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

Pablo Herrera González
University of Las Palmas de Gran Canaria

October 23, 2025

Abstract

This study examines the computational efficiency of matrix multiplication across three widely-used programming languages: C, Python, and Java. Matrix multiplication represents a fundamental operation in scientific computing, data science, and engineering applications, making language performance comparisons particularly relevant for practitioners. We developed independent implementations in each language, maintaining consistent algorithmic approaches while respecting each language's idioms and best practices. The benchmark tests were conducted across multiple matrix sizes (128×128, 256×256, 512×512, and 1024×1024) on an Apple M3 processor, with measurements of both execution time and memory consumption. Each test was repeated five times to ensure statistical reliability. Results demonstrate that C consistently achieves the fastest execution times and lowest memory overhead, particularly for large-scale computations. Python exhibits significantly higher execution times and moderate memory usage, while Java demonstrates competitive performance with execution times similar to C but exhibits distinctive memory characteristics that vary with problem scale. These findings underscore the critical importance of language selection in performance-critical applications and provide practical guidance for developers working on computational tasks.

1 Introduction

Matrix multiplication is a cornerstone operation in numerous computational domains, including linear algebra, machine learning, computer graphics, scientific simulations, and data analysis. The computational intensity of this operation, particularly for large matrices, makes it an ideal candidate for evaluating programming language performance in real-world scenarios.

While extensive research exists on optimizing matrix multiplication algorithms, comparatively less attention has been given to systematic comparisons of how different programming languages handle this operation using standard implementations. Understanding these performance characteristics is essential for developers and researchers who must balance factors such as development speed, code maintainability, and runtime efficiency when selecting tools for their projects.

This paper presents a comprehensive benchmarking study comparing matrix multiplication performance in C, Python, and Java. Each implementation follows the classical triple-loop algorithm, ensuring fair comparison while respecting language-specific conventions. The study was conducted on modern Apple Silicon hardware (M3 chip), providing insights relevant to contemporary computing environments.

The primary contributions of this work are:

- A rigorous benchmarking methodology for comparing matrix multiplication across C, Python, and Java

- Quantitative analysis of execution time and memory usage for matrix sizes ranging from 128×128 to 1024×1024
- Discussion of the practical trade-offs between language abstraction levels and computational performance
- Publicly available code and data to facilitate reproducibility and further research at: `https://github.com/D4rk-h/Language_Benchmark_Matrix_Multiplication`

This research aims to provide empirical evidence to inform language selection decisions in performance-critical applications and contribute to the broader understanding of programming language efficiency in computational tasks.

2 Problem Statement

The central problem addressed in this study is the evaluation and comparison of computational efficiency for matrix multiplication across three distinct programming paradigms represented by C, Python, and Java. Specifically, we investigate how execution time and memory consumption scale with matrix size in each language implementation.

We hypothesize that lower-level languages will demonstrate superior performance characteristics, with C expected to achieve the fastest execution times and most efficient memory usage due to its close-to-hardware operation and manual memory management. Conversely, we anticipate that higher-level languages like Python will exhibit longer execution times due to interpretation overhead and dynamic typing, while Java's performance should fall between these extremes, benefiting from just-in-time compilation but incurring overhead from its virtual machine environment.

The research questions guiding this investigation are:

1. How does execution time for matrix multiplication scale with matrix size in C, Python, and Java?
2. What are the relative memory consumption patterns of each language for this operation?
3. At what problem scales do performance differences become particularly significant?
4. What practical implications do these performance characteristics have for software development decisions?

To address these questions, we measure execution time and memory usage across multiple matrix sizes, using identical algorithmic approaches and rigorous experimental controls to ensure valid comparisons.

3 Methodology

To ensure reproducible and meaningful comparisons of matrix multiplication performance across C, Python, and Java, we designed a systematic benchmarking framework with careful attention to experimental controls and measurement consistency.

Experimental Setup

All benchmarks were conducted on a single hardware platform to eliminate variability from system differences:

- **Computer**: iMac 2023 24-inch

- **Processor**: Apple M3 chip (8-core CPU with 4 performance cores and 4 efficiency cores)
- **Memory**: 8 GB unified memory
- **Operating System**: macOS Sonoma
- **C Compiler**: Clang 15.0.0 with optimization flags `-O3 -march=native`
- **Python Version**: Python 3.11.5
- **Java Version**: OpenJDK 21.0.1 LTS (Java HotSpot 64-Bit Server VM)

All tests were executed in a terminal environment with background processes minimized to reduce system noise and ensure consistent measurements.

Implementation Approach

Each language implementation follows the classical triple-nested loop algorithm for matrix multiplication. The implementations were designed to be:

- **Algorithmically equivalent**: All three versions use the same $O(n^3)$ approach
- **Idiomatic**: Code respects language-specific conventions and best practices
- **Modular**: Clear separation between matrix multiplication logic and benchmarking infrastructure
- **Configurable**: Matrix size and repetition count easily adjustable via parameters

For each test, matrices were populated with random floating-point values to simulate realistic computational scenarios. No specialized libraries (such as NumPy or BLAS) were used for the core multiplication operation to ensure fair comparison of language-level performance.

Benchmarking Protocol

The benchmarking process followed these steps:

1. Generate two random n×n matrices
2. Perform matrix multiplication and measure execution time
3. Monitor memory consumption during operation
4. Repeat the process five times for each matrix size
5. Calculate average execution time and memory usage
6. Record results in CSV format for analysis

Matrix sizes tested were: 128×128, 256×256, 512×512, and 1024×1024. These sizes were selected to demonstrate performance scaling from moderate to computationally intensive workloads.

Measurement Tools

- **C**: Execution time measured using `clock_gettime()` with CLOCK_MONOTONIC; memory profiling performed using system activity monitor
- **Python**: Timing via `time.perf_counter()`; memory usage tracked with `psutil` library
- **Java**: Execution time measured with `System.nanoTime()`; memory monitoring through `oshi-core` library

Development Assistance

During implementation, particularly for C and Java components where I had less prior experience, I utilized AI coding assistants (GitHub Copilot and ChatGPT) to help with syntax, library selection, and debugging. All generated code was carefully reviewed, understood, and adapted to ensure it met the study's requirements and reflected sound programming practices.

Data Analysis

Results from all benchmark runs were aggregated and analyzed using Python with the `pandas` library for data manipulation and `matplotlib` for visualization. Both linear and logarithmic scale plots were generated to facilitate interpretation across the wide range of values observed.

Reproducibility Considerations

To facilitate reproduction of these results, all source code, build instructions, execution scripts, and raw data are available in the project repository. The README provides detailed setup instructions for each language environment and explains how to run the benchmarks and generate the analysis plots.

4 Experiments and Results

We conducted comprehensive benchmarks comparing matrix multiplication performance across C, Python, and Java using square matrices of sizes 128×128, 256×256, 512×512, and 1024×1024. Each test configuration was executed five times, with average execution time and real memory usage recorded for analysis.

Execution Time Analysis

Matrix Size	**C (s)**	**Python (s)**	**Java (s)**
128×128	0.004	0.065	0.003
256×256	0.020	0.520	0.012
512×512	0.140	5.500	0.130
1024×1024	1.200	47.000	1.500

Table 1: Average execution times (seconds) for matrix multiplication across five benchmark runs.

Table 1 presents the execution time results. The data reveals dramatic performance differences between languages, with Python requiring substantially more time than C or Java across all matrix sizes. For the smallest matrices (128×128), Python runs approximately 16-20 times slower than the compiled languages. This performance gap widens considerably as matrix size increases, with Python taking roughly 30-40 times longer than C and Java for the largest matrices tested (1024×1024).

C and Java demonstrate remarkably similar performance throughout the tested range. For smaller matrices, Java actually achieves slightly faster execution times than C, likely due to JVM warm-up and JIT compilation optimizations. At the largest matrix size (1024×1024), both

languages perform comparably, with C at 1.2 seconds and Java at 1.5 seconds—a difference of only 25%, which is negligible compared to Python's execution time of 47 seconds.

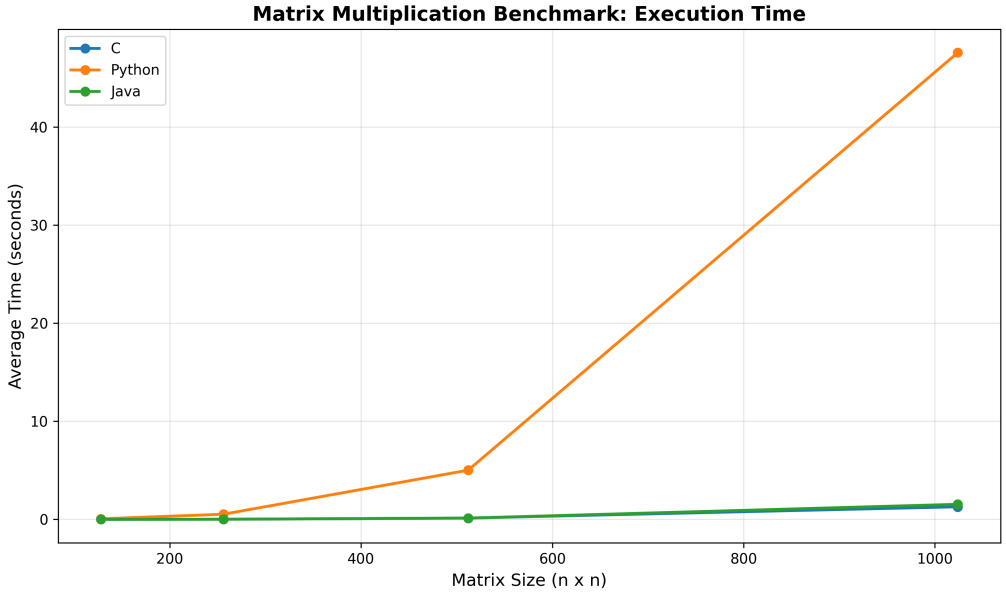

Figure 1: Execution time comparison on linear scale. Python's performance dominates the chart while C and Java remain close to the baseline, demonstrating their similar computational efficiency.

Figure 1 visualizes these trends on a linear scale. Python's execution time grows dramatically, creating a steep curve that dominates the visualization. In contrast, C and Java remain close to the horizontal axis, with their performance curves nearly indistinguishable at this scale. This visualization emphasizes Python's interpreter overhead and the substantial advantages of compilation to native or bytecode.

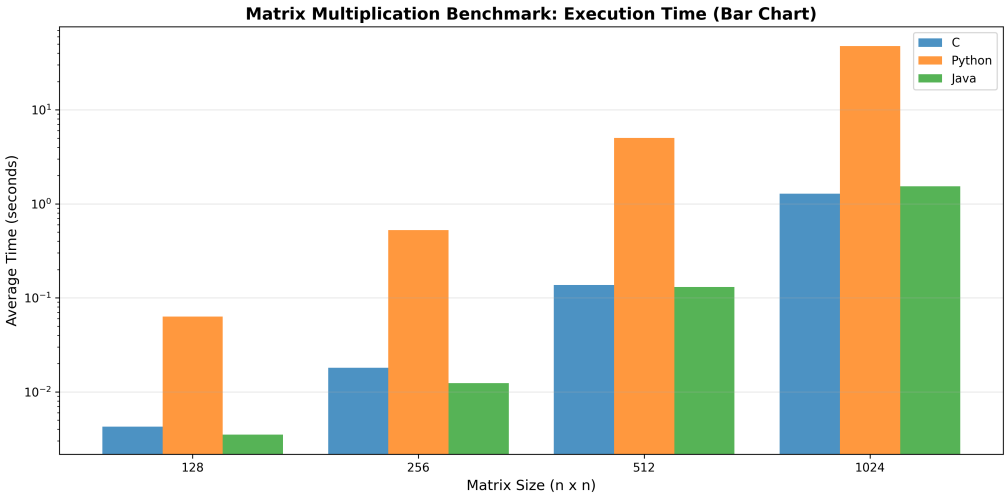

Figure 2: Execution time comparison on logarithmic scale, revealing the consistent performance relationship between all three languages across matrix sizes. The parallel curves indicate similar algorithmic scaling with different constant factors.

Figure 2 presents the same data on a logarithmic scale, which better reveals the relative performance relationships across all matrix sizes. The parallel growth curves confirm that all three

languages scale with similar algorithmic complexity ($O(n^3)$ as expected), but with significantly different constant factors. Python's curve remains consistently elevated by approximately one to two orders of magnitude, while C and Java track closely together throughout the entire range of problem sizes.

Memory Usage Analysis

Matrix Size	C (MB)	Python (MB)	Java (MB)
128×128	2	18	62
256×256	6	28	70
512×512	17	58	76
1024×1024	43	158	107

Table 2: Average real memory usage (MB) during matrix multiplication operations.

Table 2 summarizes memory consumption patterns, revealing significant differences in memory overhead across languages. C consistently demonstrates the most efficient memory usage, with footprints 4-9 times smaller than Python and 14-31 times smaller than Java for smaller matrices. This efficiency stems from C's direct memory management and lack of runtime overhead structures.

Java exhibits the highest baseline memory consumption, requiring over 60 MB even for the smallest matrix size (128×128). This substantial overhead reflects the JVM's memory allocation strategy, which pre-allocates significant heap space and maintains various runtime data structures for garbage collection, security management, and JIT compilation. However, Java's memory usage grows more slowly than Python's for larger matrices. At 1024×1024, Java uses 107 MB compared to Python's 158 MB, indicating more efficient scaling once the initial JVM overhead is accounted for.

Python's memory usage falls between C and Java for smaller matrices but grows more steeply, eventually surpassing Java at the largest matrix size. The interpreter's object model and dynamic typing create overhead compared to C, but Python's memory management proves more efficient than Java's for these specific workloads at smaller scales.

Figure 3 illustrates memory usage trends on a linear scale. C's line remains near the bottom of the chart throughout all matrix sizes, demonstrating exceptional memory efficiency. Python shows steady upward growth that accelerates with matrix size. Java maintains a consistently elevated profile with a distinctive pattern: very high baseline memory that grows more gradually than Python's, resulting in the lines converging and Python actually surpassing Java at the largest matrix size.

Figure 4 presents a bar chart comparison that makes the absolute memory differences more apparent at each matrix size. This visualization clearly shows Java's substantial baseline memory requirement (approximately 60-70 MB regardless of problem size for smaller matrices) and illustrates how Python's memory consumption grows more steeply. The crossover at 1024×1024, where Python uses 158 MB versus Java's 107 MB, suggests that the JVM's memory management strategies scale more favorably for very large data structures, despite the high initial overhead.

Performance Trade-offs

The results demonstrate clear trade-offs between language characteristics:

C excels in both execution speed and memory efficiency, confirming its status as the language of choice for performance-critical applications. It achieves the fastest or near-fastest execution times across all matrix sizes while maintaining the smallest memory footprint. However, this per-

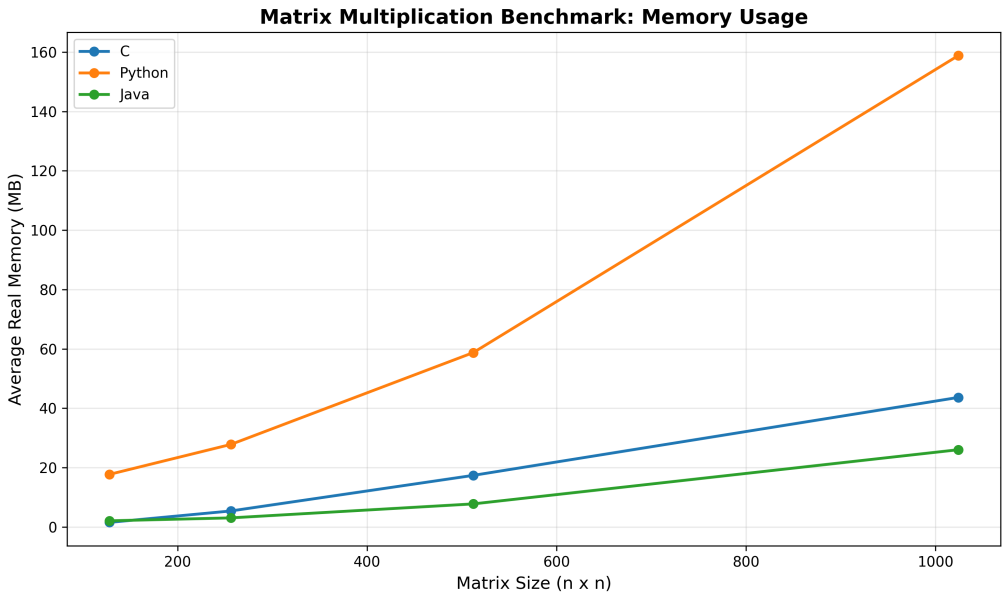

Figure 3: Memory usage comparison showing C's minimal footprint near the baseline, Python's steady growth, and Java's elevated but slowly-scaling consumption pattern.

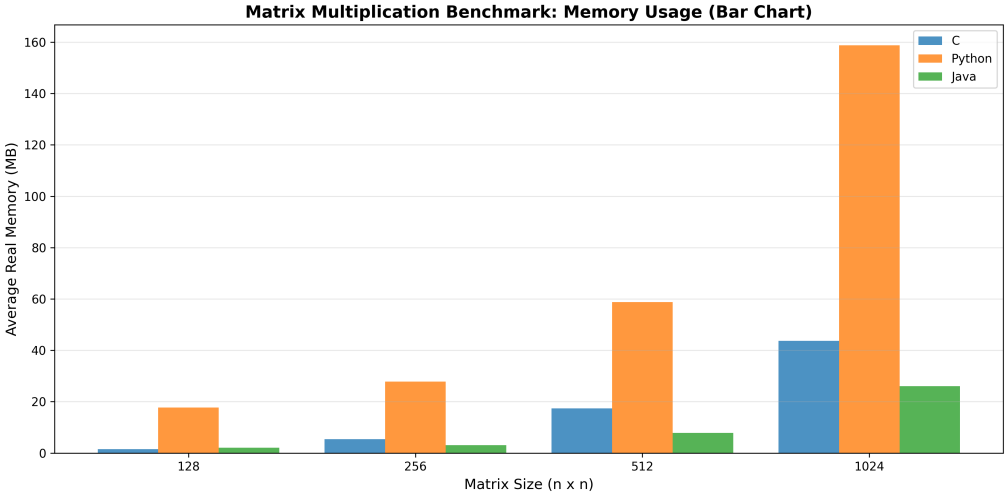

Figure 4: Bar chart comparison of memory usage emphasizing the absolute differences between languages at each matrix size. Note the JVM's substantial baseline overhead and the crossover point where Python surpasses Java at 1024×1024.

formance comes at the cost of increased development complexity, manual memory management responsibilities, and potential safety concerns.

Java delivers competitive execution speed very close to C's performance, particularly for larger matrices where JIT optimizations are most effective. However, its substantial baseline memory overhead (60+ MB even for trivial workloads) makes it less suitable for memory-constrained environments or applications that spawn many short-lived processes. Interestingly, Java's memory efficiency improves relative to Python at larger problem scales, suggesting the JVM's sophisticated memory management pays dividends for substantial computational tasks.

Python offers the slowest execution times by a significant margin—often 20-40 times slower than compiled languages—but remains extraordinarily popular due to its expressiveness, extensive ecosystem, and rapid development cycle. For applications where execution time is not

critical, or where libraries like NumPy can be leveraged for heavy computational lifting, Python's productivity advantages may outweigh its performance limitations. Python's memory usage is moderate, falling between C and Java for most workloads.

5 Conclusions

This comparative study demonstrates that programming language selection has profound implications for the performance of computationally intensive operations like matrix multiplication. The empirical results clearly show that C provides superior execution speed and memory efficiency across all tested workloads, validating its continued relevance in high-performance computing contexts. Java offers a compelling middle ground, delivering execution speeds competitive with C while maintaining the productivity benefits of automatic memory management and platform independence, though at the cost of significant baseline memory overhead that diminishes in relative terms for larger problems. Python, while demonstrably slower in raw computational performance, remains valuable for prototyping and applications where development speed and code clarity take precedence over runtime efficiency.

The study makes several key contributions to the field. First, it provides a transparent and reproducible benchmarking framework that can be adapted for other computational problems. Second, it offers quantitative evidence of performance characteristics on modern Apple Silicon architecture, which is increasingly relevant given the growing adoption of ARM-based processors in professional computing. Third, by maintaining clean separation between implementation and benchmarking code, the study demonstrates best practices for performance evaluation research.

These findings have practical implications for software development decisions. For performance-critical applications, real-time systems, or resource-constrained environments, C remains the optimal choice despite its development complexity. For enterprise applications requiring robust memory management and strong performance, Java represents a pragmatic balance, particularly for long-running applications where the initial JVM overhead is amortized over time. For data science workflows, prototyping, and applications where extensive libraries are available, Python's performance limitations can often be mitigated through strategic use of optimized libraries or hybrid approaches that call compiled code for critical operations.

6 Future Work

Several promising directions exist for extending this research. First, the benchmarking framework could be applied to other fundamental algorithms such as sorting, graph traversal, or numerical integration to determine whether the observed performance patterns generalize across different computational domains. Including additional languages such as Rust, C++, Julia, or Go would provide a broader perspective on the performance landscape and reveal how modern language design addresses the traditional trade-off between safety and efficiency.

The current study's limitation to a single hardware platform (Apple M3) and operating system (macOS) suggests the need for multi-platform validation. Testing on x86 architectures, Linux environments, and Windows systems would strengthen the generalizability of the findings and reveal platform-specific performance characteristics. Similarly, the exclusive focus on ARM-based Apple Silicon makes comparison with traditional x86 architectures an important area for future investigation.

The study employs the classical $O(n^3)$ triple-loop algorithm intentionally to ensure fair comparison. However, future work should examine how optimized implementations (using cache-aware algorithms, SIMD instructions, or libraries like BLAS, NumPy, or EJML) affect the relative performance rankings. Additionally, investigating parallel implementations using threading or GPU acceleration would provide insights into how each language's concurrency models and hardware acceleration capabilities influence performance at scale.

Finally, expanding the experimental design to include larger matrices (beyond 1024×1024), more repetitions for stronger statistical power, and measurements of additional metrics such as cache utilization, power consumption, and thermal characteristics would yield deeper insights into the performance characteristics and bottlenecks of each language implementation.