# Language Benchmark of Matrix Multiplication

Casimiro Torres Kimberly

October 20, 2024

[1]Faculty of Computer Science,
Las Palmas de Gran Canaria University, Spain

**ABSTRACT**

*This project aims to conduct a comparative analysis of the performance of matrix multiplication in three widely-used programming languages: Python, Java, and C++. Matrix multiplication is a key operation in multiple high-performance applications, especially in the context of Big Data and Artificial Intelligence, where the efficient handling of large volumes of data and operations is crucial. In this analysis, the classical matrix multiplication algorithm, which has $O(n^3)$ complexity, is implemented, and essential metrics such as execution time, memory usage, and CPU utilization are evaluated.*

*The study is conducted in a controlled environment using high-precision benchmarking tools, allowing for accurate quantitative results for each language. The results of the analysis provide a clear view of each language's strengths and weaknesses, offering developers and scientists relevant data to select the most appropriate technology for computationally intensive operations, such as those found in Big Data and machine learning scenarios.*

**KEYWORDS:** Matrix multiplication, Benchmarking, Python, Java, C++, Big Data, Execution Time, Memory Usage, CPU Usage

## 1 Introduction

In modern computing, matrix multiplication plays a fundamental role in a variety of disciplines, ranging from artificial intelligence and machine learning to numerical simulation and bioinformatics. In particular, with the explosion of Big Data, where the ability to manage large volumes of information in real-time is crucial, the efficiency of operations such as matrix multiplication becomes central. As the size of matrices increases, the necessary resources, such as memory and execution time, grow exponentially, making this operation a significant computational challenge.

Over time, several approaches have emerged to improve the performance of these operations. Advanced algorithms, such as Strassen's and Coppersmith-Winograd, have proven effective in reducing the number of operations required for matrix multiplication, thereby improving performance. Likewise, optimized libraries, such as BLAS (Basic Linear Algebra Subprograms) and Eigen, have taken advantage of parallelization and vectorization techniques, offering more efficient use of hardware resources. These advances have been particularly beneficial for low-level languages like C and C++, where control over memory and resource management is granular.

However, the growth of high-level languages such as Python and Java, driven by their accessibility and extensive library ecosystems, has generated increasing interest in evaluating their performance in computationally expensive operations. Python, with libraries such as `numpy`, has been adopted in data science and machine learning applications, while Java, with its Virtual Machine (JVM), offers significant portability, although with certain limitations in terms of raw performance.

This project aims to compare the base implementations of matrix multiplication in these three languages: Python, Java, and C++, and analyze their performance in terms of execution time, memory usage, and CPU utilization. By doing so, it aims to highlight the fundamental differences in how each language handles complex operations, which is particularly relevant in Big Data applications, where scalability and computational efficiency are essential.

## 2 Problem Statement

In today's technological landscape, characterized by the rise of Big Data and the growing need for processing in artificial intelligence and machine learning applications, optimizing the performance of matrix operations has become a key challenge. These operations are used in training machine learning models, simulations, and analyzing large volumes of data, where scalability and efficient processing can be crucial to an application's success.

The problem to be addressed in this project is to conduct a detailed comparative evaluation of the performance of matrix multiplication in three programming languages: Python, Java, and C++. Each of these languages has characteristics that can significantly affect how computationally intensive operations are handled. For example, while C++ offers low-level optimization and absolute control over memory, Python and Java are recognized for their ease of use, flexibility, and ability to integrate specialized libraries, though with some disadvantages in terms of speed and resource usage.

The need to improve performance in large-scale operations, such as those found in Big Data environments, raises several questions: How does matrix size affect the efficiency of each language? What differences arise in memory usage and CPU utilization? To what extent do the internal optimizations of each language handle these workloads without compromising the scalability of solutions? This analysis aims to address these questions, providing a quantitative basis to help professionals make informed decisions about which language to use in scenarios involving massive matrix operations.

The goal is to provide a detailed comparison that identifies not only the advantages and disadvantages of each language but also their potential limitations when facing complex large-scale data management problems. With this evaluation, the aim is to provide valuable insights for selecting the appropriate technology in contexts where performance is critical for optimizing and scaling data-intensive processing systems.

## 3 Methodology

This section provides a detailed description of the process carried out to implement, execute, and evaluate the different implementations of matrix multiplication in Python, Java, and C++. The execution environments, evaluated metrics, and tools used for performance analysis are examined, all under a rigorous and systematic approach.

### 3.1 Hardware and Software Environment

The experiments were conducted using consistent hardware to ensure comparability between the different programming languages. The equipment used is powered by a 13th generation Intel Core i7-13700H processor, with a base frequency of 2.40 GHz, 16 GB of RAM, and a 64-bit Windows 11 Home operating system. This configuration ensures adequate processing capacity and data access speed, preventing possible bottlenecks during the execution of the experiments.

### 3.2 Tools Used

For each of the languages used in the experiments (Python, Java, and C++), specific tools were selected to optimize execution and facilitate performance analysis. The tools used in each environment are detailed below.

#### 3.2.1 Python

For Python, the integrated development environment (IDE) PyCharm was used, which is widely recognized for its powerful debugging features and its native integration with virtual environments. PyCharm makes it easy to create isolated environments that allow efficient management of the dependencies and modules needed to execute complex operations, such as matrix multiplication. Additionally, PyCharm offers advanced code analysis and performance optimization tools, which allowed for detailed monitoring of resource usage during the execution of the experiment. This combination of features makes PyCharm an ideal tool for projects requiring intensive calculations in Python.

#### 3.2.2 Java

For Java implementation, IntelliJ IDEA was used, one of the most comprehensive and robust development environments for Java. IntelliJ provides advanced integration with the Java Virtual Machine (JVM), facilitating debugging and code performance analysis. Additionally, IntelliJ has integrated tools for monitoring memory and CPU usage during the execution of applications, allowing real-time tracking of resources used by the program. This was essential to ensure that the Java experiments were executed efficiently and that the data collected was accurate and comparable with the other languages.

### 3.2.3 C++

In the case of C++, the implementation was carried out on a virtual machine configured with the Linux Fedora operating system. This virtual environment provided a controlled and isolated setting, ideal for executing experiments without external interference, which allowed for consistency and precision in performance measurements. For C++ code development and analysis, advanced profiling tools were used, which allow exhaustive monitoring of memory and CPU usage, as well as detecting possible memory leaks during program execution. Thanks to these tools, a detailed view of the program's behavior in terms of efficiency was obtained, ensuring that the results were comparable and optimized in relation to the other languages analyzed. The combination of the Fedora virtual machine and analysis tools ensured precise and thorough evaluation of C++ performance in the experiments.

## 3.3 Matrix Multiplication Implementation

The matrix multiplication implementation in each language was divided into two phases: one without optimization, which used the classical $O(n^3)$ complexity algorithm, and another with optimization, where specific libraries and tools were employed to improve performance. Below is a detailed description of how the operations were implemented in each language and the applied optimizations.

### 3.3.1 Python

The matrix multiplication implementation in Python was done in two phases: one without optimization, using a manual implementation with nested loops, and an optimized phase using the `numpy` library. Both phases are detailed below.

**Unoptimized Code:** In the unoptimized phase, matrix multiplication was manually implemented using three nested loops to traverse the matrices and calculate the row-by-column product. This basic approach allows measuring the native performance of Python without using any external libraries.

- The function `multiply_matrices(a, b)` takes two input matrices and multiplies them using three nested loops: the first loop iterates over the rows of matrix $A$, the second loop iterates over the columns of matrix $B$, and the third loop sums the corresponding products between the rows and columns.
- The code does not use optimized libraries but works with nested lists in Python to represent the matrices. This implementation is useful as a reference to compare performance with optimized versions.

**Optimized Code:** In the optimized phase, the `numpy` library was used, which is highly efficient in linear algebra operations and is written in C to optimize performance in Python.

- The function `multiply_matrices(a, b)` was replaced by `numpy.dot(a, b)`, an optimized matrix multiplication operation. `numpy.dot()` takes advantage of data parallelization and vectorization, which significantly reduces execution time.
- `numpy` is specifically designed for large-scale operations, allowing calculations to be performed more efficiently compared to manual implementation, especially in large matrices.

### 3.3.2 Java

In Java, the matrix multiplication implementation was divided into two phases: one without optimization and another optimized. The matrix multiplication algorithm was manually implemented in both phases, using nested loops to traverse the matrices, but with key differences in the optimized phase to improve performance.

**Unoptimized Code:**

- Manual implementation using three nested loops to iterate over the rows of matrix $A$ and the columns of matrix $B$, multiplying and summing the corresponding products.

- No external optimization libraries were used, providing a baseline to measure Java's native performance in matrix operations.

**Optimized Code:** In the optimized phase, several improvements were applied to reduce execution time and improve resource usage efficiency:

- Transposing matrix $B$: To improve memory access and cache locality, matrix $B$ was transposed before multiplication. This technique allows more efficient access to the columns of $B$ (now rows of the transposed matrix) during calculations.
- Benchmarking with JMH: Java Microbenchmark Harness (JMH) was used to accurately measure code performance. JMH was configured to perform multiple iterations, both in the "warmup" phase and in the measurement phase, ensuring stable results.
- Measuring memory and CPU usage: During the benchmarking execution, memory usage was recorded through the `memoryUsage` function and CPU time using `cpuTime`. These metrics allowed for precise measurement of the optimization's impact on system resources.

### 3.3.3 C++

In C++, the matrix multiplication implementation was carried out in two phases: one without optimization and another optimized. In the unoptimized phase, the matrix multiplication algorithm was manually implemented using three nested loops, while in the optimized phase, advanced techniques were applied to improve performance.

**Unoptimized Code:** In the unoptimized phase, the code was manually implemented with nested loops to calculate the row-by-column product of two matrices.

- The function `multiplyMatrices(A, B)` takes two input matrices and multiplies them using three nested loops: the first loop iterates over the rows of matrix $A$, the second iterates over the columns of matrix $B$, and the third sums the corresponding products.
- No external libraries or advanced optimization techniques were employed. The code was written using only standard C++ capabilities, and the matrices were represented using two-dimensional arrays.

**Optimized Code:** In the optimized phase, several techniques were implemented to improve the efficiency of matrix operations, including loop unrolling.

- Loop unrolling with an unrolling factor of 4 (`UNROLL_FACTOR = 4`) was used, allowing multiple operations to be performed within a single loop cycle, improving calculation efficiency and reducing execution time.
- Within the `multiplyMatrices(A, B)` function, the unrolling technique was applied in the inner loop, dividing the multiplication calculation into four partial sums (`sum0`, `sum1`, `sum2`, `sum3`) that are performed in parallel, improving data access efficiency.
- After the unrolling, the code processes the remaining elements if the matrix dimension is not divisible by the unrolling factor.

## 3.4 Benchmarking and Performance Metrics

To evaluate the performance of matrix multiplication implementations in Python, Java, and C++, various tools were employed to measure the following key metrics: execution time, memory usage, and CPU usage. Each language used tools suitable for its environment, and the tests were performed over 10 iterations to ensure the stability and consistency of the results.

**Execution Time:** Execution time is a critical metric to evaluate the efficiency of the implementations. The total time it took for each program to complete matrix multiplication for different sizes was measured:

- Python: The `time.time()` function was used to capture the start and end time of the operation. The elapsed time between these two points was recorded in milliseconds.
- Java: `System.nanoTime()` was used to measure the elapsed time between the start and end of the matrix multiplication. The times were calculated in nanoseconds and then converted to milliseconds.

- C++: The `std::chrono` library with high-resolution clocks (`std::chrono::high_resolution_clock`) was used to capture the start and end time of matrix multiplication, also in milliseconds.

**Memory Usage:** Memory usage is another important metric that reflects how efficient an implementation is in terms of resource management. The amount of memory used before and after matrix multiplication was measured:

- Python: The `memory_profiler` library was used, which through the `memory_usage()` function allowed memory usage to be recorded during the execution of the code.
- Java: The `Runtime` class was used to measure memory usage before and after matrix multiplication. The `memoryUsage()` method was called at both points to record memory usage in megabytes.
- C++: In C++, the `/proc/self/statm` file was accessed, which provides statistics on the process's memory usage. The values were recorded in kilobytes to measure the consumed memory.

**CPU Usage:** CPU usage measures how many processing resources are being used by the program. This allows evaluating whether the implementation efficiently leverages the capacity of multiple CPU cores.

- Python: The `psutil` library was used to measure the percentage of CPU used during the execution of the program. This measurement reflected CPU time versus real time (wall clock time).
- Java: To measure CPU usage, the `getCurrentThreadCpuTime()` method was used before and after the matrix multiplication operation. The CPU usage percentage was calculated by comparing CPU time to the total elapsed time.
- C++: In C++, the `getrusage()` function was employed to measure the CPU time used by the process. This function returned CPU usage in seconds and microseconds, which allowed calculating the CPU percentage used relative to real time.

**Iterations and Tests:** To ensure the accuracy and stability of the results, 10 iterations were performed for each matrix size in each language. This allowed obtaining representative averages of the performance metrics for the different matrix sizes.

## 4 Experiments

The experiments conducted in this study aimed to compare the performance of matrix multiplication in Python, Java, and C++. Three key metrics were analyzed: execution time, memory usage, and CPU usage, using matrices of different sizes (64x64, 128x128, 256x256, 512x512, and 1024x1024). For the optimized version, the 2048x2048 size was also included. A detailed analysis of the results obtained in each metric is presented below.

### 4.1 Execution Time

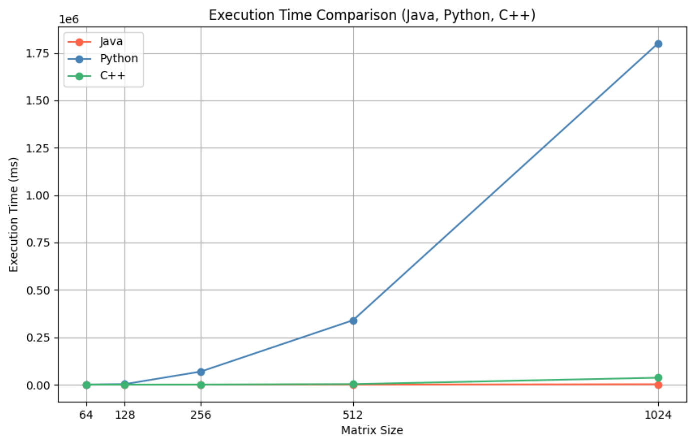

Figure 1: Comparison of execution time between Python, Java, and C++ for different matrix sizes.

Figure 1 shows the execution times in milliseconds for matrices of different sizes, which allows for the evaluation of Python, Java, and C++ performance.

**Results Analysis**

- Python (blue): Python shows the longest execution times, especially as matrix size increases. This is due to the interpreted nature of the language and the overhead introduced by its automatic resource management. Although Python is popular for its ease of use, this overhead significantly hampers its performance when handling complex and large-scale operations such as matrix multiplication.
- Java (orange): Java shows the best performance in terms of execution time. As matrices grow in size, Java continues to outperform both C++ and Python, making it the best option in this case. Its ability to efficiently manage resources allows it to excel, particularly in large-scale operations.
- C++ (green): C++ shows solid and consistent performance, standing out for its ability to manually manage memory and its highly optimized compiler code. Although its performance is good, in this scenario, C++ is outperformed by Java. However, C++ remains an efficient option, especially for larger matrices, where its low-level control over resources allows it to maintain reasonable execution times.

## 4.2 Memory Usage

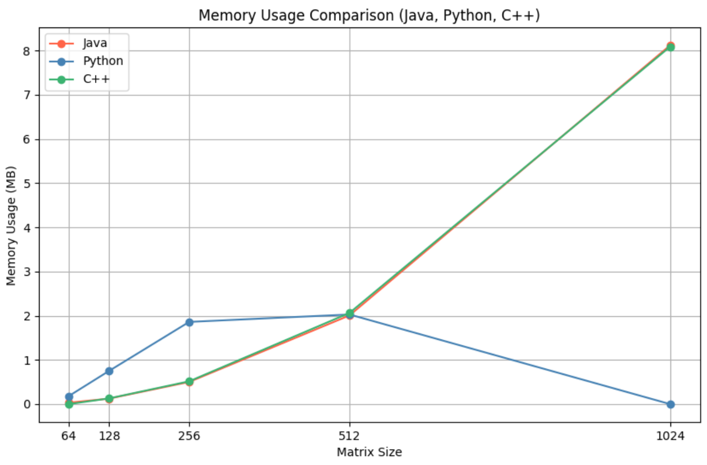

Figure 2: Comparison of memory usage between Python, Java, and C++ for different matrix sizes. The graph shows memory usage in megabytes, highlighting the differences between the languages.

Figure 2 shows the memory usage (in megabytes) during the execution of experiments in the three languages.

**Results Analysis**

- Python (blue): Python shows moderate memory consumption compared to the other languages. It still presents higher memory usage than Java, particularly for smaller matrices. However, memory consumption decreases significantly for larger matrices such as 1024x1024.
- Java (orange): Java, while efficient with smaller matrices, shows considerably higher memory usage as the size of the matrices increases. As operations become more intensive, Java's memory consumption increases significantly, surpassing that of Python for larger matrices.
- C++ (green): C++ also shows low memory usage for smaller matrices, but like Java, its memory consumption grows significantly for larger matrices. Although C++ allows for more granular control over memory allocation, in this case, the implementation has resulted in lower efficiency compared to Python when handling larger matrices.

### 4.3 CPU Usage

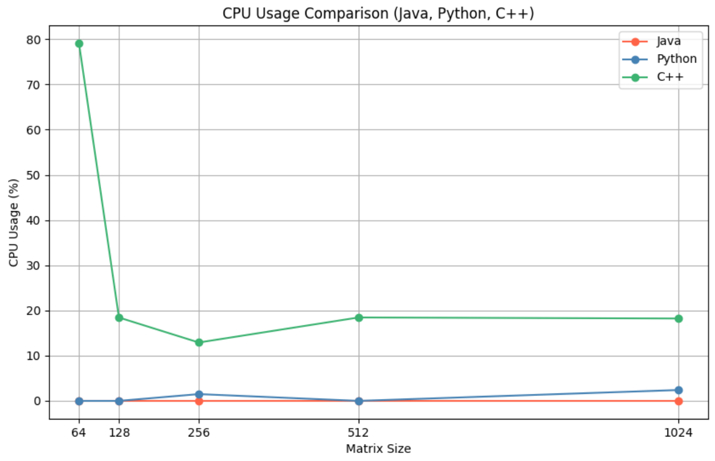

Figure 3: Comparison of CPU usage between Python, Java, and C++ for different matrix sizes.

Figure 3 shows the percentage of CPU usage during matrix multiplication execution in each language.

**Results Analysis**

- Python (blue): Python shows a more stable and slightly higher CPU usage than Java, although considerably lower than C++. CPU usage in Python does not vary significantly with matrix size, suggesting that CPU management in Python is not as well-adjusted to different operation sizes as it is in C++. Additionally, the lower CPU usage indicates that Python is not fully leveraging available resources, contributing to its longer execution times.
- Java (orange): Java maintains a low and consistent CPU usage across all matrix sizes. Compared to Python and C++, the CPU usage percentage is significantly lower, suggesting that it prioritizes resource stability and efficient management rather than maximizing CPU usage. Despite its lower CPU usage, Java manages to maintain shorter execution times than Python, indicating a balanced resource management approach that favors both efficiency and processing speed.
- C++ (green): C++ shows high CPU usage for smaller matrices, especially the 64x64 size, where it reaches over 80%. This is due to the speed with which it completes operations, causing it to use a larger proportion of CPU during execution. However, as matrix size increases, CPU usage in C++ stabilizes around 15-20%, indicating more efficient resource management as complexity increases.

## 5 Conclusion

After conducting the experiments on the three key metrics—execution time, memory usage, and CPU usage—in Python, Java, and C++, several detailed conclusions can be drawn regarding the performance of each language in the context of matrix multiplication.

In terms of execution time, Java stood out as the most efficient of the three languages. Across the different matrix sizes, Java proved to be the fastest, with consistently low execution times. This indicates that its automatic resource management, although introducing some overhead, is well-optimized for handling intensive tasks such as matrix multiplication. C++ offered solid execution times, consistently outperforming Python, although it was surpassed by Java with larger matrices. Python, on the other hand, showed the worst results in terms of execution time, which is attributable to the interpreted nature of the language and the overhead of its memory management.

Regarding memory usage, the trend was somewhat more complex. Java and C++ showed low memory consumption for smaller matrices, but their memory usage increased drastically with larger matrices. This may be due to how both languages manage memory in more intensive tasks, where object management and manual resource handling affect performance. Surprisingly, Python showed moderate efficiency in memory usage with larger matrices, outperforming both Java and C++ in this aspect. However, for smaller matrices, Python had higher memory usage compared to Java.

Finally, in terms of CPU usage, C++ presented significantly higher usage with smaller matrices, reflecting its ability to complete operations quickly. As matrix size increased, CPU usage in C++ stabilized, demonstrating efficient resource management. Java showed relatively low and constant CPU usage, suggesting that it prioritizes efficient resource management at the expense of not maximizing CPU usage. Python, like Java, showed low CPU usage, but its higher execution time indicates that it does not fully leverage the available resources, especially in more complex operations.

## 6 Future Work

This project has allowed for a comparison of the performance of three programming languages (Python, Java, and C++) in matrix multiplication. Based on the results obtained, several interesting avenues for future research emerge. Below, some possible improvements and extensions of this work are detailed:

### 6.1 Use of More Efficient Algorithms

This project used the classic matrix multiplication algorithm. As future work, it would be interesting to explore other algorithms that could reduce execution time. More complex algorithms that offer significant advantages when working with large matrices. Implementing these algorithms and comparing them with the results obtained in this project could provide a more complete view of performance in matrix operations.

### 6.2 Application of Optimizations in All Languages

In this project, simple optimizations were applied to each language. Future work could focus on applying more advanced optimizations. This would allow for a more precise measurement of performance differences when each language is optimized to its fullest potential. For example, the impact of using more optimized libraries such as `Numba` in Python or additional optimization tools in Java could be studied.

### 6.3 Evaluation on Different Operating Systems

This project was conducted using Linux for C++ and Windows for the implementations in Java and Python. As future work, it would be interesting to repeat the tests on other operating systems or additional Linux distributions. Additionally, the use of virtual machines to run Python and Java on Linux could be considered, which would allow for a more direct comparison between the three languages in the same operating environment. This evaluation would offer a more complete view of how each language behaves under different conditions and how the operating system affects the performance of the implementations.

### 6.4 Analysis of Real-World Applications

Finally, as future work, it would be interesting to apply the implementations developed in this project to real-world applications that use matrix multiplication, such as image processing or data analysis. This would allow us to see the real impact of optimizations and performance in contexts closer to industry, rather than isolated matrix multiplication tests.

## GitHub Link

All the code implemented for this project can be found on my GitHub repository at the following link: GitHub Repository.