U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

# Performance evaluation of a single and multi-core implementation

1º Projeto

**Computação Paralela e Distribuída**
Turma 01 Grupo 12
**Professor** Carlos Miguel Ferraz Baquero-Moreno


Alberto Serra **up202103627**
Cristiano Rocha **up202108813**
António Ferreira **up202108834**

# 1. Problem description

The project examines the effect of memory hierarchy on processor performance during matrix product calculations, considering the architectural and performance distinctions between various programming languages and comparing sequential and parallel code. In Part 1, the efficiency of matrix multiplication is assessed on a single core using C/C++, Java, and Python, analyzing how each language manages matrices of varying dimensions. Part 2 explores the advantages of multi-core processors by parallelizing the matrix multiplication process with OpenMP. This phase highlights the performance gains achievable by distributing computations across several processing units.

# 2. Algorithms explanation

To assess single-core processor performance with different data sizes, this section explores three different matrix multiplication algorithms: **Basic Matrix Multiplication**, **Line Matrix Multiplication**, and **Block Matrix Multiplication**, which are implemented in C++, Python, and Java. Java's selection allows for easier direct language comparison because of its syntactic resemblance to C++ and slightly slower execution speeds. The execution speed differences are highlighted by Python, which was chosen due to its noticeably slower performance, particularly with bigger datasets. However, because of its performance limitations, Python's tests were limited to smaller and medium-sized matrices to prevent unfeasible execution times and were executed only three times, where the others were six times. Because each algorithm uses a different memory utilization approach, it is possible to see how architectural and linguistic variations affect computing performance.

## 2.1 Basic Matrix Multiplication

The **Basic Matrix Multiplication algorithm** takes a straightforward method to create the output matrix: it multiplies each row of the first matrix by each column of the second matrix. Despite being straightforward, this method requires a large amount of processing power due to its **O(n^3) complexity**, which means that the calculation time increases rapidly with matrix size. This occurs because the program multiplies using three nested loops, illustrating how big data sets impact single-core processor performance.

```
void OnMult(int m_ar, int m_br)
```

```
def on_mult(m_ar, m_br):
```

```
public static void onMult(int m_ar, int m_br)
```

## 2.2 Line Matrix Multiplication

The **Line Matrix Multiplication** method increases efficiency by multiplying each element in the first matrix by the appropriate row in the second matrix. This technique maximizes the usage of data kept in memory while retaining the same **O(n^3)** computational complexity as the standard method. It seeks to lessen the performance impact common to large dataset operations in single-core systems by accessing contiguous memory.

```
void OnMultLine(int m_ar, int m_br)
```

```
public static void onMultLine(int m_ar, int m_br)
```

```
def on_mult_line(m_ar, m_br):
```

### 2.3 Block Matrix Multiplication

The input matrices are divided into smaller blocks via the **Block Matrix Multiplication** technique, enabling independent calculations and later aggregation. Even with its **O(n^3)** computational cost, this method greatly improves the efficiency of memory access. The approach reduces the latency in obtaining data from higher memory levels by guaranteeing that more data is stored in the quicker, lower-level cache memories. This is achieved by dealing with smaller data blocks. This technique takes advantage of the memory hierarchy to increase performance, especially in situations involving a single core where execution timings depend on memory access speeds.

```
void OnMultBlock(int m_ar, int m_br, int bkSize)
```

```
def on_mult_block(m_ar, m_br, bk_size):
```

```
public static void onMultBlock(int m_ar, int m_br, int bkSize)
```

## 3. Performance Metrics

Two distinct computers - a **home computer (computer A)** and a computer used at the **FEUP labs (computer B)** - were used for the measurements. **Computer B, in FEUP,** were used to measure the times in relation to the cache misses, while all basic execution time averages were calculated with the values from **computer A**. To ensure control and independence of the collected data, **C++** and **Java** algorithms were run **six times** under the same isolated settings, while **Python** implementations were run only **three times**. The values of each graphic and statistic that are displayed are based on the average of those six runs. The home system, called machine A, was equipped with an i7-8750H single-running CPU clocked at 2.20 GHz with a maximum frequency of 4.1 GHz, 16 GB of installed RAM, and distinct L1, L2, and shared L3 caches for every core. It was running Ubuntu 22.04. In total, Computer A has 10 hardware counters and 12 cores. The FEUP lab computer (machine B), which features an i7 9700 single-clocked CPU with a clock speed of 3.00 GHz and a maximum of 4.7GHz, 16 GB of RAM loaded, and an L1, L2, and shared L3 cache for each core, was also running Ubuntu 22.04. There were 8 cores and 10 hardware counters on machine B. Both were using the same version of PAPI, 7.1.0.0.

To create relevant metrics, the **Performance API (PAPI)** was used, ensuring access to CPU metrics and the levels of CPU Cache memory utilized by the process, to assess the performance of the algorithms for the C/C++ versions. Apart from the algorithm's execution time, the number of mega floating-point operations per second (**MFLOP**) was calculated with success and cache misses for both L1 and L2 were considered (using computer B). The program compiled in the C/C++ version used the -O2 optimization flag, which takes into consideration the PAPI output values and enhances the compilation time and performance of the resulting code.To achieve consistency in the numbers utilized in the subsequent statistical analysis, the average of the values was calculated. This was done to acquire results across measurements, somewhat diverse surroundings, and isolated situations originating from the computer's state. The team modified the files to run every algorithm in every language to produce three distinct text files of each distinct programming language to make it easier to run all of the algorithms in the versions of C++, Python, and Java.

# 4. Results and analysis

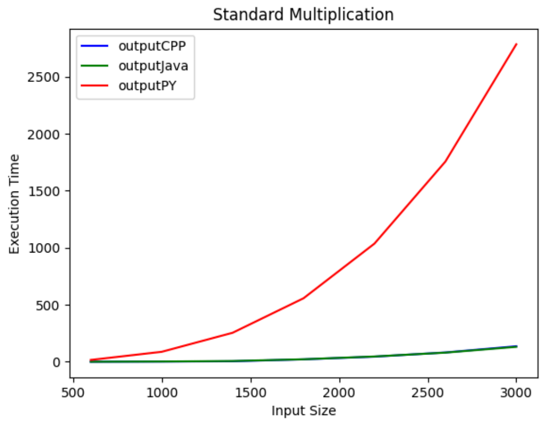
The average computations of six successive runs of each programming language file, producing three distinct output files, are shown in the following results. The **outputCPP** is connected to the C++ output results, the **outputJava** views the output from Java as well as the **outputPy** has to do with the Python outcomes. Most of our graphs show the execution time on the Y-axis and the matrix size on the X-axis. We used the formulas the professors presented in class to calculate the **MFLOPS, SpeedUp, and Efficiency** to compare the C++ sequential versus parallel programs.

$$MFLOPS = (2 * n^3)/executiontime$$

$$Efficiency = SpeedUp/Ncores$$

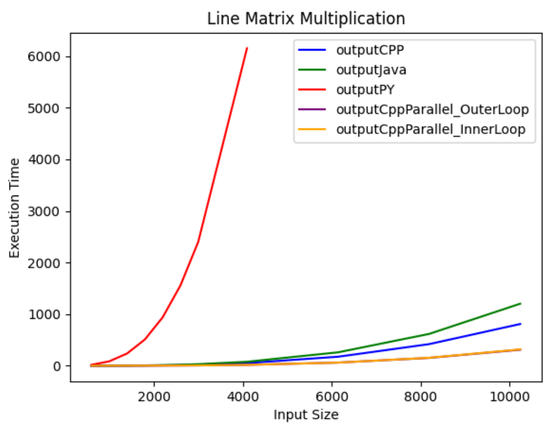
$$SpeedUp = SequentialTime/ParallelTime$$

## 4.1 Comparing the three languages' Standard Matrix Multiplication Algorithm execution times



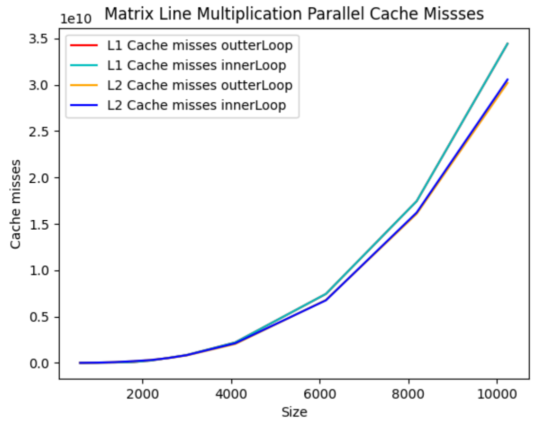

As performance-optimized compiled languages, Java and C++ have **similar execution times** in the matrix multiplication performance graph. Java's virtual machine overhead is the reason for its low latency. Python runs **substantially slower** than other programming languages because of its interpreted nature and dynamic type, which require extra work at runtime and cause exponential performance declines for bigger matrix sizes.

## 4.2 Sequential and Parallel Comparison of execution times of Line Matrix Multiplication algorithm



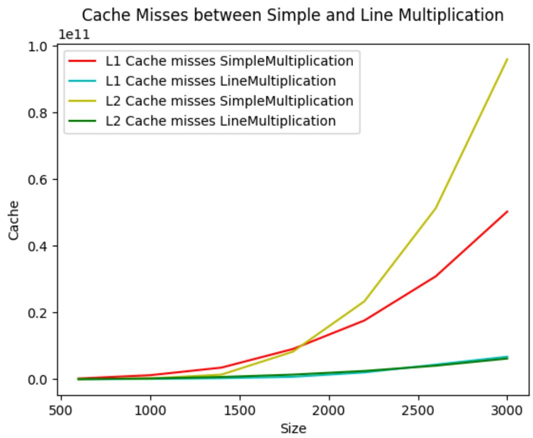

The **Line Matrix Multiplication** graph presents intriguing results about language efficiency and parallelization techniques. Java typically lags behind C++ in terms of sequential performance because of its Just-In-Time compilation overhead as opposed to **C++'s direct compilation** to machine code. Because Python is interpreted, it **delays** a lot. This causes overheads in type verification and dynamic lookup, which worsen execution speeds. C++ **executes more quickly when it is parallelized,** with inner loop parallelization outperforming outer loop parallelization. This performance results from the inner loop's improved utilization of core-level parallelism, enabling more concurrent computations.

This occurs as a result of the outer loop taking use of spatial proximity by maintaining the row-wise memory access pattern. On the other hand, the inner loop parallelization experiences **more cache misses,** particularly at bigger matrix sizes, due to numerous synchronization points and a disturbed caching pattern.

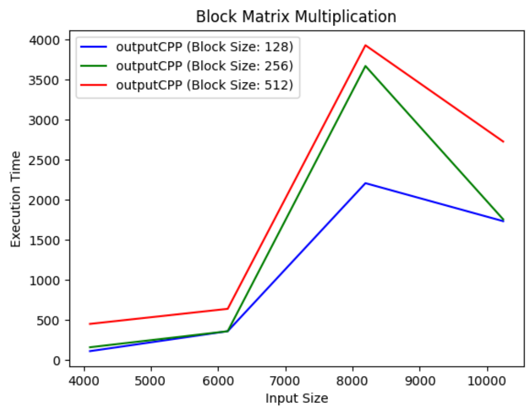



## 4.3 Sequential and Parallel Comparison of cache misses of Line Matrix Multiplication algorithm

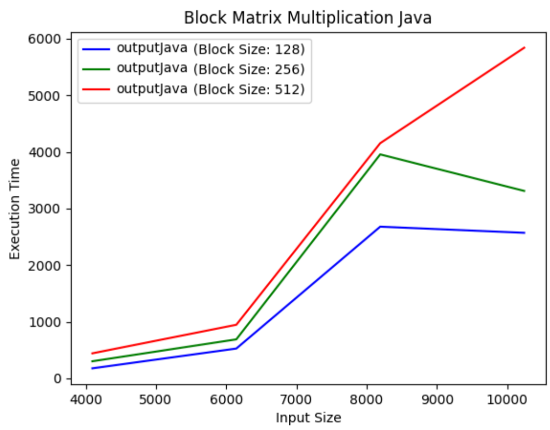
The graphs' cache miss patterns provide insight into how various algorithms and their **parallelization** techniques handle memory access. Because the **Line Multiplication technique** accesses memory line by line, as shown in the first graph, it is optimized for data proximity and, as size increases, results in smaller cache misses than **Simple Multiplication**. Better cache utilization is the result of wider job division among cores, as evidenced by the second graph, where the outer loop parallelization exhibits fewer L1 and L2 cache misses than the inner loop.




## 4.4 Block Matrix Multiplication Execution Time and Cache Misses Comparison for C++ and Java

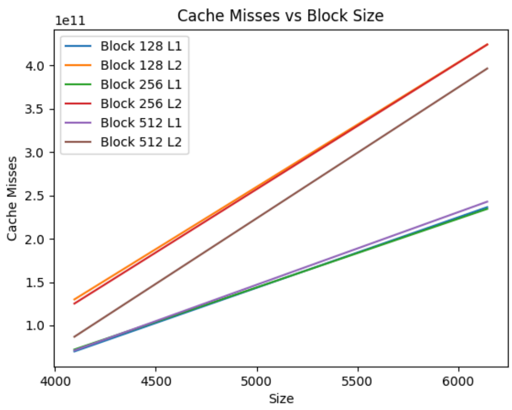



When the **Block Matrix Multiplication** algorithm is used in Java and C++, it exhibits different execution patterns depending on the block size. Block sizes of 128, 256, and 512 in C++ have a significant effect on execution time; smaller blocks typically lead to **faster computations**, especially when bigger matrix dimensions are involved. Because they fit within the cache line size more effectively, smaller blocks, especially the 128 size achieve **faster execution times**. This reduces the amount of cache misses and fetching operations from higher memory levels.



This pattern suggests that there is a sweet spot for block size that makes use of the memory architecture of the CPU, weighing the advantages of spatial proximity within the cache against the overhead of memory access.

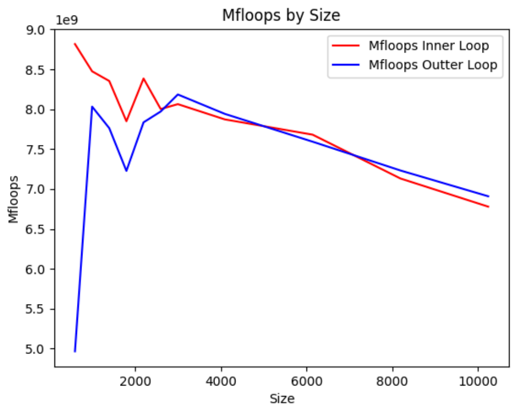
Java's runtime environment and garbage collection system are responsible for the modest increase in overall execution time when compared to C++, even if its execution durations for various block sizes follow the same general pattern. The relationship between **cache misses** and **block size** emphasizes the need to strike a balance between memory efficiency and computational overhead. While smaller blocks are **more effective in L1 cache**, larger blocks take advantage of spatial locality to **reduce L2 cache misses**. The knowledge that the ideal block size can greatly improve performance by matching the processor's memory architecture and reducing the expensive cache-to-memory fetch operations is further supported by the examination of cache misses.

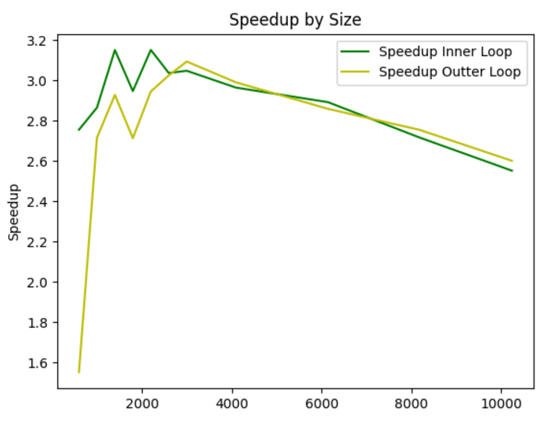


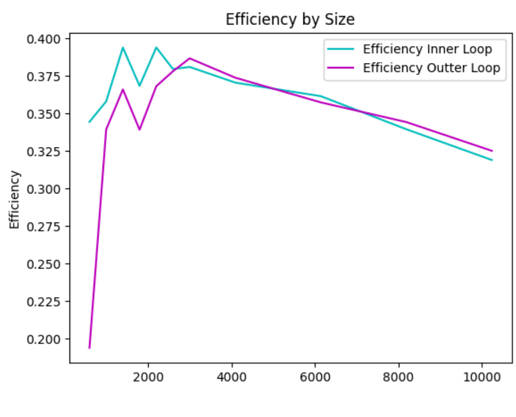

## 4.5 MFLOPS, Efficiency, and SpeedUp calculations concerning the execution of the parallel code

The graphs show the parallelized **Line Matrix Multiplication** performance metrics in C++ using OpenMP. According to the **MFLOPS** graph, both inner and outer loop parallelization experience fluctuations in computing efficiency as matrix size increases, with the outer loop exhibiting a more consistent drop in floating-point operations per second. The **Efficiency** graph illustrates how efficiency drops with increasing matrix sizes. It is computed as the **SpeedUp** divided by the number of cores.

This shows that **while parallel computing speeds up computations**, the benefit doesn't increase a lot with the number of cores because of synchronization and thread management complexity in our machines and environments. The **SpeedUp** graph, which illustrates a higher level of performance increase with outer loop parallelization, contrasts the time savings from sequential to parallel processing. This suggests that allocating workload to the outer loop makes better use of spatial proximity, allowing for greater cache hierarchy benefits.



# 5. Conclusions

This study's result emphasizes the crucial role **memory management** plays in maximizing **program performance**, which goes beyond **parallel computing**. Sequential programs can operate more efficiently when memory is managed well. This is especially true when lower-level memory caches are strategically used to lessen the need for slower, higher-level storage alternatives. This research highlighted the benefits of parallelization while shedding light on the differences between **single-core and multi-core** processing. Through contrasting the matrix multiplication implementations in **sequential and parallel** settings, we have gained a thorough grasp of the best use case for each technique. This information is priceless since it lays the groundwork for further developments in computing and parallel programming methods.

# 6. References and Webography


Oficial Papi website: Performance API (PAPI). The University of Tennessee, Knoxville. Retrieved from: https://icl.utk.edu/papi/

Simple PAPI Instrumentation. University of Maine. Retrieved from: https://web.eece.maine.edu/~vweaver/projects/papi/papi_simple.html

CPPreference chrono. Retrieved from: https://en.cppreference.com/w/cpp/chrono

Python 3.12.2 documentation. Retrieved from: https://docs.python.org/3/

Java documentation. Retrieved from: https://docs.oracle.com/javaee/7/index.html

Moodle course content. Retrieved from: https://moodle2324.up.pt/

# 7.Annexes

## A1. Default Matrix Multiplication
### A1.1 C/C++ version - execution time(s)
### Computer 1

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.265 |
| 1000 x 1000 | 1.707 |
| 1400 x 1400 | 4.279 |
| 1800 x 1800 | 21.211 |
| 2200 x 2200 | 44.927 |
| 2600 x 2600 | 81.425 |
| 3000 x 3000 | 136.136 |

## A1.1.2 C/C++ version - execution time(s)
## FEUP PC

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.195 |
| 1000 x 1000 | 1.272 |
| 1400 x 1400 | 3.867 |
| 1800 x 1800 | 18.753 |
| 2200 x 2200 | 39.178 |
| 2600 x 2600 | 69.687 |
| 3000 x 3000 | 116.565 |

## A1.2 - Java version - execution time (s)

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.263 |
| 1000 x 1000 | 2.276 |
| 1400 x 1400 | 6.501 |
| 1800 x 1800 | 21.783 |
| 2200 x 2200 | 46.215 |
| 2600 x 2600 | 80.161 |
| 3000 x 3000 | 131.144 |

## A1.3 - Python version - execution time (s)

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 16.066 |
| 1000 x 1000 | 87.343 |
| 1400 x 1400 | 254.375 |
| 1800 x 1800 | 557.063 |
| 2200 x 2200 | 1036.394 |
| 2600 x 2600 | 1755.855 |
| 3000 x 3000 | 2784.909 |

**A2. Line x Line Multiplication**
**A2.1 C/C++ version - execution time (s)**
**Computer 1**

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.135 |
| 1000 x 1000 | 0.676 |
| 1400 x 1400 | 2.07 |
| 1800 x 1800 | 4.379 |
| 2200 x 2200 | 8.004 |
| 2600 x 2600 | 13.339 |
| 3000 x 3000 | 20.41 |
| 4096 x 4096 | 51.749 |
| 6144 x 6144 | 174.647 |
| 8192 x 8192 | 418.68 |
| 10240 x 10240 | 808.457 |

### A2.1.2 C/C++ version - average execution time
### FEUP PC

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.095 |
| 1000 x 1000 | 0.445 |
| 1400 x 1400 | 1.447 |
| 1800 x 1800 | 3.216 |
| 2200 x 2200 | 6.018 |
| 2600 x 2600 | 10.091 |
| 3000 x 3000 | 15.69 |
| 4096 x 4096 | 41.259 |
| 6144 x 6144 | 138.682 |

## A2.2 - Java version - average execution time (s)

| Matrix Dimension | Average execution time |
| --- | --- |
| 600 x 600 | 0.235 |
| 1000 x 1000 | 1.028 |
| 1400 x 1400 | 2.51 |
| 1800 x 1800 | 6.499 |
| 2200 x 2200 | 11.916 |
| 2600 x 2600 | 19.672 |
| 3000 x 3000 | 30.186 |
| 4096 x 4096 | 77.002 |
| 6144 x 6144 | 260.065 |
| 8192 x 8192 | 616.605 |
| 10240 x 10240 | 1201.704 |

### A2.3 Python version - execution time (s)

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 17.926 |
| 1000 x 1000 | 85.627 |
| 1400 x 1400 | 237.154 |
| 1800 x 1800 | 508.674 |
| 2200 x 2200 | 937.681 |
| 2600 x 2600 | 1555.833 |
| 3000 x 3000 | 2399.715 |
| 4096 x 4096 | 6150.09 |

**A3 Block Matrix multiplication**
**A3.1 - C/C++ version - execution time (s)**
**Computer 1**

| Matrix Dimension | Block Size | Average execution time |
|---|---|---|
| 4096 x 4096 | 128 | 114.5 |
| 4096 x 4096 | 256 | 162.987 |
| 4096 x 4096 | 512 | 453.991 |
| 6144 x 6144 | 128 | 362.785 |
| 6144 x 6144 | 256 | 530.163 |
| 6144 x 6144 | 512 | 641.823 |
| 8192 x 8192 | 128 | 2208.257 |
| 8192 x 8192 | 256 | 3668.738 |
| 8192 x 8192 | 512 | 3927.408 |
| 10240 x 10240 | 128 | 1734.76 |
| 10240 x 10240 | 256 | 1957.232 |
| 10240 x 10240 | 512 | 2727.357 |

### A3.1.2 C/C++ version - execution time(s)
### FEUP PC

| Matrix Dimension | Block Size | Average execution time |
|---|---|---|
| 4096 x 4096 | 128 | 88.261 |
| 4096 x 4096 | 256 | 107.108 |
| 4096 x 4096 | 512 | 352.078 |

### A3.2 Java version - execution time (s)

| Matrix Dimension | Block Size | Average execution time |
|---|---|---|
| 4096 x 4096 | 128 | 177.971 |
| 4096 x 4096 | 256 | 302.873 |
| 4096 x 4096 | 512 | 442.59 |
| 6144 x 6144 | 128 | 526.989 |
| 6144 x 6144 | 256 | 689.556 |
| 6144 x 6144 | 512 | 946.949 |
| 8192 x 8192 | 128 | 2679.851 |
| 8192 x 8192 | 256 | 3956.755 |
| 8192 x 8192 | 512 | 4154.829 |
| 10240 x 10240 | 128 | 2571.603 |
| 10240 x 10240 | 256 | 3311.612 |
| 10240 x 10240 | 512 | 3756.356 |

### A3.3 Python version - execution time(s)

| Matrix Dimension | Block Size | Average execution time |
|---|---|---|
| 4096 x 4096 | 128 | 6531.254 |
| 4096 x 4096 | 256 | 6636.134 |

### A4 C/C++ Line x Line Parallel version - execution time(s)

### A4.1 - OuterLoop

| Matrix Dimension | Average execution time |
|---|---|
| 600 x 600 | 0.087 |
| 1000 x 1000 | 0.249 |
| 1400 x 1400 | 0.707 |
| 1800 x 1800 | 1.614 |
| 2200 x 2200 | 2.718 |
| 2600 x 2600 | 4.41 |
| 3000 x 3000 | 6.598 |
| 4096 x 4096 | 17.307 |
| 6144 x 6144 | 61.091 |
| 8192 x 8192 | 152.031 |
| 10240 x 10240 | 310.834 |

## A4.2 - InnerLoop

| Matrix Dimension | Average execution time |
| --- | --- |
| 600 x 600 | 0.049 |
| 1000 x 1000 | 0.236 |
| 1400 x 1400 | 0.657 |
| 1800 x 1800 | 1.486 |
| 2200 x 2200 | 2.54 |
| 2600 x 2600 | 4.393 |
| 3000 x 3000 | 6.697 |
| 4096 x 4096 | 17.459 |
| 6144 x 6144 | 60.393 |
| 8192 x 8192 | 154.171 |
| 10240 x 10240 | 316.797 |