Performance Analysis of Matrix Multiplication across Programming Languages

Gabriel Munteanu Cabrera

October 23, 2025

Abstract

This paper presents a comparative performance analysis of the standard matrix multiplication algorithm implemented in C, Java, and Python. The experiments were conducted across various matrix sizes to evaluate the impact of different programming paradigms and execution environments—compiled, Just-In-Time (JIT) compiled, and interpreted—on computational efficiency. The results demonstrate that compiled languages like C offer the best performance for smaller workloads, while Java's JIT compiler shows remarkable optimization capabilities for larger matrices. Python, as an interpreted language, consistently exhibits the highest execution time, highlighting the performance trade-offs for its high-level abstractions.

1 Introduction

Matrix multiplication is a fundamental operation in linear algebra and a core component of countless algorithms in science, engineering, and data analysis. Its computational complexity, typically $O(n^3)$, makes its performance a critical factor in application efficiency. The choice of programming language can significantly influence this performance due to differences in their execution models, memory management, and optimization capabilities.

This study aims to quantify these performance differences by implementing and benchmarking a standard matrix multiplication algorithm in three distinct languages: C, representing low-level compiled languages; Java, representing JIT-compiled languages running on a virtual machine; and Python, representing high-level interpreted languages.

2 Background and Related Work

This section provides the theoretical context for the algorithm being tested and the execution models of the chosen programming languages.

2.1 The $O(n^3)$ Matrix Multiplication Algorithm

The standard algorithm for multiplying two $n \times n$ matrices, A and B, to produce a result matrix C, is defined by the formula:

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

This calculation requires a triply nested loop structure (over indices i, j, and k). For each of the $n \times n$ elements in the resulting matrix C, we perform n multiplications and $n-1$ additions. This results in a total of n^3 multiplications and $n^3 - n^2$ additions, leading to an overall computational complexity of $O(n^3)$.While highly intuitive, this "naive" algorithm is known to be inefficient for large matrices due to its poor data locality, which does not take full advantage of modern CPU caches.

2.2 Alternative Algorithms and Implementations

It is important to note that more advanced algorithms exist, such as the Strassen algorithm (circa 1969), which reduces the complexity to approximately $O(n^{2.81})$ by using a recursive divide-and-conquer method. However, Strassen's algorithm typically only outperforms the naive method on very large matrices and introduces a higher constant factor overhead.Furthermore, production-level scientific computing relies on highly optimized libraries (e.g., BLAS, cuBLAS, Eigen, Intel MKL). These libraries achieve significant speedups not by changing the $O(n^3)$ complexity, but by using techniques like cache-blocking, vectorization (SIMD instructions), and parallelization.

3 Methodology

To ensure a fair and reproducible comparison, a consistent methodology was applied across all three languages. This section details the implementation, verification, and experimental setup.

3.1 Implementation in C

The C implementation was developed to serve as a performance baseline. The core logic (ijk variant) was isolated in a `matmul_ijk` function operating on one-dimensional arrays. The benchmarking driver handled command-line arguments (`--size`, `--repeats`), dynamic memory allocation with `malloc`, and time measurement using the high-precision `gettimeofday` function. The code was compiled with GCC using the `-O2` optimization flag.

3.2 Implementation in Java

The Java implementation followed an object-oriented approach. The multiplication algorithm was encapsulated in a static method, `MatrixOps.matmul`. The main driver class, `MatrixCLI`, managed experiment parameterization and used `System.nanoTime()` for high-resolution timing. The code was executed on a standard JDK, relying on the Just-In-Time (JIT) compiler for runtime optimizations.

3.3 Implementation in Python

The Python version was structured with the `matmul` function in its own module. The main script utilized standard libraries: `argparse` for command-line arguments, `time.perf_counter()` for accurate timing, and the `csv` module for data collection. As an interpreted language, this implementation serves as a baseline for dynamically-typed languages.

3.4 Code Verification

Before benchmarking, the correctness of each implementation was verified through a suite of unit tests. These tests included validation against known results for a 2x2 matrix and checks for mathematical properties, such as multiplication by the identity and zero matrices. This ensures that the performance measurements are valid and reliable.

4 Results

The experiments were executed for matrix sizes of 64x64, 128x128, 256x256, and 512x512, with each test being repeated 3 times to obtain an average. The collected data is summarized in Table 1 and visualized in Figure 1.

Table 1: Average execution time (in seconds) for matrix multiplication.

Matrix Size	C	Java	Python
64x64	0.000528	0.002857	0.051600
128x128	0.002952	0.026322	0.367444
256x256	0.033309	0.038695	3.439129
512x512	0.365053	0.330681	29.999540

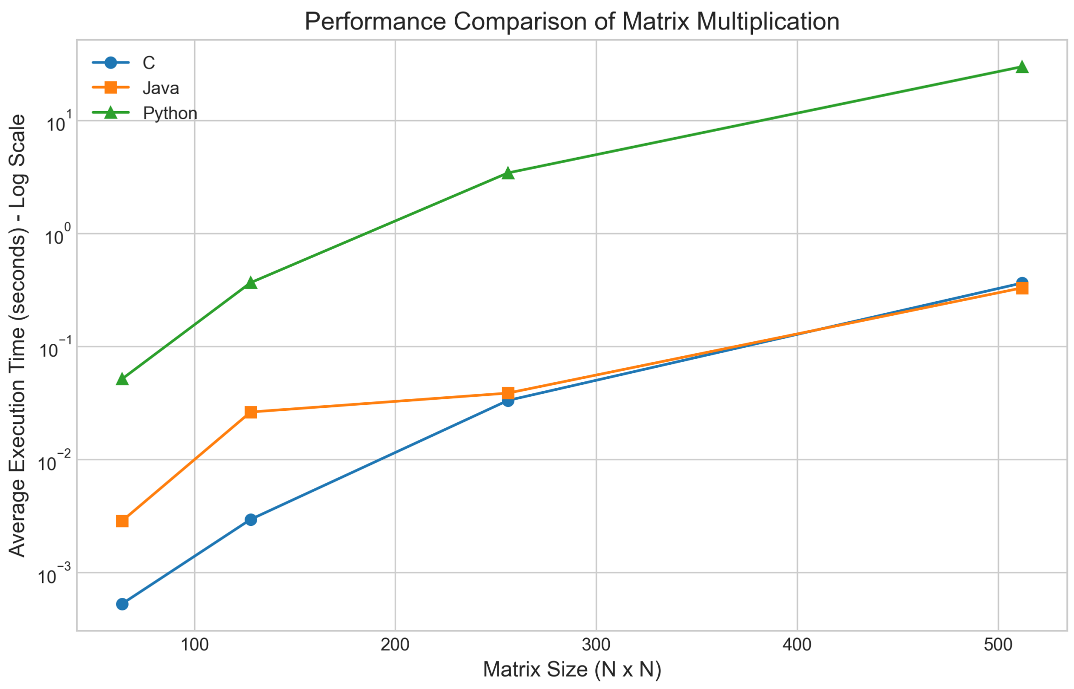

Figure 1: Execution time vs. Matrix Size for C, Java, and Python. Note the logarithmic scale on the y-axis.

5 Discussion

The results align with theoretical expectations but also reveal interesting nuances. Python's performance is several orders of magnitude slower than C and Java, which is attributable to its interpreted nature.

The most notable result is Java's performance surpassing C's for the 512x512 matrix. This is a classic demonstration of the power of the JVM's Just-In-Time (JIT) compiler. During execution, the JIT can perform profile-guided optimizations on "hot" code paths, sometimes achieving a level of optimization that surpasses the static, ahead-of-time compilation of C.

Finally, the execution times grow consistently with the $O(n^3)$ complexity of the algorithm, which is particularly visible in the steep curve of the performance plot.

6 Conclusion

This study successfully benchmarked a naive $O(n^3)$ matrix multiplication algorithm across C, Java, and Python. The findings confirm the significant impact of a language's execution model on raw computational performance.

As hypothesized, the Python implementation, being interpreted and dynamically-typed, was several orders of magnitude slower than its compiled counterparts. This confirms its unsuitability for raw, number-crunching tasks unless it acts as a wrapper for underlying C or Fortran libraries (like NumPy).

The comparison between C and Java yielded the most nuanced results. While C, as an Ahead-of-Time (AOT) compiled language, provided the best and most predictable performance on small-to-medium-sized matrices, a critical finding emerged with the 512x512 matrix. Here, the Java implementation marginally outperformed C. This phenomenon is a powerful demonstration of the Java Virtual Machine's Just-In-Time (JIT) compiler. The JIT was able to identify the multiplication loop as a "hotspot" and apply aggressive, profile-guided optimizations at runtime that ultimately surpassed C's static `-O2` optimization.

Ultimately, the findings show there is no single "best" language, but rather a spectrum of trade-offs. C offers raw, predictable speed. Java offers a powerful combination of portability ("write once, run anywhere") and "good enough" performance that can dynamically adapt and optimize long-running tasks. Python offers unparalleled development speed and simplicity, representing a clear trade-off between programmer productivity and computational efficiency.

A Repository Link

The complete source code, test suites, and raw data for this project are publicly available on GitHub: github.com/Gabrii8/matrix-assignment