Performance Analysis Report: Matrix Multiplication

October 19, 2025

1 Introduction and Objectives

This project compares the performance of a basic matrix multiplication algorithm implemented in three programming languages: Python, Java, and C++. The objective is to analyze the efficiency of each language considering execution time, memory usage, and computational overhead. The classic algorithm with $O(n^3)$ complexity was implemented and tested with matrices of increasing sizes (64×64, 128×128, 256×256, and 512×512).

2 Experimental Results

2.1 Performance Comparison Table

Table 1: Performance comparison across Python, Java, and C++

Size	Python Time (s)	Java Time (s)	C++ Time (s)	Python Mem (MB)	Java Mem (MB)	C++ Mem (MB)
64	0.4938	0.0037	0.0002	0.13	0.24	0.00
128	1.8428	0.0033	0.0017	0.51	0.44	0.12
256	23.6532	0.0525	0.0171	2.04	1.55	0.50
512	236.7505	0.7081	0.1487	8.06	6.22	2.00

2.2 Summary Statistics

Total Time Summary:

- Python: 262.74 seconds (4 min 23s)
- Java: 0.77 seconds
- C++: 0.17 seconds

Average Memory Summary:

- Python: 2.68 MB
- Java: 2.11 MB
- C++: 0.66 MB

3 Performance Analysis

The results show dramatic differences among the three languages. C++ proved to be the fastest with a total time of 0.17 seconds, establishing the performance baseline. Java achieved excellent performance with 0.77 seconds total, being approximately 4.5 times slower than C++, which represents very competitive performance considering the development advantages it offers. On the other hand, Python showed significantly inferior performance with 262.74 seconds total, being 1,545 times slower than C++ and 341 times slower than Java, evidencing that pure Python is not suitable for intensive numerical computation without optimized libraries like NumPy.

The time scaling respects the theoretical $O(n^3)$ complexity. When doubling the matrix size (for example, from 256 to 512), time increases approximately 8 times (2^3), as observed in all languages. For Python, going from 256×256 to 512×512 implied an increase from 23.65s to 236.75s (10x factor). In Java, the increase was from 0.0525s to 0.7081s (13.5x factor), and in C++ from 0.0171s to 0.1487s (8.7x factor). Small variations in these factors are due to cache effects, compiler optimizations, and operating system overhead.

An interesting result was Java's behavior with small matrices. The time to process the 128×128 matrix (0.0033s) was slightly less than for 64×64 (0.0037s), a phenomenon attributable to the "warm-up" effect of the JVM's JIT (Just-In-Time) compiler. During the first execution with 64×64, the JIT analyzes and optimizes the code, resulting in better performance in subsequent executions. This effect is characteristic of Java and demonstrates why it's important to include warm-up iterations in serious benchmarks.

Regarding memory usage, C++ was the most efficient with an average of 0.66 MB, using approximately one-third of the memory of Java (2.11 MB) and Python (2.68 MB). This is because C++ works directly with contiguous memory without the overhead of a virtual machine or interpreter. Java and Python showed similar consumption, reflecting the cost of their respective abstractions: the JVM in Java and the interpreter with its object management system in Python. For the largest matrix (512×512), Python reached 8.06 MB, Java 6.22 MB, and C++ only 2.00 MB, maintaining the proportion observed in smaller matrices.

4 Factors Affecting Performance

4.1 Python

Python's low performance is mainly explained by its interpreted nature and dynamic typing. Each arithmetic operation in Python requires multiple steps: checking data types, finding the appropriate method, executing interpreted Python code, and managing memory references. Python's list implementation, although flexible, stores references to objects rather than primitive values, adding additional indirections. The Global Interpreter Lock (GIL) also limits parallelism, although in this case it wasn't a factor since the implementation is single-threaded. However, solutions exist: NumPy can provide 50-100x acceleration by using C implementations, and tools like Numba or Cython can compile critical Python code.

4.2 Java

Java presents an interesting balance between performance and productivity. Its JIT compiler analyzes code during execution and generates optimized machine code, reaching speeds close to C++ after the warm-up period. The Garbage Collector automatically manages memory, eliminating common errors but introducing occasional pauses. The JVM performs bounds checking on arrays, guaranteeing safety but with a small performance cost. Despite these overheads, Java achieved respectable times, being only 4.5 times slower than C++, an acceptable difference for many applications considering its advantages in development, debugging, and portability.

4.3 C++

C++ obtained the best performance by compiling directly to native machine code without intermediaries. The compiler applies aggressive optimizations such as dead code elimination, loop unrolling, and automatic vectorization (with `-O2/-O3` flags). The use of `std::vector` provides automatic memory management while maintaining the performance of contiguous arrays, which favors cache locality. The absence of runtime checks and direct hardware access allow C++ to maximize system resources. The programmer has total control over memory management and can apply low-level optimizations such as cache blocking or SIMD instructions if necessary.

5 Implications and Recommendations

For real-world applications, language choice should be based on specific requirements. Python is ideal for rapid prototyping, exploratory data analysis, and scripts where development time is more important than execution time, but should always be used with NumPy/SciPy for serious numerical operations. Java is excellent for enterprise applications requiring good performance while maintaining ease of maintenance, being especially suitable for distributed systems and cross-platform applications. C++ is the preferred choice for high-performance scientific computing, video games, embedded systems, and any application where performance is critical, although it requires greater investment in development and debugging.

The limitations of this study include single-threaded implementation (doesn't leverage multi-core processors), use of a basic algorithm without algorithmic optimizations (alternatives like Strassen with $O(n^{2.807})$ exist), and tests limited to matrices up to 512×512. Future extensions could include parallel implementations with OpenMP (C++), parallel streams (Java), or multiprocessing (Python), the use of optimized libraries like Intel MKL or OpenBLAS, and tests with GPU acceleration via CUDA.

6 Conclusions

This study demonstrates that the choice of programming language has a dramatic impact on computational performance for intensive numerical operations. C++ clearly dominated with a time 1,545 times faster than Python and memory usage three times more efficient. Java proved to be an excellent intermediate option, offering 78% of maximum performance (compared to C++) while maintaining significant advantages in productivity and safety. Python, without optimized libraries, proved inadequate for this type of

intensive computation, confirming the critical importance of using NumPy in numerical applications.

The main lesson is that there is no universally superior language; each has its optimal niche. The recommended strategy is to prototype in Python for rapid development, use Java for production systems requiring a balance between performance and maintainability, and reserve C++ for critical components where every millisecond counts. In practice, many successful projects combine languages: Python for the interface and business logic, with computational cores in C++ called as native extensions, obtaining the best of both worlds.