Comparative Performance Analysis of Matrix Multiplication in Python, Java, and C

Leonoor Antje Barton

GitHub Repository

October 23, 2025

Abstract

This paper investigates the computational performance of a basic matrix multiplication algorithm implemented in Python, Java, and C. The purpose is to analyze how language design, memory management, and compilation models affect execution efficiency. Using benchmark tests across increasing matrix sizes (128×128, 256×256, and 512×512), we measured the time, CPU time, and memory usage. The results show that C consistently outperforms both Java and Python in all categories. These findings suggest that while high-level languages offer ease of development, low-level compiled languages remain optimal for computationally intensive tasks.

1 Introduction

This study compares how three widely used languages Python, Java, and C, perform when executing the same $O(n^3)$ matrix multiplication algorithm. The objective is to quantify differences in execution time, memory, and CPU usage, and to identify how language-level abstractions impact computational performance.

2 Problem Statement

This work benchmarks a consistent $O(n^3)$ implementation across three distinct language paradigms: interpreted (Python), managed (Java), and compiled (C).

We hypothesize that:

1. C will deliver the best performance due to direct memory management and compiled execution.
2. Java will perform moderately well because of its JIT compiler and efficient garbage collection.
3. Python will be the slowest due to interpretation overhead and dynamic typing.

3 Methodology

3.1 Algorithm and Implementation

The same triple-nested loop algorithm was implemented in all languages:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

The production code (matrix generation and multiplication) was separated from the benchmarking scripts to ensure modularity and clean testing environments.

3.2 Benchmark Setup

Each implementation was executed five times for each matrix size (128×128, 256×256, 512×512). For Python, the `time` and `psutil` modules were used for timing and memory measurement; Java used `System.nanoTime()`; and C used `gettimeofday()` and `clock()`.

3.3 Hardware and Software Environment

All tests were conducted on the same Windows system under identical conditions using Visual Studio Code for all builds. For C, GCC (MinGW) was used; for Java, OpenJDK 21; and for Python, version 3.11.

3.4 Metrics

The performance was evaluated based on:

- **Execution time:** total elapsed wall time.
- **CPU time:** processing time consumed by the CPU.
- **Memory usage:** measured in megabytes (MB) during execution.

4 Results and Analysis

Tables and figures below summarize the average results for each language across all matrix sizes. Each value represents the average of five runs.

Table 1: Average performance metrics for each language.

Language	Matrix Size	Time (s)	Memory (MB)
Python	128.000	0.425	28.700
Python	256.000	3.620	34.900
Python	512.000	40.780	55.500
Java	128.000	0.004	0.380
Java	256.000	0.020	1.500
Java	512.000	0.184	6.000
C	128.000	0.023	0.380
C	256.000	0.131	1.500
C	512.000	1.245	6.000

4.1 Execution Time

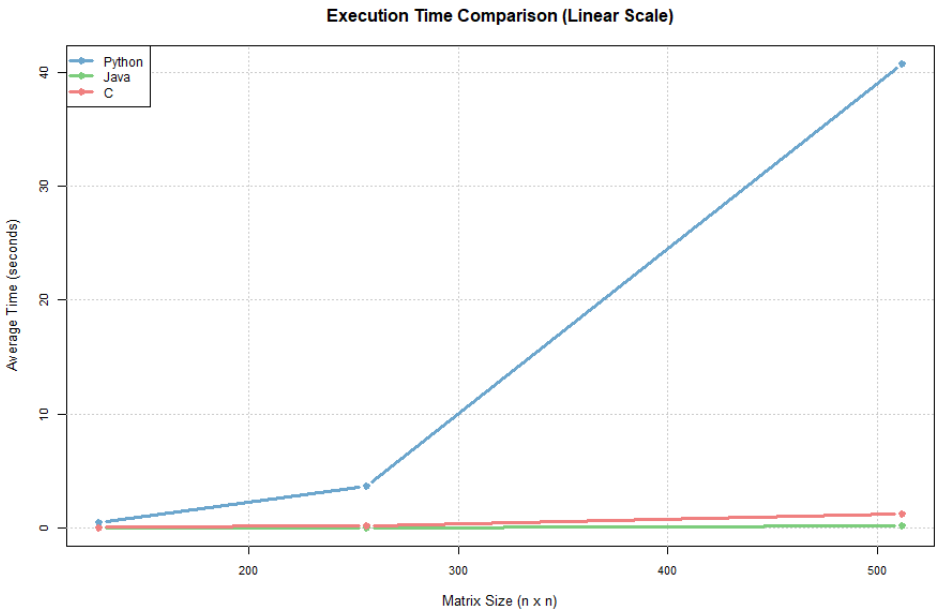

Figure 1: Execution time comparison (linear scale).

The difference in runtimes between Python, Java and C can generally be traced back to differences in execution models. C benefits from minimal overhead runtime and enables the compiler to apply optimizations prior to execution. In contrast, Java uses a JIT compiler that translates byte code into native machine code at runtime, allowing significant improvements in performance while maintaining platform independence. Meanwhile, Python is executed by an interpreter, employing dynamic type checking and object management, resulting in the performance being slower compared to with statically compiled or JIT-compiled languages.

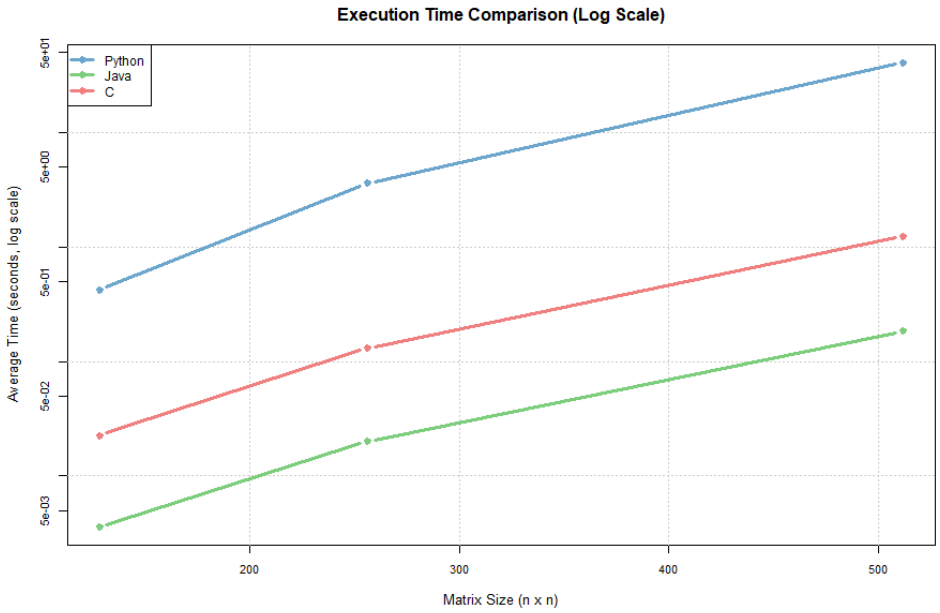

Figure 2: Execution time comparison (log scale).

The following figure shows a log-scale comparison of execution times, it reinforces that while all implementations exhibit cubic time complexity, compiled and JIT-compiled languages (C and Java) scale far more efficiently than interpreted languages like Python, particularly as matrix dimensions grow.

4.2 Memory Usage

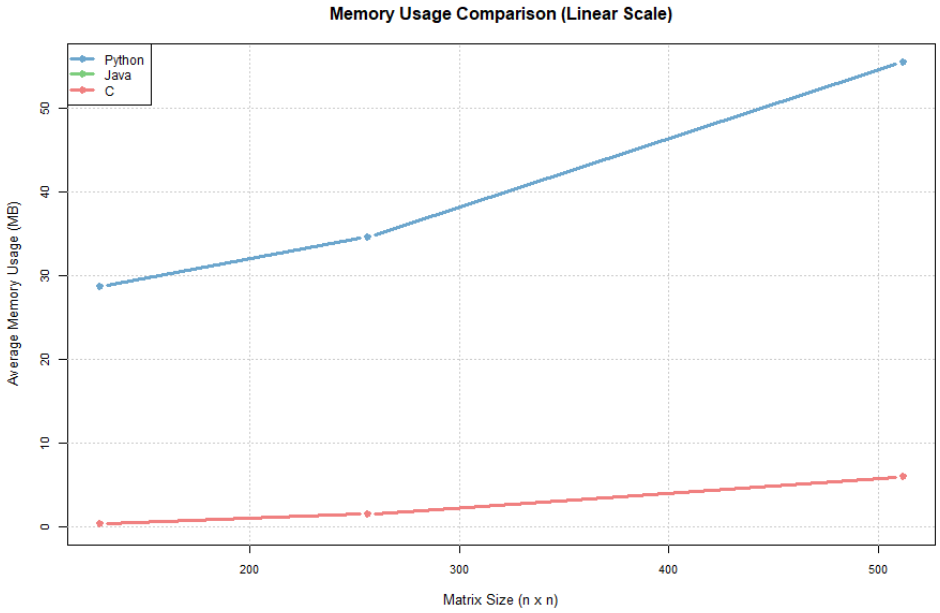

Figure 3: Memory usage comparison (linear scale).

This figure illustrates the memory usage of Python, Java, and C across varying matrix sizes. We can clearly observe how memory consumption increases linearly

with the matrix dimensions, which is in line with the theoretical expectation since each extra matrix element needs proportional storage. C and Java have nearly identical memory behavior, approximately 0.38 MB for 128×128 matrices, 1.5 MB for 256×256 and 6 MB for 512×512. This is due to both languages using fixed size data structures and contiguous memory allocation, in C using malloc, providing direct control and in Java with heap based allocation and automatic garbage collection. In contras Python has a substantially higher memory consumption, this stems from the fact that each numeric value is represented as an object with additional metadata, reference counting and dynamic type information. Improving flexibility but needing more space.

4.3 CPU Time

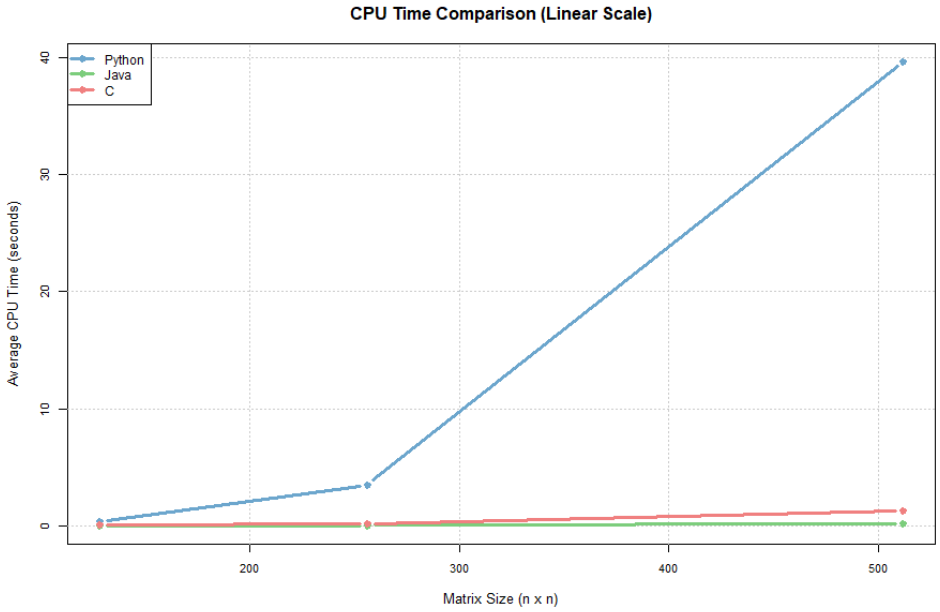

Figure 4: CPU time comparison (linear scale).

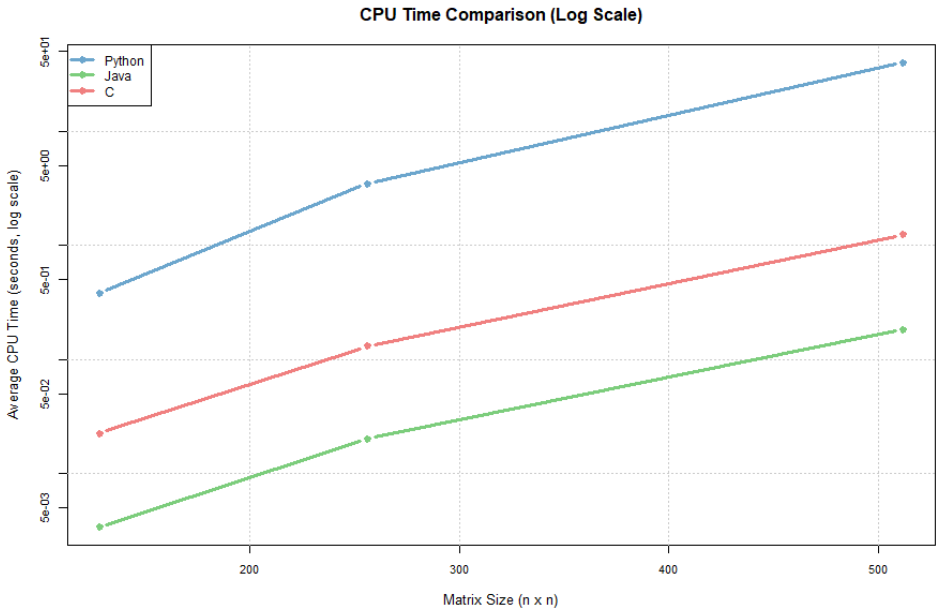

Figure 5: CPU time comparison (log scale).

These figures display the CPU time comparison between Python, Java, and C during matrix multiplication. Overall they are similar to that of the total execution time, with C and Java showing efficient scaling and Python lagging significantly behind. C demonstrates the most effective CPU utilization, keeping the processor consistently busy with minimal overhead due to its direct, compiled execution. Java shows similar scaling, with its JIT compiler dynamically optimizing loops and reusing compiled bytecode to improve throughput. In contrast, Python struggles to maintain CPU efficiency since most of its processing power is spent managing objects, performing type checks, and interpreting instructions. These results reinforce findings that compiled and JIT-compiled languages make far more efficient use of CPU cycles than interpreted ones

5 Discussion

The experimental findings align with expectations:

- **C** achieved the best performance due to its compiled nature and direct memory access.
- **Java** exhibited strong results thanks to Just-In-Time (JIT) compilation, although slightly slower than C.
- **Python** was substantially slower but easier to implement, making it suitable for prototyping rather than production-grade computation.

The difference between CPU and wall time in Python reflects its Global Interpreter Lock (GIL) and overhead in managing dynamic data types. Java's balance between abstraction and performance demonstrates the advantages of runtime optimization in managed environments.

6 Conclusion

This study demonstrates that while high-level languages simplify development, they suffer from performance costs in computationally intensive tasks. C remains the most efficient for numerical operations, followed by Java, with Python trailing far behind.

Repository Structure

- **/code** – Contains `matrix` and `benchmark` scripts for each language.
- **/data** – Includes all result files and the consolidated `summary.csv`.
- **/paper** – This LaTeX document and compiled PDF.

References

Nanz, S., & Furia, C. A. (2014). *A Comparative Study of Programming Languages in Rosetta Code*. Available at: https://arxiv.org/abs/1409.0252

Wikipedia (2024). *Just-in-time Compilation*. Available at: https://en.wikipedia.org/wiki/Just-in-time_compilation

Netguru (2023). *Static vs Dynamic Typing: Pros, Cons, and Key Differences*. Available at: https://www.netguru.com/blog/static-vs-dynamic-typing