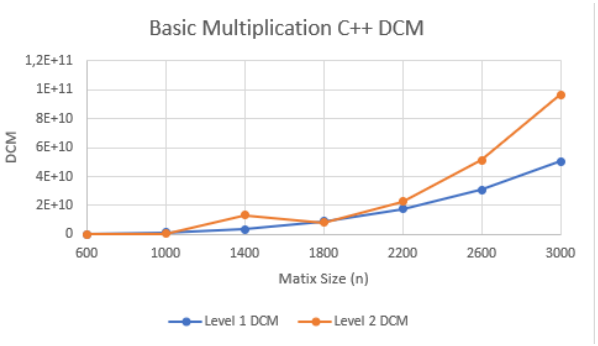



**Figure 2:** Basic Multiplication DCM C++

From the analysis of figure 1, we verified an increase in the elapsed time for bigger matrix sizes in both programming languages, as expected. Something that we, at first, were not expecting was the time it took to execute the algorithm in C++ being higher for every size compared to the algorithm in Python. That can be explained by the specific implementation details of the algorithm in C++ and Python. C++ implementation is using nested loops for matrix multiplication, which in our case is less efficient than the Python implementation that uses NumPy, a library specifically designed for numerical computations.
As for the figure 2, the results were also expected. For bigger matrix sizes, more memory accesses occur, therefore higher DCM.

## Line Multiplication

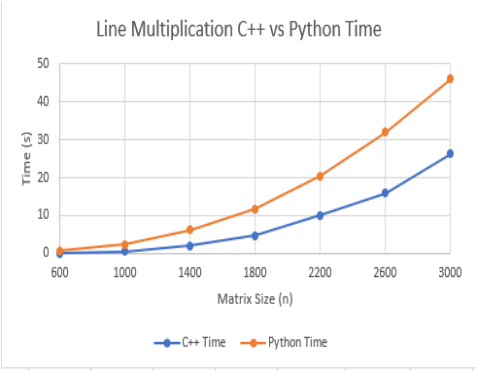



**Figure 3:** Time performance C++ vs Python

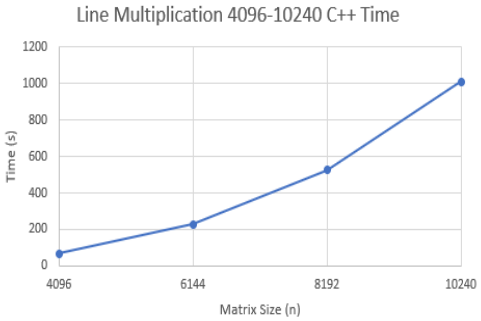



**Figure 4:** Time performance for bigger matrix sizes in C++

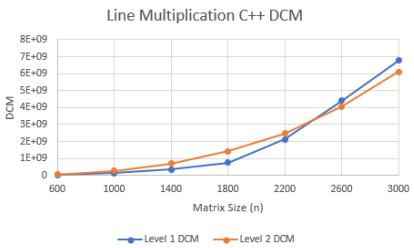



**Figure 5:** Line Multiplication DCM C++

In this algorithm there is a significant improvement in all performance metrics compared to the previous algorithm in C++. This improvement is attributed to the use of C++ memory allocation by the Line Multiplication algorithm.

### C++ vs Python:

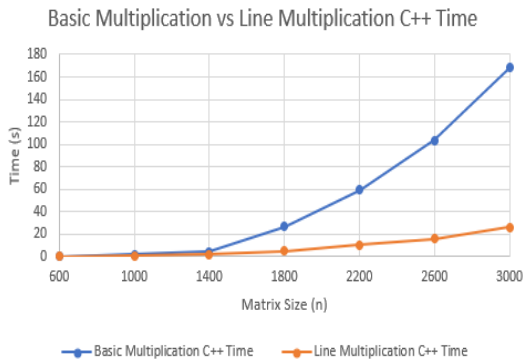



**Figure 6:** C++ time performance comparison

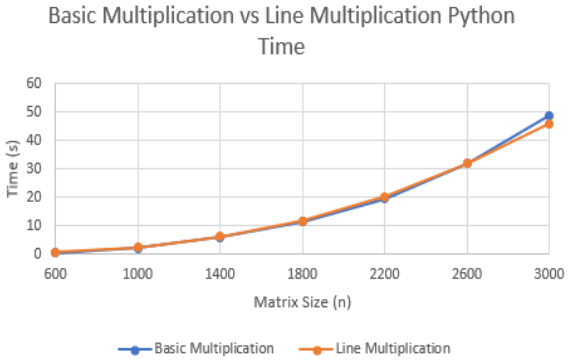



**Figure 7:** Python time performance comparison

From Fig.6 and Fig.7, we can see that apart from the Basic Multiplication algorithm, which we discussed before, python has a worse performance in comparison to C++. A possible reason for this is the fact that Python relies on an interpreter and executes many operations during runtime, while C++ programs are precompiled and only run the resulting program.

## Block Multiplication

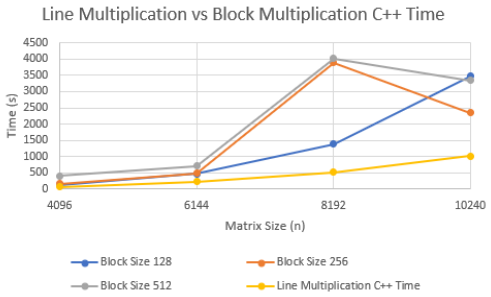
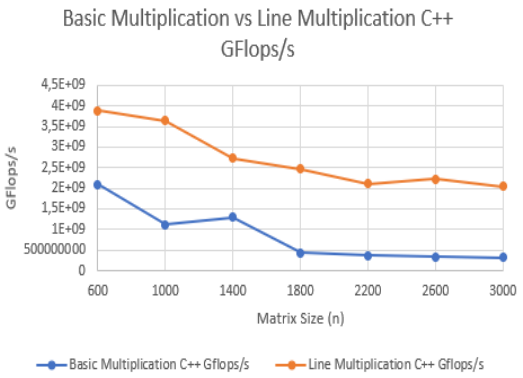



**Figure 8:** C++ Time Performance and comparison to Line Multiplication

Since the memory is divided into blocks, each access to memory reads one block at a time. Minimizing these accesses becomes the game changer for a better performance program when working with huge volumes of data.
In comparison to the Line Multiplication algorithm, which had a reduction in data cache misses, the Block algorithm approach makes use of the memory layout to some extent by decreasing memory calls.

## Floating Point Operations Comparison



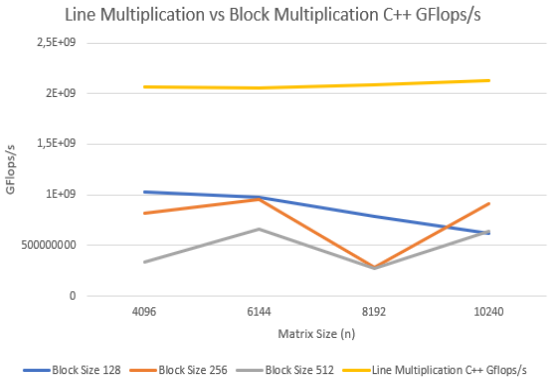




**Fig 9:** Number of Floating Point Operations Comparison C++

**Fig 10:** Number of Floating Point Operations Comparison C++

With the use of Fig.9 and Fig.10 measurements, we were able to more easily compare the effectiveness of the three algorithms. Having said that, it is clear why the line multiplication algorithms outperformed the most fundamental algorithm by a significant margin due to their higher GFLOPS/s values.

# Conclusão

Through the progress of this project, we realized the significance of writing efficient code that takes into account the details of the processor's functioning and the structure of the memory hierarchy to achieve optimal performance. By comparing all the algorithms we implemented, we noticed that small modifications to even the most basic code can significantly increase algorithm performance. Furthermore, our results confirm that C++, a language recognized for its speed, outperforms Python in most algorithms execution times, which tells us that the choice of programming language is a crucial aspect of program design and analysis. The only time Python doesn't outperform C++ is in the Basic Multiplication algorithm, which we discussed in the results and analysis chapter by giving the possible reason. In summary, this project provided us with an opportunity to learn by doing, applying the theoretical knowledge we gained during the course.