Benchmarking Java, Python, and C in Matrix Multiplication: Performance Analysis

Jorge González Benítez

October 18, 2024

1 GitHub Link

BenchmarkingMatrixMultiplication Repository.

2 Introduction

Benchmarking plays an indispensable role in software development, especially when optimizing performance-critical applications. It provides a systematic way to assess the computational efficiency, memory consumption, and scalability of programs under varying workloads. This is particularly important for tasks such as matrix operations, where the size and complexity of data can significantly impact runtime and resource usage.

In this study, I conduct a comprehensive performance analysis of matrix multiplication across three different programming languages: Java, Python, and C. Each of these languages represents a distinct paradigm in software development, offering unique trade-offs in terms of performance, ease of use, and memory management. Java, as a platform-independent and object-oriented language, is commonly used for enterprise-level applications but relies heavily on the Java Virtual Machine (JVM), which introduces certain overheads. Python, a high-level and dynamically-typed language, is renowned for its ease of use and rich ecosystem of libraries, but its interpreted nature makes it slower for computationally intensive tasks. To address this limitation, I include Python's NumPy library, a widely-used numerical computing extension that leverages optimized C-based routines to enhance performance. Finally, C, as a low-level language, is known for its unparalleled execution speed and fine-grained control over system resources, making it a natural candidate for high-performance computing tasks.

My primary objective in this analysis is to elucidate the strengths and weaknesses of each language with respect to matrix operations, specifically focusing on matrix multiplication. Through rigorous benchmarking, I aim to uncover the runtime performance and memory utilization of these programs, providing insight into how well each language handles increasing computational loads. Additionally, I explore how leveraging specialized libraries such as NumPy can

dramatically improve Python's performance, allowing it to compete with lower-level languages like C in specific use cases.

3 Methodology

We used different tools and libraries to benchmark each program:

- Java: The Java Microbenchmarking Harness (JMH) was used to measure the time taken by a matrix multiplication method with varying input sizes.
- Python: The built-in time utilities were used to measure performance in native Python and Python using the NumPy library.
- C: The `perf` tool was used to collect performance statistics on a C program that implements matrix multiplication.

All tests were performed on a machine with [specifications], and the data was collected over multiple runs to ensure statistical accuracy.

4 Results and Discussion

4.1 Java Performance

As shown in the table below, the performance of the Java matrix multiplication method exhibits a clear degradation as the input size increases. This is a common characteristic of computationally intensive tasks, where the time complexity and resource demands scale with the size of the data being processed. While Java demonstrates good performance for smaller input sizes, its efficiency diminishes significantly when handling larger matrices.

Input Size (n)	Avg Time (ms)	Error (ms)	Units
10	0.001	0.001	ms/op
100	0.830	0.178	ms/op
1000	1708.460	228.352	ms/op

- **Input Size (n)** This represents the size of the matrix used in the multiplication operation. In these tests, matrices of sizes 10x10, 100x100, and 1000x1000 were used. As the input size increases, the number of operations performed increases exponentially due to the time complexity of matrix multiplication, which is $O(n^3)$ for a straightforward implementation. This means that for each unit increase in matrix size, the number of multiplication and addition operations grows cubically, leading to a rapid increase in computational load.

- **Avg Time (ms)** This column displays the average time taken to perform the matrix multiplication, measured in milliseconds per operation (ms/op). For the smallest input size of $n = 10$, the average execution time is a negligible 0.001 ms per operation, which highlights the ability of Java to handle small tasks efficiently. However, as the input size grows to $n = 100$, the average time jumps to 0.830 ms, representing a sharp increase in execution time. By the time the input size reaches $n = 1000$, the average execution time skyrockets to 1708.460 ms, indicating that Java's performance is severely affected by larger matrix sizes.

- **Error (ms)** This column indicates the standard deviation or error margin associated with the average time measurement. It reflects the variability in performance across multiple test runs. For smaller matrices ($n = 10$), the error is minimal at 0.001 ms, which suggests that Java's performance is consistent for small-scale computations. However, as the matrix size increases, the error grows considerably, with $n = 1000$ showing an error of 228.352 ms. This significant variation for larger inputs could be attributed to Java's memory management system (garbage collection) and the underlying dynamic allocation of resources within the JVM. Larger matrix sizes stress Java's memory system, which results in inconsistent performance due to background tasks such as garbage collection and heap resizing.

4.2 Python Performance with and without NumPy

Python's performance in matrix multiplication is drastically improved by using the NumPy library, as seen in the table below. While native Python takes an average of 13,785 microseconds for a matrix operation, NumPy reduces this to just 155.99 microseconds.

Test	Min (us)	Max (us)	Mean (us)	Std Dev
Native Python	11,090.50	32,900.80	13,785.65	6,734.74
Python with NumPy	106.30	184.00	155.99	24.45

The benchmarking results highlight several important aspects:

- **Performance Comparison:** The mean execution time for Native Python is approximately 87 times slower than that of NumPy.
- **Consistency:** The standard deviation for Native Python (6,734.74) is significantly higher than that for NumPy (24.45), indicating that Native Python's performance is more variable.
- **Suitability for Large Datasets:** NumPy is optimized for operations on large arrays or matrices, making it a better choice for numerical computations.

– **Overhead Considerations:** Although NumPy may introduce some initial overhead, its performance benefits for bulk operations typically outweigh this disadvantage.

In conclusion, based on the benchmarking results, it is generally **better to use NumPy** for numerical computations in Python. NumPy not only provides faster execution times but also offers more consistent performance, making it the preferred choice for scientific computing and data analysis.

4.3 C Performance

The C implementation of matrix multiplication proved to be the fastest, as indicated by the low task-clock time and absence of branch-misses, as shown in the performance stats.

Metric	Value	Unit
Task-Clock	79.78	msec
Context-Switches	13	/sec
CPU Migrations	0	/sec
Page Faults	6.196	K/sec
CPUs Utilization	0.970	CPUs
Time Elapsed	0.082	sec
User Time	0.022	sec
System Time	0.045	sec

Table 1: C Benchmark Results

The benchmarking results provide valuable insights into the performance characteristics of the C implementation:

– **Task-Clock:** The task-clock time of 79.78 milliseconds indicates the total time taken by the CPU to execute the matrix multiplication task. This low value demonstrates the efficiency of the C implementation.

– **Context-Switches:** A rate of 13 context switches per second suggests that the program maintained a low level of interruption during execution, which helps in optimizing performance by reducing overhead.

– **CPU Migrations:** The absence of CPU migrations (0 /sec) indicates that the process was able to stay on the same CPU core throughout execution. This stability enhances cache performance and reduces latency.

- **Page Faults:** With a rate of 6.196 thousand page faults per second, the implementation shows that memory access was generally efficient, though a few page faults occurred. High page fault rates can degrade performance, but this rate appears manageable.
- **CPUs Utilization:** The CPU utilization of 0.970 suggests that the implementation effectively used nearly all available CPU resources, indicating good scalability and efficient resource management.
- **Time Elapsed:** The total elapsed time of 0.082 seconds reflects the complete duration from the start to the end of the matrix multiplication operation, confirming the fast execution.
- **User Time:** The user time of 0.022 seconds indicates the time spent executing user code (the matrix multiplication itself), while the remaining time is spent in the system/kernel space.
- **System Time:** The system time of 0.045 seconds represents the time the operating system spent managing tasks on behalf of the process, such as system calls. A low system time relative to user time suggests efficient execution.

These results highlight the strengths of the C implementation in terms of speed and efficiency, making it well-suited for performance-critical applications such as matrix multiplication.

5 Conclusion

In conclusion, this benchmarking experiment reveals that while Python's NumPy library improves performance, C remains the most performant language for matrix multiplication. Java's performance is sufficient for smaller datasets, but it struggles with large-scale operations due to JVM overhead.