U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

CPD - First Project Report

Afonso Castro @202208026
Gonçalo Ferros @202207592
Pedro Oliveira @202000149

1. Problem Description

The goal of this project is to evaluate the CPU performance of three different matrix multiplication algorithms: standard multiplication, line multiplication and block multiplication. The project also aims to compare the performance with parallelization and using different programming languages.

The workload was divided in parts:

- Implementation of the standard multiplication algorithm in a language other than C++. We chose C#. Registration of the processing times in both languages for input matrices from 600x600 to 3000x3000 elements with increments in both dimensions of 400.
- Implementation in both languages of an algorithm that uses line multiplication. Registration of the processing times in both languages for input matrices from 600x600 to 3000x3000 elements with increments in both dimensions of 400. Registration of the processing times in C++ for input matrices from 4096x4096 to 10240x10240 elements with increments in both dimensions of 2048.
- Implementation in C++ of an algorithm that uses block multiplication. Registration of the processing times for input matrices from 4096x4096 to 10240x10240 elements with increments in both dimensions of 2048, and for different block sizes (128, 256 and 512).
- Implementation in C++ of two parallel versions of the line multiplication algorithm and performance verification (MFlops, speedup and efficiency).

We used PAPI (Performance API) to measure the performance of the C++ algorithms by requesting data from the CPU.

2. Algorithms Explanation

2.1. Standard Matrix Multiplication

This already given algorithm is a basic and non-optimized implementation of matrix multiplication. It multiplies two matrices A and B (represented as simple one dimensional arrays) and stores the result in a third matrix C. The main computation part is done by a triple nested for loop.

Complexity Analysis

- **Time Complexity:** $O(N^3)$, where N is the size of the matrices
- **Space Complexity:** $O(N^2)$, where N is the size of the matrices

2.2. Line Matrix Multiplication

This algorithm implements line multiplication by accumulating contributions row by row instead of computing each element separately. The loop order is slightly changed in a way that it accumulates each row of A into the corresponding row of C before moving to the next row. This version processes C row by row, which improves cache locality and, therefore, the performance.

Complexity Analysis

- **Time Complexity:** $O(N^3)$, where N is the size of the matrices
- **Space Complexity:** $O(N^2)$, where N is the size of the matrices

Despite the complexity being the same as the basic algorithm, this algorithm is faster due to better cache efficiency and the reduced number of accesses to matrix A.

2.3. Block Matrix Multiplication

This algorithm implements block matrix multiplication by processing small blocks, or submatrices, instead of computing individual elements or entire rows at once. This approach helps improve the cache efficiency and memory access patterns since entire rows of large matrices may not fit in the CPU's cache. There are extra loops to iterate through the blocks. Inside each block, the standard multiplication process is performed.

Complexity Analysis

- **Time Complexity:** $O(N^3)$, where N is the size of the matrices
- **Space Complexity:** $O(N^2)$, where N is the size of the matrices

Although the complexity is still the same, this algorithm is in practice much faster than the others for large matrices.

2.4. Parallel Line Matrix Multiplication

We implemented two parallel versions of the line matrix multiplication using OpenMP:

- The **first version** fully parallelizes the multiplication loops, reducing thread contention. The initialization of the three matrices happens in a single parallel region. The two nested loops are collapsed into a single loop for better load balancing. Each thread operates on different elements of C independently, avoiding synchronization issues and reducing contention since each thread writes to unique memory locations.

- The **second version** requires atomic operations to prevent race conditions. The main difference is that the outer loop runs serially, despite the innermost loop still being parallelized. This version is slower and its efficiency can even be similar to the normal line multiplication because atomic operations introduce overhead and the memory access patterns are worse.

3. Performance Metrics

3.1. Hardware

All performance tests were executed on a computer with the AMD Ryzen 7 5800H processor, with 16 CPUs and with the x86_64 architecture.

3.2. PAPI

As suggested, we used the Performance API (PAPI) to analyse the performance of each algorithm (except those of C#) by collecting data directly from the CPU. PAPI is a library that provides access to hardware performance counters in CPUs. It allows measuring events like cache misses, instructions executed, and branch mispredictions. To compare cache miss variations between the different algorithms and at the different cache levels, we chose to capture 3 events:
PAPI_L1_DCM - Level 1 data cache misses
PAPI_L2_DCM - Level 2 data cache misses
PAPI_L2_ICM - Level 2 instruction cache misses

4. Results and Analysis

The data from the experiments was saved in .csv files and then transferred to a Sheets document where we could better understand it and create the following graphics.

4.1. Matrices from 600x600 to 3000x3000

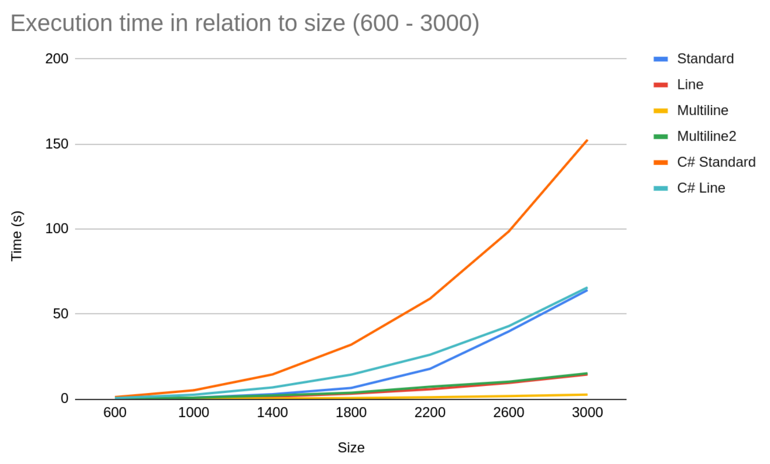
When comparing the execution times of Standard Matrix Multiplication (Standard), Line Matrix Multiplication (Line), and Parallel Matrix Multiplication (Multiline), we observed that the Standard method took significantly longer to execute than the other two. Among them, the first Multiline method achieved the best execution time.

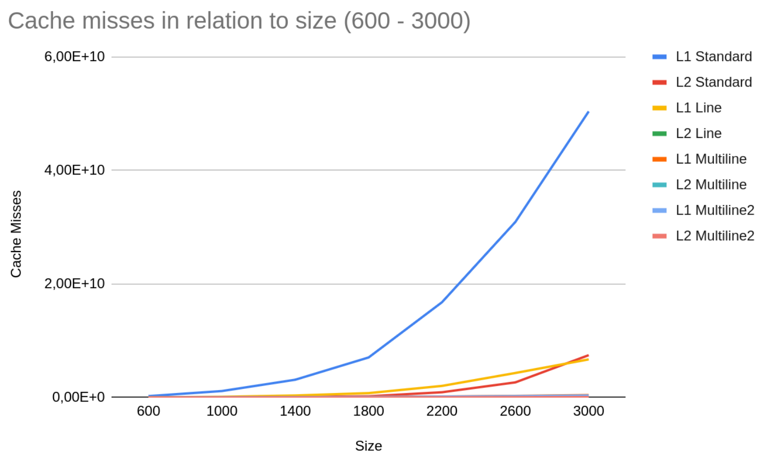
Additionally, we observed that the second Multiline method performed similarly to the Line method.

When comparing the C# implementations of the Standard and Line algorithms with their C++ counterparts, we clearly observed the poor efficiency of the C# versions. The C# Standard method performed astronomically worse than all other algorithms, while the C# Line method exhibited execution times comparable to the Standard method in C++.

In terms of cache miss analysis for all algorithms (excluding the C# implementations, where cache analysis was not possible), we observed that the L1 cache consistently had more cache misses than the L2 cache across all methods.

The Standard method exhibited significantly more cache misses compared to the others, with values substantially higher than the rest. The Line method showed a considerable improvement in cache efficiency, though it was obviously still outperformed by the Multiline methods.

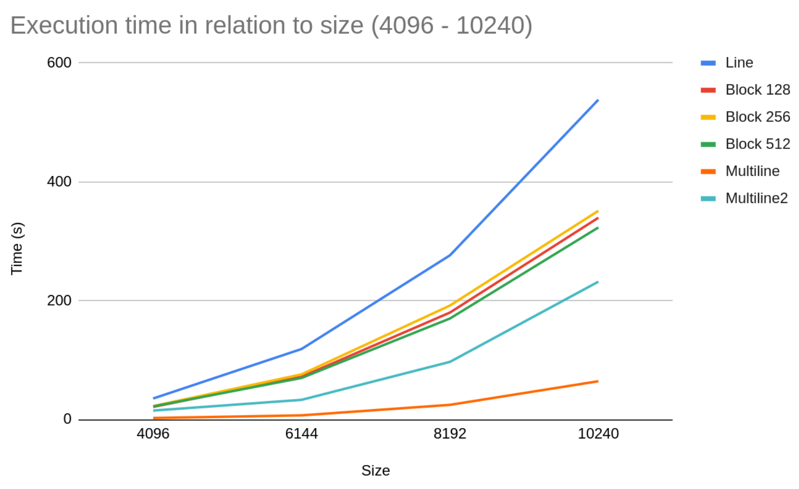


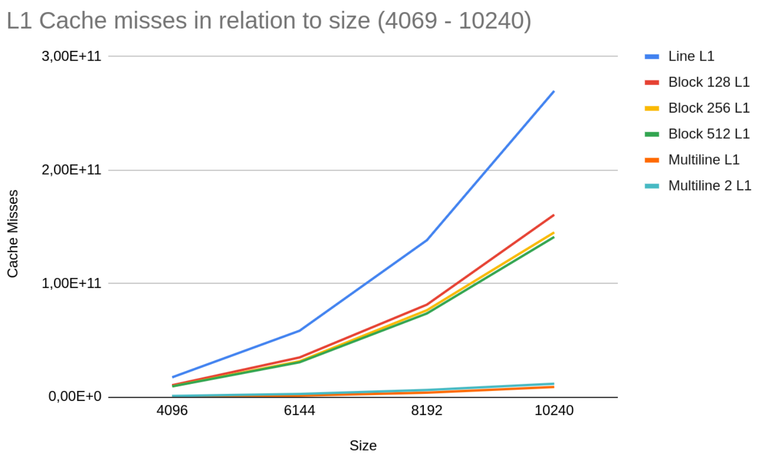


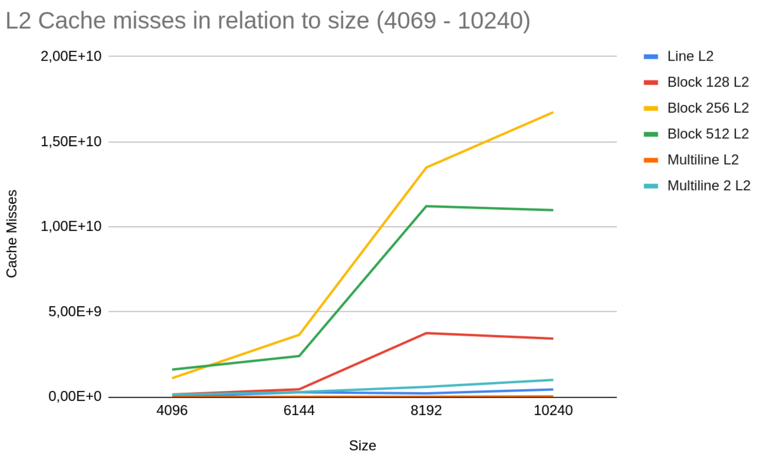
4.2. Matrices from 4096x4096 to 10240x10240

Our general observations regarding the execution times of the Line and Parallel algorithms from the previous section remained consistent for larger matrix sizes, where they performed worse and better, respectively, compared to the Block algorithms. As for the cache misses, while the L1 caches remained consistent with the above observation, in the L2 caches, the Block algorithms severely underperformed in comparison to the others.

Within the Block algorithms, we observed no significant difference in execution times across different block sizes. In terms of cache misses, the L1 cache differences were negligible. However, the L2 cache misses showed a notable variation, with the 128-block size having significantly fewer misses than the others, followed by the 512-block size, and then the 256-block size.







5. Conclusions

This project's main goal was to evaluate the performance of three matrix multiplication algorithms (standard, line, and block multiplication) along with some parallel implementations.

Through our executions of the various methods in both C++ and C#, we analyzed execution times, cache efficiency, and the impact of optimization techniques in the performance.

Our results demonstrated that cache efficiency plays a crucial role in performance, with line and block multiplication generally outperforming the standard method. The block method, in particular, was significantly more efficient for large matrices, although variations in block size affected cache performance, particularly at the L2.

Parallelization led to substantial speed improvements, with the first Multiline method achieving the best execution times. However, memory access patterns and thread contention influenced efficiency, as seen in the second Multiline method, which performed similarly to the Line method.

When comparing C++ and C# implementations, we observed that C# was far less efficient, with the C# Standard method performing significantly worse than any other algorithm tested. The C# Line method, while better, still struggled to match the performance of even the C++ Standard approach.

The use of the PAPI in C++ provided a deeper analysis into cache behavior, confirming that optimized algorithms significantly reduce cache misses and improve computational efficiency. In particular, we found that block sizes affected L2 cache performance, with the 128-block configuration yielding the lowest number of misses.

These findings show the importance of choosing the correct implementation for a given situation, be it the task to be performed or the hardware in which it will be run, where these choices can cause a significant loss of performance.