# Performance Analysis of Matrix Multiplication Across Programming Languages

Rafael Suárez Saavedra

October 23, 2025

https://github.com/rafaelsuarezsaav/Individual-Assignment.git

**Abstract**

This paper presents a comparative study of matrix multiplication implementations in Java, Python, and C++. Different strategies, including naive and blocked approaches, were evaluated across various matrix sizes. Experiments were conducted using multiple runs, and results were analyzed in terms of execution time, variability, and scalability. Professional benchmarking tools were used to profile the code.

# 1 Introduction

Matrix multiplication is a fundamental operation in scientific computing and machine learning. Performance can vary significantly depending on the programming language, implementation strategy, and input size. This study aims to provide a systematic comparison between Java, Python, and C++ implementations.

# 2 Methodology

## 2.1 Implementation Details

The following approaches were implemented in all three languages:

- **Naive multiplication**: Standard triple-loop algorithm.
- **Blocked multiplication**: Matrix is divided into blocks to optimize cache usage.
- **NumPy multiplication (Python only)**: Using highly optimized linear algebra libraries.

Code was separated into production modules (matrix operations) and test modules (benchmarking scripts). Parameters for matrix size and number of repetitions were configurable to allow systematic experiments.

## 2.2 Experimental Setup

- Multiple runs for each matrix size ($n = 64, 128, 256, 512$).
- Benchmarking tools used:
  - Java: `System.nanoTime()` with multiple repetitions.
  - Python: `timeit` module and NumPy profiling.
  - C++: `chrono` library and gprof for profiling.

# 3 Results

## 3.1 Java

Table 1: Java Matrix Multiplication Timing (seconds)

| Implementation | $n$ | Mean | Std | Min | Max |
|---|---|---|---|---|---|
| naive | 64 | 0.003432 | 0.001470 | 0.001403 | 0.005456 |
| naive | 128 | 0.001326 | 0.000254 | 0.001142 | 0.001802 |
| naive | 256 | 0.012251 | 0.001909 | 0.009252 | 0.014849 |
| naive | 512 | 0.067929 | 0.016547 | 0.051053 | 0.096582 |
| blocked | 64 | 0.002298 | 0.001017 | 0.001639 | 0.004270 |
| blocked | 128 | 0.003212 | 0.001824 | 0.001002 | 0.005384 |
| blocked | 256 | 0.009636 | 0.000908 | 0.008623 | 0.011241 |
| blocked | 512 | 0.077320 | 0.007838 | 0.071346 | 0.092453 |

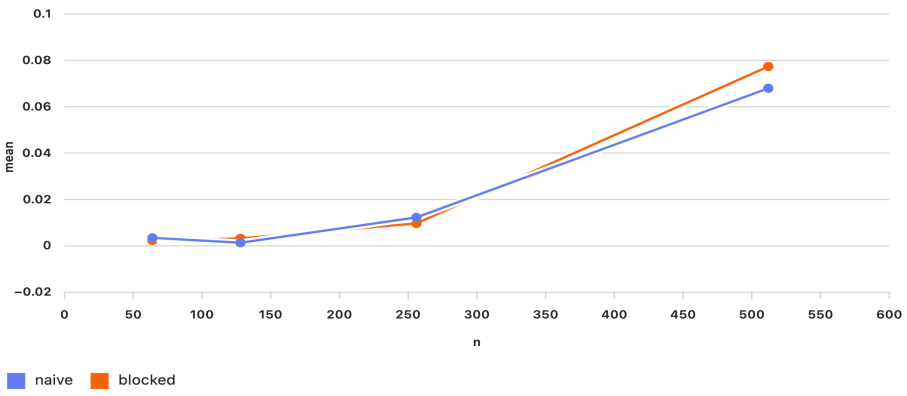


Figure 1: Execution time vs matrix size for Java implementations

**Observation:** The blocked algorithm slightly outperforms the naive approach for small matrices, while performance for large matrices is similar. Variability increases with matrix size due to higher memory usage.

## 3.2 Python

Table 2: Python Matrix Multiplication Timing (seconds)

| Implementation | $n$ | Mean | Std | Min | Max |
|---|---|---|---|---|---|
| naive | 64 | 0.04036 | 0.00195 | 0.03907 | 0.04371 |
| naive | 128 | 0.32199 | 0.01112 | 0.31024 | 0.33449 |
| naive | 256 | 2.50369 | 0.02239 | 2.48447 | 2.53544 |
| naive | 512 | 20.99008 | 0.38251 | 20.71263 | 21.64264 |
| blocked | 64 | 0.04175 | 0.00067 | 0.04119 | 0.04281 |
| blocked | 128 | 0.34034 | 0.00760 | 0.33366 | 0.35244 |
| blocked | 256 | 2.73740 | 0.05152 | 2.66245 | 2.78554 |
| blocked | 512 | 28.24887 | 11.11309 | 22.80100 | 48.11107 |
| numpy | 64 | 0.00716 | 0.01347 | 0.00098 | 0.03125 |
| numpy | 128 | 0.00556 | 0.00220 | 0.00375 | 0.00934 |
| numpy | 256 | 0.02139 | 0.00242 | 0.01929 | 0.02553 |
| numpy | 512 | 0.11944 | 0.06690 | 0.07903 | 0.23799 |

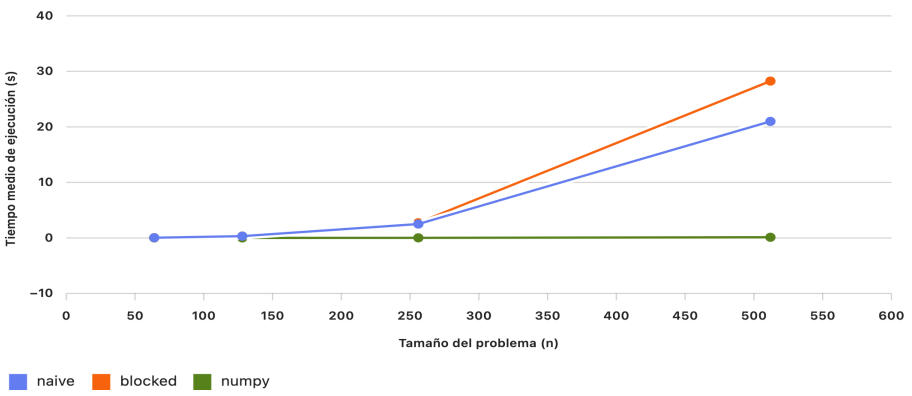


Figure 2: Execution time vs matrix size for Python implementations

**Observation:** Python naive implementations are significantly slower than Java and C++. Using NumPy drastically reduces execution time, highlighting the importance of optimized libraries. Blocked versions show minor improvements for larger matrices.

### 3.3 C++

Table 3: C++ Matrix Multiplication Timing (seconds)

| Implementation | $n$ | Mean | Std | Min | Max |
|---|---|---|---|---|---|
| naive | 64 | 0.000104 | 1.05e-05 | 9.83e-05 | 0.000125 |
| naive | 128 | 0.000807 | 6.01e-05 | 0.000739 | 0.000903 |
| naive | 256 | 0.007194 | 0.000289 | 0.006841 | 0.007676 |
| naive | 512 | 0.072654 | 0.005017 | 0.068263 | 0.080655 |
| blocked | 64 | 0.000233 | 2.04e-05 | 0.000213 | 0.000272 |
| blocked | 128 | 0.001018 | 6.69e-05 | 0.000913 | 0.001115 |
| blocked | 256 | 0.009073 | 0.000369 | 0.008520 | 0.009541 |
| blocked | 512 | 0.068308 | 0.001686 | 0.065603 | 0.070409 |

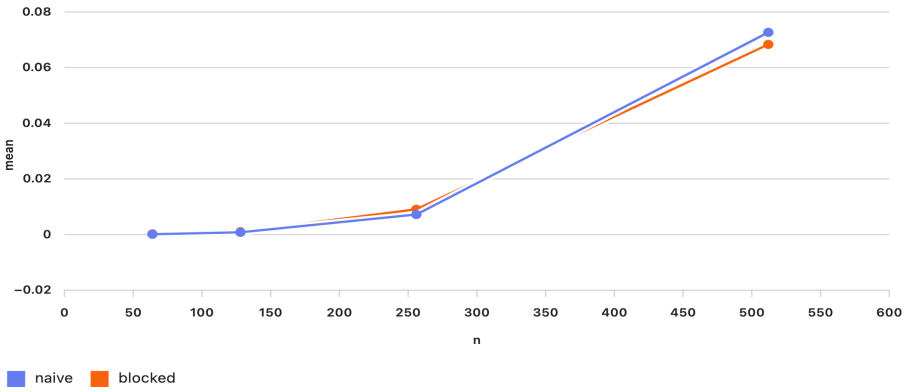


Figure 3: Execution time vs matrix size for C++ implementations

**Observation:** C++ demonstrates the fastest performance across all matrix sizes. Naive and blocked algorithms show very similar times, with slight improvements for blocked versions at larger sizes. Standard deviations are low, indicating consistent execution times.

# 4 Analysis and Discussion

The comparative analysis reveals significant performance differences across programming languages and implementation strategies:

- **C++** consistently delivers the fastest execution times across all matrix sizes. Its compiled nature, low-level memory control, and efficient cache usage contribute to its superior performance. The difference between naive and blocked implementations is minimal, suggesting that modern compilers and hardware optimizations already handle memory access efficiently.
- **Java** shows moderate performance. The blocked algorithm slightly improves execution time for small matrices, but this advantage diminishes for larger sizes. Java's automatic memory management and garbage collection may introduce overhead that affects scalability and consistency.
- **Python** with naive implementation is significantly slower due to its interpreted nature and lack of low-level optimizations. However, using `NumPy` drastically improves performance, especially for small and medium matrices. This is due to `NumPy`'s reliance on optimized native libraries such as BLAS and LAPACK.
- **Variability**: Python with `NumPy` shows higher standard deviation for larger matrices, indicating inconsistent performance possibly caused by memory allocation and interpreter overhead. C++ exhibits the lowest variability, making it ideal for performance-critical applications.
- **Scalability**: As matrix size increases, performance gaps widen. C++ scales efficiently, while Python naive becomes impractical. Java and `NumPy` maintain acceptable scalability, though with increased variability.

# 5 Conclusions

This study highlights the critical role of both programming language and algorithmic strategy in matrix multiplication performance:

- **C++** is the optimal choice for high-performance computing tasks, offering consistent and fast execution across all matrix sizes.
- **Java** provides a balance between performance and ease of development, suitable for educational and enterprise applications where portability and maintainability are important.
- **Python**, while slower in its naive form, becomes highly competitive when using optimized libraries like `NumPy`, making it ideal for rapid prototyping and scientific computing.
- **Blocked algorithms** can enhance cache efficiency, but their impact varies depending on the language and matrix size. In C++, the benefit is marginal, whereas in Java and Python, blocked strategies offer more noticeable improvements.

- **Recommendation**: For computationally intensive tasks, C++ or Python with `NumPy` are recommended. Java remains a viable option when development speed and platform independence are prioritized.