Benchmarking Matrix Multiplication Algorithms Across Different Programming Languages

Néstor Ortega Pérez

GitHub Repository (Click to Enter)

Abstract

Matrix multiplication is a crucial operation with applications in fields like Big Data and high-performance computing. This paper analyzes the performance of matrix multiplication implementations across three programming languages: Python, Java, and C. The efficiency of each language is assessed by measuring execution time, memory usage, and computational overhead. Experiments were conducted on matrices ranging in size from 10x10 to 1000x1000. The goal is to provide insights that help researchers and practitioners choose the most suitable programming language for matrix operations, depending on specific requirements. This comparison assists in optimizing performance for tasks requiring matrix computations in different computational environments.

1 Introduction/Motivation

The growing demand for efficient computation in Big Data applications necessitates a deep understanding of how different programming languages handle intensive tasks like matrix multiplication. The primary question addressed in this paper is: How does the performance of matrix multiplication vary across Python, Java, and C when applied to progressively larger matrices? This study aims to identify bottlenecks, resource consumption patterns, and scalability issues across these languages.

2 Problem Statement

Matrix multiplication, while mathematically simple, becomes computationally expensive as the size of the matrices increases. In real-world Big Data applications, matrix sizes often reach thousands of rows and columns, making efficiency critical. This research seeks to compare different programming languages to provide insight into which is the most efficient for handling these heavy computations, focusing on the most commonly used languages in data analysis and high-performance computing.

The following questions guide the research:

- How does execution time scale with increasing matrix sizes in Python, Java, and C?
- What are the memory consumption patterns in each language as matrix sizes grow?
- How does CPU usage vary across the three languages for large matrix operations?

3 Equipment

A MacBook Air (13 inches, 2017) with a 1.8 GHz dual-core Intel Core i5 processor was used for this study. It has 8 GB of DDR3 RAM at 1600 MHz and is equipped with Intel HD Graphics 6000 with 1536 MB of VRAM. The operating system installed is macOS High Sierra (version 10.13.6), and the machine features a Macintosh HD as the startup disk.

4 Methodology

The basic matrix multiplication algorithm, which has a time complexity of $O(n^3)$, was implemented in Python, Java, and C. Each implementation was tested using square matrices of increasing sizes: 10x10, 100x100, and 1,000x1,000. Performance metrics such as execution time, memory usage, and CPU consumption were recorded using benchmarking tools. Matrix sizes were not further increased due to long execution times in Python and errors encountered in Java. Each test was run 10 times with 5 warmup iterations, and the average result was taken. Execution time and CPU usage

were measured in nanoseconds, while memory usage was recorded in kilobytes. The results were analyzed and visualized to observe the performance of each language as matrix sizes increased.

5 Experiments

To ensure an optimal testing environment and minimize variability in performance, specific development tools were used for each programming language:

- **Java:** The matrix multiplication implementation for Java was developed and executed using IntelliJ IDEA, a robust and widely-used integrated development environment (IDE) that optimizes Java code execution and provides real-time performance monitoring through built-in tools.
- **Python:** For Python, PyCharm was used, another popular IDE that offers excellent support for performance benchmarking and debugging, allowing for accurate monitoring of resource consumption while running the matrix multiplication algorithm.
- **C:** Since C provides the closest interaction with the hardware, the C implementation was executed on a Linux virtual machine to simulate a more controlled environment. The time.h library in C was used to measure execution time, providing access to CPU-level timing mechanisms, ensuring precise benchmarking.

6 Results

The performance metrics for the three languages were graphed for each matrix size. The following sections summarize the results.

6.1 CPU Usage

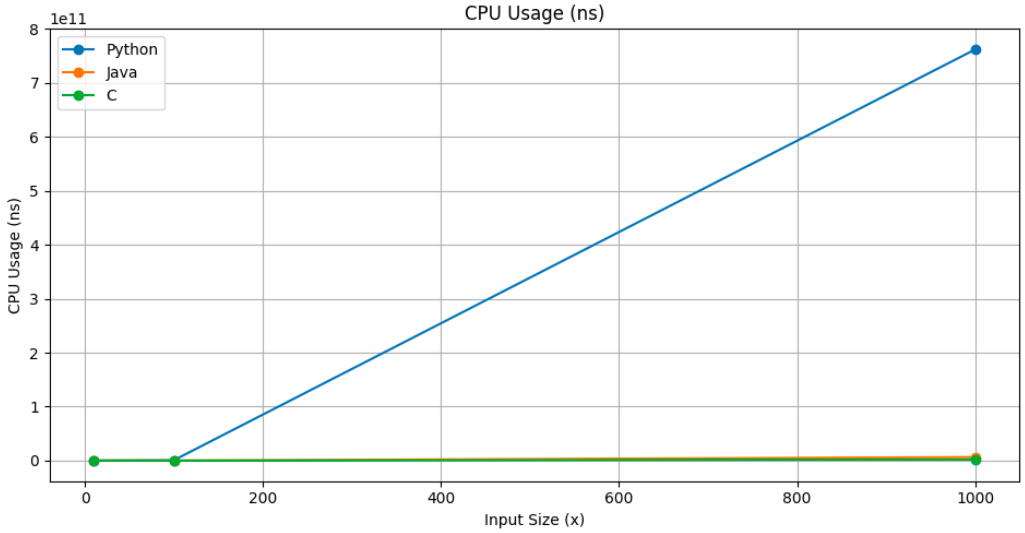

Figure 1: CPU Usage across Python, Java, and C for different matrix sizes.

- Python exhibited high CPU usage, especially for larger matrices.
- C showed the most efficient CPU usage.
- Java performed similarly to C, though with slightly higher CPU usage.

6.2 Execution Time

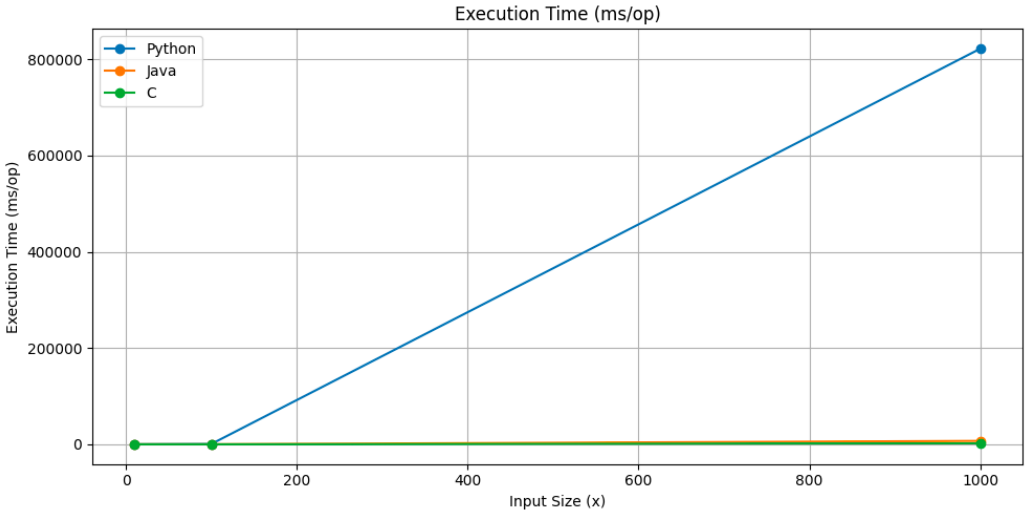

Figure 2: Execution Time across Python, Java, and C for different matrix sizes.

- Python's execution time increased dramatically with matrix size.
- C was the fastest in terms of execution time.
- Java had similar performance to C, but was slightly slower.

6.3 Memory Usage

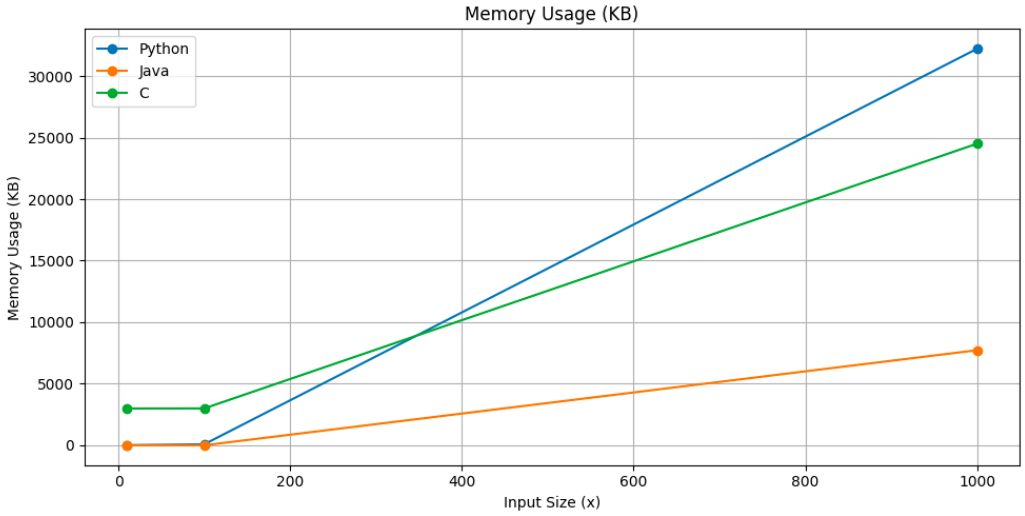

Figure 3: Memory Usage across Python, Java, and C for different matrix sizes.

- Python's memory usage increased significantly with matrix size.
- C used more memory for smaller matrices, but was more efficient for larger matrices.
- Java used minimal memory for smaller matrices, but increased slightly for larger matrices.

7 Language-Specific Optimizations and Features Affecting Performance

In comparing the matrix multiplication implementations in Python, Java, and C, several language-specific optimizations and features impacting performance were identified:

7.1 Python

Memory Management: Python uses automatic memory management (via garbage collection), but its memory usage is higher than lower-level languages like C due to the overhead of its dynamic typing and object model.

Execution Speed: Python is interpreted, making it slower than compiled languages. Its for-loop implementation of matrix multiplication is inefficient compared to optimized libraries like NumPy (which is optimized in C/C++ under the hood).

CPU Usage: Python's CPU usage tends to be higher due to its higher-level abstractions and interpreter overhead.

7.2 Java

Memory Management: Java also uses garbage collection but is generally more optimized for long-running applications.

Execution Speed: Java is faster than Python due to Just-In-Time (JIT) compilation, which optimizes code execution dynamically.

CPU Usage: Java handles CPU-bound tasks well, with the JVM offering optimizations that improve performance for matrix multiplication.

7.3 C

Memory Management: C allows manual memory management, leading to low overhead and more predictable memory usage.

Execution Speed: C is the fastest of the three due to its low-level nature and direct access to hardware.

CPU Usage: C compiles directly to machine code, making CPU usage the most efficient among the three languages.

8 Conclusions and Future Work

The analysis of matrix multiplication implementations in Python, Java, and C shows significant differences in CPU usage, execution time, and memory consumption. While Python is easy to use, it is not suitable for resource-intensive tasks like large-scale matrix multiplication. C, on the other hand, offers the best performance but comes with the complexity of manual memory management. Java strikes a balance between efficiency and ease of use,

offering good performance with more flexibility than C, so probably Java would be the best possible option because it finds the perfect equilibrium. Future research could explore other programming languages, such as Julia or Rust, which promise near-C performance with greater safety and ease of use. Additionally, more advanced matrix multiplication algorithms, such as Strassen's algorithm or parallelization using GPUs, could be tested to improve performance in Big Data applications. It would also be interesting to explore the handling of sparse matrices or even larger dimensions common in real-world Big Data scenarios.