Comparative Performance Analysis of Matrix Multiplication in Python, Java, and C++

Enrique Reina Hernández

October 23, 2025

Abstract

This paper presents a comparative performance study of matrix multiplication implemented in Python, Java, and C++. We evaluate computation time, scalability, and profiling results for varying matrix sizes. Each implementation follows software engineering best practices, including separation of production and testing code, parametrization of matrix operations, and multiple benchmark runs to ensure consistency. The study aims to highlight trade-offs between performance, ease of implementation, and maintainability.

1 Introduction

Matrix multiplication is a fundamental operation in numerical computing, with applications spanning linear algebra, machine learning, and computer graphics. As computational demands increase, choosing the right programming language for performance-critical tasks becomes essential.

This work compares the performance of matrix multiplication across three programming languages—Python, Java, and C++—to evaluate differences in execution time, scalability, and memory efficiency.

The author utilized AI assistance (ChatGPT, OpenAI) for guidance in drafting parts of this paper and for support in implementing the C++ version of the matrix multiplication, as this language was new to the author. All experiments, benchmarking, and final code verification were performed independently.

2 Methodology

Each implementation was designed following principles of software modularity:

- **Production code** contains the core matrix multiplication logic.
- **Testing code** manages benchmarking, input/output operations, and result validation.
- **Parameterization** allows dynamic configuration of matrix dimensions (n) and number of experimental runs.

Benchmarking tools and methods:

- **Python**: `time.perf_counter()` and `pytest-benchmark`
- **Java**: `System.nanoTime()` and Java Microbenchmark Harness (JMH)
- **C++**: `std::chrono`

All measurements were taken as the average of multiple runs to minimize the effects of system load and caching.

2.1 Python Implementation

The original Python script combined matrix initialization, multiplication, and timing within a single file. The revised version adopts a modular design with two components: **MatrixMultiplier.py**, responsible for matrix generation and multiplication, and **MatrixBenchmark.py**, which performs benchmarking and memory measurement.

An indexing bug (`B[k][k]` → `B[k][j]`) was corrected, and timing precision improved using `time.perf_counter()`. A new function, `get_memory_usage_kb()`, based on `psutil`, was added to track resident memory usage.

The program now performs multiple runs (for matrices of 128–1024) and reports average runtime and memory usage as:

Python: NxN average = X.XXXXXX s, memory = XXXX KB

2.2 Java Implementation

The original monolithic Java class was refactored into **MatrixMultiplier**, handling matrix generation and multiplication, and **MatrixBenchmark**, responsible for timing and memory profiling.

Timing was upgraded from `System.currentTimeMillis()` to `System.nanoTime()`, and memory tracking uses the `Runtime` API (`totalMemory() - freeMemory()`). Garbage collection (`System.gc()`) is invoked between runs to ensure consistency.

Benchmarks cover matrix sizes from 128 to 1024, averaging five runs. Results are output as:

Java: NxN average = X.XXXXXX s, memory = XXXX KB

2.3 C++ Implementation

The original C code was modernized in C++, separating functionality into **MatrixMultiplier.cpp** (matrix logic) and **MatrixBenchmark.cpp** (timing and memory analysis).

Static arrays were replaced with `std::vector` containers, and an accumulation bug (`=` → `+=`) was fixed. Timing now uses `std::chrono::high_resolution_clock`, and memory is obtained from `/proc/self/statm` (RSS in KB).

Benchmarks are repeated across several matrix sizes, reporting average execution time and memory as:

C++: NxN average = X.XXXXXX s, memory = XXXX KB

3 Experimental Setup

All experiments were conducted on a GitHub Codespace with the following configuration:

- CPU: 2 vCPUs, Intel Xeon Platinum 8370C @ 2.80 GHz (virtualized)
- RAM: 7.8 GB
- Operating System: Ubuntu 24.04.2 LTS (64-bit)

Matrix sizes of 128×128, 256×256, 512×512 and 1024×1024 were tested. Each configuration was executed five times, and the results were averaged.

3.1 Execution Procedure

Each implementation was executed from the command line under identical benchmarking conditions to ensure fair performance comparison. The following commands were used to run the experiments for each programming language:

- **Python:**

```
python MatrixBenchmark.py
```

- **Java:**

```
javac *.java
java MatrixBenchmark
```

- **C++:**

```
g++ -std=c++17 -O2 -o MatrixBenchmark MatrixBenchmark.cpp
./MatrixBenchmark
```

All executions were performed in the same virtual environment described previously, ensuring identical hardware and operating system configurations. Before each run, background processes were minimized and system caches cleared to reduce measurement noise.

3.2 Parameter Configuration

In all three implementations, the matrix sizes and the number of benchmark runs were defined directly within the source code to ensure identical experimental conditions and avoid dependency on command-line arguments. This approach guarantees that each language executes the same workload under controlled parameters.

For all experiments, the benchmark iterated through four matrix sizes: `128×128`, `256×256`, `512×512`, and `1024×1024`. Each configuration was executed five times (`runs = 5`), and the average runtime and memory usage were recorded.

The following code excerpts illustrate how these parameters were defined in each language:

- **Python:**

```
if __name__ == "__main__":
    sizes = [128, 256, 512, 1024]
    runs = 5
    for n in sizes:
        avg_time, mem_kb = benchmark(n, runs)
        print(f"Python: {n}x{n} average = {avg_time:.6f} s, memory = {mem_kb:.2f} KB")
```

- **Java:**

```
public static void main(String[] args) {
    int[] sizes = {128, 256, 512, 1024};
    int runs = 5;
```

```
        for (int n : sizes) {
            long[] memUsage = new long[1];
            double avg = benchmark(n, runs, memUsage);
            System.out.printf("Java: %dx%d average = %.6f s, memory = %d KB%n",
                              n, n, avg, memUsage[0]);
        }
    }
```

- **C++:**

```
int main() {
    std::vector<int> sizes = {128, 256, 512, 1024};
    int runs = 5;

    for (int n : sizes) {
        size_t mem_kb;
        double avg_time = benchmark(n, runs, mem_kb);
        std::cout << "C++: " << n << "x" << n
                  << " average = " << avg_time << " s, "
                  << "memory = " << mem_kb << " KB" << std::endl;
    }
    return 0;
}
```

This consistent parameterization across implementations facilitates a fair comparison of execution times and memory usage while simplifying reproducibility of the experimental results.

4 Results

This section presents the performance comparison among Python, Java, and C++ for matrix multiplication tasks of different sizes. The results include both execution time and memory consumption, averaged over five independent runs to ensure consistency.

Table 1 summarizes the mean execution times (in seconds) for the three languages.

Matrix Size	**Python (s)**	**Java (s)**	**C++ (s)**
128×128	0.123	0.005	0.002
256×256	1.157	0.030	0.018
512×512	12.489	0.201	0.160
1024×1024	95.904	2.045	1.645

Table 1: Average execution times over five runs for different matrix sizes.

Table 2 summarizes the mean memory usage (in kilobytes) for the three programming languages.

Matrix Size	Python (kb)	Java (kb)	C++ (kb)
128×128	640	187	3712
256×256	1320	569	5248
512×512	1348	2267	9932
1024×1024	1664	8448	28 444

Table 2: Average memory usage over five runs for different matrix sizes.

5 Discussion

C++ achieved the highest performance due to its compiled nature, low-level memory management, and cache efficiency. Java's performance was moderately high, benefiting from the Just-In-Time (JIT) compiler, though slightly limited by the Java Virtual Machine (JVM) overhead. Python showed significantly slower results, attributed to interpreter overhead and dynamic typing.

However, Python excels in development speed and ecosystem support, making it ideal for prototyping and data analysis tasks. Java provides a balance between performance and maintainability, while C++ remains the preferred choice for high-performance computing.

Profiling results from `gprof` (C++), JMH (Java), and `cProfile` (Python) identified inner loop operations and cache misses as key bottlenecks.

6 Conclusion

This study shows that all three languages can perform matrix multiplication, albeit with significant differences in performance. C++ exhibits the lowest execution times, followed by Java, while Python is considerably slower.

Repository

All code and the full LaTeX document are available at: `https://github.com/ellupe/Individual_Assigment_Task1`

The repository includes:

- `/C++/` folder: Source code for the C++ implementation.
- `/java/` folder: Source code for the java implementation.
- `/python/` folder: Source code for the python implementation.
- `paper.pdf`: The final report.