Comparative Performance Analysis of Matrix Multiplication Algorithms Across Programming Languages

Guillermo Cubas Granado
github.com/guillecubas/Matrix-Benchmark

June 14, 2025

Abstract

Matrix multiplication constitutes a cornerstone operation in computational data science, with the standard algorithm exhibiting $O(n^3)$ time complexity. As matrix dimensions scale to accommodate big data applications, computational efficiency becomes paramount. This study presents a comprehensive performance evaluation of the classical matrix multiplication algorithm implemented across multiple programming languages. Our analysis aims to provide empirical guidance for language selection in large-scale matrix computation scenarios, comparing execution times and resource utilization patterns across different programming environments while maintaining algorithmic consistency.

1 Introduction

1.1 Matrix Operations in Modern Computing

Matrix multiplication pervades numerous computational domains, serving as a fundamental building block for complex algorithmic processes. Social network analysis exemplifies this ubiquity, where user relationships are encoded in adjacency matrices $A \in \mathbb{R}^{n \times n}$. In such representations, element a_{ij} quantifies the relationship strength between users i and j, with non-zero entries indicating active connections.

These network structures facilitate advanced analytical tasks, including community detection and anomaly identification. Community detection algorithms typically formulate the problem as a matrix optimization challenge, requiring efficient multiplication routines for practical implementation. The computational demands of such operations scale dramatically with network size, necessitating careful consideration of implementation choices.

1.2 Computational Complexity Challenges

The prevalence of matrix operations extends beyond social networks to encompass machine learning, deep learning, natural language processing, computer vision, and bioinformatics. In each domain, the ability to efficiently multiply large matrices directly impacts algorithmic feasibility and practical applicability.

Given the $O(n^3)$ complexity of standard matrix multiplication, performance bottlenecks emerge rapidly as matrix dimensions increase. This scalability challenge motivates our comparative analysis of programming language performance characteristics, focusing on identifying optimal implementation strategies for large-scale computational scenarios.

2 Mathematical Foundation and Algorithm Design

2.1 Matrix Multiplication Definition

Consider two matrices $A \in \mathbb{M}_{n,p}(\mathbb{R})$ and $B \in \mathbb{M}_{p,m}(\mathbb{R})$. Their product $C = A \cdot B \in \mathbb{M}_{n,m}(\mathbb{R})$ is defined such that each element c_{ij} satisfies:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} \cdot b_{kj} \tag{1}$$

This formulation requires a triple-nested iterative structure: the outer loops traverse the result matrix dimensions, while the innermost loop accumulates the dot product for each matrix element.

2.2 Square Matrix Specialization

For computational simplicity and practical relevance, we restrict our analysis to square matrices where $A, B \in \mathbb{M}_n(\mathbb{R})$, yielding $C \in \mathbb{M}_n(\mathbb{R})$. Each result element becomes:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{2}$$

2.3 Algorithmic Implementation

The standard matrix multiplication algorithm employs three nested loops to compute all matrix elements systematically:

Algorithm 1 Standard Matrix Multiplication

Require: Matrices $A, B \in \mathbb{M}_n(\mathbb{R})$
Ensure: Matrix $C \in \mathbb{M}_n(\mathbb{R})$ where $C = A \cdot B$

```
for i = 1 to n do
    for j = 1 to n do
        C[i][j] ← 0
        for k = 1 to n do
            C[i][j] ← C[i][j] + A[i][k] · B[k][j]
        end for
    end for
end for
```

This algorithm maintains $O(n^3)$ time complexity and $O(n^2)$ space complexity, providing a consistent baseline for cross-language performance evaluation.

3 Implementation Strategy

3.1 Language-Agnostic Design

To ensure fair comparison across programming languages, we maintain identical algorithmic structure in all implementations. The core computation follows this pattern, illustrated in Python syntax:

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

3.2 Cross-Language Consistency

Our comparative study encompasses multiple programming languages including Python, Java, and C. Each implementation preserves the fundamental algorithmic structure while utilizing language-specific syntax and optimization patterns. This approach isolates language-specific performance characteristics from algorithmic variations.

3.3 Alternative Algorithms

While advanced algorithms such as Strassen's method or AlphaTensor's innovations offer superior asymptotic complexity for large matrices, our focus remains on the standard algorithm. This constraint enables direct language comparison without confounding algorithmic sophistication with implementation efficiency.

4 Experimental Methodology

4.1 Hardware Specifications

All performance benchmarks were conducted on a standardized testing platform to ensure consistent and reproducible results. The experimental hardware configuration consists of:

Component	Specification
Processor	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Graphics	Intel(R) Iris(R) Plus Graphics (128 MB)
Memory Speed	2667 MHz
Available RAM	20.0 GB

Table 1: Testing Platform Hardware Configuration

The benchmarking carried out while the machine was not performing any other tasks and remained connected to a power supply throughout the entire process.

5 Performance Comparison Between Languages

This section presents a performance comparison of the matrix multiplication algorithm implemented in Python, Java, and C++. Execution time is the key metric, measured across increasing matrix sizes to assess how each language scales under computational pressure. These benchmarks are intended to provide insights into the suitability of each language for data-intensive operations.

To ensure a fair and informative comparison, we measured the execution time for square matrices of sizes ranging from 2^1 to 2^{11} (i.e., up to order 2048). This range is sufficient to highlight the differences in performance, particularly when considering practical applications in Big Data and scientific computing, where such operations are common.

5.1 Execution Times

Figure 1 illustrates the execution times observed for each language. As expected, Python performs significantly worse compared to Java and C++, due to

its interpreted nature and lack of low-level memory control.

Surprisingly, the gap between Java and C++ is relatively small. Java's Just-In-Time (JIT) compilation and effective memory management allow it to closely match — and occasionally outperform — native C++ in certain scenarios. Java also benefits from mature multithreading support, enabling better parallelization on multi-core systems.

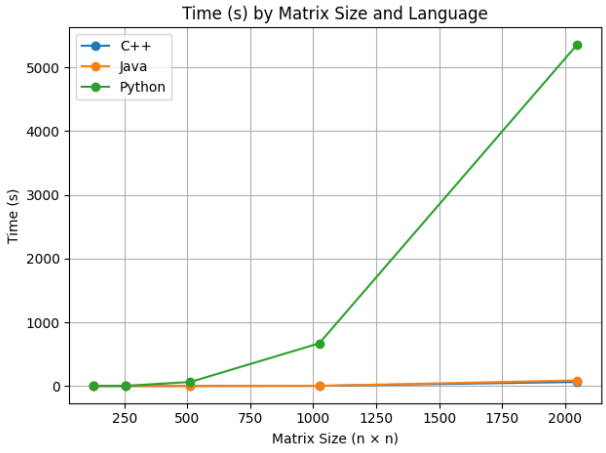

Figure 1: Execution time by matrix size and language (C++, Java, Python)

This behavior demonstrates the importance of understanding how languages interact with hardware and compilers. High-level language optimizations can sometimes bridge the expected performance gap, particularly when code is well-optimized and the runtime system is sophisticated.

5.2 Memory Usage

In addition to execution time, memory consumption is a key factor when evaluating the performance of programming languages, especially in resource-constrained systems or when processing large datasets.

Figure 2 illustrates the memory usage observed for matrix multiplication across increasing matrix sizes. As expected, all three languages exhibit a growing memory footprint as matrix size increases. However, notable differences arise in the growth rate and absolute usage.

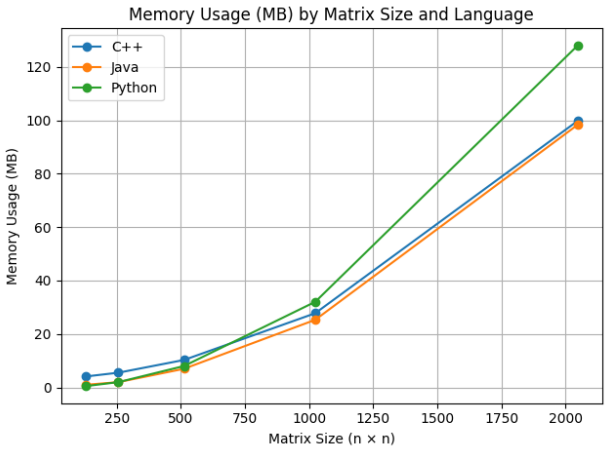

Figure 2: Memory usage by matrix size and language (C++, Java, Python)

Python consumes the most memory overall, especially for larger matrices, which aligns with its dynamic typing and object-heavy implementation. Java, interestingly, demonstrates slightly lower memory usage than C++ at larger scales. This could be attributed to efficient memory management by the Java Virtual Machine (JVM), which may outperform manual memory handling in C++ in certain scenarios.

C++ still performs competitively and maintains a predictable memory profile, but the slight overhead might stem from internal data structures or less aggressive memory optimization compared to the JVM.

Overall, while Python clearly lags in memory efficiency, the close performance between Java and C++ challenges common assumptions and highlights the optimizations modern virtual machines can achieve.

5.3 CPU Usage

CPU usage was also measured to explore how efficiently each language utilizes processing resources during matrix multiplication. The results are shown in Figure 3.

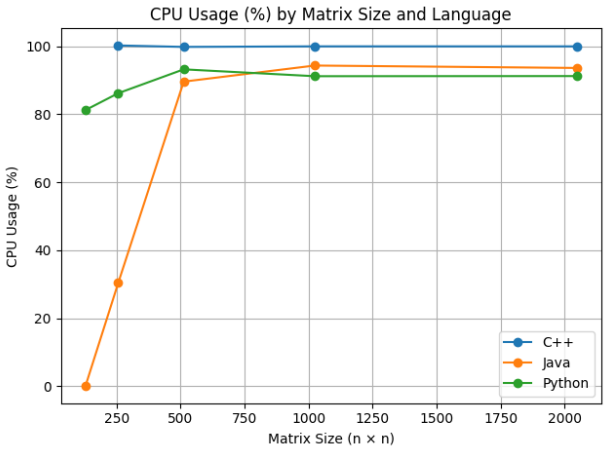

Figure 3: CPU usage by matrix size and language (C++, Java, Python)

C++ consistently reached 100% CPU usage across all matrix sizes, suggesting full utilization of a single core. Python's CPU usage stabilized around 90–93%, slightly lower likely due to interpreter overhead and the presence of the Global Interpreter Lock (GIL), which limits concurrent execution of threads. Java started with low CPU usage on smaller matrices but quickly rose above 90% as matrix size increased, likely due to Just-In-Time (JIT) compilation optimizing performance during runtime.

While all three languages eventually made near-maximal use of the CPU, these results highlight some of the runtime behavior differences, especially in warm-up or initialization phases.

6 Conclusion

This study evaluated the performance of matrix multiplication across three widely-used programming languages: C++, Java, and Python. Metrics considered included execution time, memory usage, and CPU utilization, with tests run on square matrices of increasing order.

The results reveal clear differences in performance:

- **Python** exhibited the worst performance overall. While it had the shortest code and highest-level syntax, it consistently showed the slowest execution times (e.g., over 4800 seconds at size 2048), the highest memory usage, and lower CPU saturation, especially for smaller matrices. This is likely due to the interpreted nature of Python and its Global Interpreter Lock (GIL).
- **C++** delivered extremely fast execution and full CPU utilization across all matrix sizes. However, it used slightly more memory than Java at larger scales and requires manual memory management, which introduces development complexity.
- **Java** achieved a strong balance across all metrics. It maintained low execution times (e.g., 1.9 seconds at size 1024), low memory usage, and high CPU utilization, approaching C++ in raw speed while retaining higher developer productivity through automatic memory management and runtime optimizations via the Java Virtual Machine (JVM).

Given this analysis, **Java stands out as the most balanced and efficient choice** for implementing matrix multiplication in environments where both performance and maintainability are important.

Relative Speedup Comparison

To quantify the differences in execution time, we can compute the speedup of Java relative to Python for large matrices. At size 1024:

$$\text{Speedup}_{\text{Java vs. Python}} = \frac{611.0}{1.9} \approx 321\times$$

This dramatic speedup highlights the limitations of Python for high-performance numerical tasks, and underscores the benefits of using a compiled, optimized language like Java or C++.

For even larger matrices (e.g., 2048), Java still shows a $\sim 59\times$ speedup over Python. Therefore, Java is highly recommended in performance-critical settings where dynamic typing or interpreter overhead would become a bottleneck.