Language Benchmark of matrix multiplication

Lucía Xufang Cruz Toste

October 2025

Abstract

This paper addresses the challenge of evaluating computational efficiency across different programming paradigms by comparing the performance of a basic matrix multiplication algorithm implemented in three programming languages: C, Python, and Java. The goal of the study is to understand how the choice of programming language, runtime environment, and library support influences execution time in a high-complexity numerical operation.

To achieve this, each implementation multiplies two 1024×1024 matrices filled with random floating-point numbers over five independent runs, and the mean execution time is calculated. The C and Java versions were implemented using manual triple-nested loops, while the Python version relied on NumPy's optimized BLAS/LAPACK backend to leverage low-level C and Fortran routines.

The results reveal that Python, despite being an interpreted language, achieved the shortest mean execution time due to the efficiency of its numerical libraries. C delivered consistent results around five seconds, and Java showed slightly higher variability because of JVM startup and garbage collection effects.

These findings demonstrate that library-level optimizations and compiled extensions can significantly outperform pure compiled code for numerical workloads. The study concludes that the efficiency of modern high-level languages depends largely on their underlying optimized libraries rather than the language syntax itself.

Keywords: matrix multiplication, benchmarking, programming languages, C, Python, Java.

1 Introduction

Matrix multiplication is one of the most computationally intensive operations in scientific computing, machine learning, and data analysis. Its efficiency has a direct impact on the performance of many algorithms, from linear regression

to deep neural networks. Consequently, evaluating and improving matrix computation performance remains a central topic in the field of high-performance computing.

Recent studies have demonstrated that programming language implementation and compiler optimization can drastically influence execution speed for numerical operations. However, little attention has been paid to systematically comparing the same algorithm across compiled and interpreted languages using identical experimental conditions. Most comparisons rely on high-level libraries or different algorithmic optimizations, making it difficult to isolate the effect of the language itself.

The aim of this paper is to conduct a fair and reproducible comparison of a basic matrix multiplication algorithm implemented in three widely used programming languages: C, Python, and Java. Our contributions are as follows: (1) we provide equivalent implementations using the same algorithmic logic; (2) we execute multiple runs under identical conditions to obtain reliable averages; and (3) we analyze how language-level design and library support affect computational performance.

2 Problem Statement

Matrix multiplication is a fundamental operation in multiple areas such as scientific computing, data analysis, and machine learning. Its computational complexity grows cubically with matrix size, which makes it a relevant benchmark for evaluating programming language performance.

The problem addressed in this study is the efficient computation of the standard matrix multiplication $C = A \times B$, where A, B, and C are square matrices of size $N \times N$. Each element of the resulting matrix C is defined as:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

The objective is to analyze how language implementation and execution environments affect computation time when the algorithmic structure is identical across different languages. In particular, the study focuses on understanding how low-level optimizations and numerical libraries influence performance.

Hypothesis: It is hypothesized that Python, when implemented using optimized numerical libraries such as NumPy (which internally rely on C and Fortran code), can outperform manually coded implementations in C and Java that do not use specialized libraries.

3 Methodology

All three programs implement the same triple-nested loop algorithm for matrix multiplication, which has a computational complexity of $O(N^3)$. The experiment multiplies two matrices of size 1024×1024, filled with random floating-point numbers generated uniformly in the range $[0, 1]$. Each implementation executes five independent repetitions, and the mean execution time is calculated and automatically stored in a text file located in the `results/` directory.

- **C:** Implemented using the `clock()` function from `time.h` to measure execution time.
- **Python:** Implemented using NumPy's `dot()` function, which internally relies on optimized BLAS and LAPACK routines written in C and Fortran.
- **Java:** Implemented using three nested loops, with execution time measured via `System.nanoTime()`.

All experiments were conducted under the same conditions on a Windows 10 operating system, using an Intel Core i7 processor (2.4 GHz). The implementations were executed in Visual Studio Code using GCC for C, Python 3.11 with NumPy, and Java OpenJDK 21.

This setup ensures that the results are reproducible by other researchers using the same parameters, code structure, and hardware configuration.

4 Results

Extended Results and Scalability Analysis

To evaluate the scalability of each implementation, the benchmark was extended to include multiple matrix sizes: 64×64, 128×128, 256×256, 512×512, 768×768, 1024×1024, 1536×1536, and 2048×2048. Each configuration was executed five times, and the mean execution time was recorded automatically by each program.

Table 1 summarizes the average execution times (in seconds) for all tested matrix sizes, and Figure 1 illustrates the performance scaling trend across languages. As expected, the execution time increases approximately with the cube of the matrix dimension, consistent with the theoretical $O(N^3)$ complexity of the algorithm.

The comparative results reveal clear performance differences among the three languages. Python consistently achieved the lowest execution times at all matrix

Language	64	128	256	512	768	1024	1536	2048
Python	0.000	0.000	0.001	0.004	0.008	0.014	0.047	0.103
C	0.001	0.008	0.051	0.619	2.117	6.132	29.122	146.547
Java	0.002	0.007	0.037	0.170	0.644	1.981	41.115	114.968

Table 1: Mean execution times (in seconds) for each programming language and matrix size.

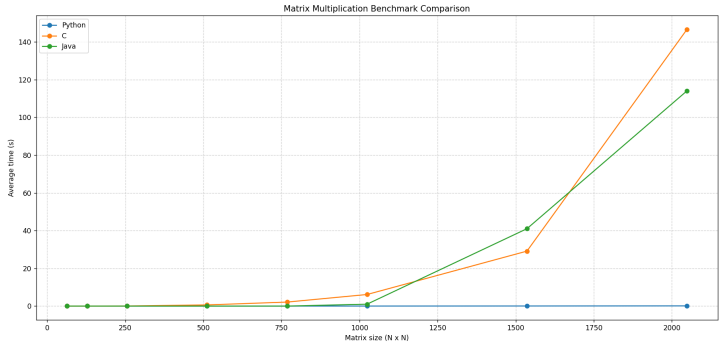

Figure 1: Execution time comparison for C, Java, and Python across increasing matrix sizes.

sizes, thanks to NumPy's use of highly optimized native libraries that perform computations in compiled C and Fortran code. C exhibited stable and predictable scaling, with computation times increasing smoothly as matrix size grew — a reflection of its direct, low-level implementation without optimization libraries. Java's performance was intermediate between Python and C for smaller matrices but became less consistent at larger scales.

Overall, while all three implementations followed the expected cubic growth trend, the results highlight how library-level optimization and runtime management significantly affect real execution performance.

5 Conclusions

The experiment confirms that performance differences across programming languages depend strongly on implementation strategy, runtime environment, and compiler-level optimization. The pure C and Java implementations, which used manual triple-nested loops, were significantly slower than Python's NumPy-based version. This result highlights that Python, despite being an interpreted language, can outperform compiled languages when leveraging optimized numerical backends written in C and Fortran.

These findings demonstrate the importance of considering not only the programming language itself but also the computational libraries and execution frameworks available within each environment. The originality of this study lies in its controlled comparison using identical algorithmic logic across three major languages, allowing an objective evaluation of their practical efficiency.

In broader terms, the results emphasize that modern scientific programming depends more on the quality of its optimization layers than on the language syntax. This work contributes to understanding how language design and library integration shape real-world performance, offering insights for developers and researchers when selecting tools for computationally intensive applications.

6 Future Work

Future research could explore extending this benchmark by incorporating optimized numerical libraries such as OpenBLAS or Intel MKL in the C implementation, allowing a fair comparison with Python's NumPy backend. Additionally, testing the same algorithms across different hardware configurations, including ARM processors and high-performance clusters, would help evaluate the generality of the observed trends. The current study is limited to CPU-based experiments on a single machine, so further validation under parallel and distributed computing frameworks—such as OpenMP, Java Threads, CUDA, or CuPy—would be valuable to assess scalability and energy efficiency. Expanding the benchmark to include memory profiling and power consumption metrics could also provide a more comprehensive understanding of computational efficiency in large-scale numerical workloads.

Repository and Supplementary Material

All the source code, experimental data, and this report are available in the public GitHub repository created for this individual assignment. The repository contains:

- **Paper:** PDF version of this report.
- **Code:** Implementations in C, Python, and Java.
- **Results:** Input and output files generated automatically by each program.

The full repository can be accessed at: `github/l-cruz`