Faculdade de Engenharia da Universidade do Porto

CPD Project 1

Performance evaluation of a single core

Computação Paralela e Distribuída 2022/23 – Licenciatura em Engenharia Informática e Computação:

Turma 11 Grupo 14

Estudantes & Autores:
Rui Pires up202008252@up.pt
Pedro Gomes up202006322@up.pt
João Reis up202007227@up.pt

March, 2023
Porto, Portugal

Índice:

1. Problem description and algorithms explanation

This project is centered around the study of the performance of the memory hierarchy when accessing large amounts of data. The product of two matrices will be used for this study, which involves implementing several versions of the matrix multiplication algorithm. The algorithms will be implemented in C++ and Rust.

We will be using The Performance API (PAPI) to collect relevant performance indicators, like cache misses and the number of floating operations. We measure the performance indicators in FEUP's computers that have an intel core i7 9th gen. However, we were unable to install Rust on the college computers, so we used another computer (intel core i7 6th gen) to do the tests involving this language by running both the Rust and C implementations.

With the first algorithm, we make the calculations in the order we typically think of when multiplying matrices: we calculate the dot product between each line in the first matrix and each column in the second.

With the second, instead of multiplying each line by each column, we multiply each element in the first matrix by the corresponding line in the second. In other words, for each element in the first matrix, we iterate the line in the second matrix corresponding to the column of the element we are currently iterating in the first.

With the third, we divide the matrix into blocks, calculate the product of those blocks and then combine their result. First, we initialize the result matrix as a zero matrix. Then, we iterate through each of the matrices. We do this in the same order as the second algorithm, as if the blocks were single elements (we iterate the first matrix in a row-major fashion, and then for each block of the first matrix we iterate the line in the second matrix corresponding to the column of that block). Then, we use the second algorithm to multiply each pair of blocks (corresponding to a pair of matrices) and add the result to the current result matrix.

2. Performance metrics

In this project we used the Performance API (PAPI), to collect relevant performance indicators of the program execution.

For this project we chose as metrics:

- Execution Time
- Level 1 Data Cache Misses
- Level 2 Data Cache Misses
- Double Precision Floating-Point Operations

Execution time is perhaps the most straightforward performance metric, as it simply measures the time taken for the program to complete a particular task. In this project, the execution time is measured for both the C++ version of the program and also the version of the program that was written using Rust, since PAPI does not provide their list of metrics for

the Rust language. By comparing the execution times of these two implementations for matrices of different sizes, we can gain insight into the relative performance of the two languages and the impact of different sizes of the matrices on execution time.

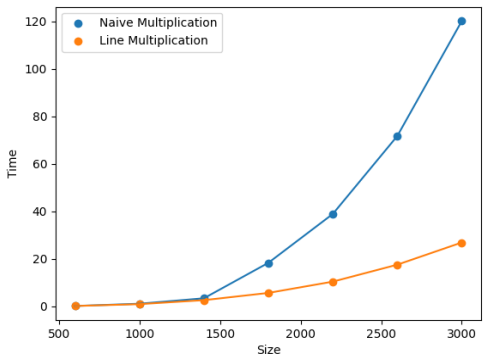
Cache misses is also a relevant metric because, when accessing large amounts of data, such as when multiplying large matrices, cache utilization becomes critical to the overall performance of the algorithm. By measuring the cache misses, we can gain insight into how well the algorithm is utilizing the cache memory and how often it is being forced to retrieve data from the slower main memory.

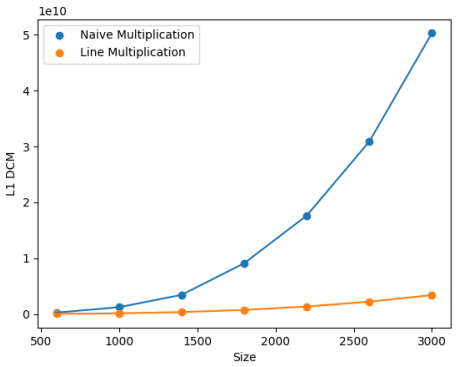
Double precision floating-point operations are also an important performance metric in the context of our project. This metric measures the number of double precision floating-point operations that the algorithm is able to perform. Since matrix multiplication involves a large number of floating-point operations, this metric provides an indication of how efficiently the algorithm is able to perform these calculations.

Overall, by considering a range of different performance metrics, we can gain a more comprehensive understanding of the performance characteristics of each algorithm version and how well they are able to utilize the available hardware resources. This can help us to identify potential areas for optimization and improve the overall efficiency of the algorithm.

We also meant to use level 2 data cache accesses and use that performance metric to derive data cache hits. However, for some unknown reason the number of data cache accesses was repeatedly lower than the number of data cache misses, which is an impossibility. For this reason, we decided to not use these 2 metrics.

3. Results and analysis





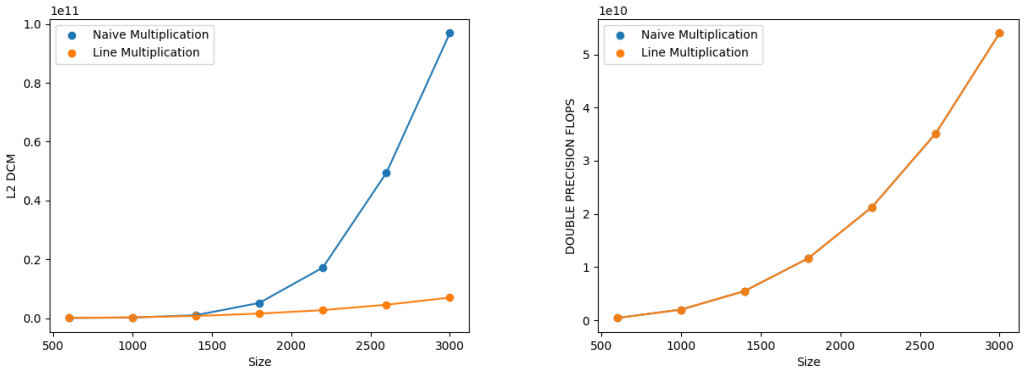


Imagem 1: Naive Multiplication comparison with Line Multiplication in C

When using the naive algorithm, in each iteration we need to get a value of a different line of the second matrix. Assuming the cache size is small enough so that it doesn't contain more than one line, the cache block retrieved in the previous iteration due to a cache miss doesn't contain the value needed for the new iteration (since it is in a different line), resulting in a cache miss in every single iteration.

With the line multiplication algorithm, we multiply each element in the first matrix by the corresponding line in the second. Because of this, we iterate the second matrix in a row-major order. Since rows are contiguous in memory, there are a lot less cache misses, because when we get the first value of a line, a large part of that line is stored in cache, which means that the following iterations result in cache hits.

This was verifiable when comparing naive multiplication and line multiplication Level 1 and Level 2 cache misses. These two metrics are a lot smaller for the second algorithm. Obviously, this results in a smaller execution time for the second algorithm because more cache misses means needing to access the main memory a lot more times, resulting in a time penalty. This is also verifiable on the above graphs.

Although not very perceivable on the graph, FLOPS also increased a tiny percentage (0.1%) from the naive algorithm to the line algorithm. This is expected, since the first algorithm spends more time accessing main memory than the second, which means the second has more time to perform operations per time unit.

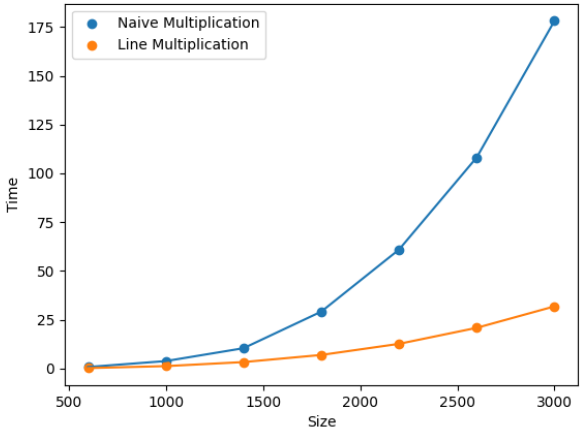


Imagem 2: Naive Multiplication comparison with Line Multiplication in Rust

In Rust, the exact same thing happens and causes the same time differences between the naive and the line multiplication algorithms.

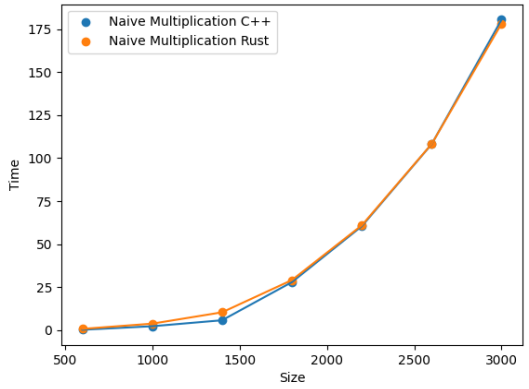


Imagem 3: Naive Multiplication comparison in C and Rust

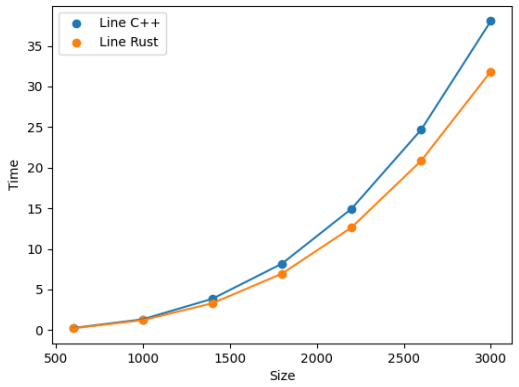
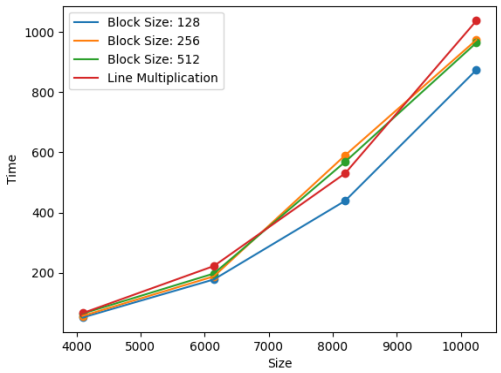
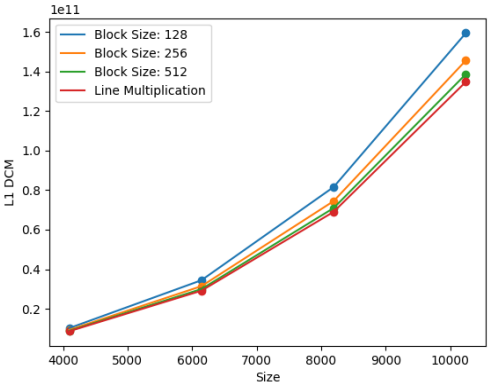


Imagem 4: Line Multiplication comparison in C and Rust

In the figures above we can compare the C++ version of the naive and line multiplication algorithms with the rust version of these same algorithms. For this comparison, the only metric we possess is the execution time for both languages, since PAPI does not provide its metrics for the Rust language. When compiling the rust source code, we used the `-C opt-level=2` flag.

Analyzing the graphs, we notice that both lines are incredibly similar, this is because both C++ and Rust are compiled programming languages that offer low-level control over hardware resources and are designed for performance-critical applications. Additionally, both C++ and Rust use a similar approach to memory management, where we are responsible for allocating and deallocating memory manually. Since we are running the same algorithms in both languages, we can observe minimal overheads between the execution time.





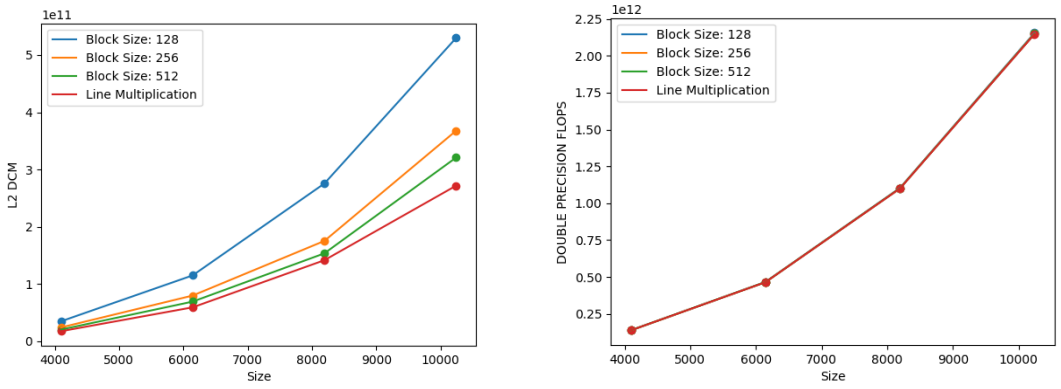


Imagem 5: Line Multiplication comparison with Block Multiplication (128, 256, 512) in C

The results when comparing these two algorithms are a bit contradictory. The block algorithm's purpose is to decrease data cache misses by using blocks that can fit into cache. However, this was not the case according to the L1 DCMs and L2 DCMs. At first, we thought that maybe the algorithm was incorrectly implemented, but we are in fact dividing the matrices in blocks and using the line multiplication algorithm for each of those blocks. If it was in fact incorrectly implemented, then line multiplication would be expected to be faster, which it almost never is for any matrix or block size.

In addition, the line algorithm is increasingly slower than the block algorithm for bigger matrices, in spite of having less cache misses. It doesn't make much sense for an algorithm with less cache misses to take more time, when the number of operations is exactly the same and the FLOPS are very similar. The only metric that seems to point in the direction of explaining this is FLOPS. FLOPS decreases about 0.1% when the block size increases and also decreases 0.1% from a block size of 512 to the line multiplication algorithm. We don't think this decrease is significant enough to explain the time decrease, but FLOPS do decrease each time the time execution decreases.

The purpose of the block algorithm is to fit 3 blocks into cache: the block of the first matrix, the block of the second and the result block. If this doesn't happen, more cache misses will occur.

We measured these metrics in a computer whose processor is a 9th generation i7 intel CPU. This processor has a level 2 cache size of 2MB per core. Since a double is 4 bytes, this means that block sizes up to ~700 could fit into the L2 cache. For the bigger block size used (512), only one block at a time could fit into the cache, which we expected to result in more L2 cache misses than when using smaller block sizes.

The level 1 cache size is 256KB, exactly the size to fit one 256 block. Again, we expected this to result in more L1 cache misses for block sizes of 256 and 512 when compared to 128.

In summary, we expected there to be more cache misses in general for the line multiplication algorithm when compared to the block algorithm for any block size, and then we expected the L1 cache to have less misses for a block size of 128 when compared to 256 and 512, and the L2 cache to have less misses for a block size of 128 and 256 when compared to 512. This would be expected to result in a decreasing execution time from the line multiplication algorithm to the block multiplication algorithm and from a block size of 512 to 256 and to 128. The time decrease was verified except for an anomaly when multiplying 8192x8192 matrices, but the expected L1 and L2 cache misses were not what we expected.

4. Conclusions

To sum it all up, we developed 3 different algorithms that all accomplished the same objective, multiplying two matrices.

It was quite easy to see that the naive algorithm was much inferior in terms of performance compared to the other algorithms. In terms of language comparisons, what we have to conclude is that the performance of C++ and the performance of Rust are quite similar due to the factors listed above. As for the block version of the algorithm, when comparing different block sizes, we obtained somewhat contradictory results, which we did not expect.

The instructions for installing the dependencies, compiling and running code are in the README.md file inside the assing1 directory.