Performance evaluation of a single and multi-core implementation

Computação Paralela e Distribuída (CPD)

2024/2025

Class 09, Group 16

Gonçalo Cunha Marques - 202206205

Miguel Lopes Guerrinha - 202205038

Rui Pedro da Silva Cruz - 202208011

Index

Problem Description

In this project, we will analyze the impact of memory hierarchy on processor performance when handling large amounts of data. To conduct our tests, we use matrix multiplication, a fundamental operation widely applied in fields such as computer graphics and machine learning. This method is particularly useful for our study because it is computationally intensive and heavily dependent on memory access patterns, making it an effective benchmark for evaluating performance bottleneck in modern processors.

Algorithms

This project was divided into two distinct parts. At first, we implemented three different algorithms for matrix multiplication, each designed to evaluate the performance of a single core when handling large amounts of data. These algorithms primarily differ in how they optimize memory allocation and access patterns, allowing us to assess the impact of memory hierarchy on computational efficiency.
These algorithms are:

1. **Simple Matrix Multiplication**
2. **Line x Line Matrix Multiplication**
3. **Block Matrix Multiplication**

We were tasked with developing code in C/C++ and another programming language to implement both Simple Matrix Multiplication and Line-by-Line Matrix Multiplication. For this task, we chose Java due to its syntactic similarity to C/C++. This similarity facilitated the translation process between the two languages, making the implementation more efficient and manageable. For Block Matrix Multiplication, we were only tasked with developing the code in C/C++ and analysing its performance.
Afterwards we were asked to test the parallel version of the Line x Line algorithm, to verify the CPU performance compared to the single-core version.

Simple Matrix Multiplication

For this algorithm, we were already given a C/C++ program that multiplies matrices the traditional way, i.e. multiplies one line of the first matrix by each column of the second matrix. Here is the code of the algorithm in C/C++:

```
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;
        }
    }
```

Line x Line Matrix Multiplication

Here we implemented an improved algorithm that multiplies an element from the first matrix by the corresponding line of the second matrix. This algorithm has the same complexity of the last one but as we will see forward, has a better performance. The following is the C/C++ code for the algorithm:

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

Block Matrix Multiplication

For the Block Matrix Multiplication, we implemented an algorithm that splits the matrices into smaller blocks that are calculated separately using the previous Line x Line Matrix Multiplication algorithm and added up in the end.

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

Performance Metrics

To evaluate the performance of the algorithms in the C/C++ implementations, we analysed metrics such as execution time, the number of cache misses for both L1 and L2, and the number of Floating Point Operations Per Second (MFLOPS) while running the algorithms. To collect this data, we used the PAPI (Performance API), which provides access to critical CPU performance metrics.

These metrics were gathered on the same computer to ensure consistency in the data collection. The system used was running Ubuntu 22.04, with an Intel i5-1135G7 processor, a base clock of 2.40 GHz, 96 KB of L1 cache per core, 1.25 MB of L2 cache per core, and 8 MB of shared L3 cache. Calculating the average values accounts for minor variations in measurements due to factors beyond our control.

When compiling in C++, we used the -O2 flag to enhance performance and improve compilation time. Subsequently, we compared the execution time of each algorithm in C++ against its Java implementation.

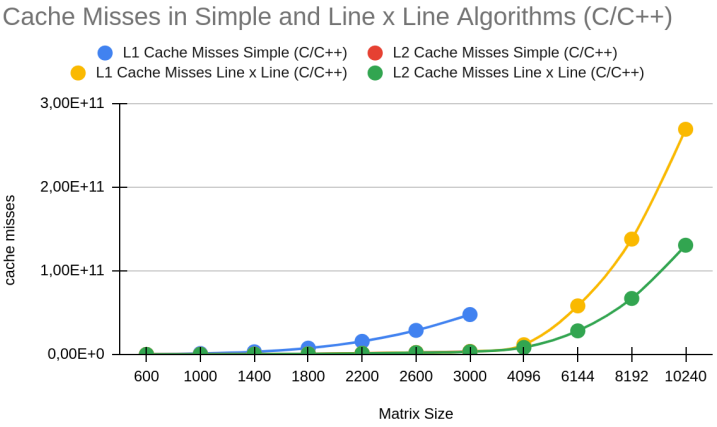
Results and Analysis

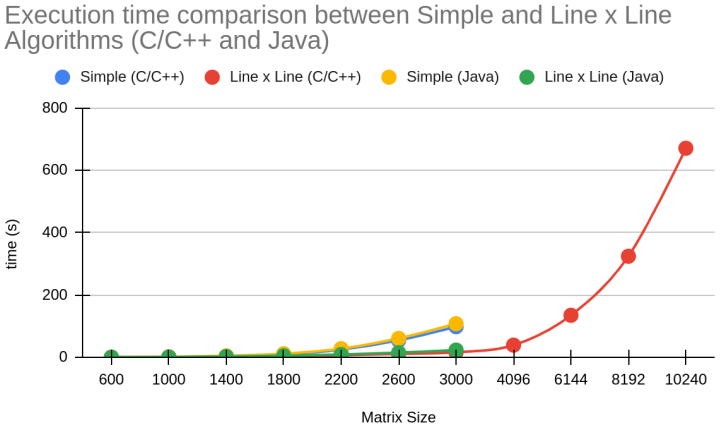
The performance of each algorithm was analysed by doing several tests using varying matrix sizes. To ensure accuracy, we performed five consecutive tests for each matrix size. Additionally, we created graphs to visualize and better interpret the obtained results.

Simple and Line x Line Matrix Multiplication Algorithms in C/C++ and Java

These algorithms were tested in both C/C++ and Java, and both languages achieved nearly identical execution times. By analyzing the graphs below, we concluded that the Line x Line algorithm is more efficient than the simple algorithm, as it results in fewer cache misses, leading to faster execution times.

This efficiency is due to the Line x Line algorithm accessing memory in a more sequential manner, improving cache utilization and reducing the number of costly memory accesses compared to the simple algorithm, which frequently loads data from slower memory levels.



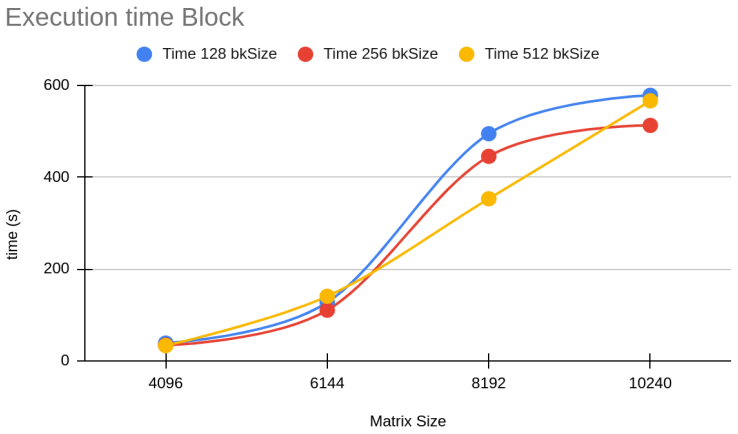


Cache Misses Using Block Matrix Multiplication Algorithm

As previously mentioned, this algorithm was implemented only in C/C++ and analyzed using various matrix and block sizes. The execution time and cache miss results are presented in the graphs below.

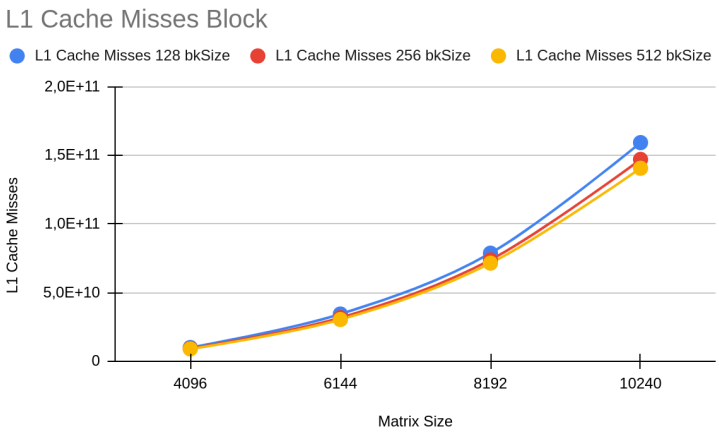
It was observed that the Block Matrix Multiplication algorithm has nearly the same execution time as the Line x Line algorithm for matrix sizes up to 8192×8192.

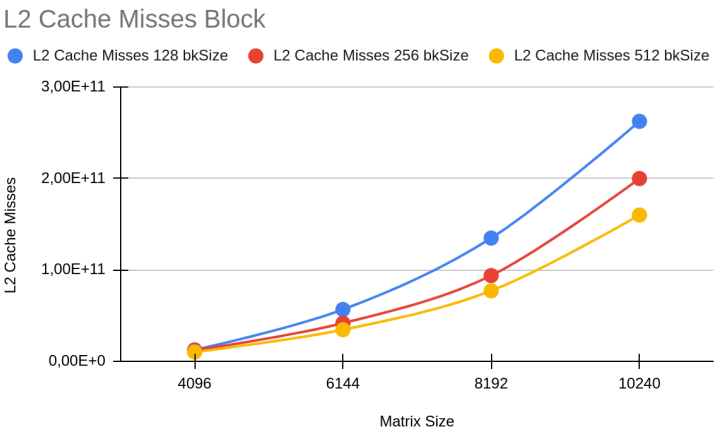
However, for larger matrices, the Block Matrix Multiplication algorithm demonstrates a significantly lower execution time. Additionally, the execution time remains relatively consistent across different block sizes, with minimal variation.



Analysing the Cache Misses at L1 and L2 level we observe that in the Block Matrix Multiplication algorithm we have less L1 cache misses and more L2 cache misses compared to the Line x Line one. Despite both algorithms having similar execution times, this suggests that the time lost fetching data from L1 cache to higher memory levels ends up being more costly than the time of doing that from the L2 level, impacting overall performance.

At the L1 cache level, the block size does not significantly impact performance, as L1 is the primary and fastest cache. However, at the L2 cache level, we observe that larger block sizes result in fewer cache misses. This can be explained by better spatial locality, larger blocks allow more data to be loaded into the cache at once, reducing the need for frequent memory accesses and improving overall efficiency.



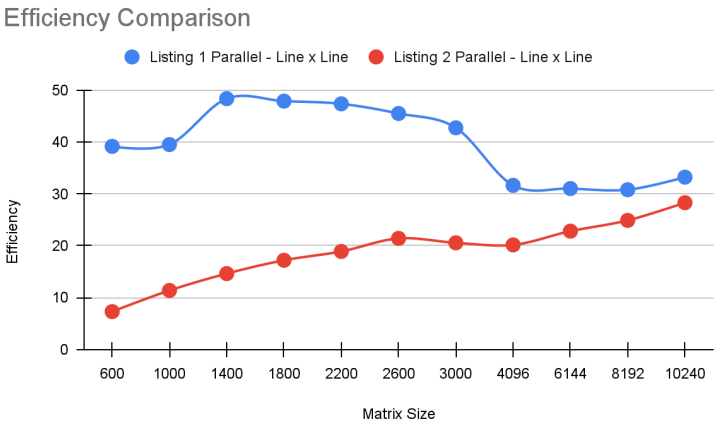


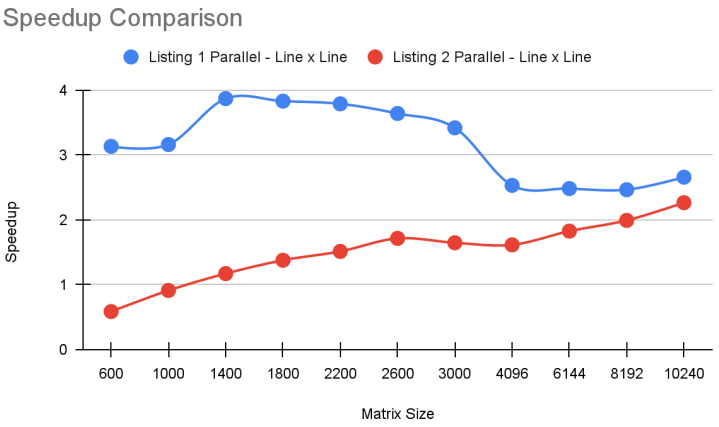
Performance Evaluation of a Multi-core Implementation

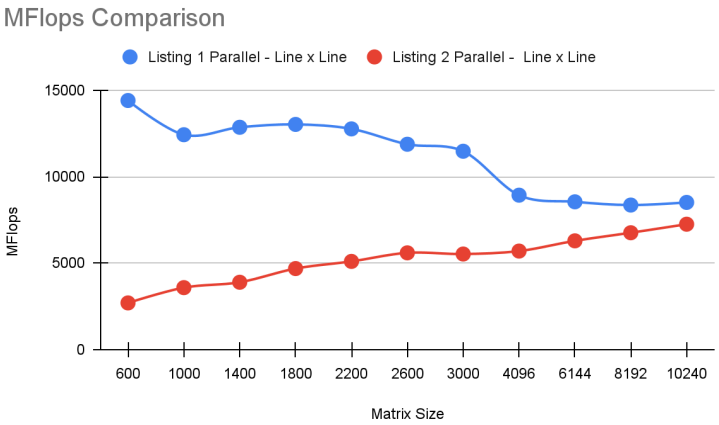
For the Line x Line algorithm we have been tasked to implement two parallel versions to compare the performance of the CPU when using just a single core and when using multi-core. The first version (Listing 1) consists in providing parallelism to the outermost level, whereas the second version (Listing 2) provides parallelism at the innermost level.

To verify their performance we calculate the MFlops, speedup and efficiency for both versions. This parameters were calculated by the following formulas:

- **Number of Operations:** $2 * MatrixSize^3$
- **MFlops:** $NumberOperations / (ExecutionTime * 10^6)$
- **Speedup:** $SequentialExecutionTime / ParallelExecutionTime$
- **Efficiency:** $Speedup / NumberThreadsUsed$







Analyzing the efficiency graph, we observe that both parallel implementations outperform the single-core execution. However, when comparing the two parallelization strategies, we notice distinct trends in their efficiency behavior. The

first version (Listing 1) demonstrates higher efficiency across all tested matrix sizes but tends to decline in efficiency as the matrix size increases. In contrast, the second version (Listing 2) shows an increasing efficiency trend with larger matrix sizes, suggesting better scalability. This is due to the fact that in smaller matrices, Listing 1 is more efficient thanks to minimal overhead: each thread processes contiguous rows and needs fewer synchronizations. As matrices grow, Listing 2 distributes the innermost loop more flexibly for better load balancing, so it scales better. However, in the tested range, Listing 1 still outperforms Listing 2 overall.

Examining the speedup graph, we observe that the first parallel implementation consistently achieves a significant speedup compared to single-core execution. In contrast, the second version only begins to show a noticeable speedup for matrix sizes larger than 1000. Similar to the efficiency trends, the first version experiences a decline in speedup as the matrix size increases, whereas the second version demonstrates an upward trend in speedup with larger matrices. Despite this difference, the first version remains the more efficient approach across all tested cases.

From the MFlops perspective, Listing 1 consistently achieves higher floating-point throughput. Even though its MFlops decline as matrix size grows, it still outperforms Listing 2, thanks to the reduced overhead from parallelizing the outer loop and effectively distributing work among threads.

Conclusions

In conclusion, we explored various matrix multiplication algorithms and saw how memory accesses directly influence performance. The ‘line-by-line’ method proved to be superior to the ‘simple’ method due to its more efficient sequential access. The ‘block’ approach stood out for larger sizes, thanks to improved spatial locality.

On the parallel side, the distribution of work in the outer loop revealed less overhead, but the inner loop showed better scalability for very large matrices.

Overall, this project has shown how parallelisation decisions and memory access patterns are key to optimising performance in modern multicore systems.