Performance Comparison of Basic Matrix Multiplication in Python, Java, and C

Judith Portero Pérez
judporper1911@gmail.com

October 13, 2025

Abstract

This study evaluates the performance of a basic $O(n^3)$ matrix multiplication algorithm implemented in Python, Java, and C. Benchmarks were conducted for matrices of sizes 50, 100, 500, 1024, and 2000, measuring execution time, CPU usage, and peak memory. The results highlight significant differences in computational efficiency and memory consumption among the languages, providing insights for high-performance numerical computing.

Keywords: matrix multiplication, benchmarking, Python, Java, C, performance analysis

1 Introduction

Matrix multiplication is one of the most fundamental operations in computer science, mathematics, and engineering. It appears in countless applications such as linear algebra computations, image processing, data analysis, and especially in modern machine learning models. Even though it seems conceptually simple, its computational cost can become very high as the size of the matrices grows. The classical algorithm has a time complexity of $O(n^3)$, which means that doubling the matrix size increases the number of operations roughly by a factor of eight.

The goal of this project was to measure, analyze, and compare the performance of matrix multiplication implemented in three different programming languages:

- **C** (compiled and executed directly from the terminal).

- **Java** (executed from IntelliJ IDEA).
- **Python** (executed from IntelliJ IDEA).

The comparison focuses on execution time, CPU time, and peak memory usage for increasing matrix sizes. This is not just a theoretical comparison, it aims to observe the real impact of using a compiled language (C) versus an interpreted language (Python) and a managed language (Java) in a simple but computationally intensive task.

2 Problem Statement

The problem tackled in this project consists of computing the product $C = A \times B$ of two square matrices of size $N \times N$, using the classical triple nested loop algorithm:

```
for i in 0..N:
    for j in 0..N:
        for k in 0..N:
            C[i][j] += A[i][k] * B[k][j]
```

This algorithm, although simple, is computationally expensive. Its time complexity is $O(N^3)$, meaning that as N grows, the number of arithmetic operations increases cubically. For small matrices ($N = 50$), execution is almost instantaneous in any language. However, for large matrices ($N = 2000$), performance differences between languages become critical.

The objective of the benchmark is to:

- Measure the wall time (real elapsed time) of the matrix multiplication.
- Measure the CPU time (amount of CPU resources actually consumed).
- Measure the peak memory usage during execution.

The experiment aims to provide quantitative evidence of how language design, execution model, and runtime environment affect raw performance in a computationally intensive but conceptually simple task.

3 Methodology

3.1 Experimental Setup

All experiments were executed on the same machine to guarantee fair performance comparison. System background processes were minimized to reduce external interference in the measurements.

The execution setup for each language was as follows:

- **C:** compiled using `gcc` with the optimization flag `-O3`, executed directly from the Windows terminal.
- **Java:** executed from IntelliJ IDEA using the project SDK module `openjdk-22`, with the Oracle OpenJDK 23.0.1 runtime. This setup runs on the Java Virtual Machine (JVM), which includes Just-In-Time (JIT) compilation during execution.
- **Python:** executed from IntelliJ IDEA using a configured Python 3.12 interpreter (CPython). The interpreter was explicitly set in the IDE to ensure consistent execution across runs.

3.2 Matrix Sizes

Tests were performed with square matrices of sizes:

$$N \in \{50, 100, 500, 1024, 2000\}.$$

These values were chosen to:

- Evaluate behavior on small inputs (very fast execution).
- Observe scaling on medium inputs.
- Stress the implementation on large inputs ($N = 2000$).

3.3 Benchmark Design

- **Multiple runs:** Each configuration was executed 5 times to minimize the impact of noise or outliers, and the average was computed.
- **Isolating multiplication time:** Only the time taken by the actual multiplication loop was measured. The time spent generating matrices or printing results was excluded.

- **Metrics collected:**
  - *Wall time (s)*: measured using the system clock.
  - *CPU time (s)*: to capture effective processor usage.
  - *Peak memory (KB)*: maximum memory footprint during execution.
- **Data collection:** All results were stored in CSV files for later analysis and plotting using Python with `pandas` and `matplotlib`.

3.4 Plot Generation

Two types of plots were generated:

- Comparisons among the three languages (Python, Java, C).
- Focused comparisons only between Java and C to highlight their close performance, since Python results are much larger and visually dominate the plot.

This allowed a more detailed analysis of the shape of performance curves without losing detail due to Python's slower execution.

4 Experiments and Results

The experiments were carried out for different matrix sizes ($N = 50, 100, 500, 1024, 2000$) in order to observe how execution time, CPU usage, and memory consumption scale with problem size. Each experiment was repeated 5 times to reduce the effect of noise, and the average was used for analysis.

4.1 Execution Time

Figure 1 shows the average execution time for the three programming languages as the matrix size increases.

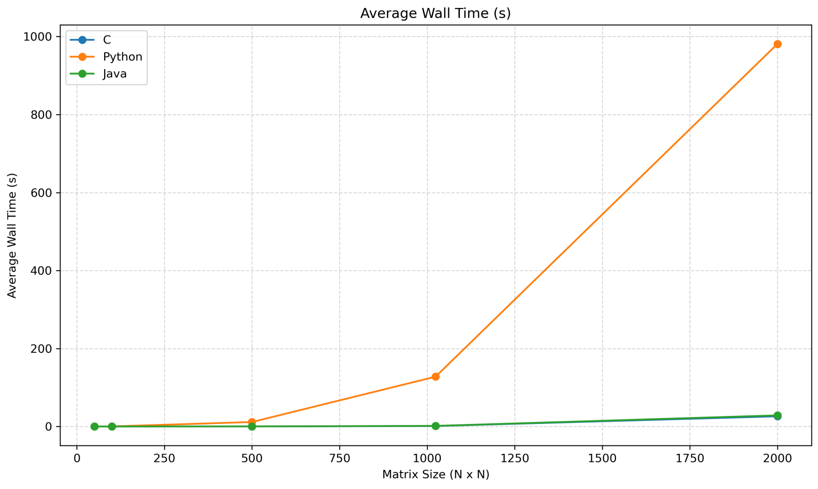

Figure 1: Execution time vs. matrix size for Python, Java and C.

For small matrix sizes (e.g., $N = 50$ or 100), the execution time for all languages is very low. However, as N grows, Python exhibits a much higher growth in time.

This behavior is consistent with the underlying execution models of each language:

- **C** is compiled to native machine code and runs directly on the CPU, which minimizes overhead.
- **Java** uses the Java Virtual Machine (JVM) and JIT compilation. Although this introduces some startup overhead, the JIT optimizer improves performance during execution.
- **Python** is an interpreted language (CPython), which incurs significant overhead in loops and arithmetic operations, leading to much slower execution as complexity grows.

To better visualize the differences between C and Java without Python dominating the scale, a second graph (Figure 2) compares only these two languages.

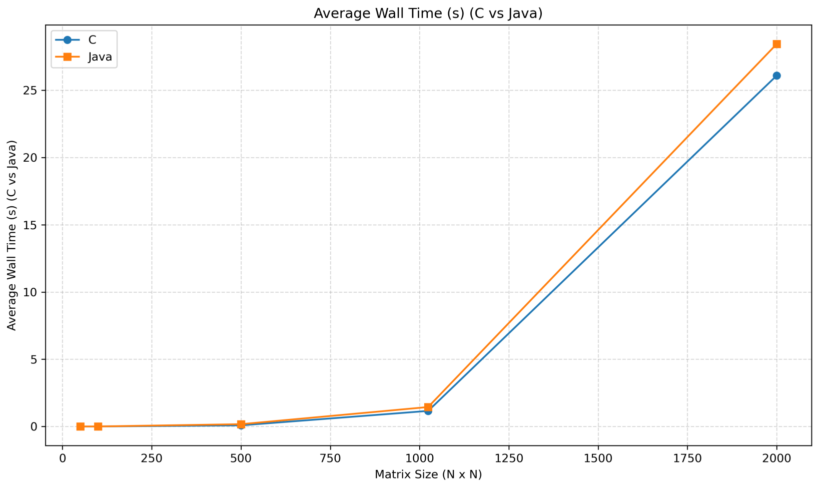

Figure 2: Execution time vs. matrix size for C and Java.

As shown, C consistently outperforms Java for all tested sizes. This gap increases with larger matrices because the JVM introduces additional runtime management overhead, such as garbage collection and JIT compilation time, whereas C executes deterministically once compiled.

It is worth noting that the difference observed in this experiment is not as large as commonly reported in benchmarks conducted on Linux systems. This can be attributed to several factors:

- **Windows overhead:** In Windows, native C programs tend to have slightly higher process and memory management overhead compared to Linux, which can reduce the performance advantage of compiled code.
- **JVM optimizations:** Modern JVM implementations (such as OpenJDK 23 used in this experiment) include highly optimized JIT compilers that can produce near-native performance for computationally intensive loops, especially after a warm-up period.

Therefore, although C is still faster than Java, the relative difference appears smaller in this environment than what would typically be expected on other systems.

4.2 Memory Usage

Figure 3 illustrates peak memory usage across all languages.

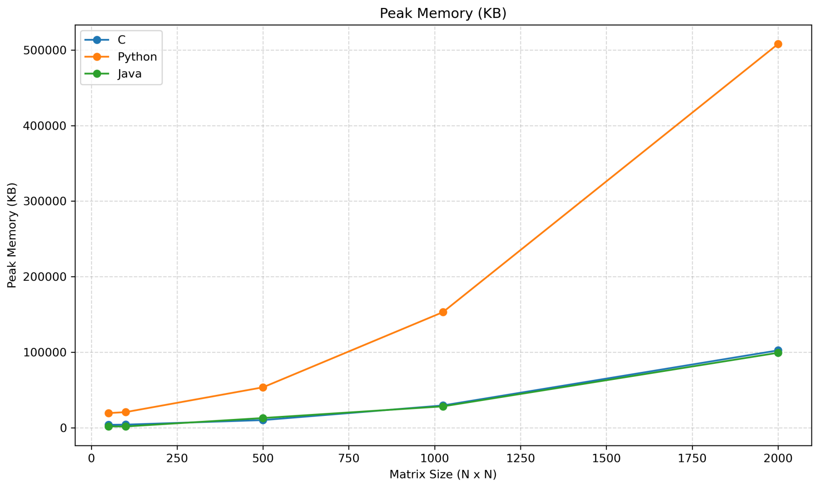

Figure 3: Peak memory usage vs. matrix size.

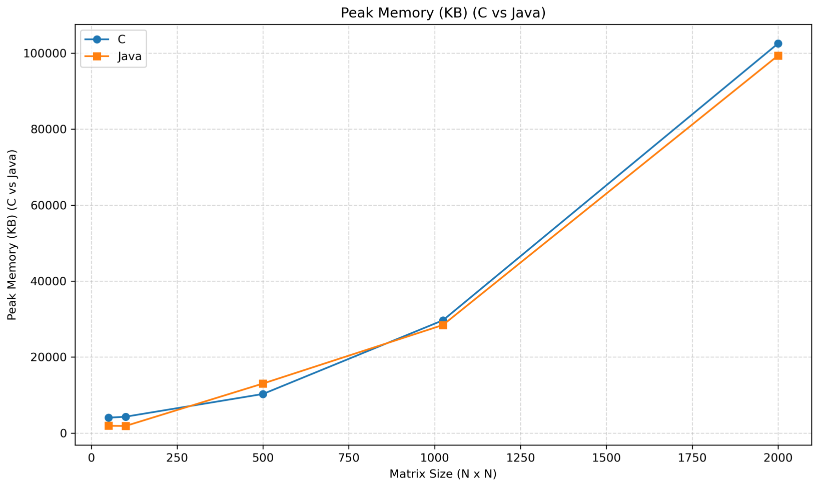

Figure 4: Peak memory usage vs. matrix size for C and Java.

Among the three languages, Python clearly exhibits the highest memory consumption across all matrix sizes. This behavior is expected due to several intrinsic characteristics of the language and its runtime environment:

- Each numerical element in Python is represented as a full-fledged object, which includes not only the value but also additional metadata (type information, reference counters, etc.).
- Python lists are dynamic and store references to objects rather than

raw values, introducing additional pointer layers and internal fragmentation.

- The Python interpreter maintains various runtime structures that contribute to the overall memory footprint, including garbage collection metadata and dynamic allocation buffers.

As a result, even for moderate matrix sizes, Python's memory usage grows significantly faster than C and Java. For $N = 2000$, Python exceeds 500 MB of peak memory usage, while both C and Java remain around 100 MB.

In contrast, C and Java show very similar memory profiles throughout the experiments. C benefits from leaner allocations and lower-level memory control, while Java incurs a small overhead from the JVM for metadata and runtime services. However, at larger sizes the difference between both becomes negligible, and in this particular setup, the peak usage of C was even slightly higher than Java. This may be due to differences in how memory is allocated and released: while Java reuses internal buffers efficiently thanks to the JVM, C performs raw allocations for each run without such optimizations.

These results illustrate how the underlying memory model of a language and its runtime system can have a major impact on performance and resource usage, even when running the same algorithm.

4.3 CPU Usage

Figure 5 presents the CPU usage measured during the runs.

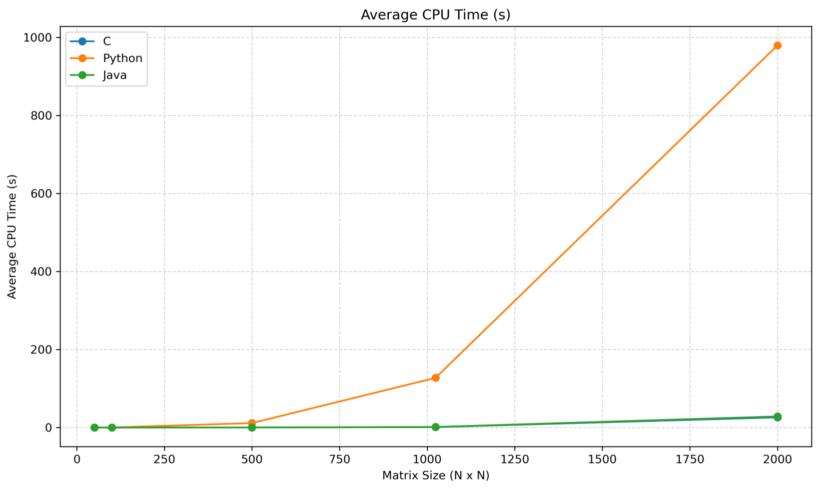

Figure 5: Average CPU usage vs. matrix size.

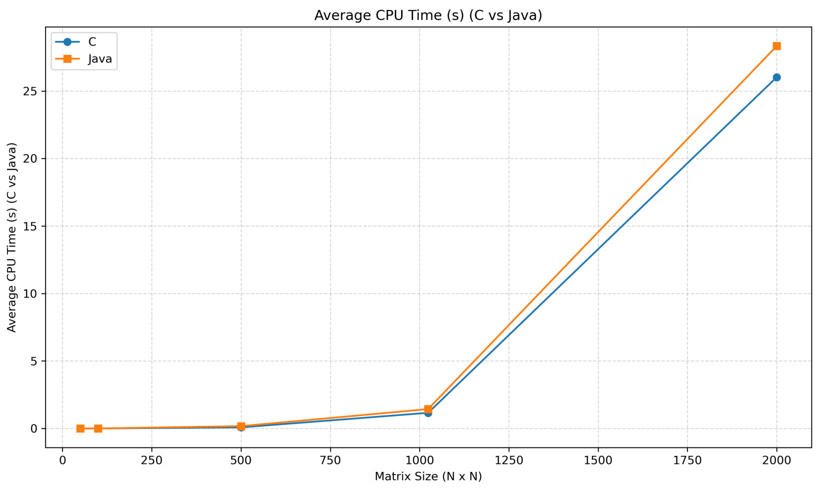

Figure 6: Average CPU usage vs. matrix size for C and Java.

Java and C exhibit CPU usage patterns that are closely aligned with their actual computation times, reflecting efficient utilization of CPU cycles. C shows the most stable behavior across all matrix sizes, benefiting from direct native execution without significant runtime overhead. Java maintains slightly higher variability, primarily due to JVM management activities such as garbage collection and JIT compilation.

In contrast, Python consistently shows significantly higher CPU usage than both Java and C. This elevated CPU consumption reflects the heavy computational cost of its interpreted execution model and inefficient handling of numerical loops at the Python level. Unlike C and Java, Python does not efficiently translate arithmetic operations into optimized machine instructions, resulting in prolonged CPU engagement to execute the same mathematical workload.

4.4 Summary of Results

Execution Time

Matrix Size	C (s)	Java (s)	Python (s)
50	0.000090	0.000983	0.010408
100	0.000616	0.000908	0.098243
500	0.083417	0.170395	11.416150
1024	1.156953	1.445575	127.629322
2000	26.093038	28.440847	981.094325

Table 1: Average execution time across different languages and matrix sizes.

The results show a clear performance hierarchy. C achieves the lowest execution times across all matrix sizes, followed closely by Java. Python exhibits significantly higher execution times, particularly for larger matrices.

CPU Usage

Matrix Size	C (s)	Java (s)	Python (s)
50	0.000000	0.000000	0.009375
100	0.000000	0.000000	0.100000
500	0.081250	0.171875	11.406250
1024	1.159375	1.434375	127.331250
2000	26.028125	28.340625	979.050000

Table 2: Average CPU usage across different languages and matrix sizes.

C and Java exhibit CPU usage values that closely match their execution times, indicating efficient CPU utilization with minimal idle time.

Python, in contrast, shows CPU usage that grows dramatically with matrix size, far exceeding the other two languages. This reflects the heavy computational burden caused by its interpreter and the lack of compiled numeric optimization.

Peak Memory Usage

Matrix Size	C (KB)	Java (KB)	Python (KB)
50	4048	1919	19544
100	4312	1873	20848
500	10288	13037	53660
1024	29680	28442	153112
2000	102536	99363	507888

Table 3: Peak memory usage during execution.

The peak memory analysis reveals that Python consumes significantly more memory than Java or C for all matrix sizes, particularly for larger matrices. This is due to Python's object-oriented memory model and additional overhead from dynamic type management. C and Java show more contained and predictable memory usage.

5 Conclusions

The results clearly show a significant performance gap between the three languages. C achieves the best execution times and the most stable performance, followed closely by Java, while Python is orders of magnitude slower for large matrix sizes. This is mainly due to C and Java being compiled languages that make efficient use of CPU cycles, in contrast to Python's interpreted execution model, which introduces substantial overhead.

Regarding memory usage, Python consumes considerably more memory than both C and Java, as its dynamic object model adds significant overhead per element. C and Java, on the other hand, have similar and much lower memory footprints, with Java showing slightly more variability due to JVM management.

Overall, C provides the highest performance and efficiency. Java represents a good balance between speed and ease of development. Pure Python is not suitable for large-scale matrix multiplication unless optimized libraries such as NumPy are used. These results highlight the strong influence of language design and runtime models on computational performance.

6 Future Work

Future work should focus on further improving performance and understanding resource behavior:

- **Optimize Resource Usage:** Identify bottlenecks in memory, CPU, and storage utilization, and apply targeted optimizations to reduce overhead. This may involve improving data structures, using memory-efficient representations, or delegating computations to optimized libraries.
- **Scalability Analysis:** Extend the benchmarking to much larger matrices and datasets to assess how execution time, CPU usage, and memory consumption scale. This is particularly important for applications dealing with large-scale or Big Data problems.
- **Informed Decision-Making:** Use the benchmarking results to guide the choice of programming language and algorithm for specific tasks. Future studies could systematically evaluate trade-offs between execution speed, resource consumption, and development convenience to support data-driven decisions.

7 Repository

All code, data, and scripts are available in the GitHub repository: `https://github.com/judporper/-Language-Benchmark-of-matrix-multiplication`