Faculty of Engineering University of Porto

‘Assignment 1 - CPD’

Class 11 Group 11

Authors:

Carlos Costa (up202205908@up.pt)	Participation: 33%
Luciano Ferreira (up202208158@up.pt)	Participation: 33%
Tomás Telmo (up202206091@up.pt)	Participation: 33%

1 - Problem Description & Algorithm Explanation

This assignment aims to analyze the impact of memory hierarchy access on processor performance when handling large datasets, as well as the effect of parallelization on overall performance. To investigate this, we will study different matrix multiplication algorithms written in two different languages, C++ and Haskell, and we'll compare their performance based on these given metrics.

Implemented Algorithms

Each algorithm is implemented in a function, as shown below:

Single-Core:

-> C++ Algorithms (located in *matrixproduct.cpp*)

- ***OnMult()*** – Standard matrix multiplication
- ***OnMultLine()*** – Optimized version using row-wise operations
- ***OnMultBlock()*** – Block-based multiplication aimed at cache efficiency

-> Haskell Algorithms (located in *matrixprod.hs*)

- ***matrixMultiply()*** **-** Standard matrix multiplication,
- ***onMultLine()*** **-** Optimized version using row-wise operations

Important **-** In order to run the haskell algorithms, you will need to **download ghci** (haskell's compiler), then run the following commands on a terminal inside the project folder:

windows

```
>  cd .\src\
> ghci .\matrixprod.hs
ghci> main
```

linux

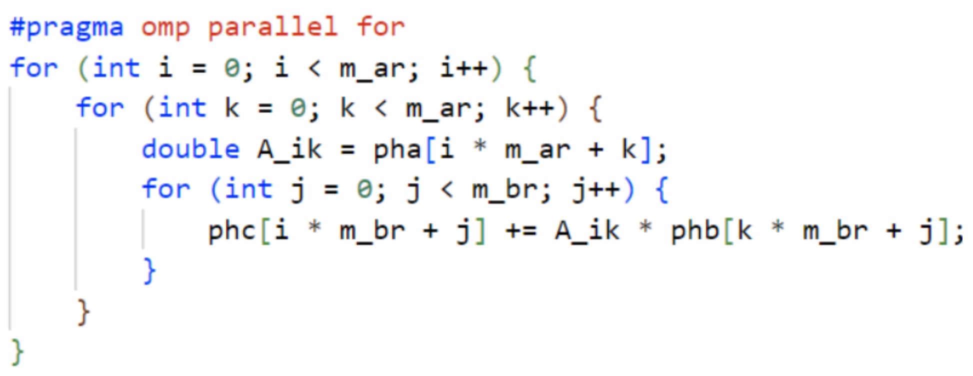
```
>> cd src
>> ghci matrixprod.hs
ghci> main
```

(afterwards choose the matrix size and the algorithm you want to test)

(Multi-Core):

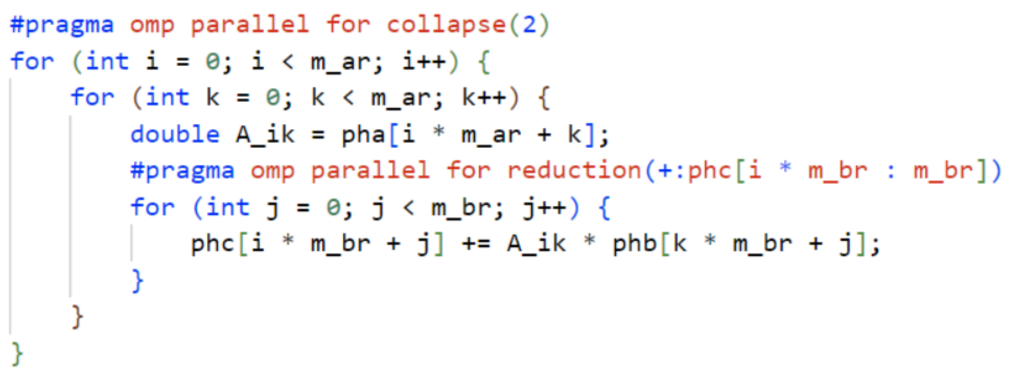
The multi-core functions, written in C++, use the same row-wise multiplication algorithm as the OnMultLine() function. The main difference is that they will use the **Open Multi-Processing** (openMP) API to achieve **parallel execution on multi-core processors.** In our code, we will demonstrate this by using pragmas provided in the OpenMP API.

As requested, we will implement the parallel processing in two different ways:



```
#pragma omp parallel for
for (int i = 0; i < m_ar; i++) {
    for (int k = 0; k < m_ar; k++) {
        double A_ik = pha[i * m_ar + k];
        for (int j = 0; j < m_br; j++) {
            phc[i * m_br + j] += A_ik * phb[k * m_br + j];
        }
    }
}
```

***Fig1** - listing 1*



```
#pragma omp parallel for collapse(2)
for (int i = 0; i < m_ar; i++) {
    for (int k = 0; k < m_ar; k++) {
        double A_ik = pha[i * m_ar + k];
        #pragma omp parallel for reduction(+:phc[i * m_br : m_br])
        for (int j = 0; j < m_br; j++) {
            phc[i * m_br + j] += A_ik * phb[k * m_br + j];
        }
    }
}
```

***Fig2**-listing 2*

OnMultLineParallel() -

Implements **listing 1** as is provided in the assignment packet.

OnMultLineParallel2() -

Implements **listing 2**, which slightly differs from the provided assignment code in the fact that it **parallelizes i and k together** *(collapse(2))* and distributes them across threads, making the iteration space evenly divided among threads.

On top of this, we added a **reduction to remove race conditions on key variables,** so that threads do not have to compete over who gets to update the variable. This works by providing **each thread** with a private copy of, in this case, ***phc[i * m_br : m_br]***, allowing them to **compute partial results independently in parallel.** In the end, OpenMP efficiently merges the partial sums into the final result.

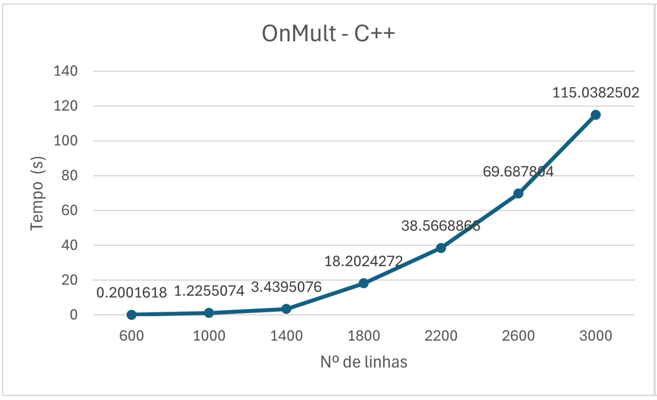
2 - Performance Metrics

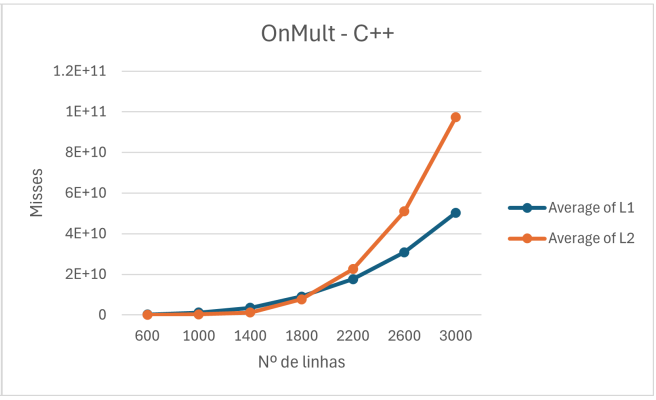
- For the single-core algorithms written in C++, we'll use the Performance API (PAPI) to collect program **execution time**, **L1 cache misses,** and **L2 cache misses.** Additionally, we will vary the matrix size as a means to test how the metrics vary.
- For the single-core algorithms written in Haskell, we will measure only the **execution times**, varying the matrix size, aiming to compare the results with those of the single-core C++ implementations.
- For the multi-core algorithms written in C++, we'll use the **same metrics as the single-core version**, with the addition of **MFlops**, **speedup**, and **efficiency**, while additionally varying the matrix size.

Note - for a given matrix size, we will run the experiment five times, using the median of the results for analysis.

3 - Results & Analysis

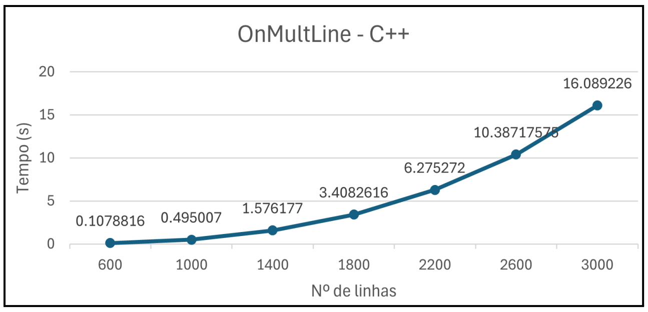
OnMult()- C++:

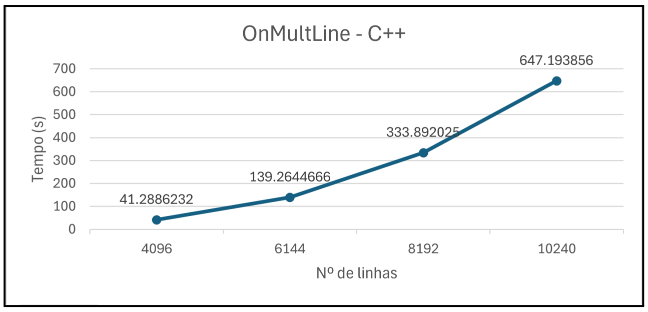


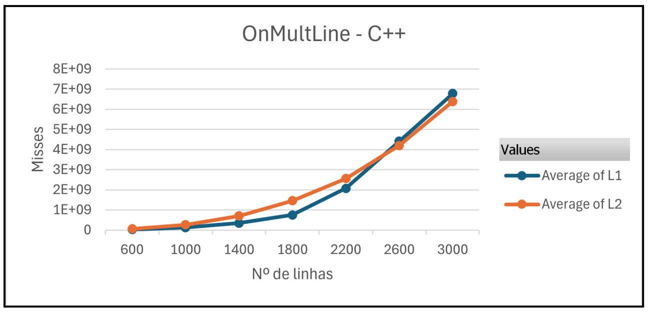


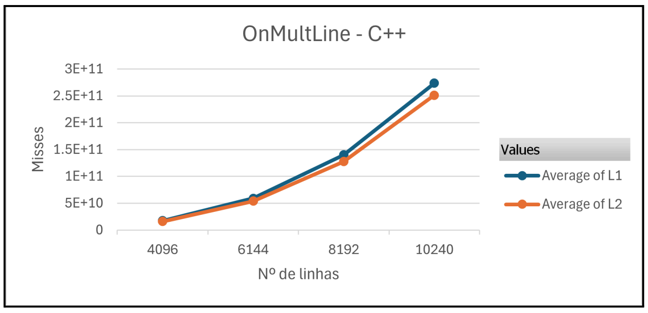
The execution time plot for *OnMult()* as a function of matrix size follows **a polynomial growth**, suggesting that execution time increases non-linearly with matrix size. This also suggests an *O(n^3)* complexity, which is expected for naive matrix multiplication (O(N^3), due to three nested loops). Additionally, we observe a **high average number of L2 cache misses**, indicating that the CPU frequently **has to fetch data from main memory**, which is significantly more expensive in terms of latency and efficiency.

OnMultLine() - C++:









Likewise, the execution time plot for *OnMultLine()* as a function of matrix size also exhibits a polynomial growth pattern, resembling *O(n³)* complexity. When dealing with big matrices, the **average number of L1 cache misses** appears to be **on par with L2**, while sometimes being a bit higher.

OnMultBlock() - C++

Matrix Size	Block Size	Time(s)	Misses L1	Misses L2
4096	128	91.081335	70016563772	1.02064E+11
4096	128	90.983427	70015592187	1.0198E+11
4096	256	143.207782	75116018296	1.09147E+11
4096	256	134.526631	74774699750	1.08565E+11
4096	512	304.922192	71395397606	86854279091
4094	512	386.259758	70762041591	77468400607
6144	128	290.019814	2.36379E+11	4.21181E+11
8192	128	711.3586	5.60235E+11	1.07346E+12

10240	128	1408.523955	1.09404E+12	1.55525E+12

Just like before, as the matrix size increases, both the execution time and the number of cache misses display polynomial growth, suggesting an O(n^3) complexity. Regarding block size, increasing the block size leads to a higher execution time. This can be attributed to the better cache locality provided by smaller block sizes, and more frequent memory accesses when using larger blocks. As for the cache misses, both L1 and L2 misses increase from block size 128 to 256, but then decrease at block size 512 (especially in L2). This reduction in L2 cache misses can be explained by the larger blocks (512) making better use of the L2 cache, even though they don't fit as efficiently into the L1 cache. As for the L1 misses at block size 512, this decrease can be attributed to better spatial locality in the larger blocks, which results in fewer L2 evictions and fewer L1 misses as the data stays in cache longer. The L1 cache in block size 512 might evict data less frequently, benefiting from better cache usage at both L1 and L2 compared to block size 256. Overall, it seems to depend on how well the block size aligns with the cache size.

OnMult() VS OnMultLine()

A notable observation upon comparison is the significant difference in **execution time** for multiplying two matrices of the same size, depending on the function used—especially for larger matrices.

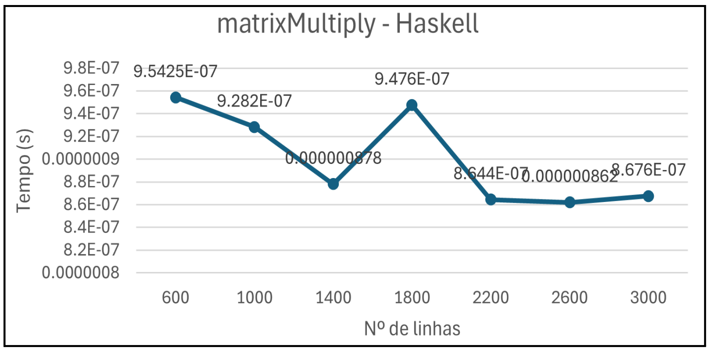
For instance, when multiplying **two 3000 × 3000 matrices**:

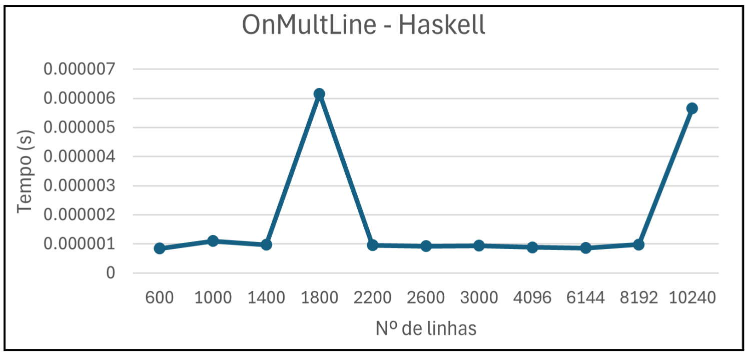
- **OnMultLine()** completes the computation in **16 seconds**
- **OnMult()** takes **115 seconds**—it is more than **7 times slower**

Perhaps more interesting is the **difference in cache misses** between the two functions. For a 3000 x 3000 matrix multiplication, the L2 cache misses in *OnMult()* can reach exponents of 11, which is 2 orders of magnitude higher than the exponents of 9 observed in *OnMultLine()*. Only in the case of an 8192 x 8192 matrix multiplication does *OnMultLine()* reach L2 cache misses of the same magnitude as OnMult().

These improvements over *OnMult* serve as evidence to support the fact that *OnMultLine's* use of **row-wise access patterns** reduces cache misses, therefore making ***OnMultLine*** the **better function of the two**, both in terms of execution time and memory usage.

Haskell Functions:





Immediately we notice that both Haskell implementations run extremely quickly when compared to their C++ counterparts.

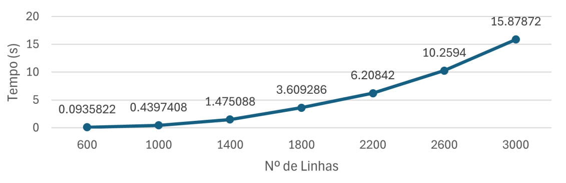
This is due to Haskell's innate **memoization and lazy evaluation capabilities**. In other words, the fact that the language avoids running redundant calculations by **storing** (in heap) **and reusing previously calculated results** in conjunction with **expressions being only evaluated when their values are actually needed,** makes it possible to compute this matrix multiplication extremely quickly.

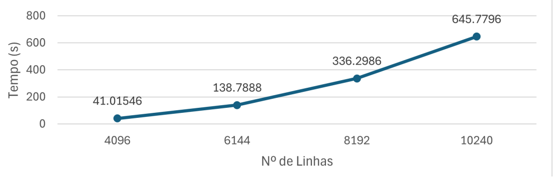
There are some visibles outliers when it comes to execution time, but we found no exact evidence of what could be causing such fluctuations, therefore we are leaving it to possible hardware inconsistencies. That being said, since the variations are very small we chose to ignore them in our analysis.

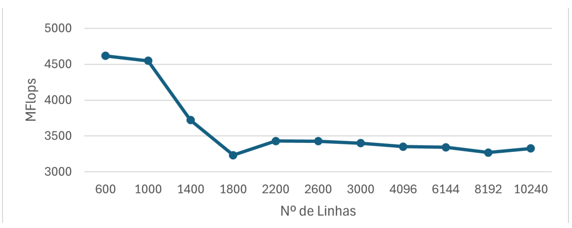
C++ Multi-Core Algorithms:

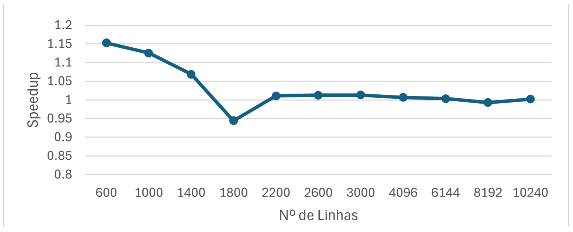
Note: There was no need to create four additional graphs, L1 and L2 misses for both functions, since the values were nearly identical, resulting in flat lines.

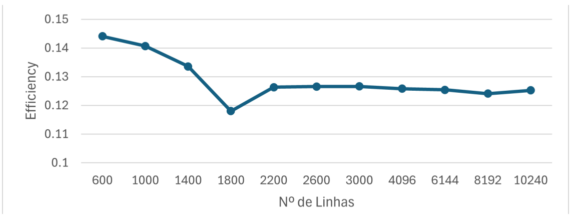
OnMultParallel():



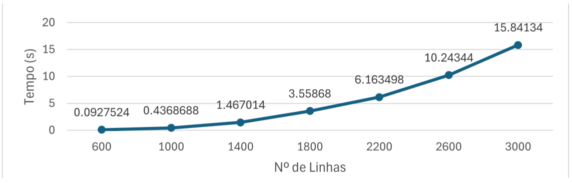


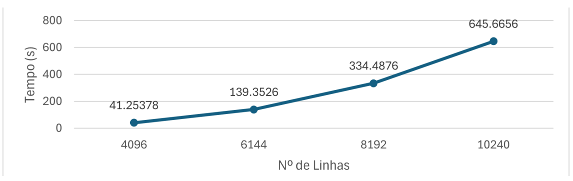


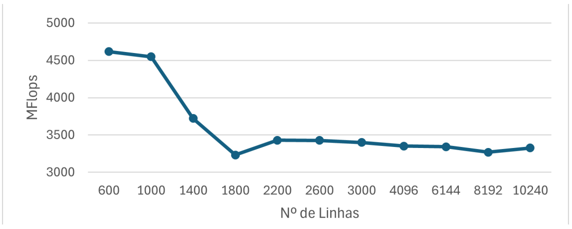


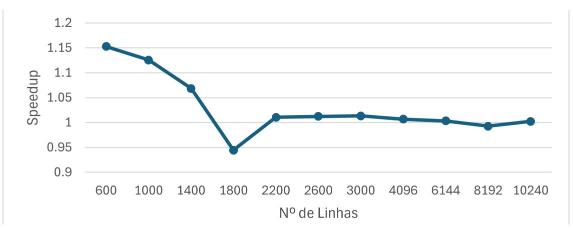


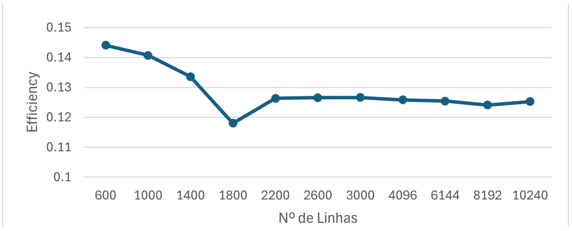
<u>**OnMultParallel2():**</u>











About *OnMultLineParallel()*, as the matrix size increases, the execution time grows significantly, reflecting the O(n³) complexity of matrix multiplication. For smaller matrices, the function performs efficiently due to its straightforward parallelization strategy. However, as the matrix size grows, the limitations of parallelizing only the outer loop become apparent. The lack of finer-grained parallelism leads to suboptimal cache utilization, resulting in higher L1 and L2 cache misses.

About *OnMultLineParallel2()* even though it employs a more advanced parallelization strategy, the execution times and cache miss values were almost the same as those of the first parallelized function. This similarity suggests that the additional overhead from nested parallelism and the reduction operation may offset the potential benefits of finer-grained parallelism.

4 - Conclusions

The results of our project highlight the significance of memory hierarchy and access patterns on processor performance.

The comparison between the OnMult() and OnMultLine() functions showed that depending on how the memory access is done, the improvements in execution time and cache efficiency are drastic. Furthermore, the contrast between Haskell and C++ implementations showed that Haskell executed matrix multiplication significantly faster, likely due to its inherent lazy evaluation and memoization features.

Finally, the implementation of parallelized matrix multiplication in C++ using OpenMP highlighted the challenges of achieving significant performance improvements through parallelization. Both functions demonstrated limited gains over single-core implementations, with execution times and cache miss values showing little to no improvement