Performance Evaluation of a Single Core

Parallel and Distributed Computing – 1st Project

3LEIC03 G15

António Ferreira – up202004735@fe.up.pt
Hugo Gomes – up202004343@fe.up.pt
João Moreira – up202005035@fe.up.pt

Contents

1- Introduction……..2

2- Algorithms……...2

1. Naïve Algorithm
2. Line Multiplication Algorithm
3. Block Multiplication Algorithm

3- Metrics……..3

4- Results……..4

1. Comparison between the different algorithms
2. Comparison between the different block sizes
3. Comparison between cache performance of the algorithms
4. Comparison between FLOPS
5. Comparison between the C++ and Java algorithm implementations

5- Conclusions……7

1. Introduction

The main objective of this project is to analyze the effects of processor performance on the memory hierarchy big volumes of data are accessed. For this study, we implemented and compared 3 distinct matrix multiplication algorithms (in C++ and Java) and utilized a Performance API (PAPI) to collect key performance indicators of the program execution.

- **Hardware Information:**

CPU - Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
Cache Information -

Cache	Total Size	Line size	Number of Lines	Associativity
L1 Data Cache	32 KB	64 B	512	8
L1 Instruction Cache	32 KB	64 B	512	8
L2 Unified Cache	256 KB	64 B	4096	4
L3 Unified Cache	12288 KB	64 B	196608	16

2. Algorithms

2.1 – Naïve Algorithm

The simplest algorithm of matrix multiplication consists of applying the dot product to each line of the first matrix with each column of the second matrix. It should be noted that to represent a matrix, we resort to a one-dimensional array where the values are stored in a row-major order (the consecutive elements of a row of the matrix reside next to each other, whereas elements of the same column or a different matrix are separated).

```
for(i=0; i<m_ar; i++){
    for( j=0; j<m_br; j++){
        temp = 0;
        for( k=0; k<m_ar; k++){

            temp += pha[i*m_ar+k] * phb[k*m_br+j];
        }
        phc[i*m_ar+j]=temp;
    }
}
```

(fig.1 - Naïve Algorithm written in C++)

2.2 – Line Multiplication Algorithm

In this algorithm, we chose to change the order of the operations so that we could improve the performance of the prior algorithm. Consequently, we can utilize the spatial and temporal locality of the elements kept in memory. Since, in this project, the elements are stored in row-major order, to reduce the number of Data Cache Misses,

we can multiply the matrix's elements by each element of the corresponding line of the second matrix.

```
for(i=0; i<m_ar; i++){
    for( k=0; k<m_ar; k++){
        for( j=0; j<m_br; j++){
            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
        }
    }
}
```

(fig.2 - Line Multiplication Algorithm written in C++)

2.3 – Block Multiplication Algorithm

This algorithm consists of dividing the matrices into blocks of a fixed size and the multiplication is performed block by block, before being done on an element level with the line algorithm shown previously.

Also, although block-oriented algorithms are generally more efficient than line-oriented algorithms for larger matrices since they minimize the number of memory accesses and cache misses, the block size must be small enough so that a single line may fit in cache.

```
size_t numBlocks = m_ar / bkSize;
for(size_t iBlock = 0; iBlock < numBlocks; iBlock++){

    for(size_t kBlock = 0; kBlock < numBlocks; kBlock++){

        for(size_t jBlock = 0; jBlock < numBlocks; jBlock++){

            for(size_t i=iBlock * bkSize; i < ((iBlock + 1) * bkSize); i++){

                for(size_t k=kBlock * bkSize; k < ((kBlock + 1) * bkSize); k++){

                    for(size_t j=jBlock * bkSize; j < ((jBlock + 1) * bkSize); j++){
                        phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];

                    }
                }
            }
        }
    }
}
```

(fig.3 - Block Multiplication Algorithm written in C++)

3. Metrics

To test and analyze the effects of processor performance on the memory hierarchy, we measured the **elapsed time** (in seconds) and hardware metrics during the execution of the algorithms previously shown, for different matrix sizes. More specifically, we tracked the number of **L1 and L2 data cache misses** (they provide insight into overall program performance, computer efficiency, hardware limitations, cache hierarchy utilization, and memory access patterns). Also, taking into consideration that, in this project, the number of floating-point operations, for all the

algorithms presented, is $2n^3$ (n is the matrix size), we also determined the **GFLOPS** (Giga Floating-point Operations Per Second – a useful unit of measurement to compare the performance of different computer systems or processors), for each algorithm, by using the following formula:

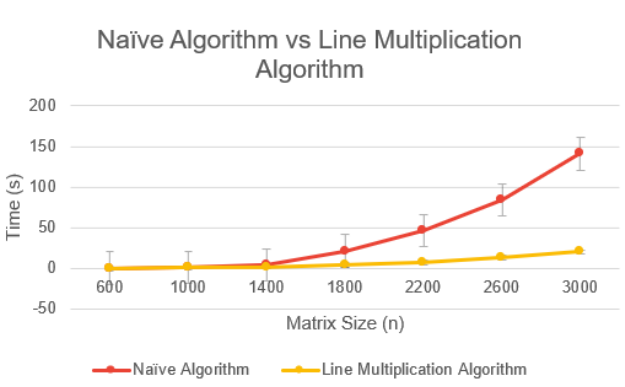
$$FLOPS = \frac{\text{Nº Floating Point Operations}}{\text{t (seconds)}}$$

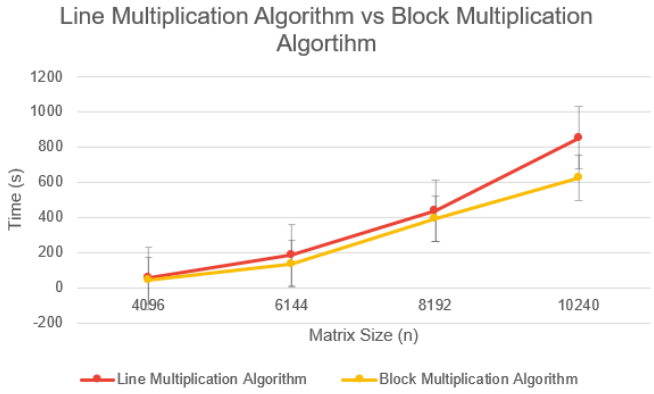
Lastly, in order to compare and draw conclusions regarding the influence of the programming language choice on the execution of algorithms, besides C++, we also implemented the algorithms in Java.

4. Results

It should be noted that values presented in this report were obtained by calculating the average of a set of measures, to reduce uncertainties.

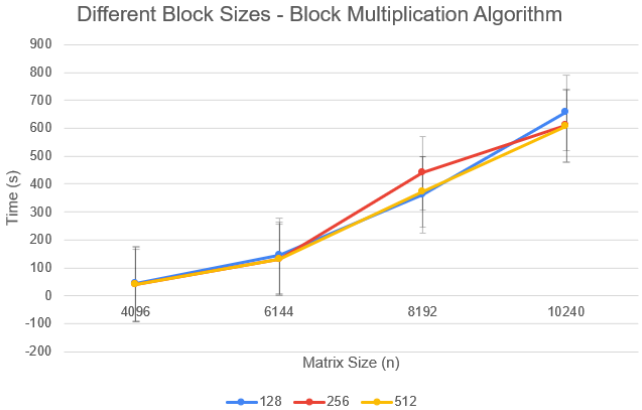
4.1 – Comparison between the different algorithms





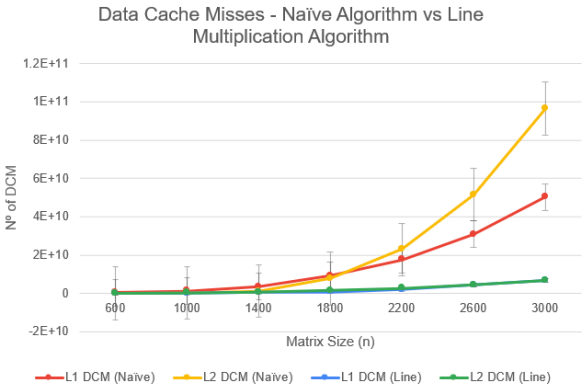
To do the comparison between the different algorithms previously mentioned, we measured the execution time of each C++ implementation for increasing values of matrix sizes. It must be noted that, for the block multiplication algorithm, we used the average time of the block sizes for each matrix dimension. As we can see, out of all the algorithms, the worst was the naïve implementation, since it was the slowest. On the other hand, although the difference between the line and block multiplication algorithms is quite small, we can verify a slight improvement in the performance speed when using the block-oriented multiplication algorithm.

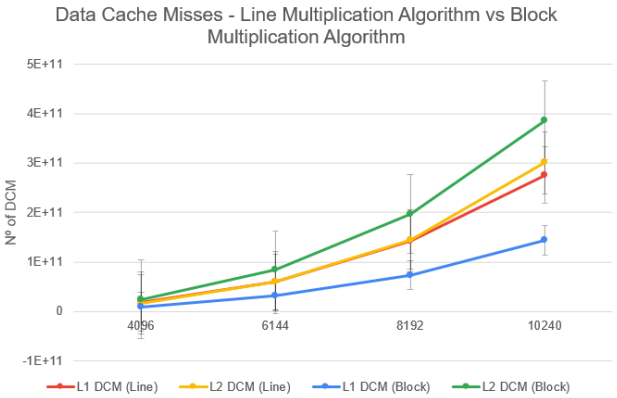
4.2 – Comparison between the different block sizes

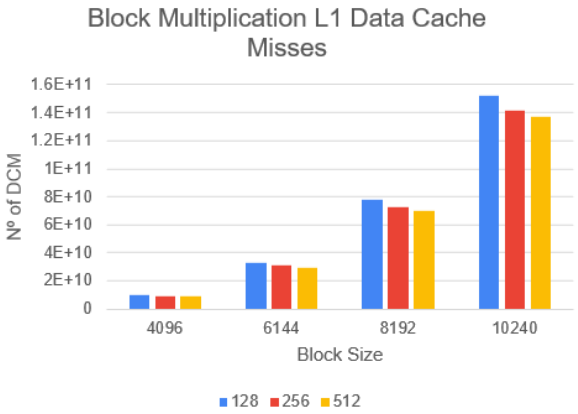


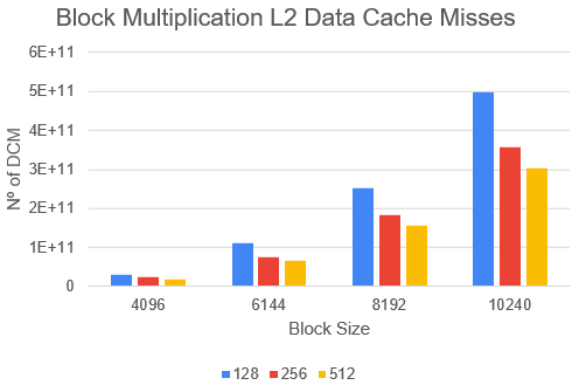
Taking into consideration the results shown, even though we only notice slight differences between the different block sizes, we can conclude that the block of size 512 has the best efficiency.

4.3 – Comparison between cache performance of the algorithms



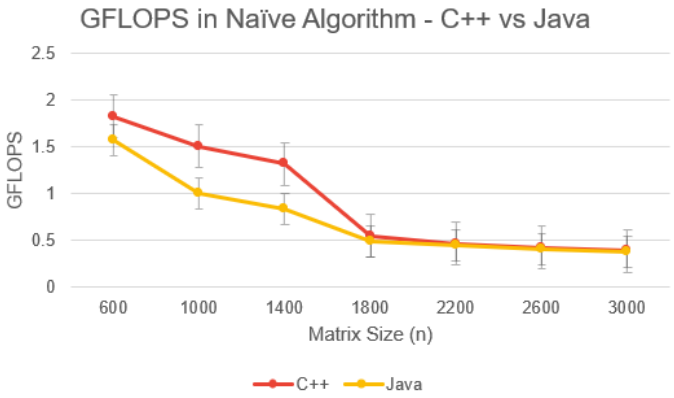


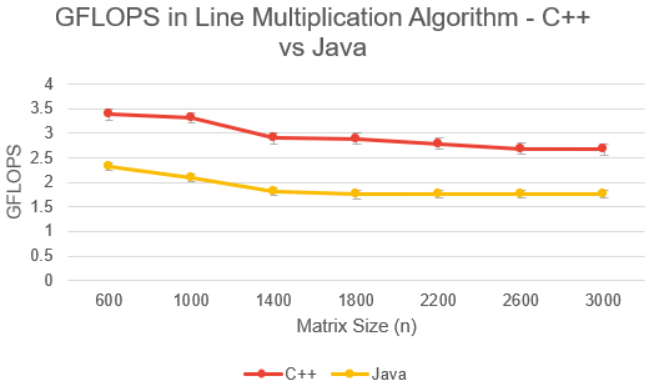


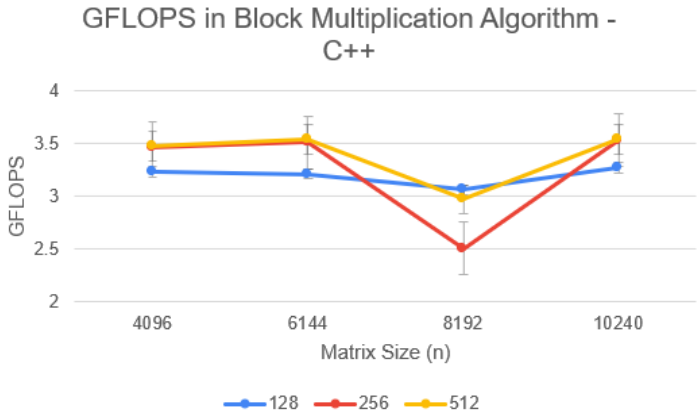


The three techniques were displayed across a range of progressively larger values for the matrix sizes to demonstrate how the algorithm improvements affect cache consumption. Considering the graphics above, it's possible to affirm that, once again, the naïve algorithm presents the worst performance. When comparing the line and block-oriented multiplication algorithms block, although the latter presents a lower number of L1 DCMs, the number of L2 DCM is much bigger. The number of DCMs, on the other hand, decreases as the block size increases, as can be seen when comparing block sizes.

4.4 – Comparison between FLOPS

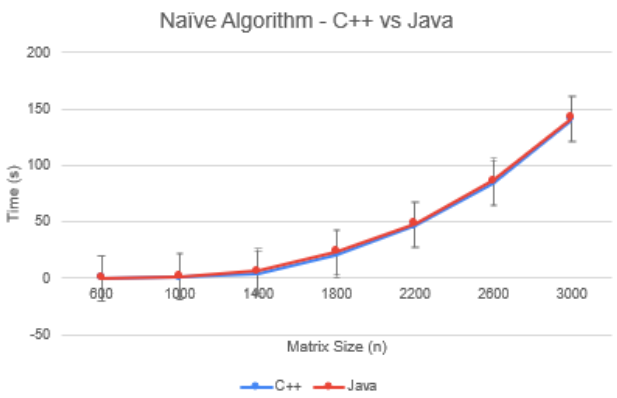


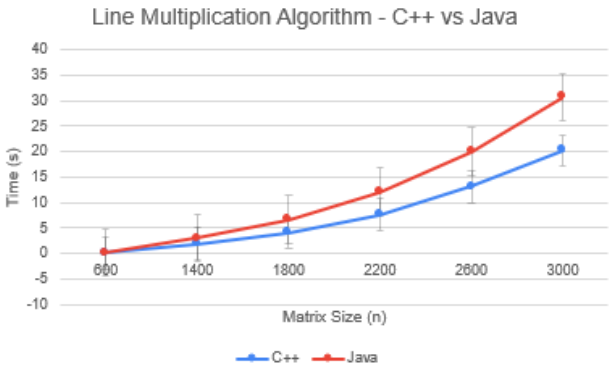




After examining the graphs above, it is clear that C++ outperforms Java versions in terms of GFLOPS calculated for each determined matrix size. On the other hand, for increasing block size, it's possible to observe that the blocks of 512 have the highest number of GFLOPS.

4.5 – Comparison between the C++ and Java algorithm implementations





For continuously growing matrix sizes, with the aim of evaluating the performance of the algorithms when implemented in various programming languages, we choose to

directly compare and assess the execution times of the naïve and line multiplication methods in C++ and Java. Taking this into account, and the graphics above, we can conclude that C++ version performed better and that its performance on improved algorithms increases with larger matrix sizes when compared to slower algorithms.

5. Conclusions

With this project, we had the opportunity to measure the performance of a single core, by utilizing different matrix multiplication algorithms to observe and compare various performance behaviors due to memory (especially cache memory) management. One aspect that should be highlighted is that, by taking advantage of the spatial and temporal locality of the elements stored in cache, we could see a decrease in data cache misses, which, as consequence, leads to the maximization of the efficiency and overall speedup of the program run time.

Consequently, we can affirm that the block-oriented multiplication algorithm is the best one. On the other hand, considering the results obtained in this project, it's possible to conclude that the choice of the block size in a block matrix multiplication algorithm can have a very significant effect on the performance. Normally, the optimal size block depends on many factors like the sizes of the matrices, the implementation of the algorithm, and the specific hardware architecture of the computer, for example. This leads to a no one-size-fits-all answer to the question of the best block size to use. However, in this context, we can conclude that the best block size is 512.