Individual assigement BIG DATA

Grado en Ciencia e Ingenier´ıa de Datos a

Universidad de Las Palmas de Gran Canaria

Matrix multiplication

Jakub Król

Link to repository: GitHub repository

Contents

Part I

Abstract

This document presents a individual assignment for the Big Data course, centered on matrix multiplication. Matrix multiplication is a fundamental operation in various computational fields, especially in Big Data and high-performance computing. The main objective of the assignments is to benchmark different approaches to matrix multiplication, focusing on scalability, efficiency, and resource usage. I will conduct comparative studies across multiple matrix sizes and at least three alternative algorithms.

The assignments emphasize benchmarking as a critical method for evaluating the performance of algorithms under different computational conditions. In the context of Big Data, where applications often require real-time processing of massive datasets, benchmarking helps identify optimal algorithms by analyzing execution time, memory consumption, and scalability.

Key objectives of these tasks include:

- Evaluating the performance of various matrix multiplication approaches.
- Optimizing resource usage by identifying bottlenecks.
- Analyzing scalability as the matrix size increases.
- Making informed decisions based on performance metrics such as execution time and resource efficiency.

I will test my implementations using square matrices of varying sizes, from small (e.g., 10×10) to large (e.g., $10,000 \times 10,000$). Assignments include implementing basic matrix multiplication algorithms in Python, Java, and C, followed by performance benchmarking.

Part II

Introduction

Matrix multiplication is a fundamental operation in various scientific and engineering computations, and its performance can significantly affect the efficiency of applications. This report discusses the analysis and benchmarking of matrix multiplication code in C, Python, and Java, focusing on execution time, CPU usage, and memory consumption. Each language's implementation involves measuring these factors to provide insight into how resources are utilized during the matrix multiplication process.

Part III

Problem Statement

The primary objective is to evaluate the performance of matrix multiplication by benchmarking code implementations in C, Python, and Java. Key performance metrics include:

- **Execution Time**: How long it takes for the matrix multiplication operation to complete.
- **CPU Usage**: The proportion of CPU resources used during the operation.
- **Memory Usage**: The amount of memory consumed by the program during execution.

These benchmarks aim to provide a baseline for comparison and identify opportunities for optimizing matrix multiplication performance, particularly when working with larger matrix sizes.

Part IV

Methodology

In the C implementation, random values are assigned to matrices `a` and `b`, and the classical $O(n^3)$ matrix multiplication algorithm is applied to compute matrix `c`. Time measurement is done using the `gettimeofday()` function, while CPU usage is tracked via the `clock()` function and memory consumption is estimated using system-specific tools such as `getrusage()`.

For the Python implementation, performance testing is conducted using the `pytest` library for time measurement and the `psutil` library for tracking CPU and memory usage. Similarly, the Java code utilizes the `JMH` framework for benchmarking, with time and resource tracking implemented using `java.lang.management` tools.

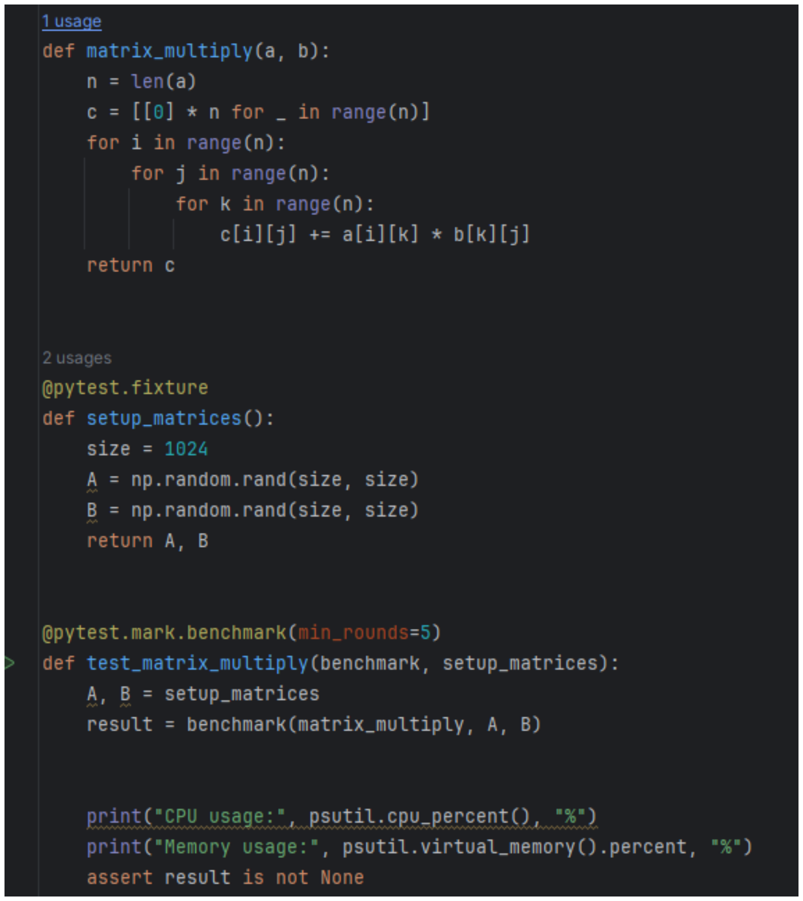
By benchmarking across these three languages, a comprehensive view of how each language handles resource-intensive operations like matrix multiplication is obtained. The results are compared to assess the performance and resource efficiency of each implementation.

Part V

Experiments

1 Python

Results of measuring time, CPU and Memory in Python with code given on Virtual Campus. By installing "pytest-benchmark" I have done benchmarks of matrices of size of 32 to 2048.



```
1 usage
def matrix_multiply(a, b):
    n = len(a)
    c = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                c[i][j] += a[i][k] * b[k][j]
    return c


2 usages
@pytest.fixture
def setup_matrices():
    size = 1024
    A = np.random.rand(size, size)
    B = np.random.rand(size, size)
    return A, B


@pytest.mark.benchmark(min_rounds=5)
def test_matrix_multiply(benchmark, setup_matrices):
    A, B = setup_matrices
    result = benchmark(matrix_multiply, A, B)


    print("CPU usage:", psutil.cpu_percent(), "%")
    print("Memory usage:", psutil.virtual_memory().percent, "%")
    assert result is not None
```

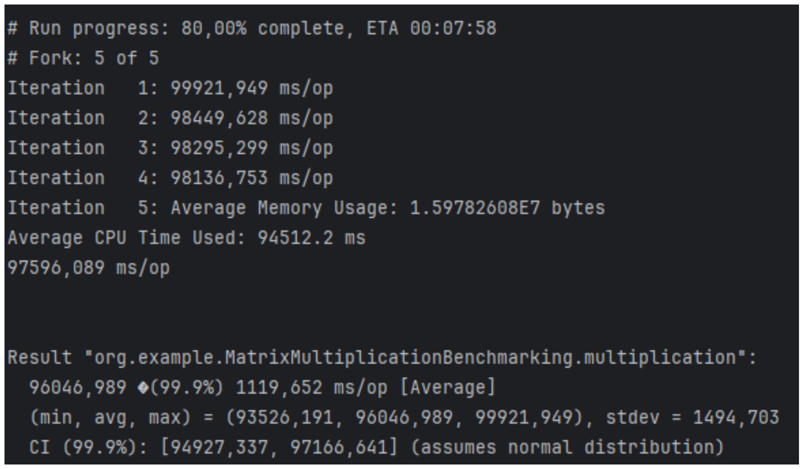
Figure 1: Python code

Python			
size	**Time [s]**	**CPU [%]**	**Memory [%]**
32	0,01634	8,7	48,4
64	0,13824	7,9	48,5
128	1,049	8,1	48,5
256	8,28	8,5	48,7
512	67,26	9,7	50,1
1024	555,21	8,9	41,4
2048	x	x	x

Table 1: Performance metrics for different sizes in Python

2 Java

Results of measuring time, CPU and Memory in Python with code given on Virtual Campus. By installing "jmh" I have done benchmarks of matrices of size of 32 to 2048 and here are the results.



```
# Run progress: 80,00% complete, ETA 00:07:58
# Fork: 5 of 5
Iteration   1: 99921,949 ms/op
Iteration   2: 98449,628 ms/op
Iteration   3: 98295,299 ms/op
Iteration   4: 98136,753 ms/op
Iteration   5: Average Memory Usage: 1.59782608E7 bytes
Average CPU Time Used: 94512.2 ms
97596,089 ms/op


Result "org.example.MatrixMultiplicationBenchmarking.multiplication":
  96046,989 ◆(99.9%) 1119,652 ms/op [Average]
  (min, avg, max) = (93526,191, 96046,989, 99921,949), stdev = 1494,703
  CI (99.9%): [94927,337, 97166,641] (assumes normal distribution)
```

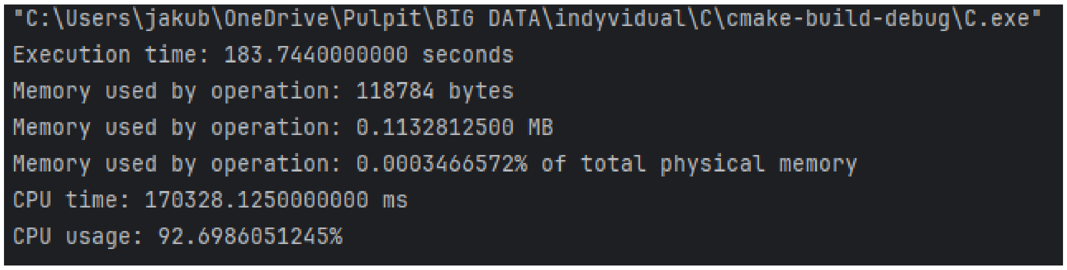
Figure 2: Example of output in terminal

Java			
size	**Time [s]**	**CPU [ms]**	**Memory [bytes]**
32	0,000000032	0,027	148,78
64	0,000000214	0,2181	57,57
128	0,02092	1,8	1549,02
256	0,021858	20,46	13639,65
512	0,206219	197,088	298949,088
1024	6,008413	5781,866	4537468,266
2048	96,046989	94512,2	15978260,8

Table 2: Performance metrics for different sizes in Java

3 C

Results of measuring time, CPU and Memory in Python with code given on Virtual Campus. By checking, at the begging time in last iteration and subtracting this from end time I have done benchmarks of matrices of size of 32 to 2048.



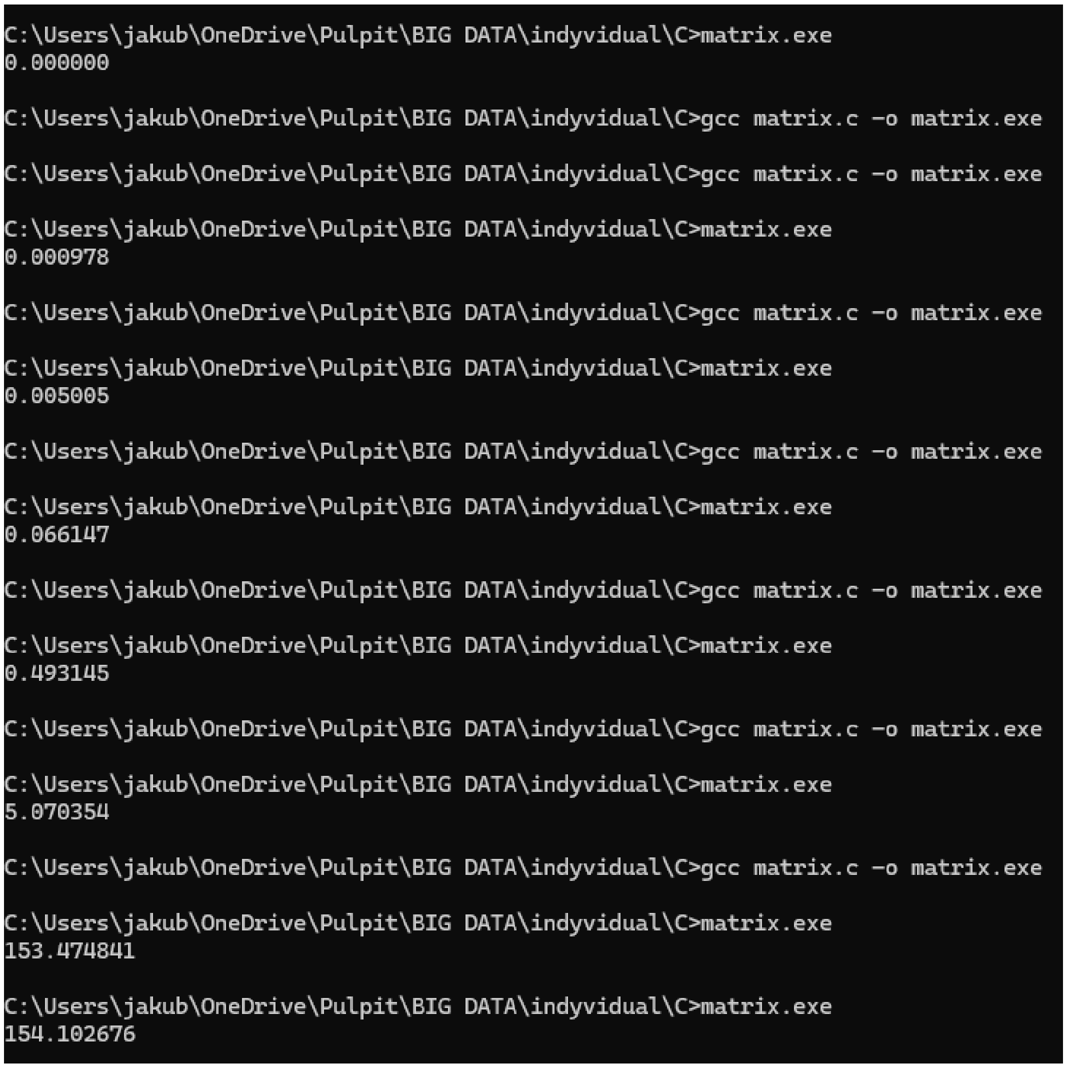
```
"C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C\cmake-build-debug\C.exe"
Execution time: 183.7440000000 seconds
Memory used by operation: 118784 bytes
Memory used by operation: 0.1132812500 MB
Memory used by operation: 0.0003466572% of total physical memory
CPU time: 170328.1250000000 ms
CPU usage: 92.6986051245%
```

Figure 3: Example of output from terminal

C				
size	**Time [s]**	**CPU [%]**	**Memory [%]**	**Memory [ms]**
32	0,000075	0	0,00038	0
64	0,000978	0	0,00038	0
128	0,0050	0	0,00038	0
256	0,0661	76,54	0,00038	46,86
512	0,4931	68,73	0,00038	468,75
1024	5,07	85,42	0,00038	10281,25
2048	153,474	91,4	0,00038	169015,625

Table 3: Performance metrics for different sizes in C

Result of time of C in comparison to Java's time result made me try to make another attempts of making measures. I thought it might be because of the environment in which I run the code so I tried to run it in Clion, VCS and in cmd terminal and results were that C was still slower than Java in larger matrices like 2048. It is something unexpected and should be given in further consideration.



```
C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
0.000000

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
0.000978

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
0.005005

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
0.066147

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
0.493145

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
5.070354

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>gcc matrix.c −o matrix.exe

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
153.474841

C:\Users\jakub\OneDrive\Pulpit\BIG DATA\indyvidual\C>matrix.exe
154.102676
```

Figure 4: Example of output from cmd terminal

4 Comparison

Time [s]			
SIZE	**Python**	**Java**	**C**
32	0,01634	0,000000032	0,000075
64	0,13824	0,000000214	0,000634
128	1,049	0,02092	0,00505
256	8,28	0,021858	0,06614
512	67,26	0,206219	0,4931
1024	555,21	6,008413	5,0703
2048	X	96,046989	153,47

Table 4: Comparison of execution time for different sizes in Python, Java, and C

Memory				
SIZE	**Python [%]**	**Java [bytes]**	**C [bytes]**	**C [%]**
32	48,4	0,027	131072	0,00038
64	48,5	0,2181	131072	0,00038
128	48,5	1,8	131072	0,00038
256	48,7	20,46	131072	0,00038
512	50,1	197,088	131072	0,00038
1024	41,4	5781,866	131072	0,00038
2048	X	94512,2	131072	0,00034

Table 5: Memory usage comparison for different sizes in Python, Java, and C

CPU usage comparison for different sizes in Python, Java, and C

Comparison of Python, Java, and C Performance

The comparison of execution times shows that C is generally faster than Python but slower than Java for small input sizes. As the size increases, C maintains a competitive time performance, while Python becomes significantly slower. Java, while initially very fast, sees a substantial increase in time at larger sizes, but remains faster than Python.

Memory usage varies across languages. Python reports memory usage in percentage terms, which remains relatively constant across different sizes. In contrast, Java (I could not get Java memory and CPU in % in no mater what cost) and C report memory in bytes, with Java's usage increasing significantly as the input size grows. C, on the other hand, keeps a stable memory footprint across input sizes due to fixed memory allocation.

CPU				
SIZE	**Python [%]**	**Java [ms]**	**C [ms]**	**C [%]**
32	8,7	0,027	0	0
64	7,9	0,2181	0	0
128	8,1	1,8	0	0
256	8,5	20,46	46,86	76,54
512	9,7	197,088	468,75	89,74
1024	8,9	5781,866	10281,25	88,27
2048	X	94512,2	169015,625	92,89

Table 6: CPU usage comparison for different sizes in Python, Java, and C

Regarding CPU usage, Python and C maintain relatively lower CPU percentages compared to Java, especially at smaller sizes. However, C's CPU usage spikes considerably for larger input sizes, aligning with increased execution times. Java's CPU usage in milliseconds follows a similar trend, becoming particularly high for larger sizes.

Overall, Python is more resource-intensive in terms of time, while Java demonstrates efficiency for smaller tasks but demands more memory and CPU at scale. C strikes a balance between time and memory, making it suitable for applications where speed is crucial, and memory can be managed efficiently.

4.0.1 Comparison of the Two Graphs (Time vs. CPU Usage) for Java and C

Time Graph (First Image) The time taken to execute programs for different matrix sizes is depicted here for Python, Java, and C.

- **Java** shows a slower increase in execution time compared to C and Python, with a moderate rise as the matrix size grows beyond 512.
- **C**, on the other hand, begins with a smaller time than Python but eventually surpasses Java in terms of execution time for larger matrices.

CPU Usage Graph (Second Image) This second graph focuses on CPU usage, and compares Java and C as matrix sizes increase.

- **C** consumes significantly more CPU resources as the matrix size grows, especially after 1000. The usage appears to grow at an exponential rate, reaching well over 160,000 for the largest matrix size (around 2000).

- **Java** also shows an increase in CPU usage but at a slower rate. The CPU usage rises much more gradually, staying well below C's peak, at around 80,000 for the largest matrix size.

Analysis of Both Graphs **Performance (Time):**

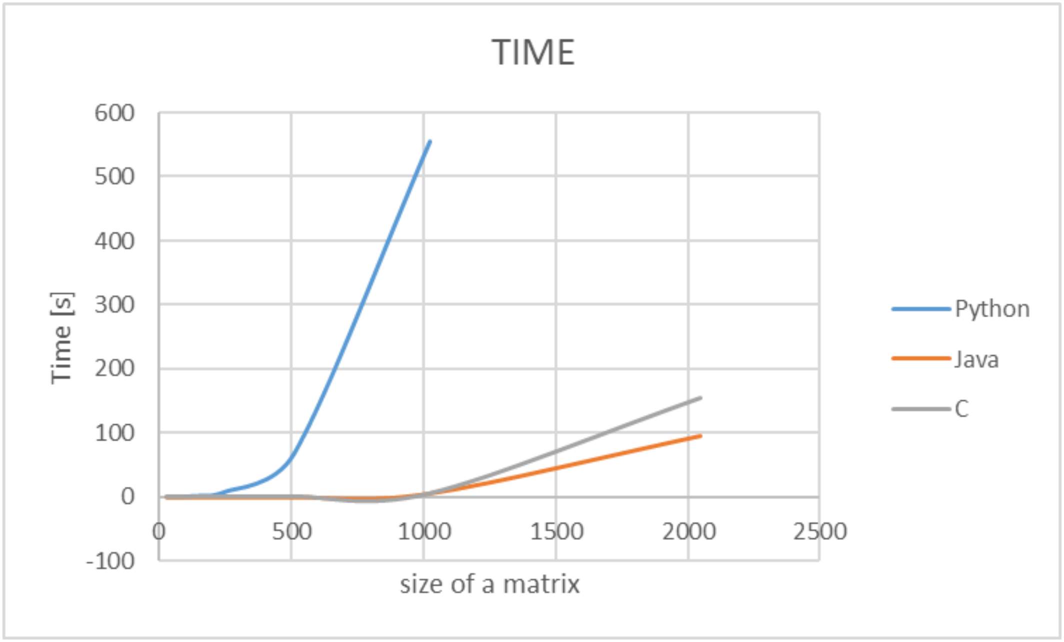


Figure 5: Time comparison chart

- **Java** shows better time efficiency for larger matrix sizes compared to **C**, as it consistently takes less time to complete the same tasks beyond a matrix size of 1000.
- **Python** is the slowest, with execution times ballooning exponentially for larger matrices.

CPU Usage:

- **C** is more CPU-intensive, as seen from the second graph. Its CPU usage increases drastically for larger matrices, which may indicate that it is performing more low-level operations and possibly optimizing the computation differently.
- **Java** is more conservative in CPU usage, even as the matrix size increases. This suggests that while Java takes a bit longer than C to execute certain operations (as seen in the time graph), it does so with less strain on the CPU.

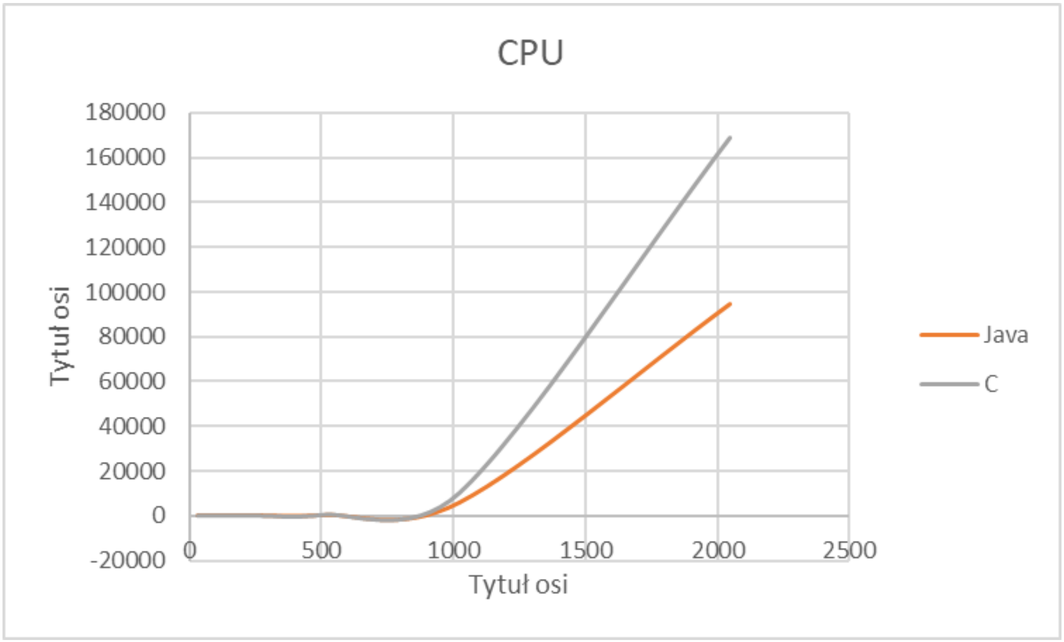


Figure 6: CPU comparison chart in ms

- **Java** appears to strike a better balance between CPU usage and execution time for larger matrix sizes. While it is not the fastest in execution time, it is more efficient in terms of CPU resources compared to **C**.
- **C**, despite being faster than Python, becomes quite resource-heavy in terms of CPU usage as the matrix size increases, which may lead to performance issues on systems with limited CPU resources. However, it is still faster than Java in terms of execution time.

Part VI

Conclusion

Despite of expecting to be the fastest in that task because of the fact that C is low level language Java appeared to be faster in bigger matrices (2048). By comparison time in CPU we can assume that Java is using Central Processing Unit's resources more efficient that C. Python presented the worst scores in bigger matrices and by the end of this experiment should not be consider as a tool to work with Big Data.

Part VII

Future work

In future I will be trying to make code more efficient by making and checking those changes:

- **Parallel algorithms:** Split matrix multiplication into smaller tasks that can be processed concurrently across threads or clusters, like using Strassen's algorithm.
- **Sparse matrices:** Use techniques for handling matrices with many zeros to save both memory and computation time.
- **Optimized libraries:** Leverage libraries like `BLAS` or `cuBLAS` for faster matrix operations.
- **MapReduce:** Distribute matrix calculations across clusters using frameworks like Hadoop.
- **Block representation:** Break down matrices into smaller blocks to improve cache efficiency.
- **Using GPUs:** Accelerate calculations with the parallel processing power of GPUs.
- **Batch size optimization:** Adjust batch size to make the best use of available memory.
- **Minimizing data transfer:** Reduce data movement between memory locations, like RAM and GPU.