FACULDADE DE ENGENHARIA DA
UNIVERSIDADE DO PORTO

COMPUTAÇÃO PARALELA E DISTRIBUÍDA

CPD Project 1 - Performance evaluation

António Henrique Martinz Azevedo (202108689)
José António Pereira Martins (202108794)
Tomás Eiras Silva Martins (202108776)

March 14, 2024

Contents

1 Problem Description

This project goal is to analyse the impact of memory hierarchy on processor performance during matrix multiplication. We aim to understand how data access patterns influence computational efficiency through various implementations and performance measurements across different input sizes.

2 Algorithms

In order to understand the influence of different algorithms on processor performance during matrix multiplication, we implemented three algorithms that vary primarily in their memory access patterns. To analyze the impact of the programming languages on the performance we implemented the algorithms in **C++** and **Rust**. This choice allows us to compare the efficiency of a compiled, system-level language **- C++ -** with a compiled language known for its memory safety and performance focus **- Rust**. Both languages offer high levels of control over memory management, making them suitable for investigating how memory access patterns affect performance.

- **Basic matrix multiplication algorithm** - implemented in C++ and Rust
- **Line matrix multiplication algorithm** - implemented in C++ and Rust
- **Block matrix multiplication algorithm** - implemented in C++

2.1 Basic matrix multiplication algorithm

This algorithm implements a basic matrix multiplication approach, where each element in the resulting matrix is computed by multiplying one row of the first matrix by one column of the second. As a result, the algorithm demonstrates a computational time complexity of $O(n^3)$ and a space complexity of $O(n^2)$, owing to its composition of three nested loops. Here, n represents the size of the input matrices.

The resulting matrix can be calculated using the following equation:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j] \tag{1}$$

where C is the resulting matrix, A and B are the input matrices, and n is the size of the matrices.

2.2 Line matrix multiplication algorithm

Instead of directly multiplying a row of the first matrix with a column of the second, this algorithm computes the product of a row of the first matrix with each row of the second. This adjustment results in accessing matrix elements in a cache-friendly row-major order, reducing cache misses and improving execution time, particularly for large matrices. Despite maintaining the same time and space complexity as the algorithm before , this algorithm's efficient cache usage often makes it faster than the basic matrix multiplication algorithm in practical scenarios.

2.3 Block matrix multiplication algorithm

The Block Matrix Multiplication algorithm is designed to further optimize memory access patterns by dividing the matrices into smaller blocks. This strategy enhances cache utilization and reduces cache misses compared to the Line Matrix Multiplication approach. The algorithm operates by multiplying submatrices within these blocks, thereby minimizing the distance between accessed elements and exploiting spatial locality. Although sharing identical space and time complexities with the preceding algorithms, block matrix multiplication excels in performance, particularly with large matrices. This superiority derives from its capability to partition matrices into smaller blocks, effectively decreasing the incidence of cache misses.

3 Performance Metrics

3.1 Metrics

To assess the performance of the C++ implementations, we utilized the PAPI (Performance API), providing us with comprehensive hardware performance monitoring. Our evaluation comprised the following key metrics:

- **Execution Time** - This metric quantifies the duration taken by the program to complete its execution. It serves as a fundamental benchmark for computing overall performance and facilitating comparative analysis.
- **L1 and L2 Cache Misses** - These metrics quantify the number of cache misses occurring in the L1 and L2 caches, respectively. A cache miss arises when the requested data is not found in the cache and require retrieval from main memory. This metric provides valuable insights into the memory access patterns of the program, as frequent cache misses can lead to increased execution times, which means that minimizing cache misses is crucial for optimizing program performance!
- **GFlops** - This metric, derived from "**G**iga **Fl**oating-point **O**perations **p**er **S**econd", provides a measure of a computer's performance in executing floating-point arithmetic operations. It quantifies the rate at which a system can perform such operations and is calculated using the following formula:

$$GFlops = \frac{Number\ of\ Floating\ Point\ Operations}{Execution\ Time\ in\ Seconds} \times 10^{-9} \tag{2}$$

- **Speedup** - This metric quantifies the amount of improvement gained by using a particula parallelization technique compared to a baseline implementation. It is calculated as the ratio of the execution time of the baseline implementation to the execution time of the optimized or parallelized version. The formula for speedup is as follows:

$$Speedup = \frac{Execution\ Time\ of\ Baseline\ Implementation}{Execution\ Time\ of\ Optimized\ Implementation} \tag{3}$$

- **Efficiency** - Efficiency measures how effectively a parallelization technique utilizes the available computational resources, such as CPU cores. It is calculated as the ratio of the speedup to the number of cores used. The formula for efficiency is as follows:

$$Efficiency = \frac{Speedup}{Number\ of\ Cores} \tag{4}$$

3.2 Hardware used

- **Processor** - Intel Core i7-10700 CPU @ 2.90GHz 8-core (16 threads)
- **L1 Cache** - 512 KB
- **L2 Cache** - 2 MB

4 Results and Analysis

We ran the algorithms three times and recorded the values of the chosen metrics. This section analyzes and compares the performance of the algorithms based on the average values from these three runs.

4.1 Basic multiplication and Line multiplication comparison

Both **C++** and **Rust** are performance-oriented programming languages that provide developers with direct control over memory management and system interactions, facilitating the creation of highly optimized code. This is apparent when comparing the execution times of basic and line multiplication algorithms, where both C++ and Rust implementations demonstrate **similar performance**, as depicted in the *(Graph1).*

Comparing the execution times of both algorithms, it's clear that the line multiplication algorithm **surpasses the basic multiplication algorithm in terms of speed** - *(Graph1).* This can also be evident in the *(Graph2)*, where the line multiplication algorithm demonstrates **fewer L1 and L2 cache misses** compared to its basic counterpart, resulting in a reduced execution time. Moreover, the correlation between FLOPS (Floating Point Operations Per Second) and cache misses is significant. The line multiplication algorithm consistently maintains higher FLOPS across varying matrix dimensions, indicative of its **efficient access to cache**. While there may be a slight decline in FLOPS as matrix dimensions increase, the line multiplication algorithm exhibits greater stability in its performance compared to the basic algorithm as depicted in *(Graph3).*

4.2 Block multiplication and Line multiplication comparison

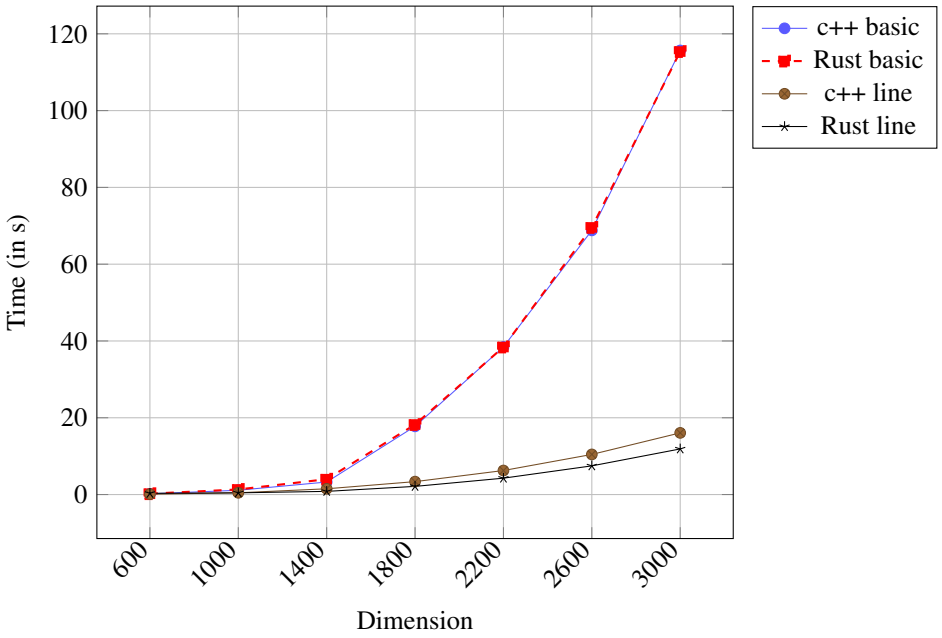
While performing the block multiplication algorithm, it was possible to observe that for most of the matrix dimensions, the block sizes of 512 and 256 outperformed the block size of 128 in terms of execution time eventhough the difference is not quiet significant concluding that block size does not influence the performance when comparing

the performance of each block size. This is evident in the *(Graph4)* and *(Graph5)* as the execution time for all block sizes are similar.

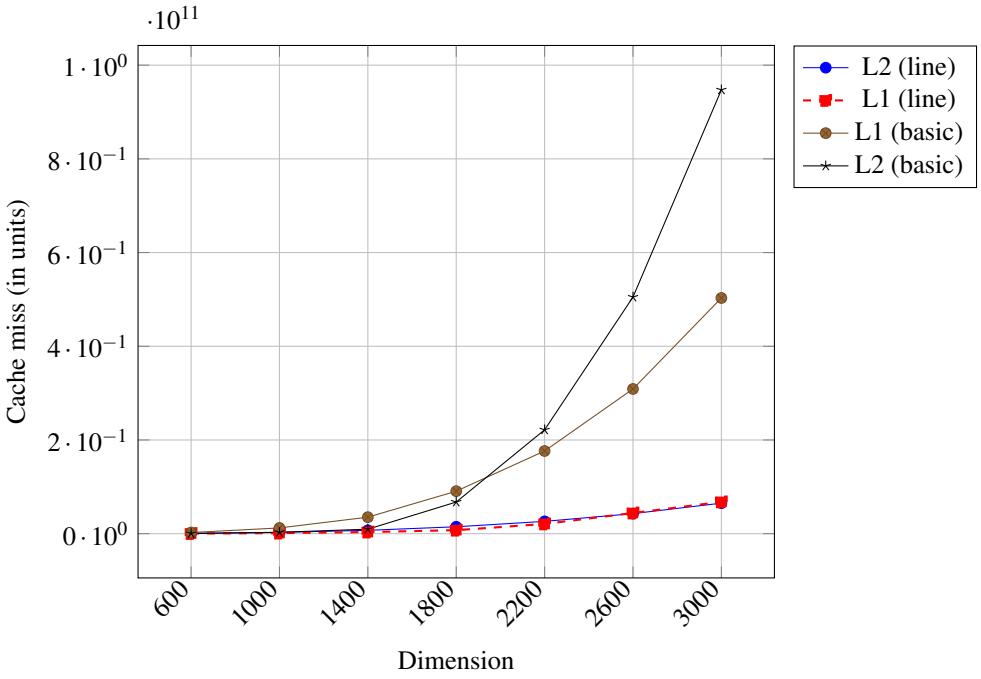
5 Conclusion

6 Annexes

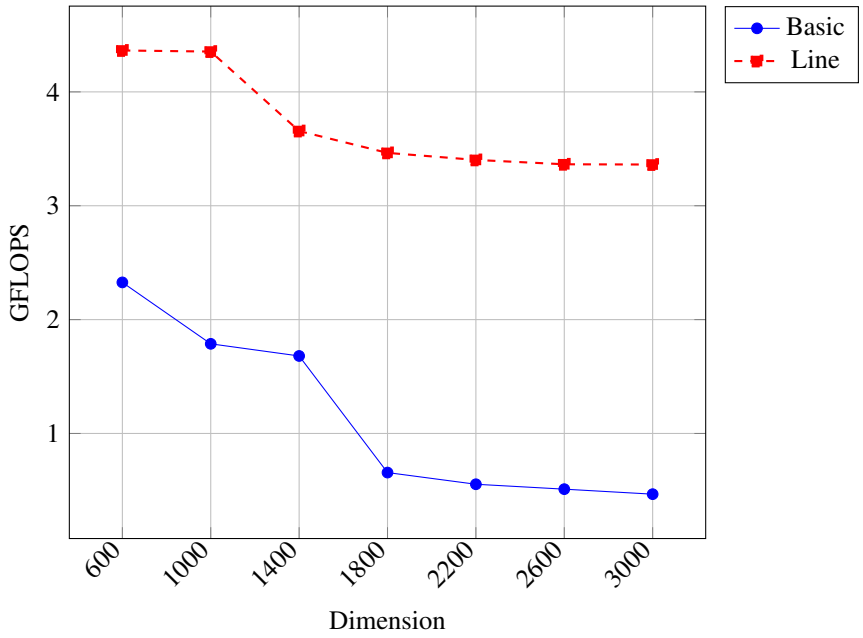
Graph 1: Execution Time: basic vs line (Rust and C++)



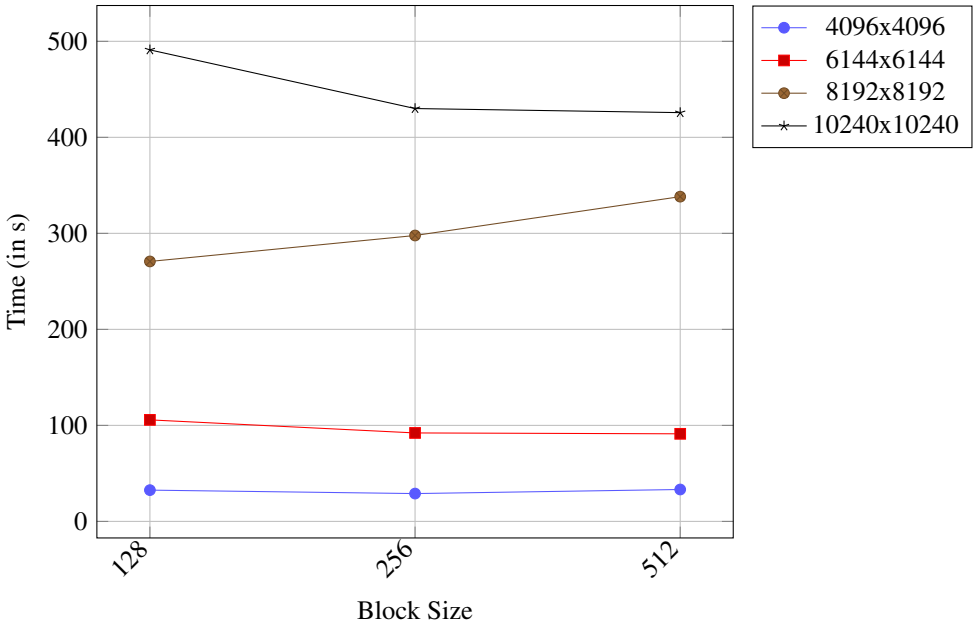
Graph 2: Cache Misses: line vs basic (C++)



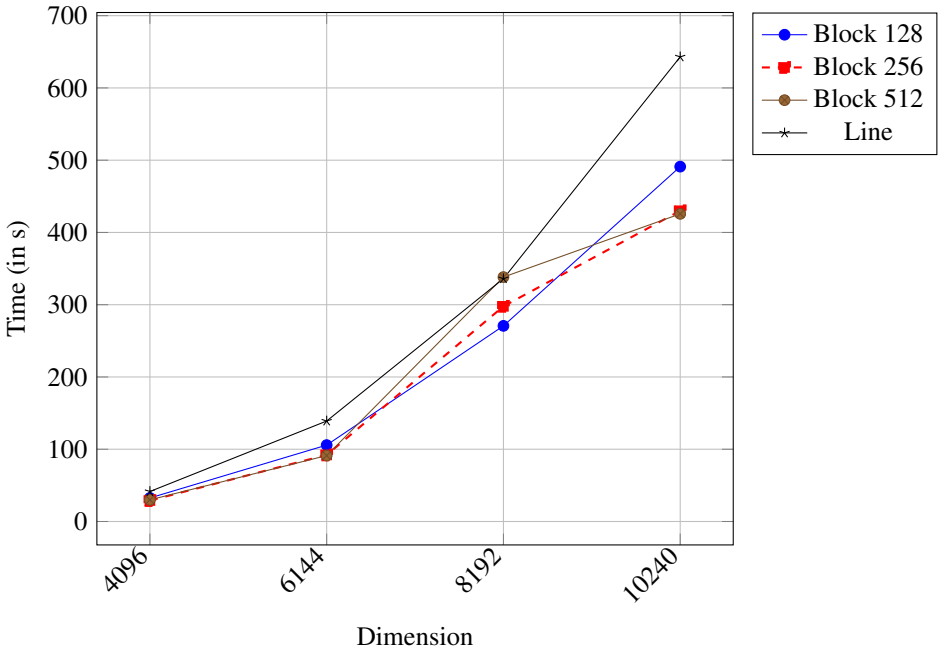
Graph 3: GFLOPS Comparison: Line vs Basic (c++)



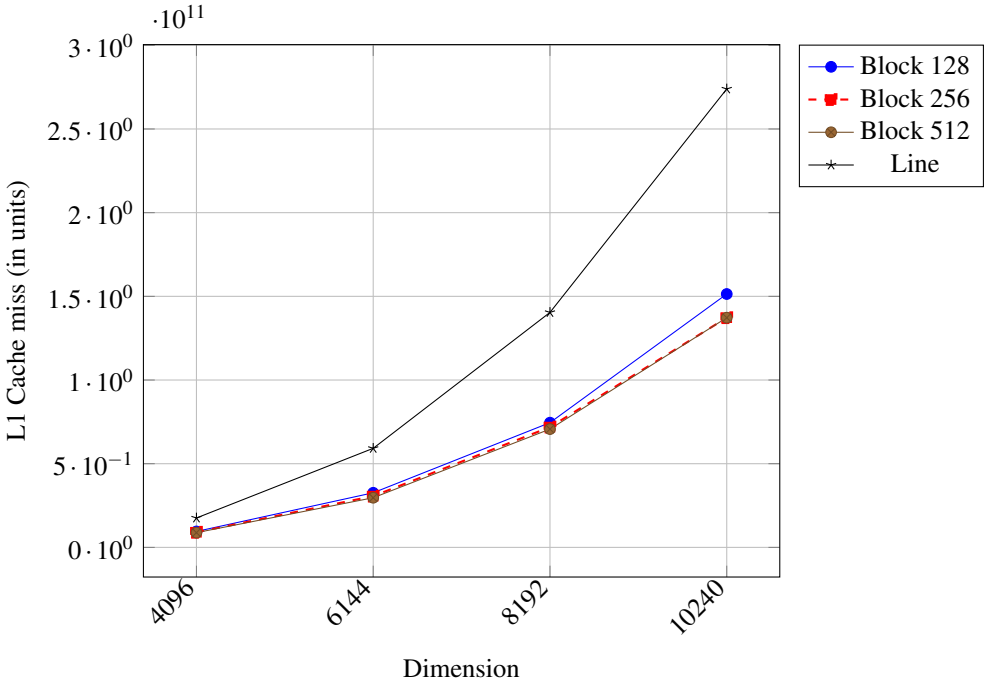
Graph 4: Execution Time: block size relationship (C++)



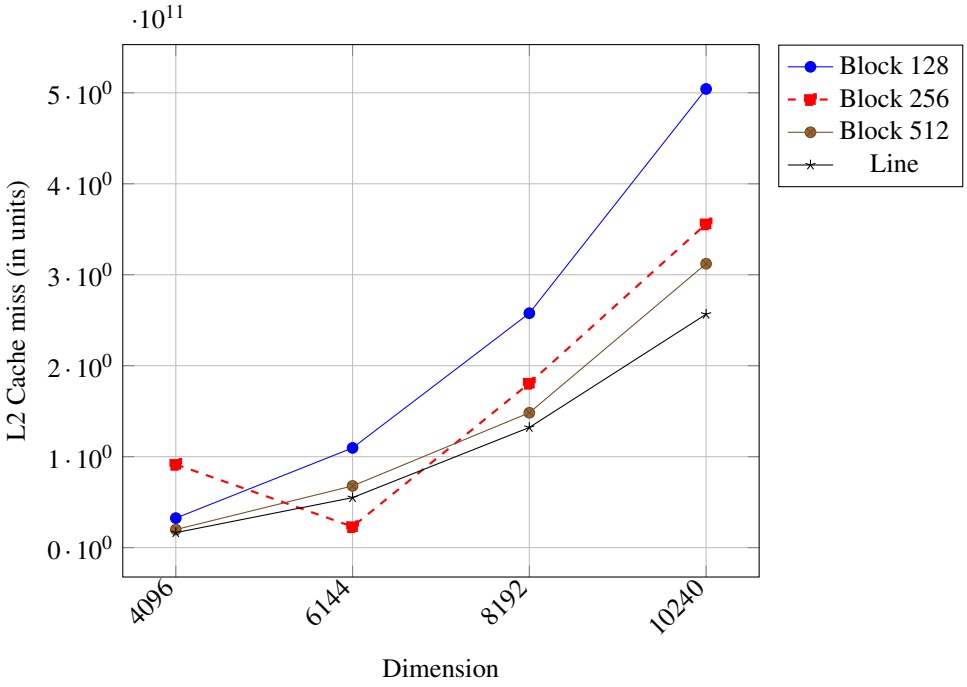
Graph 5: Execution time: Block vs Line (C++)



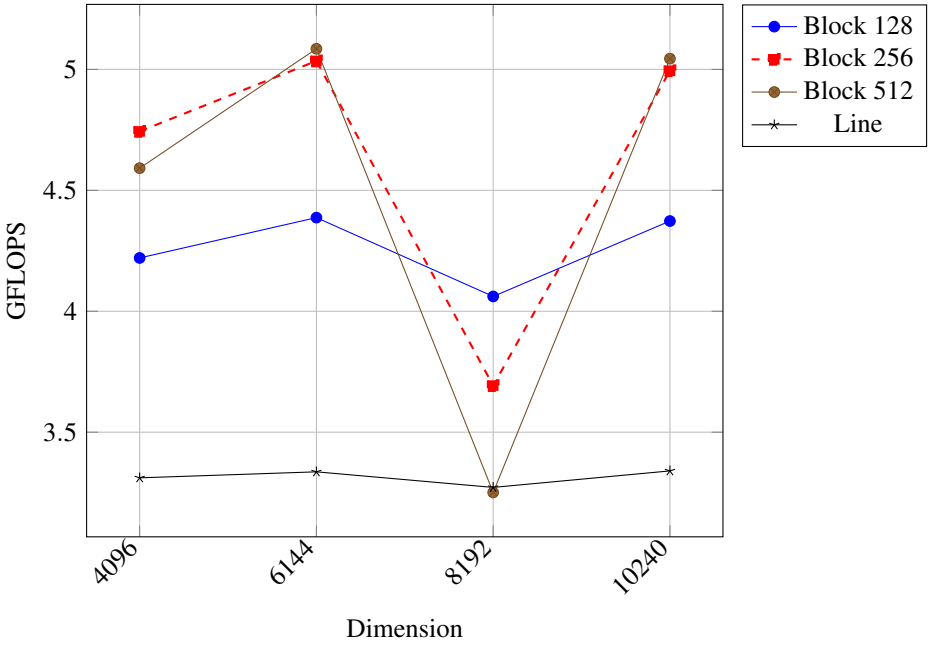
Graph 6: L1 Cache Misses: block vs line (C++)



Graph 7: L2 Cache Misses: block vs line (C++)



Graph 8: GFLOPS Comparison: Line vs Block (c++)



Graph 9: Speedup: line - Parallel 1 vs Parallel 2 (C++)