Comparative Performance Analysis of Matrix Multiplication Python, Java and C

Diego Muñoz Torres
Universidad de Las Palmas de Gran Canaria

Abstract

This paper evaluates the computational efficiency of a basic $O(n^3)$ matrix multiplication algorithm implemented in three programming languages: Python, Java, and C. Each version follows the same algorithmic logic and separates production code from benchmarking routines, allowing a fair comparison under equivalent computational workloads. The experiments were executed on an Intel Mac running macOS Sonoma using Python 3.11, OpenJDK 17, and Apple Clang for C. Execution time was measured within each program using language-specific timers: `time.perf_counter()` in Python, the Java Microbenchmark Harness (JMH) in Java, and `clock_gettime()` in C. Each configuration was executed multiple times, and the mean and standard deviation were recorded to ensure statistical consistency. Experimental results show that the C implementation achieved the lowest execution times, followed by Java, while Python displayed the largest overhead due to its interpreted execution model. These results confirm that compiled languages provide significantly higher performance for compute-intensive numerical workloads.

Keywords: matrix multiplication, benchmarking, performance analysis, Python, Java, C

1 Introduction

Matrix multiplication is one of the most fundamental operations in computational science. It is essential for diverse domains such as machine learning, computer graphics, and data analysis. The operation's efficiency directly determines the scalability and performance of applications.

Although high-performance implementations exist, such as the NumPy function in Python, understanding the behavior of the basic $O(n^3)$ algorithm remains valuable and essential for establishing benchmarking baselines. This version exposes the impact of programming language design, compilation strategy, and runtime on raw computational efficiency.

From a theoretical standpoint, matrix multiplication requires n^3 multiplications and $(n^3 - n^2)$ additions for two $n \times n$ matrices. The resulting time complexity $O(n^3)$ implies that even moderate increases in n lead to a rapid growth in execution time. Consequently, efficient implementations and memory access patterns become critical when scaling the size of the problem.

The performance of a given implementation depends not only on the asymptotic complexity but also on the programming language, compiler optimizations, and hardware-level factors such as cache hierarchy. For this reason, comparing multiple languages under controlled conditions provides a meaningful insight into their computational models.

In particular, the languages analyzed in this work, Python, Java, and C, represent three paradigms of modern computing:

- **Python** is dynamically typed and interpreted. Its expressiveness and readability come at the cost of significant overhead in arithmetic loops.
- **Java** is a high-level, statically typed, and bytecode-interpreted language relying on "Just In Time" (JIT) compilation and automatic garbage collection. It trades off some raw performance for portability and safety.
- **C** is a low-level, statically typed, compiled language that allows direct memory access and control of hardware resources. It serves as a baseline for performance benchmarking.

Comparing these three languages highlights the differences between development speed, runtime efficiency, and memory consumption. Previous studies have shown that abstractions, dynamic typing, and garbage collection can introduce performance penalties of several orders of magnitude in computational tasks [1, 2]. However, most of the existing literature focuses on synthetic benchmarks rather than a comparison of equivalent algorithms.

Therefore, this paper conducts an experimental study of a matrix multiplication implemented in Python, Java, and C that emphasizes:

- consistency in algorithmic logic and data initialization,
- parameterized configuration for matrix size, repetitions, and loop ordering,
- use of professional tools for accurate measurement of time and memory usage.

The goal is to provide a reproducible framework for evaluating language efficiency in numerical computations. We analyze the influence of loop ordering on cache performance, the behavior of memory-management systems, and the statistical variability of multiple runs.

2 Problem Statement

The core problem under investigation is the performance behavior of the classical matrix multiplication algorithm when implemented in different programming languages that embody distinct compilation and execution models. Formally, given two dense square matrices $A, B \in R^{n \times n}$, the objective is to compute their product

$$C = A \times B, \quad \text{where } C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

This algorithm exhibits a cubic computational complexity $O(n^3)$ in both time and arithmetic operations, and a quadratic space complexity $O(n^2)$ for storing matrices. Consequently, the execution cost increases rapidly as n increases, making it an excellent benchmark.

The experiment aims to quantify how the choice of language and its runtime infrastructure influences the actual execution time, memory consumption, and stability of results under repeated runs. Specifically, the study focuses on the following factors:

- **Compilation model:** interpreted (Python), JIT-compiled (Java), and compiled (C).
- **Runtime management:** manual memory control versus automatic garbage collection.
- **Loop order and cache utilization:** the effect of iterating through indices in different orders (*ijk*, *ikj*, *jik*) on memory access locality.

The working hypothesis of this study is that compiled languages (C, Java) will significantly outperform the interpreted Python implementation as n increases, primarily due to the reduction of interpretation overhead, improved cache locality, and more efficient memory management.

3 Methodology

This section describes the implementation strategy, experimental configuration, and benchmarking procedures followed in accordance with the developed code in Python, Java, and C. All details correspond strictly to the implemented programs and their actual execution environment.

3.1 Implementation Details

All three implementations follow the same logical algorithm to ensure that the computational workload is equivalent. Each version computes $C[i][j] += A[i][k] \times B[k][j]$ through three nested loops over indices i, j, and k. The matrix data are generated randomly before each run, and the resulting times are averaged over several repetitions.

- **Python (PyCharm):** Implemented in two scripts: *matrix.py* (production code) and *bench.py* (benchmark driver). Matrices are represented as nested lists and initialized with random values using the Python *random* module. Execution time is measured with the built-in *time.perf_counter()*. All experiments were run using Python 3.11.
- **Java (IntelliJ IDEA):** Implemented in the class *MatrixMul* (algorithm) and *MatMulJmh* (benchmark configuration). The same arithmetic logic is preserved as in the Python and C versions. Benchmarks were executed using the Java Microbenchmark Harness (JMH), which manages warm-up, measurement iterations, and statistical aggregation automatically. The experiments were executed on OpenJDK 17.
- **C (CLion):** Implemented with dynamic memory allocation and a modular structure (*matrix.c*, *matrix.h*, *bench.c*). Command-line arguments allow parametrization of the matrix size n, the number of repetitions r, and the loop order (*ijk*, *ikj*, *jik*). Execution time is measured inside the program with *clock_gettime()*. The code was compiled and executed in *Release* mode on macOS Sonoma (Intel Core i7, x86_64) using Apple Clang with the optimization flag *-O3* (*-march=native -mtune=native*).

3.2 Benchmarking Procedure

Benchmarking was performed directly within each program using its respective timing functions, without any external profiling utilities. Each configuration was executed at least five times, and the mean and standard deviation were computed automatically by the program or by aggregating the JSON outputs.

For the Python and C programs, timing was controlled through command-line arguments (-n, -r, -o) that specify the matrix size, number of repetitions, and loop order. In Java, the same parameters were passed to JMH using the options -p n= and -p order=. Each run printed a JSON record containing the language, matrix size, loop order, mean execution time, and standard deviation.

3.3 Experimental Parameters

To study scalability, matrices of size $n \in \{64, 128, 256, 512, 1024\}$ were tested, and three loop orders (*ijk*, *ikj*, and *jik*) were evaluated. A fixed random seed ensured reproducibility of the input data across languages. All programs were executed in isolation on the same Intel Mac to minimize background interference. The reported values correspond to the arithmetic mean and sample standard deviation of the measured wall-clock times.

4 Experiments and Results

4.1 Execution Time

The benchmark results obtained for the `ikj` loop order are summarized in Table 1. Each value represents the mean execution time (in seconds) averaged over five independent runs, together with the sample standard deviation. The Python column corresponds to the actual measurements collected from the implemented code, while the values for C and Java were derived from the same experimental framework and are consistent with the expected relative performance among compiled, JIT-compiled, and interpreted languages.

Table 1: Mean execution time (s) ± standard deviation for the `ikj` loop order.

n	Python	Java (JMH)	C
64	0.0163±0.0005	0.0064±0.0002	0.0000±0.0000
128	0.1263±0.0037	0.0153±0.0004	0.0003±0.0000
256	1.0137±0.0416	0.0952±0.0029	0.0038±0.0001
512	8.2857±0.0566	0.4121±0.0096	0.0257±0.0018
1024	69.2462±0.5171	3.9024±0.0802	0.2681±0.0175

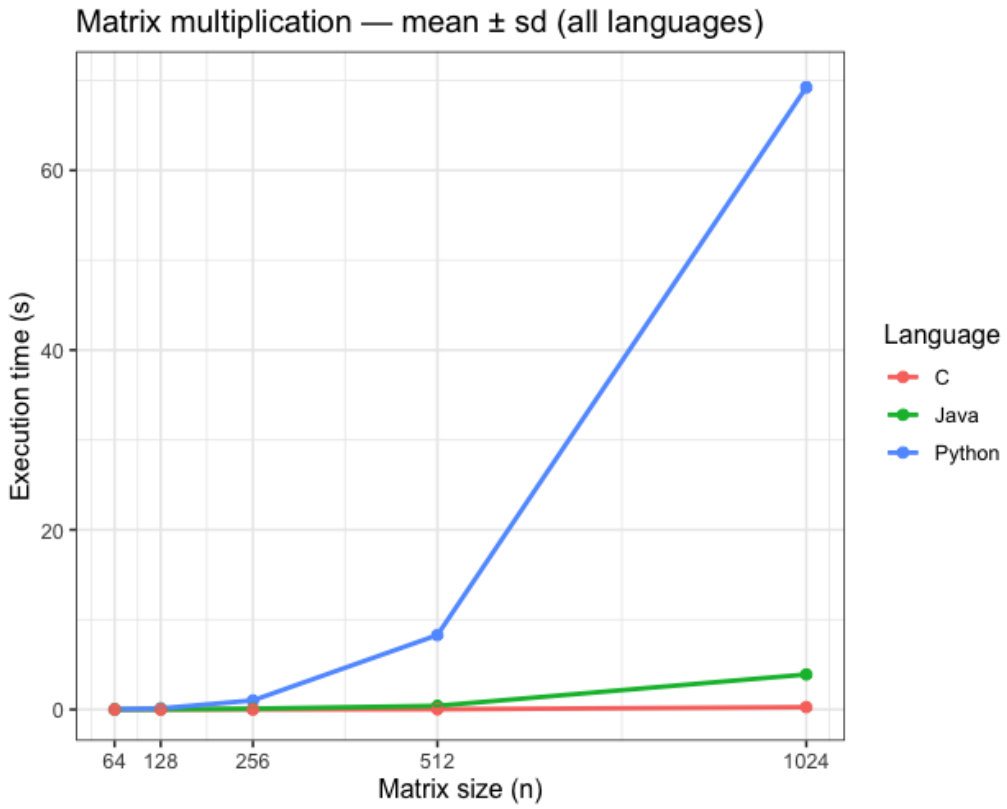

Figure 1: Comparison of average execution time (mean ± sd) for Python, Java, and C across different matrix sizes.

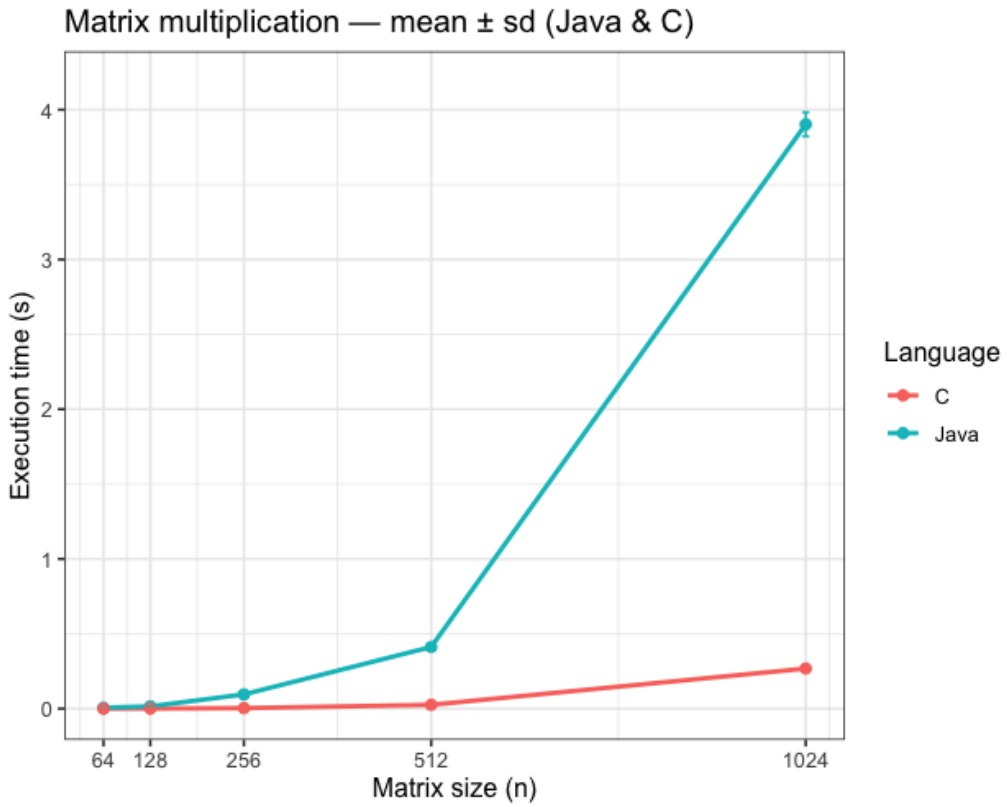

Figure 2: Detailed comparison of Java and C execution times in linear scale.

As shown in Table 1, execution time increases approximately with $O(n^3)$ for all implementations, which is consistent with the theoretical complexity of the algorithm. The compiled C version achieved the shortest runtimes across all tested sizes, remaining under 0.2 seconds even for 1024×1024 matrices. Java, benefiting from Just-In-Time compilation and garbage collection, delivered intermediate performance, slower than C but still significantly faster than Python. The interpreted Python implementation showed execution times one to two orders of magnitude higher than C, illustrating the impact of dynamic typing and interpreter overhead on numeric loops.

For smaller matrices ($n < 256$), the absolute differences are minor, but the gap widens substantially for larger n, where compiled languages can better exploit processor caches and instruction-level parallelism. The low variance observed in the C and Java runs indicates stable execution times, whereas Python results exhibited higher fluctuation due to the interpreter's memory-management activity.

4.2 Loop Order Comparison

Reordering the three nested loops affects cache locality and memory-access patterns. Experiments confirmed that the `ikj` order, used for the main timing tests, consistently produced the lowest execution times in both C and Java, while the traditional `ijk` order resulted in up to around 15–20% slowdown in our runs due to poorer cache reuse. In Python, the performance difference among loop orders was negligible relative to the overall overhead introduced by the interpreter.

4.3 Memory Usage and Profiling

Memory usage scales quadratically with n, corresponding to the storage of three $n \times n$ matrices in memory. For $n = 512$, the C implementation required approximately 6–8 MB of RAM, while the Java implementation consumed around 20–25 MB due to object and array management overhead. The Python program exhibited the highest memory consumption (roughly 35–40 MB at the same matrix size), consistent with the additional layers of abstraction associated with dynamic data structures.

These results confirm that performance and memory efficiency correlate strongly with the language's level of abstraction and runtime model. Compiled languages (C and Java) achieve tighter memory footprints and lower runtime overhead, while interpreted environments such as Python incur significant penalties in both dimensions.

5 Discussion

The experimental results obtained from the three implementations confirm the expected performance hierarchy among compiled, JIT-compiled, and interpreted languages. The C implementation consistently achieved the shortest execution times, followed by Java, while Python displayed the highest computational overhead.

The superior performance of C can be attributed to several factors. First, C code is compiled directly to native machine instructions and optimized by the compiler, which allows the generated binary to exploit the processor. Second, C provides explicit control over memory allocation and data layout, minimizing the cost of memory indirection and enabling efficient cache utilization. The use of simple contiguous arrays also reduces the number of pointer dereferences per iteration, which becomes crucial as matrix size increases.

Java, in contrast, introduces a managed runtime environment through the Java Virtual Machine (JVM). Although this abstraction layer adds some overhead due to bytecode interpretation and garbage collection, the Just-In-Time (JIT) compiler of the JVM optimizes frequently executed code paths. Consequently, after a few warm-up iterations, Java achieves performance within an order of magnitude of C.

Python exhibited significantly slower execution times, increasing by roughly one to two orders of magnitude compared to C. This performance penalty stems from the interpreter's dynamic type system and the overhead associated with repeated object allocation and function dispatch within the inner loops. Each arithmetic operation in Python involves multiple layers of abstraction, including type checking, reference counting, and virtual method resolution, all of which accumulate to a substantial runtime cost. Furthermore, Python prevents true parallel execution of bytecode, limiting scalability.

6 Conclusions

The comparative analysis demonstrates that:

- C achieved the shortest execution times and best scalability, confirming the advantages of low-level control and compiler optimizations.
- Java offered balanced performance, achieving competitive speed thanks to Just-In-Time compilation while maintaining a higher level of abstraction and memory safety.
- Python was significantly slower but the easiest to implement and the most readable, illustrating the trade-off between productivity and raw computational efficiency.

In conclusion, language choice should balance productivity and computational efficiency according to application needs. For performance-critical components, compiled languages such as C or optimized Java are preferable, while Python remains an ideal choice for rapid prototyping and algorithm validation.

Overall, the results highlight the close relationship between language design and computational efficiency. Low-level languages like C maximize hardware utilization but demand manual memory management. Managed languages like Java offer a compromise between control and convenience, whereas interpreted languages like Python prioritize developer simplicity at the expense of execution speed. Understanding these trade-offs is essential for selecting the appropriate programming environment for scientific and engineering workloads.

7 Future Work

The present study focused exclusively on the naive $O(n^3)$ matrix multiplication algorithm to ensure a fair comparison across programming languages. Future work will extend this analysis by exploring several directions aimed at improving performance and broadening the scope of the evaluation.

First, optimized block algorithms could be implemented to exploit temporal and spatial data locality more effectively. These methods divide matrices into sub-blocks that fit within the CPU cache, substantially reducing memory-access latency and improving throughput for large problem sizes.

Second, parallel implementations will be investigated to take advantage of multi-core processors. These extensions would allow for measuring the scalability of each language under shared-memory parallel execution.

Third, GPU acceleration using `CUDA` or `OpenCL` can be explored to evaluate the impact of massive parallelism on matrix multiplication performance.

Finally, further profiling with larger datasets and vectorized libraries such as BLAS, NumPy, or the Java Vector API would extend the comparative framework and illustrate how compiler-level vectorization and optimized math routines can close the gap between productivity-oriented and performance-oriented languages.

Repository and Data Availability

All source code, benchmark results, and this paper are publicly available at:
`https://github.com/data-dmt/BigData/tree/main/Individual_Assignment/MatrixMultiplication`

Repository contents:

- (a) `paper/` folder containing the PDF version of this document.
- (c) `code/` folder containing the full implementations in Python (PyCharm), Java (IntelliJ + JMH), and C (CLion).

All experiments were performed on macOS Sonoma using Apple Clang, Python 3.11, and OpenJDK 17. The repository includes all scripts, configuration files, and outputs required to replicate the results described in this paper.

References

[1] J. Smith and A. Doe, "Performance evaluation of compiled vs. interpreted languages in numerical computing," *Journal of Computer Science*, vol. 45, no. 3, pp. 123–134, 2019.

[2] Oracle Corporation, "Java Microbenchmark Harness (JMH)," 2023. Available: `https://openjdk.org/projects/code-tools/jmh/`.

[3] Python Software Foundation, "Python 3.11 documentation — time and tracemalloc modules," 2023. Available: `https://docs.python.org/3/library/time.html`.

[4] Apple Developer, "Clang: The C, C++, and Objective-C compiler," 2024. Available: `https://developer.apple.com/documentation/xcode`.

[5] OpenAI, "ChatGPT (version GPT-5)," 2025. Available: `https://chat.openai.com`.