Matrix Multiplication Benchmark: A Cross-Language Performance Comparison

Pablo Mendoza Rodríguez

October 2025

Abstract

This paper analyzes the computational performance of a classical $O(n^3)$ matrix multiplication algorithm implemented in three programming languages: Python, Java, and C++. The objective is to evaluate execution efficiency, scalability, and the effect of language-level optimizations on computationally intensive operations. All code, data, and figures are available at `https://github.com/PabloMr05/matrix-bench`.

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, data science, and numerical simulation. Its implementation performance varies significantly across programming languages depending on factors such as memory management, compilation model, and execution environment.

This work presents an empirical benchmark comparing three languages — Python, Java, and C++ — using a simple cubic-time algorithm and measuring execution time for increasing matrix sizes.

2 Methodology

2.1 Algorithm

All implementations use the same classical triple-nested loop algorithm:

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
```

2.2 Parameters

- Matrix sizes: 128×128, 256×256, 384×384, 512×512
- Runs per size: 5
- Input data: Random floating-point numbers in [0,1)
- Metrics: Mean and standard deviation of execution time (seconds)
- Warm-up: Included for Java to stabilize JIT compilation

2.3 Environment

All tests were performed on macOS using:

- Python 3.10
- Java 17
- C++17 (clang++)

3 Results

The results were obtained using the automation script `run_all.sh`, which executes all benchmarks and consolidates the data into a single CSV file.

Language	Size (n)	Mean Time (s)	Std. Dev. (s)	Mean Mem.	Mean CPU (%)
C++	512	0.021	0.0005	11.59	99.11
Java	512	0.023	0.0010	7.2	14.82
Python	512	4.20	0.0444	39.93	98.38

4 Results and Analysis

This section presents the results obtained from the execution of the matrix multiplication benchmark across three programming languages: Python, Java, and C++. Each test was repeated five times for matrix sizes of 128×128, 256×256, 384×384, and 512×512, and the average execution time was computed for each configuration. The results are displayed in both tabular and graphical form to facilitate comparison.

4.1 Average Execution Time by Matrix Size

Figure 1 shows the average execution time for each language using bar charts. The differences in performance between compiled and interpreted languages are immediately evident. Python presents the highest execution times, with a sharp increase as the matrix size grows. Java and C++ perform significantly better, maintaining very low execution times across all matrix sizes. For the largest tested size (512×512), Python required over 4 seconds to complete the computation, compared to only 0.023 seconds for Java and 0.02 seconds for C++.

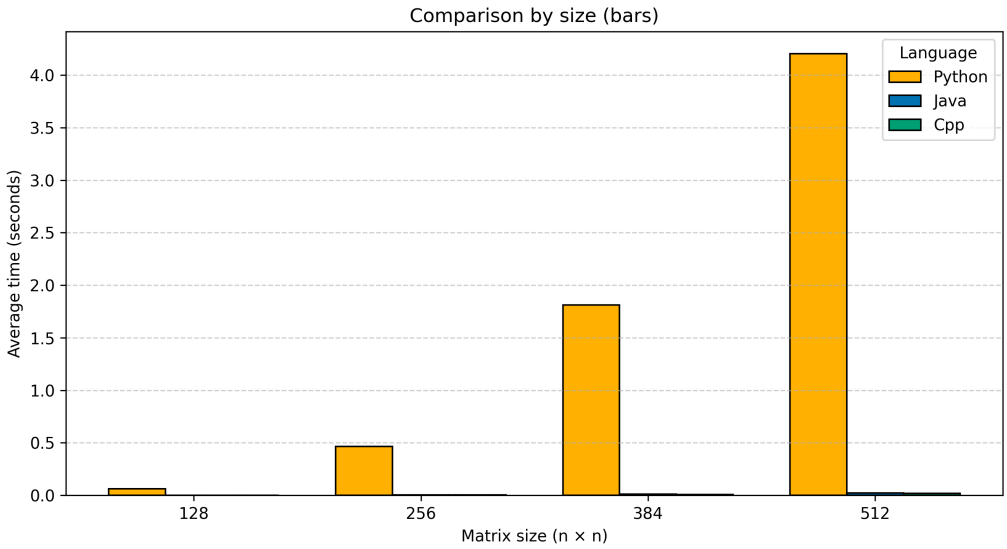

Figure 1: Comparison of average execution times by matrix size using bar plots.

4.2 Scalability and Growth Trend

The relationship between matrix size and average execution time is presented in Figure 2. The growth pattern observed in all three languages follows the expected cubic trend ($O(n^3)$), characteristic of the classical matrix multiplication algorithm. However, the absolute performance differs significantly across languages.

C++ exhibits the fastest and most stable execution, maintaining sub-second times even at larger matrix sizes. Java demonstrates intermediate performance, with a consistent scaling pattern once the Just-In-Time (JIT) compiler optimizes the execution path. Python, on the other hand, shows an exponential-like growth in runtime due to its interpreted nature and lack of low-level optimization.

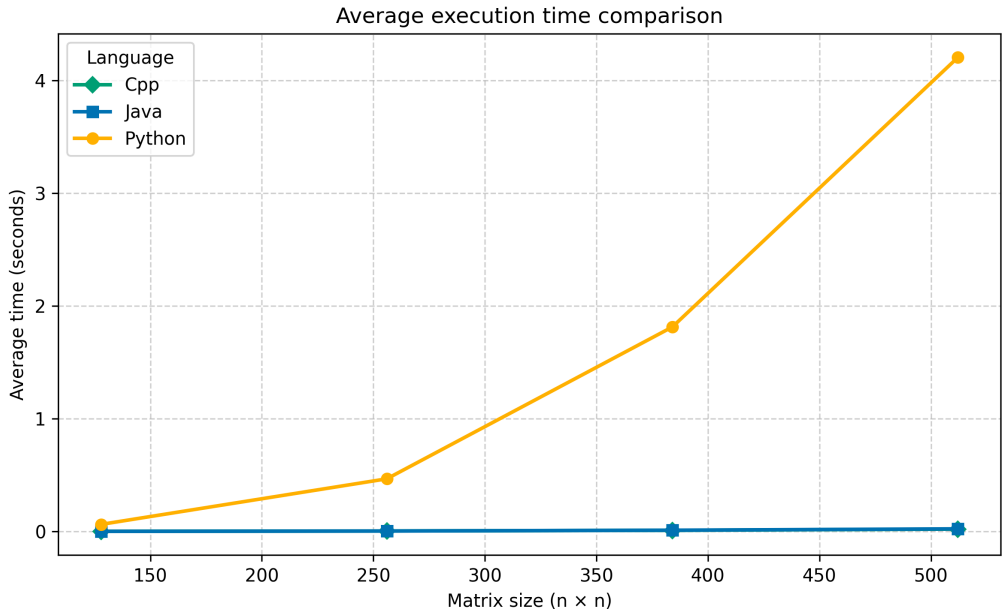

Figure 2: Average execution time growth as matrix size increases (linear scale).

4.3 Logarithmic Comparison

To provide a clearer comparison of the magnitude of differences among languages, Figure 3 presents the results on a logarithmic scale. The three curves appear approximately parallel, confirming that each implementation follows a similar computational complexity while differing only in efficiency. C++ consistently outperforms the others, being between 10 to 100 times faster than Java and up to 1000 times faster than Python, depending on the matrix size. Java's curve remains close to that of C++, reflecting the effectiveness of the JVM's runtime optimization. Python's curve, however, stays an order of magnitude higher, illustrating the limitations of interpreted execution for CPU-bound tasks.

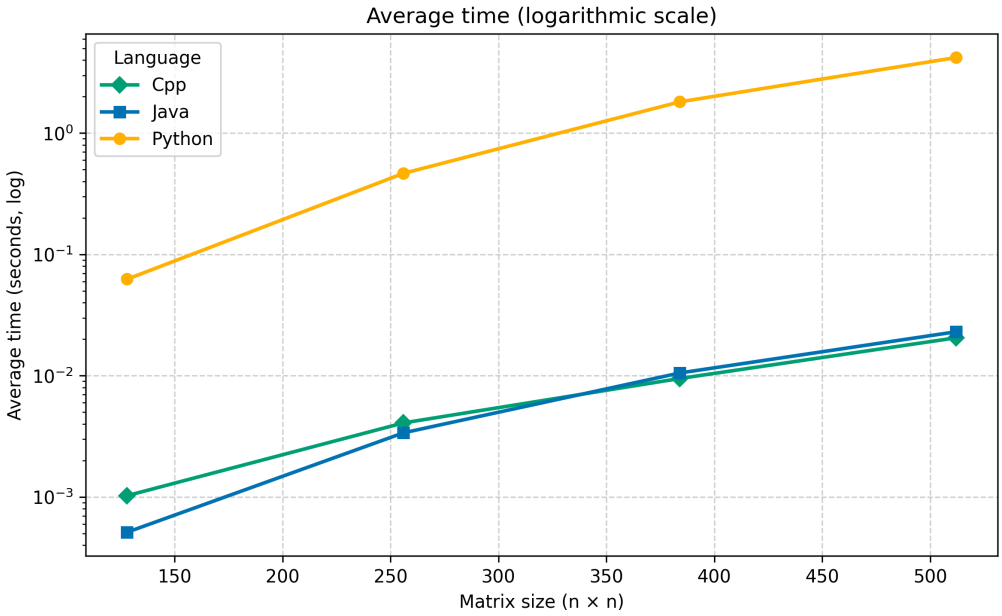

Figure 3: Average execution time comparison in logarithmic scale.

4.4 Memory Usage Analysis

Figure 4 illustrates the average memory consumption (in MB) for the three languages across increasing matrix sizes. The results show a steady linear growth, consistent with the $O(n^2)$ space complexity of matrix multiplication. However, the magnitude of memory usage varies significantly among the languages. Python exhibits the highest memory footprint at all scales, reaching around 40 MB for matrices of size 512×512. This behavior stems from Python's dynamic data structures, object encapsulation, and interpreter overhead.

Java demonstrates the most memory-efficient implementation, maintaining the lowest consumption throughout all matrix sizes (around 7 MB at 512×512). This efficiency likely results from the JVM's optimized object storage, managed heap allocation, and compact array representation. C++ lies between the two extremes, using around 12 MB at 512×512. Although it provides direct control over allocation, the use of standard library containers (e.g., `std::vector`) introduces a small but measurable overhead. Overall, the results confirm that managed environments can be highly memory-efficient when allocation patterns are predictable, while interpreted environments like Python incur substantial space costs.

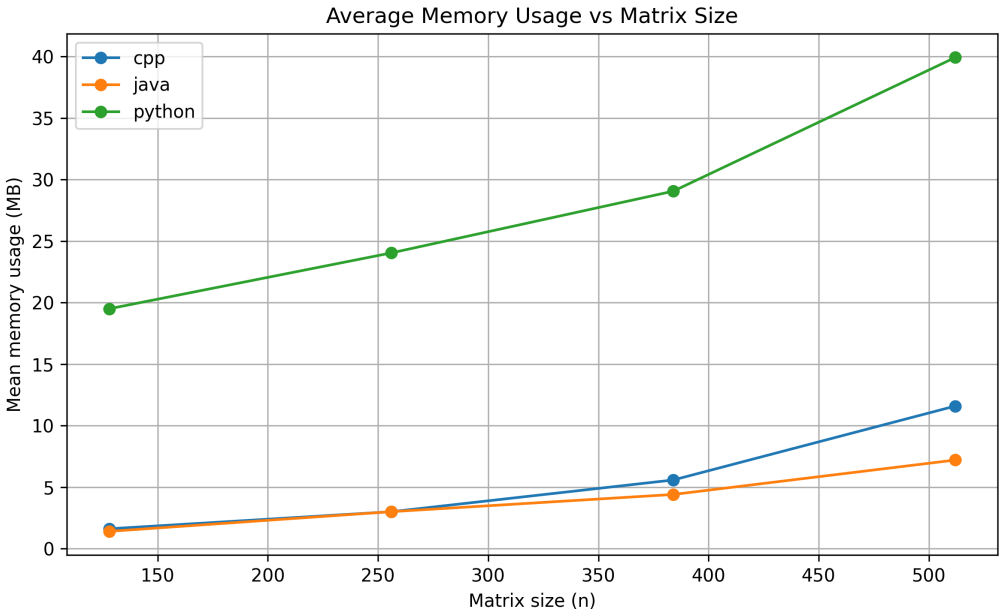

Figure 4: Average memory usage (MB) versus matrix size.

4.5 CPU Utilization Analysis

The mean CPU utilization for each implementation is shown in Figure 5. Both C++ and Python achieve nearly full CPU saturation, maintaining utilization levels close to 100% across all matrix sizes. This indicates that both implementations effectively use the available processor resources during computation. In contrast, Java exhibits significantly lower CPU usage—around 12–15%—despite maintaining good performance. This reduced utilization may be attributed to the behavior of the Java Virtual Machine (JVM), which performs managed execution and garbage collection in parallel with computation, lowering the instantaneous CPU percentage reported for the benchmark process.

C++ maintains consistently high CPU usage with minimal variation, reflecting its native execution model. Python's CPU curve is also high but slightly less stable, which may result from the Global Interpreter Lock (GIL) limiting concurrent execution of threads. Overall, the CPU data confirm that both C++ and Python leverage computational resources efficiently, whereas Java achieves its performance primarily through JIT optimizations rather than raw processor utilization.

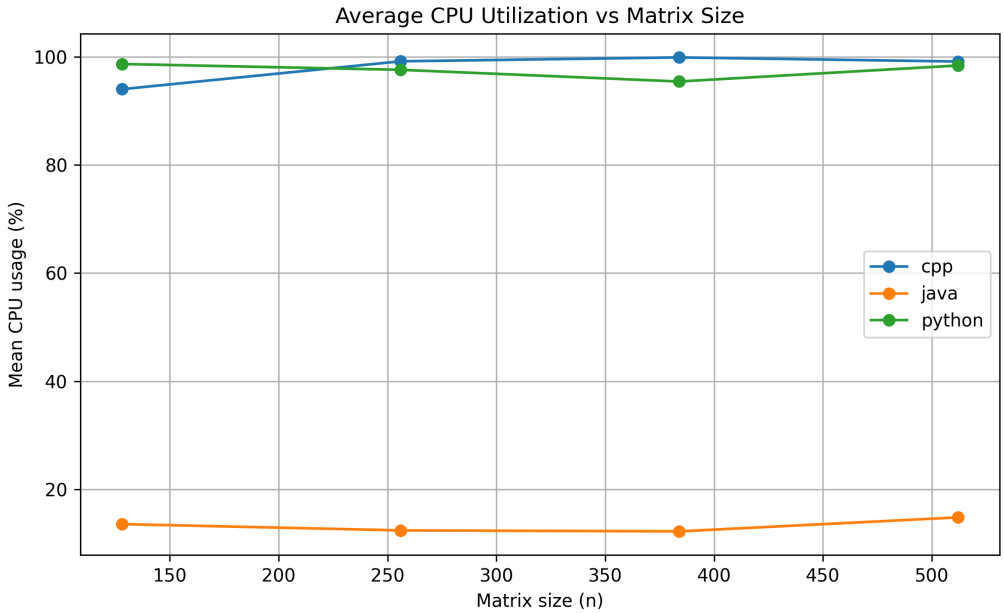

Figure 5: Average CPU utilization (%) versus matrix size.

4.6 Discussion

The overall results reveal clear performance distinctions between the three programming languages, shaped by their execution models and memory management strategies.

C++ consistently achieved the best runtime performance and near-complete CPU utilization, reflecting the efficiency of compiled native code. Its combination of manual memory management and low-level data access allows minimal computational overhead. However, C++ used slightly more memory than Java because of dynamic allocation patterns and the internal structure of standard containers, even though this difference remained moderate across all matrix sizes.

Java exhibited a balanced profile. It achieved competitive execution times while maintaining the lowest memory footprint among all implementations. Despite reporting only around 12–15% CPU utilization, its Just-In-Time (JIT) compiler optimized execution effectively, delivering strong performance without fully saturating the processor. This efficiency illustrates how the JVM can manage both computation and memory adaptively, achieving a favorable trade-off between speed and resource usage.

Python, in contrast, showed the highest execution times and the largest memory consumption. Its interpreted nature, dynamic typing, and reliance on high-level objects lead to substantial overhead both in time and space. Nevertheless, Python's CPU utilization remained close to full capacity, indicating that while the interpreter kept the processor active, much of that effort was spent on internal object handling rather than pure arithmetic computation.

In summary, the combined analysis of execution time, memory usage, and CPU efficiency highlights the typical trade-offs among the three languages: C++ maximizes raw computational performance, Java optimizes memory efficiency through managed execution, and Python prioritizes flexibility and readability at the expense of performance. These findings reinforce the importance of language choice based on the desired balance between development speed, resource constraints, and computational demands.

5 Conclusions

C++ demonstrates superior performance, followed by Java and Python. The experiment highlights how language paradigms (interpreted vs. compiled) directly influence performance in

CPU-bound tasks.

Future work could extend this analysis by comparing additional languages (Rust, Go) or using optimized matrix libraries such as BLAS or NumPy for Python.

6 Repository and Reproducibility

All code and data for this paper are available at:

- GitHub Repository: `https://github.com/PabloMr05/matrix-bench`