CPD first project

Turma 12 - Grupo 15

Performance evaluation of a single core
Performance evaluation of a multi-core implementation

Group members:

1. David Carvalho (up202208654@up.pt)
2. João Santos (up202205794@up.pt)
3. Leonardo Teixeira (up202208726@up.pt)

Index

Problem Description 3
Algorithms 3
Single Core Matrix Multiplication 3
1. Simple Matrix Multiplication 3
2. Line Matrix Multiplication 3
3. Block Matrix Multiplication 3
Multicore Matrix Multiplication 4
Parallelizing the Outer Loop 4
Parallelizing the Inner Loop 4
Comparing Thread Performance 4
Performance metrics 4
Results and analysis 5
Basic Implementation Comparison (C++ vs Go) 5
Cache Performance Analysis 5
Block Size Impact Analysis 6
Parallelization Analysis 7
Parallelization Scalability 7
Thread Count Analysis 8
Conclusions 8

Problem Description

In this report, we will study the impact of memory hierarchy on processor performance when handling large datasets. To analyze this, we focus on matrix multiplication, a fundamental operation widely used in fields such as computer graphics, scientific computing, and machine learning. Due to its high computational cost and intensive memory access patterns, matrix multiplication remains a key area of research for optimizing performance. We will evaluate different implementations and optimizations to understand how memory hierarchy influences execution time and efficiency.

Algorithms

Single Core Matrix Multiplication

Even without leveraging parallel computing, sequential programs can be optimized by considering memory management. In this project, we implemented three different algorithms to evaluate the performance of a **single core** when processing large amounts of data.

For Simple Matrix Multiplication and Line Matrix Multiplication, we were required to implement the algorithms in both C/C++ and an additional programming language. We chose Go because it offers efficient memory management, built-in concurrency support (even though not used in this context), and a syntax that is simple yet similar to C, making code translation easier. Go's garbage collection and stack allocation optimizations also influence memory access patterns, impacting execution time.

For Block Matrix Multiplication, we implemented the algorithm only in C/C++.

1. Simple Matrix Multiplication

For the baseline implementation, we were given a basic C/C++ algorithm that performs matrix multiplication by computing the dot product of each row of the first matrix with each column of the second matrix. The time complexity of this algorithm is $O(n^3)$.

2. Line Matrix Multiplication

This version introduces an optimized approach that multiplies an element from the first matrix with the corresponding row of the second matrix. Although the algorithm retains the same time complexity of $O(n^3)$, it improves memory access by modifying the order of operations, reducing cache misses.

3. Block Matrix Multiplication

In this approach, the matrices are divided into smaller sub-matrices (blocks), and computations are performed on these blocks before summing the results. This technique optimizes data locality, allowing more data to remain in lower-level, faster memory caches (such as L1 and L2). By reducing access to higher-level memory (RAM), this method improves execution efficiency.

Although the time complexity remains $O(n^3)$, this method significantly enhances cache utilization, leading to better performance.

Multicore Matrix Multiplication

We explored two distinct methods for performing matrix multiplication using multiple cores, both based on the Line multiplication approach. In both implementations, we ensured a single point for thread creation and synchronization, minimizing overhead. Excessive thread forking inside loops (potentially N or N^2 times) would have led to severe performance degradation.

Parallelizing the Outer Loop

The first strategy focused on parallelizing the outer loop. To prevent conflicts, the inner loop variables (j and k) were declared as private, as they are shared by default. The directive #pragma omp parallel for enabled automatic workload distribution across all available processing units by spawning separate threads.

Parallelizing the Inner Loop

For the second approach, we parallelized the innermost loop. Here, the iteration variables (i and k) needed to be explicitly private. Unlike the outer-loop parallelization, we couldn't just use #pragma omp parallel for on the inner loop, as it would lead to excessive thread creation. Instead, we used #pragma omp parallel to create a fixed set of threads, followed by #pragma omp for to distribute loop iterations among them. However, this method generally performed worse because of thread synchronization overhead at each loop iteration.

Comparing Thread Performance

To analyze performance differences, we tested the program with 2, 4 and 8 threads. This comparison helped us understand how scaling the number of threads affects execution time and synchronization overhead.

Performance metrics

To assess the efficiency of the algorithms implemented in C/C++, we utilized the Performance API (PAPI). This tool provides access to a wide range of CPU performance metrics and we focused on L1 and L2 cache misses, as they have a direct impact on processing speed. A high number of cache misses significantly increases memory access latency, making this metric crucial for evaluating the effectiveness of different memory access strategies.

To ensure consistency and eliminate potential biases in the results, we executed all benchmarks on the same machine and repeated each test **four times**, using FEUP’s computers and **using the average of these four benchmarks as the final result**. The system used was running Ubuntu 22.04 on an Intel Core i7-9700 processor. Averaging the results from multiple runs helped mitigate fluctuations caused by background processes and system state variations.

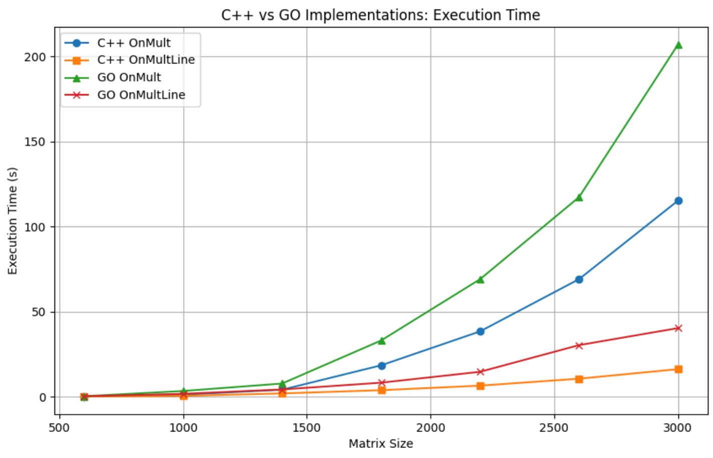
In the C/C++ implementation, we compiled the code with the `-O2` optimization flag, which balances compilation speed with performance improvements.

To accurately measure execution time, we used C++'s `<chrono>` library and Go’s `time` package. These tools allowed us to obtain precise time measurements, ensuring a direct evaluation of how efficiently each implementation executes.

Additionally, we compared the execution times of C/C++ with those of our Go implementation, highlighting the differences in memory management, execution efficiency, and overall performance.

Results and analysis

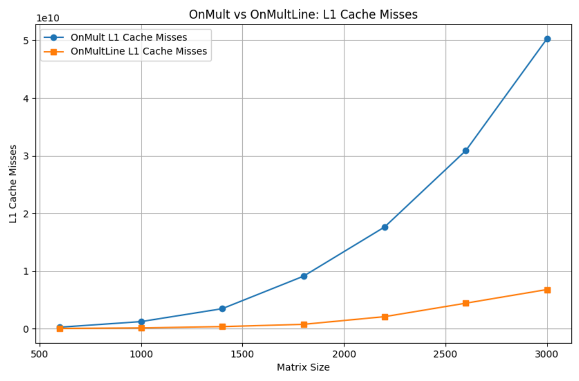
Basic Implementation Comparison (C++ vs Go)

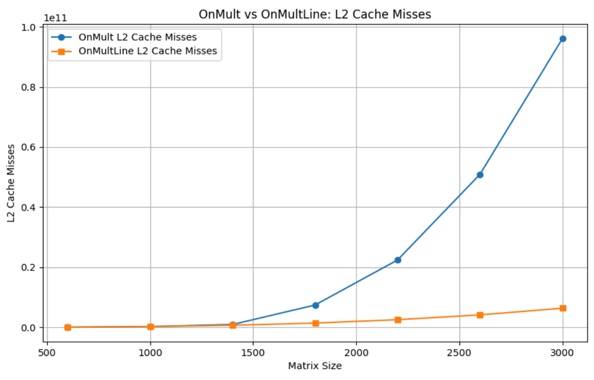


The first chart compares the basic matrix multiplication implementations in both C++ and Go for matrix sizes up to 3000×3000:

- C++ outperforms Go across all matrix sizes. The performance gap widens significantly as matrix sizes increase.
- Both languages show similar scaling patterns, with execution time increasing with matrix size according to the $O(n^3)$ complexity.
- For the largest matrices (3000×3000), the C++ implementation runs approximately 3-4 times faster than the equivalent Go code.
- The OnMultLine (element-by-line multiplication) implementation consistently outperforms the standard OnMult implementation in both languages, demonstrating the benefit of improved memory access patterns.

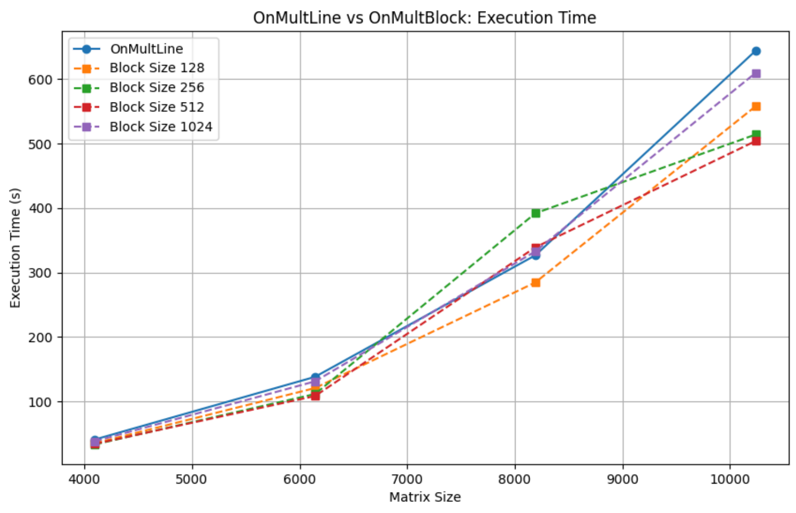
Cache Performance Analysis

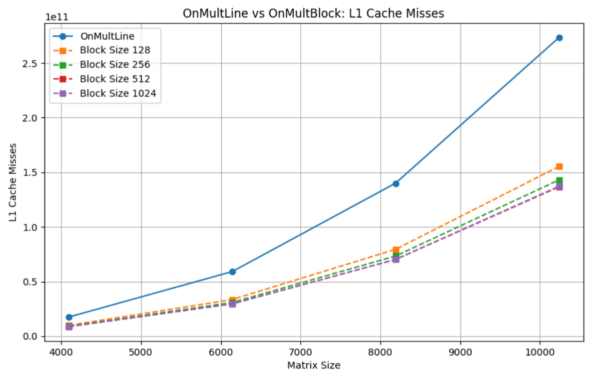


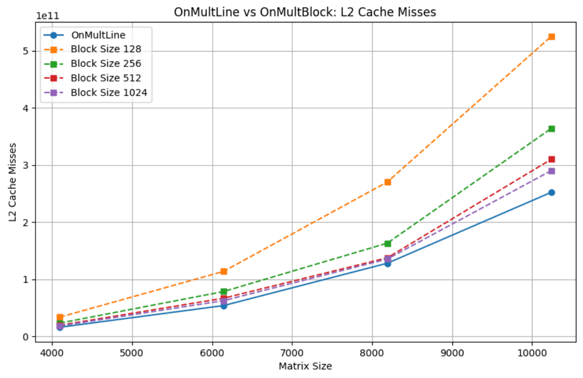


The cache miss data explains the performance advantage of OnMultLine over OnMult:
- OnMult suffers from significantly more L1 cache misses than OnMultLine (about 9-10 times more for larger matrices).
- This difference directly results from memory access patterns: OnMult accesses the second matrix in a non-contiguous pattern, causing more cache misses.
- L2 cache misses show a more variable pattern, but OnMultLine generally maintains an advantage, especially at larger matrix sizes.
- The cache behavior explains the substantial performance difference between the two implementations despite their identical computational complexity.

Block Size Impact Analysis





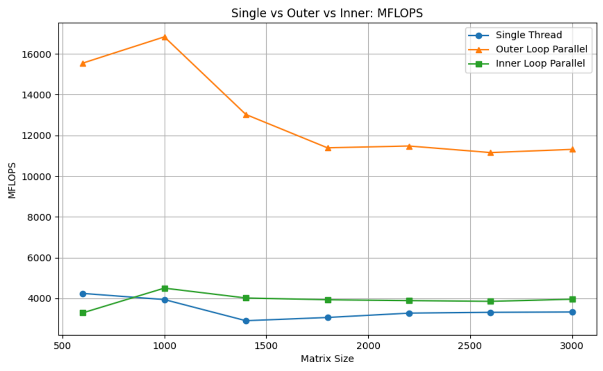


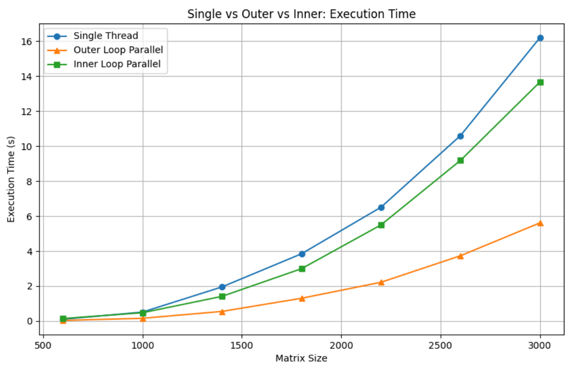
The block multiplication approach (OnMultBlock) shows interesting performance characteristics:

- Optimal block size depends on matrix size. For the tested matrices (4096×4096 to 10240×10240), a block size of 512 generally provides the best performance.
- Block sizes that are too small (128) or too large (1024) perform worse than the optimal size.

- The block approach outperforms standard OnMultLine for most configurations, with execution time improvements of 5-15%.
- L1 cache misses are significantly reduced by the block approach (by about 40-50% compared to OnMultLine).
- L2 cache misses show a more complex pattern, with the block approach actually causing more L2 misses than OnMultLine, but this is offset by the L1 cache benefits.
- The improvement comes from better spatial and temporal locality, keeping blocks of the matrices in cache.

Parallelization Analysis

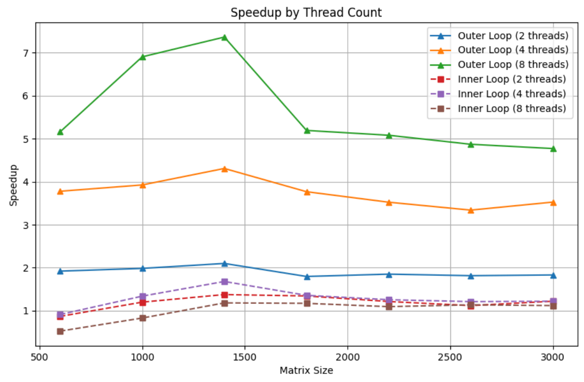


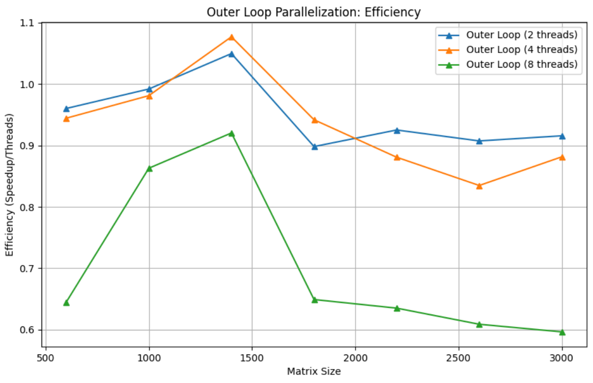


Parallelizing matrix multiplication yields significant performance improvements:

- Outer loop parallelization consistently outperforms inner loop parallelization by a substantial margin.
- Outer loop parallelization achieves 2-6x speedup compared to single-threaded execution, depending on matrix size and thread count.
- Inner loop parallelization shows limited benefits, with speedups rarely exceeding 1.5x, and sometimes performing worse than single-threaded execution for small matrices.
- The MFLOPS chart confirms these observations, showing much higher computational throughput for outer loop parallelization.
- The difference stems from work distribution and synchronization overhead; inner loop parallelization requires more synchronization, limiting scalability.

Parallelization Scalability

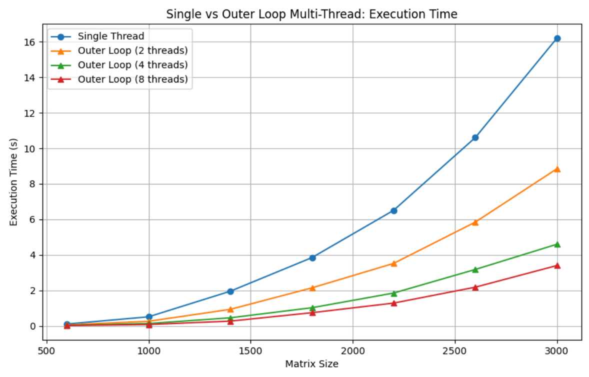


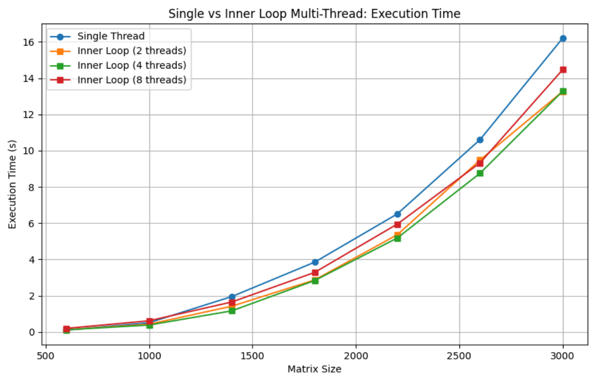


The speedup and efficiency metrics provide insights into parallel scalability:

- Outer loop parallelization scales well, with speedup increasing with thread count across all matrix sizes.
- For outer loop parallelization, the best scaling occurs at matrix sizes 1800-3000, where the 8-thread implementation achieves speedups of 5-6x.
- Inner loop parallelization scales poorly, with minimal speedup beyond 2 threads and efficiency dropping rapidly.
- Efficiency (speedup/number of threads) decreases with increasing thread count, which is expected due to coordination overhead and resource contention.
- The efficiency chart shows diminishing returns beyond 4 threads for most matrix sizes.
- Smaller matrices (600-1000) show lower parallel efficiency due to insufficient work to amortize the threading overhead.

Thread Count Analysis





The impact of thread count on execution time provides practical insights:

- For outer loop parallelization execution time decreases significantly with increasing thread count across all matrix sizes.
- The benefit from 2 to 4 threads is substantial, while the improvement from 4 to 8 threads is less pronounced but still significant.
- For inner loop parallelization, the benefit of adding threads is minimal, and in some cases, more threads actually degrade performance.
- This suggests that for this particular workload and system, outer loop parallelization with 8 threads represents the most efficient configuration.

Conclusions

In conclusion, this work provided deeper insight into the crucial role of memory management in optimizing program efficiency. Even without parallel computing, sequential programs can be significantly improved by employing techniques that optimize memory access, taking into account the hardware on which they run.

The differences in execution time between algorithms highlight the importance of designing code to maximize the use of lower-level memory and minimize reliance on higher-latency memory, reducing unnecessary performance overhead.