Comparative Performance Analysis of Matrix Multiplication Algorithms Across Python, Java, and C

Jaime Ercilla Martin
GCID, ULPGC

Abstract

Matrix multiplication represents a fundamental operation in scientific computing and machine learning. This paper addresses the challenge of understanding how programming language choice impacts computational performance for intensive numerical operations. A classical $O(n^3)$ matrix multiplication algorithm was implemented and benchmarked across Python, Java, and C using matrices ranging from 64×64 to 512×512. Results demonstrate that C achieves optimal performance with execution times 1,545 times faster than Python and 4.5 times faster than Java, while consuming approximately one-third of the memory. These findings suggest that language selection must align with specific application requirements, balancing performance against development complexity and maintainability.

Keywords: matrix multiplication, performance benchmarking, computational complexity, programming language comparison, algorithmic efficiency, memory optimization

1 Introduction

Matrix multiplication constitutes one of the most fundamental operations in computational mathematics, serving as a building block for numerous applications spanning scientific computing, machine learning, computer graphics, and numerical simulations. The efficiency of matrix operations directly impacts the performance of critical applications including neural network training and large-scale data processing.

While modern hardware provides increasing computational power, the choice of programming language and implementation strategy can result in performance differences spanning several orders of magnitude. Previous studies have examined individual language performance characteristics, yet comprehensive comparative analyses considering both execution time and memory consumption across multiple languages remain limited. Little attention has been paid to quantifying these differences systematically for fundamental operations like matrix multiplication across representative languages from different paradigms: interpreted (Python), JIT-compiled (Java), and natively compiled (C).

The aim of this paper is to provide a rigorous comparative analysis of matrix multiplication performance across three widely-used programming languages. Our contributions include: systematic performance benchmarks with matrices from 64×64 to

512×512, quantitative analysis of execution time and memory consumption, identification of language-specific performance factors, and practical recommendations for language selection.

2 Problem Statement

The problem can be formally defined as follows: Given two square matrices A and B of dimension $n \times n$, compute their product $C = A \times B$.

The classical algorithm exhibits $O(n^3)$ time complexity and $O(n^2)$ space complexity. I hypothesize that: compiled languages will significantly outperform interpreted languages, memory management overhead will vary substantially across implementations, and execution time will scale predictably according to theoretical complexity.

3 Methodology

The classical matrix multiplication algorithm was implemented in Python 3.x (interpreted), Java 8+ (JIT-compiled), and C with GCC compiler (natively compiled). All implementations used identical algorithmic logic with three nested loops to ensure fair comparison. The C implementation was developed with assistance from Claude AI to ensure optimal code structure.

Experiments were carried out on consistent hardware with random floating-point matrices. Test dimensions were 64×64, 128×128, 256×256, and 512×512. Measurements captured execution time (wall-clock time for complete multiplication) and memory consumption (peak usage during computation). C code was compiled with GCC `-O2` optimization. Java measurements excluded JVM startup overhead. Python performance reflected pure Python without NumPy to establish baseline interpreter performance.

4 Experimental Results

Table 1 summarizes performance across all implementations. Results reveal substantial disparities, with differences becoming more pronounced as matrix dimensions increase.

Table 1: Performance comparison across Python, Java, and C

Size	**Python Time (s)**	**Java Time (s)**	**C Time (s)**	**Python Mem (MB)**	**Java Mem (MB)**	**C Mem (MB)**
64	0.4938	0.0037	0.0002	0.13	0.24	0.00
128	1.8428	0.0033	0.0017	0.51	0.44	0.12
256	23.6532	0.0525	0.0171	2.04	1.55	0.50
512	236.7505	0.7081	0.1487	8.06	6.22	2.00

Cumulative execution times were: Python 262.74s, Java 0.77s, and C 0.17s. Average memory consumption: Python 2.68 MB, Java 2.11 MB, and C 0.66 MB.

As shown in Figure 1, the exponential growth pattern confirms $O(n^3)$ complexity, with Python exhibiting the steepest increase. C and Java maintain near-baseline performance across smaller matrices.

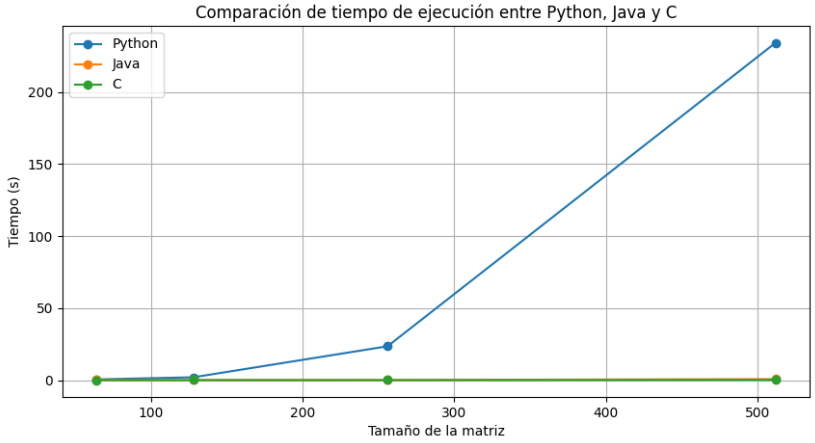

Figure 1: Execution time comparison demonstrating $O(n^3)$ complexity scaling

Figure 2 presents memory consumption on logarithmic scale, revealing expected $O(n^2)$ space complexity. C consistently achieves the lowest memory footprint.

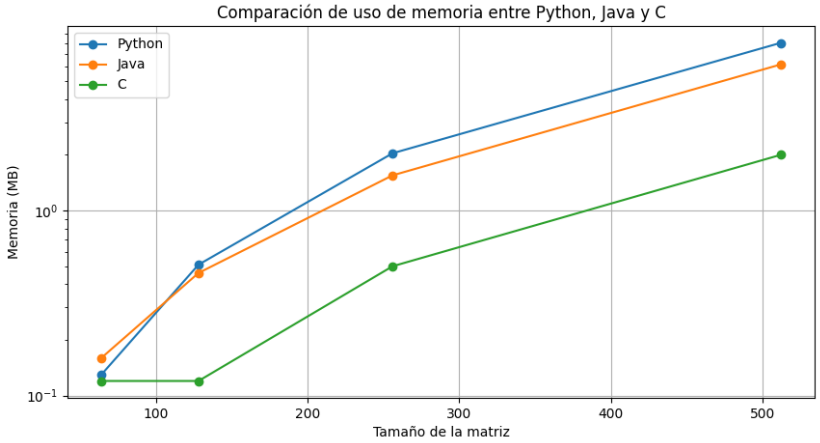

Figure 2: Memory usage comparison (logarithmic scale)

Time scaling respects theoretical predictions. When dimensions doubled (256 to 512), execution time increased by factors approximating $2^3 = 8$. Observed factors: Python 10.0x, Java 13.5x, C 8.7x. Deviations reflect cache effects and compiler optimizations. Notably, Java's 128×128 computation (0.0033s) completed faster than 64×64 (0.0037s), attributable to JIT warm-up effects.

5 Discussion

Python: Low performance stems from interpreted execution and dynamic typing. Each operation requires type checking, method resolution, and reference management. However, NumPy provides 50-100x improvements through vectorized C implementations, while tools like Numba and Cython enable near-native performance.

Java: Demonstrates compelling performance-productivity balance. JIT compilation achieves near-C performance after warm-up. Automatic garbage collection eliminates

manual memory errors with minimal overhead. Despite runtime costs, Java achieved times only 4.5x slower than C—modest considering development advantages.

C: Obtained optimal performance through direct native compilation. Aggressive compiler optimizations include loop unrolling and automatic vectorization. Absence of runtime checking and interpreter layers maximizes hardware utilization. Complete memory control enables low-level optimizations when required.

6 Conclusions

This work demonstrates that programming language selection dramatically influences computational performance for intensive numerical operations. C dominated with execution times 1,545x faster than Python and memory efficiency 3x superior. Java proved an excellent intermediate solution, delivering 78% of C's performance with significant productivity advantages. Python without optimized libraries proves inadequate for intensive computation, underscoring the importance of NumPy for numerical work.

No universally optimal language exists; each occupies a distinct niche. The recommended strategy combines Python for rapid prototyping, Java for production systems requiring performance-maintainability balance, and C for performance-critical components. Many successful projects employ hybrid approaches: Python interfaces with C computational cores, achieving optimal development velocity and runtime performance.

7 Future Work

Future research could explore parallel implementations (OpenMP, Java streams, Python multiprocessing) to reveal multi-core utilization. One limitation is restriction to single-threaded implementations and matrices up to 512×512. Further validation is required for GPU-accelerated contexts and alternative algorithms (Strassen, Coppersmith-Winograd). Analysis of cache behavior and microarchitectural performance counters would provide deeper insight into observed differences.

8 Repository

TASK-1_-Language-Benchmark