Basic Matrix Multiplication in Different Languages

Jorge Cubero Toribio

October 2025

Abstract

This report presents an empirical comparison of the scalability and performance of the basic matrix multiplication algorithm ($O(N^3)$) implemented in three distinct programming environments: C++ (native), Java (JIT virtual machine), and Python (interpreted). The results clearly demonstrate the superior efficiency of C++ and Java over Python. The analysis focuses on calculating the empirical growth factor (T_{512}/T_{256}) to assess how each language handles increasing computational load, revealing significant performance penalties related to memory hierarchy effects.

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing and serves as an excellent micro-benchmark for comparing diverse execution environments. The standard algorithm, using the i-j-k loop ordering, has a time complexity of $O(N^3)$, where N is the dimension of the matrix.

The primary goal of this experiment is to empirically measure the execution times for matrix sizes $N = 256$ and $N = 512$. By calculating the ratio of times (T_{512}/T_{256}), we evaluate the scalability and efficiency of each language when the workload is doubled. It is hypothesized that compiled and JIT-optimized environments (C++ and Java) will exhibit vastly superior performance compared to the interpreted environment (Python).

1.1 Methodology and Code Structure

To ensure rigor, all code was implemented while adhering to professional software engineering practices:

- **Separation of Concerns:** Production code (the core matrix multiplication function) was separated from the benchmarking driver code.
- **Parametrization:** Matrix sizes (N) and the number of repetitions were easily configurable parameters.
- **Benchmarking:** Each test was run five (5) times to average the results and mitigate interference from OS scheduling and JIT compilation overhead.

2 Experimental Configuration

2.1 Execution Environments

- **C++:** Compiled using **GCC** (GNU Compiler Collection) in a stable WSL/Ubuntu environment.
- **Java:** Executed on the Java Virtual Machine (JVM) using Just-In-Time (JIT) compilation.
- **Python:** Executed using the standard CPython interpreter.

3 Empirical Results and Analysis

Table 1 presents the average execution times and the empirical growth factors derived from the successful benchmarks across the three languages.

Table 1: Average Execution Time and Empirical Growth Factor (T_{512}/T_{256})

Language	$N = 256$ **(s)**	$N = 512$ **(s)**	**Growth Factor**	**Complexity**
C++ (Native)	0.020225	0.270235	13.36	$O(N^3)$
Java (JIT)	0.092672	1.606393	17.33	$O(N^3)$
Python (Interp.)	16.364865	125.869204	7.69	$O(N^3)$

3.1 Interpretation of Base Performance

- **C++ (0.020 s):** C++ was the fastest, demonstrating the efficiency of its native execution model, which compiles directly to machine code without runtime overhead.
- **Java (0.093 s):** Java was approximately 4.6 times slower than C++. This difference is attributed to the overhead of the JVM and the latency incurred during the JIT warm-up phase; although its compiled nature keeps it highly efficient.
- **Python (16.365 s):** Python was approximately 800 times slower than C++. Its interpreted nature, coupled with dynamic type checking performed at every iteration, results in massive computational overhead, confirming the absolute necessity of using native extensions (like NumPy) for performance-critical Python tasks.

3.2 Analysis of $O(N^3)$ Scalability

The theoretical growth factor for any $O(N^3)$ algorithm when doubling the input size ($N = 256$ to $N = 512$) should be $2^3 = 8$.

- **Python (7.69):** Python's growth factor is very close to the theoretical 8, suggesting that its performance is limited by the constant overhead of the interpreter.
- **C++ (13.36) and Java (17.33):** Both high-performance languages show a growth factor significantly **higher than 8**. This anomaly is a classic result of the **Memory Hierarchy Effect**, or **Cache Misses**.
  - At $N = 256$, the data fit well within the CPU's fast L1/L2 cache.
  - At $N = 512$, the matrix size (2MB for three matrices) exceeds the cache capacity. The processor must constantly fetch data from the slower main RAM, introducing a massive time penalty that effectively pushes the observed performance curve beyond the theoretical $O(N^3)$ complexity. The higher factor for Java suggests that the JVM's memory management may exacerbate this effect.

4 Conclusion and Deliverables

This experiment successfully confirmed the performance hierarchy for matrix operations: **C++ > Java > Python**. Furthermore, the analysis of the empirical growth factor (13.36 for C++ and 17.33 for Java demonstrated that the real-world performance of high-efficiency code is dominated not by computational speed, but by the **memory hierarchy and cache utilization** of the system. The

project successfully met all requirements, including the use of C++, Java, and Python, with fully separated production and testing code.

All deliverables are located at: `https://github.com/JorsuCT/Big_Data/tree/main/IndividualAssignment`