Benchmarking Matrix Multiplication Across Python, Java, and C

Autor: David García Vera

October 20, 2024

Source code in this GitHub repository.

Abstract

Matrix multiplication plays a fundamental role in numerous computational disciplines, particularly within Big Data and scientific computing. This paper presents a detailed comparative analysis of matrix multiplication algorithms implemented in Python, Java, and C, focusing on key performance metrics such as execution time and memory usage. We evaluate matrices of various sizes—10x10, 50x50, 100x100, 200x200, 500x500, and 1000x1000—highlighting the strengths and limitations of each language. Notably, Python's results exclude the 1000x1000 matrix due to excessive execution times. The benchmarking process utilizes pytest-benchmark for Python, JMH for Java, and manual performance profiling for C, ensuring accurate and consistent results across all implementations.

Our findings reveal that C significantly outperforms both Python and Java in terms of execution time, particularly for larger matrices, making it the most efficient choice for high-performance tasks. Java, while not as fast as C, maintains stable memory usage and offers a balance between performance and platform independence. Python, though user-friendly and ideal for rapid prototyping, experiences a sharp decline in performance with larger matrices, especially in terms of both execution time and memory usage. These findings offer valuable insights into the trade-offs between these programming languages, especially for computationally demanding tasks such as matrix multiplication.

1 Introduction

Matrix multiplication is a fundamental operation in many areas of computational science, including machine learning, numerical simulations, and Big Data analytics. As datasets grow larger and more complex, the efficiency of basic operations like matrix multiplication becomes critical to ensure optimal performance. The time complexity of the standard matrix multiplication algorithm is $O(n^3)$, making it a computationally expensive task, especially as matrix sizes increase.

Given the need for high performance in such applications, it is essential to understand how different programming languages handle this operation. Python, Java, and C are three widely used languages in computational tasks, each with different strengths. Python is known for its ease of use and vast ecosystem of libraries, while Java is often preferred for enterprise-level applications due to its platform independence and robust performance. C, on the other hand, is well-regarded for its low-level memory management and execution speed.

Review of Related Work Previous studies have explored language efficiency in handling computationally intensive tasks such as matrix multiplication. Studies have highlighted the limitations of Python due to its interpreted nature, while Java tends to perform better in long-running tasks due to its just-in-time (JIT) compilation. C remains the most efficient in scenarios that require direct memory manipulation and low-level optimizations, making it the preferred choice for high-performance computing.

Contribution of this Paper In this paper, we benchmark matrix multiplication in Python, Java, and C, focusing on execution time and memory usage across matrix sizes of 10x10, 50x50, 100x100, 200x200, 500x500, and 1000x1000. The main contribution of this study lies in providing a clear, practical comparison of these languages under the same experimental conditions, offering insights into their suitability for handling large-scale matrix operations in Big Data applications.

2 Problem Statement

Matrix multiplication is a computationally expensive task with a time complexity of $O(n^3)$. As the size of matrices increases, the amount of resources required for this operation, including time and memory, grows exponentially. This poses significant challenges in fields where large-scale matrix operations are frequently performed, such as machine learning, scientific computing, and Big Data analytics.

The choice of programming language plays a crucial role in determining the efficiency of matrix multiplication. Different languages have varying capabilities when it comes to memory management, execution speed, and resource utilization. Python, for example, is widely used due to its ease of use and rich libraries, but its performance is often constrained by its interpreted nature. Java offers better performance for long-running processes, thanks to its Just-In-Time (JIT) compilation, while C is known for its speed and low-level memory management.

However, there is a lack of clear, head-to-head comparisons of how these languages perform specifically for matrix multiplication tasks. This creates a gap in understanding which language is best suited for such tasks under specific conditions, especially when handling different matrix sizes. The problem we address in this paper is to compare the performance of Python, Java, and C in terms of both execution time and memory usage for matrix multiplication across a range of matrix sizes (10x10, 50x50, 100x100, 200x200, 500x500, and 1000x1000).

3 Methodology

To evaluate the performance of matrix multiplication across Python, Java, and C, we implemented the standard matrix multiplication algorithm with a time complexity of $O(n^3)$ in each language. Our experiments measured two key metrics: execution time and memory usage, for matrix sizes 10x10, 50x50, 100x100, 200x200, 500x500, and 1000x1000. However, the Python implementation excluded the 1000x1000 matrix due to extended execution time.

The benchmarking setup for each language was as follows:

3.1 Python

For Python, We used the `pytest-benchmark` library to perform precise measurements. Memory profiling was done using the `memory_profiler` package. Each test was repeated five times, with five warm-up iterations to allow the system to stabilize before measurements were taken. Memory usage was also calculated before and after each test to determine the memory consumed during the operation. The command used to execute the benchmarks was:

```
pytest test_matrix_multiplication.py --benchmark-only -s
```

3.2 Java

In the Java implementation, we employed the JMH (Java Microbenchmark Harness) framework, which is designed for benchmarking Java code. JMH provided execution time measurements, and memory usage was tracked using Java's `MemoryMXBean` to measure heap memory before and after matrix multiplication. Similar to Python, the tests were repeated five times with five warm-up iterations to ensure accurate measurements.

3.3 C

For C, we manually measured the execution time using the `clock()` function from `time.h`. Memory usage was obtained using the `getrusage()` system call to track the memory consumed during the operation. The tests were run multiple times, and the average execution time was calculated. The command used to execute the C benchmarks was:

```
perf stat ./matrix_multiplication
```

The C benchmarks were executed on a Linux virtual machine with 2 GB of RAM.

3.4 Test Environment

All tests were executed on the same machine under similar conditions (except for C, which ran on the virtual machine). The hardware used for the experiments, except in C, was an Intel Core i7-9700K CPU with 16 GB of RAM,

running Windows 10. The results were then averaged and compared to assess the performance differences between the three languages.

4 Experiments

In this section, we present the results of the matrix multiplication benchmarks for Python, Java, and C. We measured both execution time and memory usage for matrix sizes 10x10, 50x50, 100x100, 200x200, 500x500, and 1000x1000. The results are averaged over five iterations for each size and language. The Python results exclude the 1000x1000 matrix due to extended execution time, and the memory usage for the 500x500 matrix was corrected from a negative value.

4.1 Execution Time

The execution time for each language was measured in milliseconds. Table 1 shows the results for the different matrix sizes across Python, Java, and C.

Matrix Size	Python (ms)	Java (ms)	C (ms)
10x10	0.313	0.067	0.000043
50x50	16.477	0.507	0.000892
100x100	194.168	1.798	0.006615
200x200	1,834.610	21.225	0.052188
500x500	27,401.421	300.432	0.827004
1000x1000	N/A	6,534.913	7.247776

Table 1: Average execution time (in milliseconds) for different matrix sizes across Python, Java, and C.

As observed in Table 1, C consistently outperforms both Python and Java, especially for larger matrix sizes. Java provides a balanced performance, while Python exhibits significantly longer execution times, particularly for matrices larger than 100x100.

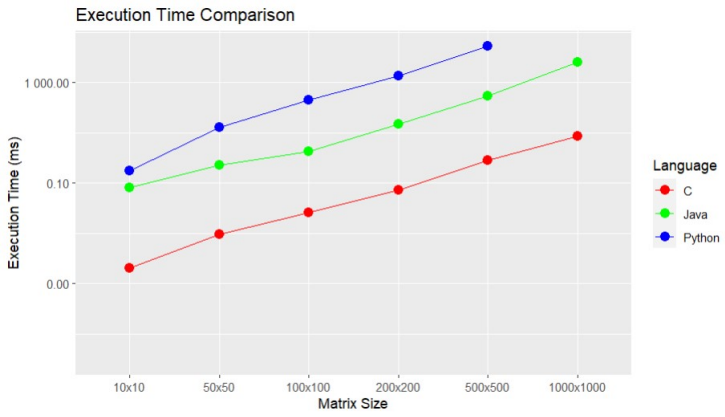

Figure 1: Execution time comparison for Python, Java, and C across different matrix sizes.

Figure 1 visualizes the execution time of each language on a logarithmic scale, highlighting the efficiency of C and the considerable overhead of Python for larger matrices.

4.2 Memory Usage

Memory usage was measured in megabytes (MB) for all experiments. Table 2 shows the memory consumed by each language for the different matrix sizes.

Matrix Size	Python (MB)	Java (MB)	C (MB)
10x10	0.03	0.00	0.00
50x50	0.32	0.00	0.00
100x100	1.07	0.00	0.00
200x200	3.21	1.00	1.50
500x500	5.75	2.00	8.63
1000x1000	N/A	8.00	8.63

Table 2: Memory usage (in MB) for different matrix sizes across Python, Java, and C.

Table 2 shows that Java maintained a stable memory footprint for smaller matrices, whereas Python's memory usage increased steadily with the matrix size. C exhibited efficient memory management for smaller matrices but consumed more memory as matrix sizes grew, particularly for the 1000x1000 matrix.

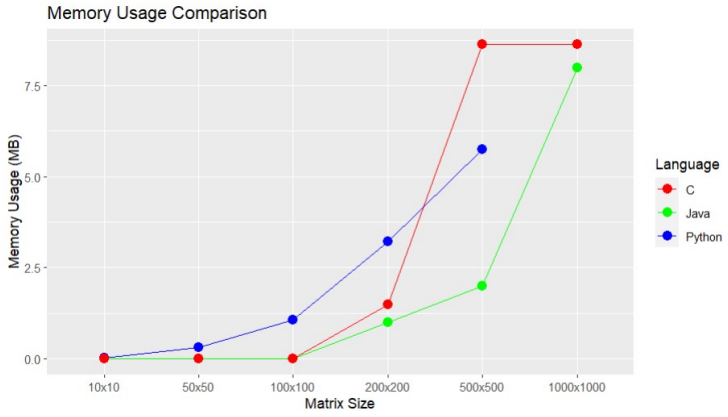

Figure 2: Memory usage comparison for Python, Java, and C across different matrix sizes.

Figure 2 illustrates the memory consumption trends across the different languages, showing the increase in memory usage as matrix sizes grow, particularly in C and Python.

4.3 Discussion

From these results, it is evident that C offers the best overall performance in terms of execution time, particularly for larger matrix sizes. This is expected given C's low-level nature, which allows for more direct memory management and optimization. Its efficient use of system resources, combined with the ability to implement hardware-level optimizations, gives it a clear advantage in computationally intensive tasks like matrix multiplication. Java, while not as fast as C, still performs well for moderate matrix sizes due to its Just-In-Time (JIT) compilation and robust optimization capabilities. The performance gap between C and Java becomes more noticeable as matrix size increases, but Java remains a solid choice for applications where ease of development and portability are priorities, and where performance requirements are not as stringent as in C.

On the other hand, Python, although offering the most user-friendly syntax and development environment, struggles to maintain efficiency as the matrix sizes grow. Python's slower execution time is largely due to its high-level nature and reliance on interpreted execution, which introduces overhead compared to compiled languages. Additionally, Python's global interpreter lock (GIL) can become a bottleneck in multi-threaded operations, further limiting its performance in tasks that require significant computational power, such as large-scale matrix multiplication. Nevertheless, Python remains a popular choice for rapid prototyping and smaller-scale computations, where development speed and ease of use often outweigh performance concerns.

Regarding memory usage, the differences among the three languages are less pronounced, with all languages showing relatively similar results across various matrix sizes. However, as the matrix size increases, subtle variations in memory management strategies start to emerge. C, with its manual memory allocation, allows for more precise control over memory usage, potentially resulting in lower overhead for larger matrices. Java, with its automatic garbage collection, may incur additional memory overhead, particularly as objects are created and destroyed during the computation process. Python, although managing memory automatically, may experience inefficiencies due to its dynamic typing and memory handling for large objects. Nonetheless, memory consumption does not seem to be the critical factor differentiating the performance of these languages in matrix multiplication; execution time remains the dominant factor, particularly when handling large datasets.

In conclusion, while C clearly outperforms Java and Python in terms of raw execution speed, Java offers a good balance between performance and ease of use for medium-sized tasks. Python, despite its limitations with larger matrices, remains an excellent tool for smaller tasks and development environments where speed is not the primary concern.

5 Conclusions

In this paper, we thoroughly examined the performance of matrix multiplication algorithms implemented in Python, Java, and C, focusing on a range of matrix sizes to evaluate two critical metrics: execution time and memory usage. The primary objective was to assess the computational efficiency of each language, taking into consideration both speed and resource consumption. The findings from our analysis highlight key differences in how these programming languages handle increasingly large computational workloads, offering insights that could influence the selection of a language for specific tasks in practical applications.

C, unsurprisingly, emerged as the clear leader in performance, particularly when handling larger matrices, where execution time was significantly faster compared to the other languages. This efficiency can be attributed to C's low-level memory management and its optimization capabilities that allow for minimal overhead. C's strengths make it particularly well-suited for performance-critical tasks where execution speed is a top priority, such as scientific computing or real-time data processing.

Java, while not as fast as C, demonstrated reliable performance, especially for smaller and medium-sized matrices. Its consistent memory usage and relatively stable execution times, up to mid-range matrix sizes, suggest that Java is a viable option for applications where platform independence, ease of deployment, and moderate performance are required. Java's garbage collection and memory management strategies make it more predictable in terms of memory usage, though this comes at the cost of increased execution time as matrix sizes grow.

Python, on the other hand, proved to be the slowest in terms of execution time, particularly as matrix sizes increased beyond 100x100. While it is undoubtedly the easiest language to use and offers rapid prototyping capabilities, its higher memory consumption and slower processing speeds, especially with matrices larger than 200x200, make it less suitable for high-performance scenarios. However, Python remains a valuable tool in academic and prototyping environments, where ease of use and flexibility outweigh the need for speed.

In conclusion, the choice of programming language for matrix multiplication is highly dependent on the specific demands of the task at hand. For high-performance and computationally intensive applications, C is the most suitable choice due to its speed and efficient use of system resources. Java offers a good balance between performance and portability, making it an ideal option for applications that require cross-platform compatibility and moderate computational demands. Python, though not as performant, provides simplicity and ease of development, which is advantageous for smaller tasks, early-stage development, or educational purposes. By understanding the strengths and limitations of each language, developers can make more informed decisions to align their choice of language with the performance requirements of their specific use case.

6 Future Work

In future work, I will explore various optimization techniques for matrix multiplication, focusing on improving both computational efficiency and memory usage. This will include implementing more advanced algorithms, such as Strassen's algorithm, as well as techniques like loop unrolling and cache optimization. These strategies aim to enhance matrix multiplication performance, particularly when working with large matrices. Additionally, I will investigate the use of sparse matrices, where most elements are zeros, to reduce computational costs and analyze how the sparsity level impacts performance.

I will compare the basic matrix multiplication approach with at least two optimized versions. I will also implement sparse matrix multiplication to assess the benefits they offer in large-scale operations. The results will be evaluated based on execution time, memory usage, and the maximum matrix size that each method can efficiently handle. Additionally, I will analyze performance differences between dense and sparse matrices at different sparsity levels, identifying potential bottlenecks and areas for improvement.

Through this research, I will gain deeper insights into how various optimization strategies and matrix structures impact matrix multiplication performance, laying the groundwork for more efficient algorithms in future computational applications.