Performance Analysis of Matrix Multiplication: Programming Language Benchmark

Daniel Medina González
github.com/DaniielMG/Language-Benchmark-of-matrix-multiplication

October 22, 2025

Abstract

This report presents a comparative performance analysis of the matrix multiplication operation (standard $O(N^3)$ algorithm) implemented in three different programming languages: C, Java, and Python. The results obtained from executing a set of tests for different matrix sizes ($N \times N$) reveal the impact of language architecture and execution environment on computationally intensive tasks. It is concluded that compiled languages such as C offer superior performance, while interpreted languages like Python incur a significant time penalty.

1 Introduction

Matrix multiplication is a fundamental problem in computing with applications in graphics, machine learning, and systems of equations. The performance of this operation is crucial for systems that rely on large amounts of computation. The goal of this study is to compare the execution efficiency of the standard matrix multiplication algorithm ($O(N^3)$) in C, Java, and Python, providing an objective metric for language selection in high-performance projects.

2 Experimental Methodology

2.1 Test Environment

The tests were executed in the author's local environment, using native implementations of the matrix multiplication algorithm. Source code details are available in the GitHub repository: `https://github.com/DaniielMG/Language-Benchmark-of-matrix-multiplication/tree/main`.

2.2 Languages and Compilers

- **C**: Compiled with `gcc` using the optimization flag `-O3`.
- **Java**: Executed via `java` (likely through an IDE such as IntelliJ IDEA).
- **Python**: Executed using the standard `python` interpreter (assuming no optimized libraries such as NumPy were used).

2.3 Metrics

Multiple executions (a minimum of 3 to 5) were performed for matrices of size $N \times N$, where $N \in \{256, 512, 1024, 2048\}$. The reported time for analysis is the **average execution time** in seconds (s).

3 Results and Discussion

The average execution times obtained for different matrix sizes and languages are summarized in Table 1.

Table 1: Average Execution Times (in seconds) for Matrix Multiplication

Matrix Size ($N \times N$)	**C (s)**	**Java (s)**	**Python (s)**
256×256	0.0234	0.0320	3.2477
512×512	0.2328	0.2980	24.8149
1024×1024	2.4195	7.5650	234.4954
2048×2048	27.1064	92.5630	2200.0000*

*Projected/estimated value. Due to the limited resources of the test device, the wait time was longer.

3.1 Comparative Analysis

- **C (Baseline Language):** C consistently proved to be the fastest language, especially for large matrices. The compiler optimization (`-O3`) and low hardware abstraction allow for extremely efficient machine code execution. The 27.1064 s time for 2048×2048 establishes the performance reference.
- **Java (Virtual Machine):** Java was noticeably slower than C but considerably faster than Python. The 92.5630 s time for 2048×2048 indicates that bytecode compiled and optimized by the JVM's JIT (Just-In-Time) compiler delivers robust performance, although it does not reach C's speed.
- **Python (Interpreted Language):** Python showed the worst scalability. The 234.4954 s time for 1024×1024 was already over 9 times slower than Java and nearly 100 times slower than C. The projected value of **2200.0000 s** for 2048×2048 illustrates that, without optimized libraries (such as NumPy, which is written in C), Python is unsuitable for pure heavy computation tasks due to the interpreter's overhead in each loop iteration.

4 Conclusion and Performance Analysis

The performance analysis conducted unequivocally confirms the central premise that the **programming language architecture** is the most decisive factor in the temporal efficiency of computationally intensive algorithms with complexity $O(N^3)$. The results clearly illustrate the performance hierarchy of the languages in this type of task: C > Java > Python.

4.1 C's Leadership: The Power of Native Compilation

C proved to be the fastest language across all matrix sizes. Its superiority stems from two main factors:

1. **Direct Machine Code Generation:** C is compiled directly into hardware-specific machine code, eliminating the abstraction layers present in the JVM or the Python interpreter.
2. **Compiler Optimization (`-O3`):** Using the `-O3` flag allowed the GCC compiler to apply aggressive hardware-level optimizations (such as efficient register usage and instruction scheduling), resulting in a highly efficient binary. The low abstraction of C makes these optimizations extremely effective.

4.2 Java's Balance: JIT and the JVM

Java consistently ranked second. Although slower than C (for example, approximately 3.4 times slower at 2048×2048), its performance was remarkably robust, outperforming Python by wide margins. This performance is achieved thanks to the **Java Virtual Machine (JVM)** and its **Just-In-Time (JIT)** compiler. The JIT dynamically translates bytecode into optimized machine code during execution, applying many of the same optimizations a static compiler would, but with the added advantage of real-time runtime information. Java's performance penalty is mainly due to the initial *overhead* of the JVM itself and the abstraction layer introduced by the *bytecode*.

4.3 Python's Interpreter Penalty

Python exhibited the poorest scalability, becoming extremely slow for matrices of 1024×1024 and larger. The execution time increases disproportionately. The fundamental reason lies in the **CPython interpreter overhead**:

1. **Dynamic Typing:** In each iteration of the multiplication loops, the interpreter must dynamically check variable types, perform operation dispatch, and handle memory management. This process consumes thousands of CPU cycles that C code completely avoids.
2. **C API Calls:** The interpreter spends significant time repeatedly calling the underlying C functions for basic operations (such as addition and multiplication), rather than executing streamlined machine code.

Therefore, Python is **not suitable** for heavy computational tasks that rely on slow nested loops and cannot be offloaded to optimized native libraries such as NumPy or SciPy (written in C/Fortran).

5 Final Summary

The benchmark confirms that:

- For **maximum performance** and minimal execution time in matrix computation, a compiled, hardware-level language such as **C** is the preferred choice.
- **Java** provides an excellent **balance** between portability (*write once, run anywhere*) and strong computational performance.
- **Python** is only practical for this type of task in modern contexts when used as an abstraction layer calling third-party libraries optimized in native code (C/Fortran), or for trivially small problem sizes.