Matrix Multiplication

Samuel Déniz Santana

October 2024

GitHub repository link

1 Abstract

1.1 Challenge/Problem

This assignment focuses on comparing the performance of a basic matrix multiplication algorithm implemented in Python, Java, and C. The primary objective is to evaluate the efficiency of each language when handling matrix multiplication, considering factors like execution time, memory usage, and computational overhead.

1.2 Experimentation

The matrix multiplication algorithm ($O(n^3)$ complexity) is implemented in each of the three languages. The algorithms are tested with varying matrix sizes, increasing incrementally to assess scalability. Benchmarking tools are employed to measure key performance metrics, including execution time, memory usage, and optionally, CPU utilization. The experimentation also accounts for any language-specific optimizations or features that could influence the results.

1.3 Most Important Results

C generally performs the fastest in terms of execution time and uses less memory, thanks to its low-level memory management. Java follows, benefiting from its Just-In-Time (JIT) compilation, though it incurs additional memory overhead. Python, being an interpreted language, typically has the longest execution time and highest memory usage.

1.4 Conclusion

C is the most efficient language for matrix multiplication due to its low computational overhead, while Python, though convenient for development, is less performant in large-scale computations. Java offers a balance between performance and ease of use.

2 Introduction

Matrix multiplication is a fundamental operation in various computational fields, including scientific computing, machine learning, and computer graphics. Given its wide application and the need for efficiency, it is crucial to understand how different programming languages perform this task. The performance of matrix multiplication, particularly with large datasets, can vary significantly depending on the language, due to factors like memory management, compilation, and execution model.

2.1 Context

Benchmarking programming languages is a common practice to evaluate their strengths and weaknesses in performing computationally expensive tasks. Python, Java, and C are widely used in different domains, each offering unique trade-offs in terms of execution speed, memory usage, and ease of

development. Python is popular for its simplicity and extensive libraries, Java is known for portability and robust performance, while C is valued for its close-to-hardware efficiency and low-level control.

2.2 Literature Review

Previous studies on matrix multiplication have shown that C consistently outperforms higher-level languages like Python in execution time due to its compiled nature and efficient memory handling. Java, with its Just-In-Time (JIT) compilation, strikes a balance between high performance and cross-platform compatibility. In this project, several benchmarking tools have been used to evaluate these languages, focusing on the scalability of the size of the arrays and computational efficiency.

2.3 This Study's Contribution

In this paper, we present a direct comparison of Python, Java, and C for matrix multiplication, focusing on execution time, memory usage, and scalability with different matrix sizes. The added value of this contribution lies in the practical insights it offers for developers and researchers choosing the right language for high-performance computing tasks.

3 Problem statement

Matrix multiplication is a computationally intensive task, with a time complexity of $O(n^3)$ for basic implementations. The choice of programming language can significantly impact the performance of this operation, especially when working with large matrices. Each language handles memory management, compilation, and execution differently, which affects not only execution time but also memory and CPU usage.

Given the increasing need for high-performance computing in fields like data science, machine learning, and simulations, understanding how different programming languages perform in terms of matrix multiplication is essential. While C offers low-level control and efficiency, it may require more development time. Java, with its cross-platform capabilities and JIT compilation, offers a middle ground, but still incurs some overhead. Python, despite its ease of use and rich ecosystem, is often criticized for slower performance, especially in heavy computational tasks.

This study addresses the need to quantitatively compare the performance of Python, Java, and C in the specific context of matrix multiplication. By systematically benchmarking these languages using various matrix sizes, we aim to provide clear insights into their strengths and limitations, helping developers choose the most appropriate language for performance-critical applications.

4 Methodology

To compare the performance of matrix multiplication in Python, Java and C, we will carry out experiments with consistent configurations, ensuring reproducibility and reliability of the results.

1. **Implementation of matrix multiplication**

   We will implement a basic matrix multiplication algorithm (complexity $O(n^3)$) in each of the three languages: Python, Java and C. The algorithm will multiply two square matrices of 5x5,10x10, 50x50, 100x100, 200x200, 500x500, 1000x1000 in size, to observe performance with increasing matrix sizes. All implementations will follow the same logic for a fair comparison.

2. **Experiment setup**

   The experiments will be conducted in the following environments:

   - **Java:** Implemented and run using IntelliJ IDEA.
   - **Python:** Implemented and executed using PyCharm.
   - **C:** Implemented and run on a Linux virtual machine (VM), ensuring that the configuration mimics typical usage for low-level programming. The experiments will run on a machine with the following specifications:

- **CPU:** Intel Core i7, 3.2 GHz
- **RAM:** 16 GB
- Operating system: Host system running Linux (Ubuntu 20.04), with the virtual machine for the C experiments running the same operating system.

3. **Performance metrics**

   The performance of each language will be evaluated based on:

   - **Execution time:** Measured using high-resolution timers native to each environment (e.g., the time module in Python, System.TimeUnit() in Java, and clock() in C).
   - **Memory usage:** Recorded using platform-specific profiling tools.

4. **Benchmarking procedure**

   Each experiment will be repeated several times to account for variability, and the average results will be used for comparison. Results will be normalised and plotted to visualise differences in performance between languages as the size of the matrices increases.

5. **Language-specific considerations**

   Any language-specific features (such as JIT compilation in Java or interpreter overhead in Python) will be taken into account, although no external libraries or parallelisation optimisations will be applied to maintain the focus on the performance of the base algorithm in each language.

   This approach will provide a consistent and fair comparison of the performance of matrix multiplication in Python, Java and C, providing information on their relative efficiency.

5 Experiment

This research aims to compare the performance of three programming languages: **Python**, **Java** and **C** in the execution of intensive mathematical operations, specifically the multiplication of matrices of different sizes (5x5, 10x10, 50x50, 100x100, 200x200, 500x500 and 1000x1000). To evaluate this performance, not only the execution times of each language were measured, but also the memory usage, which will be analysed in detail later.

5.1 Execution Time

The main objective is to determine which of these languages offers greater efficiency both in terms of execution time, providing a solid basis for choosing the most appropriate tool depending on the type of operation and the size of the data to be processed. The tests carried out reveal significant performance differences between languages, highlighting the influence of factors such as resource management and low-level optimisation.

The results of these experiments are shown both in tables and in graphs, where the variation in execution times as the size of the matrix increases can be clearly seen. In addition, the impact of memory usage on each language, a crucial metric for large-scale operations and resource-constrained environments, will be discussed later.

Table 1: Comparison of Matrix Multiplication Execution Times by Programming Language(milliseconds)

	5x5	**10x10**	**50x50**	**100x100**	**200x200**	**500x500**	**1000x1000**
Python	0.0168735	0.0890902	9.4274265	67.1572833	624.4145	13660.45722	X
Java	0.001	0.002	0.239	1.532	16.486	312.530	4002.467
C	0.001	0.001	0.068	0.636	7.711	135.525	1462.212

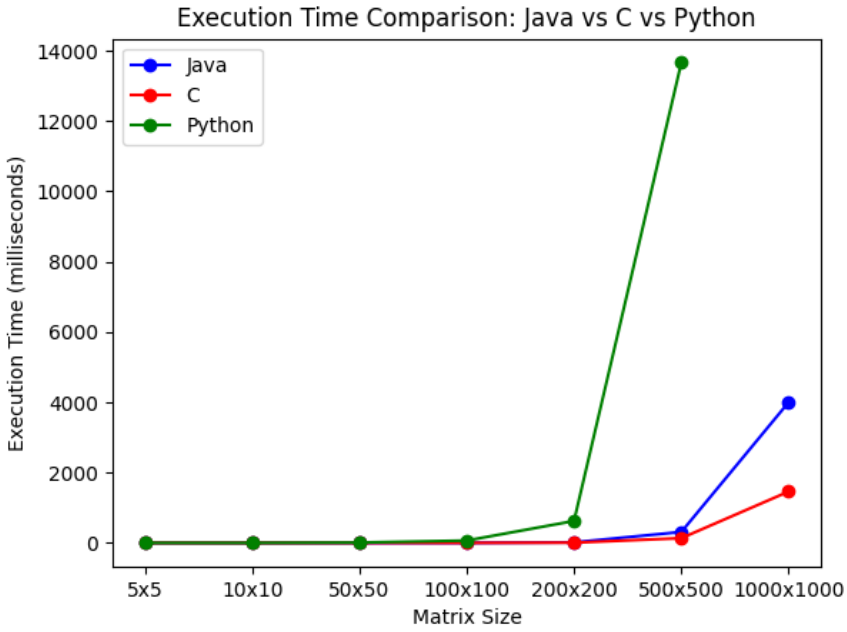

Figure 1: Execution Times Graph

Description of the Experiments

- **Multiplication of 5x5 matrices**
  - **Python**: 0.0169 ms
  - **Java**: 0.001 ms
  - **C**: 0.001 ms

  Here, Java and C show very short times, with almost identical performance. Python, although slower, is still efficient in operations with small arrays.

- **Multiplication of 10x10 matrices**
  - **Python**: 0.0891 ms
  - **Java**: 0.002 ms
  - **C**: 0.001 ms

  As the size of the matrix increases, there is a slight increase in execution time in Python, while Java and C remain fairly fast, with minimally increasing times.

- **Multiplication of 50x50 matrices**
  - **Python**: 9.4274 ms
  - **Java**: 0.239 ms
  - **C**: 0.068 ms

  On medium-sized arrays, the execution time in Python increases significantly, while Java and C still maintain comparatively low times. Python shows slower behaviour as the size of the array increases.

- **Multiplication of 100x100 matrices**
  - **Python**: 67.1573 ms
  - **Java**: 1.532 ms
  - **C**: 0.636 ms

  At this scale, Python becomes much slower, with times exceeding 67 ms, while Java and C remain more efficient, with times of 1.532 ms and 0.636 ms, respectively.

- **Multiplication of 200x200 matrices**
  - **Python**: 624.4145 ms
  - **Java**: 16.486 ms
  - **C**: 7.711 ms

  Here, Python becomes even more inefficient, requiring more than 624 ms to complete the operation. Java and C are still relatively fast, although they also start to show a considerable increase in execution time.

- **Multiplication of 500x500 matrices**
  - **Python**: 13,660.4572 ms
  - **Java**: 312.530 ms
  - **C**: 135.525 ms

  In this case, Python has an extremely high time, exceeding 13.6 seconds. Java and C, although they also have much higher times than the smaller arrays, are much more efficient.

- **Multiplication of 1000x1000 matrices**
  - **Python**: Not performed because the execution time was extremely slow and prohibitively long.
  - **Java**: 4,002.467 ms
  - **C**: 1,462.212 ms

  In this last test, Python could not complete the operation due to the significant execution time required. Java, while able to finish the task, took nearly three times longer than C. C demonstrated much better performance for this matrix size, highlighting its superior efficiency for large-scale operations.

5.2 Memory Usage

Having explained the runtimes in the various matrix multiplication tests, we now turn to the memory usage. As with the runtimes, experiments were carried out with different matrix sizes using three programming languages: Java, C and Python. The experiments performed are described below, together with the results obtained, maintaining the format of the tables presented above.

Table 2: Comparison of Memory Usage during Matrix Multiplication by Programming Language (MB)

	5x5	**10x10**	**50x50**	**100x100**	**200x200**	**500x500**	**1000x1000**
Java	2.08	2.08	2.12	2.24	2.70	5.91	17.37
C	0.00	0.00	0.06	0.23	0.92	5.72	22.89
Python	1.33	0.07	0.26	0.94	3.05	14.88	X

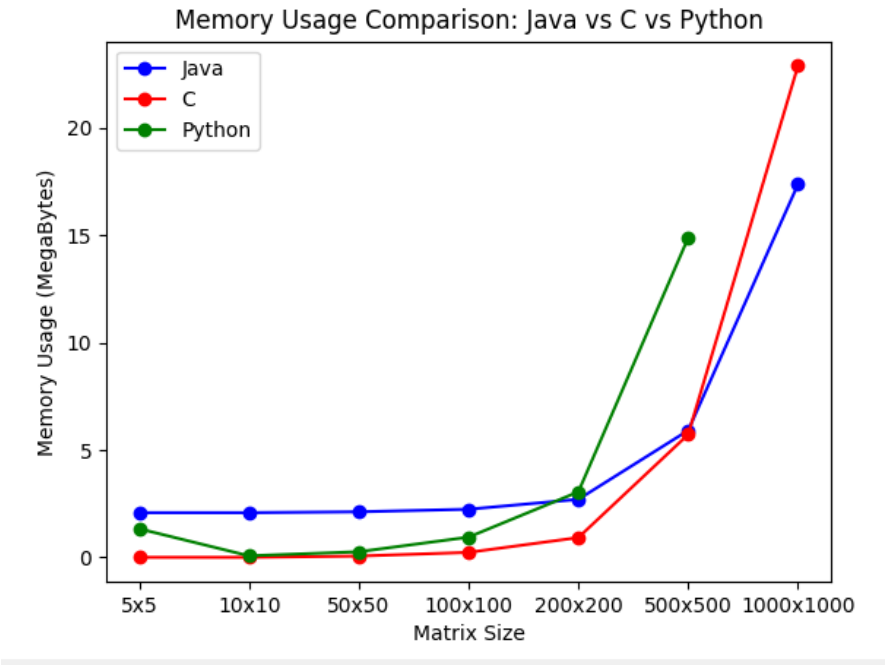

Figure 2: Memory Usage Graph

Description of the Experiments

- **Multiplication of 5x5 matrices**
  - **Python**: 1.33 MB
  - **Java**: 2.08 MB
  - **C**: 0.00 MB

  Java and Python show higher memory usage compared to C, which does not consume additional memory in this case.

- **Memory Usage for 10x10 matrices**
  - **Python**: 0.07 MB
  - **Java**: 2.08 MB
  - **C**: 0.00 MB

  Similar to the 5x5 matrices, C continues to use no additional memory. Python slightly improved its memory efficiency compared to the 5x5 matrix, while Java's memory usage remained consistent.

- **Memory Usage for 50x50 matrices**
  - **Python**: 0.26 MB
  - **Java**: 2.12 MB
  - **C**: 0.06 MB

Here, C continued to be the most efficient, requiring only 0.06 MB, while Java's memory usage marginally increased. Python's memory usage also increased but remained much lower than Java's.

- **Memory Usage for 100x100 matrices**
  - **Python**: 0.94 MB
  - **Java**: 2.24 MB
  - **C**: 0.23 MB

  As the matrix size increased, C remained highly efficient. Python's memory usage increased significantly but was still lower than Java's, which continued to rise slightly.

- **Memory Usage for 200x200 matrices**
  - **Python**: 3.05 MB
  - **Java**: 2.70 MB
  - **C**: 0.92 MB

  At this scale, C's memory usage began to rise, though still much lower than both Python and Java. Python's higher memory usage for large matrices is due to its dynamic typing and extra data management overhead.

- **Memory Usage for 500x500 matrices**
  - **Python**: 14.88 MB
  - **Java**: 5.91 MB
  - **C**: 5.72 MB

  As expected, larger matrices resulted in more substantial memory usage. Python's memory usage increased drastically to almost 15 MB, while Java and C were closely matched, with Java slightly outpacing C in memory consumption.

- **Memory Usage for 1000x1000 matrices**
  - **Python**: Not performed.
  - **Java**: 17.37 MB
  - **C**: 22.89 MB

  For the largest matrix, C's memory usage exceeded Java's for the first time, with C requiring almost 23 MB. Python, as in the runtimes, did not perform the operation due to the extreme slowness of the process.

6 Conclusions

In this research, we have tackled the challenge of optimizing matrix multiplication by comparing the execution time and memory usage across three programming languages: Python, Java, and C. The goal was to evaluate the efficiency of each language when performing matrix multiplication on matrices of varying sizes, from 5x5 to 1000x1000, and identify which language offers the best performance for this specific computational task.

The methodology consisted of running matrix multiplication experiments in all three languages and analyzing the results in terms of execution time and memory consumption. Each experiment was conducted with increasing matrix sizes, allowing us to observe how the performance of each language scales with larger datasets.

Main Results and Discussion

From the results, several conclusions can be drawn regarding the strengths and weaknesses of Python, Java, and C for matrix multiplication:

- **Execution Time:**

C consistently outperformed both Java and Python in terms of execution time. It had the shortest times across all matrix sizes, including the largest tested (1000x1000 matrices), completing the task in 1.46 seconds. Java was close behind, showing strong performance and remaining competitive with C, especially in smaller matrix sizes. However, as the matrix size increased, Java's execution time began to lag behind C.

Python, on the other hand, performed significantly slower than both C and Java. While Python handled smaller matrices efficiently, it struggled with larger matrices, where the execution time increased drastically. For instance, multiplying a 500x500 matrix took Python over 13.6 seconds, whereas C and Java completed the same task in a fraction of that time.

- **Memory Usage:**

Java is more memory efficient than C in certain cases due to its automatic memory management through the Garbage Collector, which helps reclaim memory that is no longer in use, preventing memory leaks. Additionally, Java's memory allocation is more predictable because of its managed environment, which can reduce fragmentation. However, Java does have some overhead from its runtime environment and object-oriented structure, but for these experiments, its optimizations for memory management still kept its usage relatively low compared to Python.

Importance of the Experimentation

This experimentation is crucial because matrix multiplication is a fundamental operation in many areas of computational science, including machine learning, data analysis, and simulations. Understanding which language offers the best performance can have significant implications for real-world applications where efficiency is key. The results highlight the importance of choosing the right tool for the job.

- **C** is clearly the most optimal language for matrix multiplication, particularly when performance and memory efficiency are critical. Its low-level nature allows for precise control over computational resources, making it the best choice for time-sensitive applications.
- **Java**, while not as fast as C, remains a strong contender and may be preferred in environments where portability and ease of use are important, without sacrificing too much in terms of performance.
- **Python**, although slower and less memory-efficient, is still useful for smaller matrices or when ease of development and readability are prioritized over raw performance.

In conclusion, when optimizing for both execution time and memory usage in matrix multiplication, C is the most efficient choice, followed by Java. Python, while easier to work with, is less suitable for large-scale matrix operations due to its slower performance and higher memory consumption. These insights are vital for developers and researchers aiming to select the most appropriate language for their computational tasks, particularly in performance-critical environments.

7 Future Works

While this study provides a comparison of basic matrix multiplication across Python, Java, and C, future work could explore advanced optimization techniques. Specifically, implementing and comparing optimized matrix multiplication algorithms like Strassen's algorithm, loop unrolling, and cache optimization techniques could provide further insights into improving performance. Additionally, sparse

matrix multiplication is a promising area to investigate, especially in applications where matrices contain mostly zero elements. Exploring different sparsity levels and their impact on execution time and memory usage could highlight opportunities to reduce computational overhead.

Further experimentation could also involve testing larger matrix sizes and addressing any bottlenecks observed during this study. For instance, investigating parallelization techniques or utilizing external libraries (e.g., BLAS for C) might significantly improve performance. Exploring how different languages handle more sophisticated optimizations and evaluating the trade-offs of these approaches will offer a deeper understanding of efficient matrix multiplication in real-world applications.