Basic Matrix Multiplication in Different Languages

Lucía Afonso Medina

October 18, 2024

Contents

Abstract

Matrix multiplication is a fundamental operation in various fields of mathematics and computer science, typically performed using the classical algorithm with a time complexity of $O(n^3)$ for multiplying two $n \times n$ matrices. However, this computational cost becomes inefficient as matrix size increases. This study evaluates the performance of matrix multiplication algorithms implemented in three programming languages: C, Java, and Python, with a focus on determining which language yields the lowest execution time and memory usage.

1 Introduction

In today's landscape, where numerous programming languages are available, it can be challenging for developers to determine which language is best suited for a particular project. This can lead to inefficiencies, particularly in computationally intensive operations like matrix multiplication, which are critical in fields such as scientific computing, machine learning, and artificial intelligence.

Matrix multiplication is a fundamental operation with significant impact on the performance of algorithms, especially when processing large-scale data. The standard algorithm for matrix multiplication has a time complexity of $O(n^3)$, making it computationally expensive as matrix sizes grow. As both programming languages and hardware systems evolve, it becomes increasingly important to evaluate the performance of these languages when executing such resource-intensive tasks.

Benchmarking has emerged as a valuable approach for evaluating the performance of programming languages in handling computational workloads. Python, Java, and C are three popular languages that offer distinct approaches to solving such problems, each with its own design philosophies and trade-offs.

Using tools like pytest-benchmark for Python, JMH for Java, and perf for C, we evaluate execution time and memory usage across varying matrix sizes. The results provide a detailed comparison of these three languages, helping developers make informed decisions based on the trade-offs between ease of development, computational efficiency, and resource usage.

2 Matrix Multiplication Algorithm

In this work, it has been implemented the basic matrix multiplication algorithm, commonly known as the **naive** or **brute-force** method. This algorithm is widely used due to its simplicity and ease of implementation. However, it has a time complexity of $O(n^3)$, which means that the execution time increases cubically as the size of the matrices grows.

The algorithm operates on square matrices of size $n \times n$. Since we are multiplying square matrices of increasing dimensions (for example, 10×10, 100×100, and so on), the total number of operations required to compute the resulting matrix is directly proportional to the cube of the matrix size, i.e., n^3.

Each element of the resulting matrix $C[i][j]$ is computed by taking the dot product of row i from matrix A and column j from matrix B. This requires iterating over the rows of A and the columns of B, performing n multiplications and additions for each element of the resulting matrix.

The pseudocode for the matrix multiplication algorithm is as follows:

Algorithm 1 Naive Matrix Multiplication

```
1: for i = 0 to n − 1 do
2:     for j = 0 to n − 1 do
3:         C[i][j] ← 0
4:         for k = 0 to n − 1 do
5:             C[i][j] ← C[i][j] + A[i][k] × B[k][j]
6:         end for
7:     end for
8: end for
```

The process can be broken down into the following steps:

- **Outer loop**: Iterates over each row of matrix A, executing n times.
- **Middle loop**: Iterates over each column of matrix B, also executing n times for each iteration of the outer loop.
- **Inner loop**: For each combination of row i and column j, the inner loop multiplies n elements (from the row of A and the column of B) and sums the results.

This approach, while intuitive, results in cubic complexity, $O(n^3)$, due to the three nested loops. As the size n of the matrices increases, the number of operations grows exponentially, leading to a significant increase in execution time and memory usage for large matrices.

3 Problem Statement

The goal of this work is to compare the efficiency of this basic implementation across different programming languages, specifically Python, Java, and C, to evaluate how execution time and memory usage vary as matrix size increases. Tests will be conducted for matrices of different sizes, allowing us to observe how the choice of programming language affects the performance of solving a problem with $O(n^3)$ complexity.

4 Methodology

In this section, we describe the experimental setup and methodology used to evaluate the performance of matrix multiplication algorithms implemented in Python, Java, and C. The objective of these experiments is to compare the execution time and memory usage across different matrix sizes, with a focus on determining how each programming language handles the computational complexity of the problem.

4.1 Experimental Setup

The experiments will be conducted on a machine with the following specifications:

- Processor: 13th Gen Intel(R) Core(TM) i5-1340P 1.90 GHz
- RAM: 16,0 GB
- Operating System: Windows 11 Home, Version 23H2.
- Compiler/Interpreter versions:
  - Python: Version 3.9
  - Java: 21.0.1
  - C: Executed within a Fedora Virtual Machine hosted on the same physical machine

4.2 Procedure and Performance Metrics

Each implementation will perform matrix multiplication on square matrices of varying sizes, specifically: 10×10, 100×100, 300×300, 500×500, and 1000×1000. The performance metrics of interest are:

- **Execution Time**: Measured in milliseconds using benchmarking tools specific to each programming language:
  - **Python**: The execution time will be measured using the `benchmark` library, which provides accurate and consistent timing for Python functions.
  - **Java**: The `JMH (Java Microbenchmark Harness)` framework will be used for precise performance benchmarking in Java.
  - **C**: For C, the `perf` tool will be employed to profile and measure execution time at the system level, providing detailed performance insights.
- **Memory Usage**: Memory allocation during the execution of matrix multiplication will be tracked as follows:
  - **Python**: Memory usage will be monitored using the `psutil` library, which allows tracking the memory consumed by the current process.
  - **Java**: Memory consumption will be measured using the `Runtime` class, which provides information about the total and free memory in the Java Virtual Machine (JVM).

– **C**: The `valgrind` tool will be used to analyze memory usage, offering detailed information about memory allocation, leaks, and other profiling details in C.

The following procedure will be followed for each programming language:

1. **Matrix Generation**: For each matrix size n, two random matrices A and B of size $n \times n$ will be generated, with elements randomly initialized between a predefined range.
2. **Algorithm Execution**: The matrix multiplication algorithm will be executed using the generated matrices A and B, and the resulting matrix C will be computed.
3. **Performance Measurement**: Execution time and memory usage will be measured
4. **Repetition**: Each experiment will be repeated 5 times for each matrix size and programming language to obtain an average value for both execution time and memory usage, minimizing the impact of system noise and variability. Specifically:
   - In Java, the execution is configured with 2 warmup iterations and 5 measured iterations using the @BenchmarkMode and @Measurement annotations, ensuring that the JVM has warmed up before accurate measurements are taken.
   - In Python, the benchmark.pedantic function is used, where the matrix multiplication is executed once per round with 2 warmup round and 5 measured iterations.
   - In C, the matrix multiplication code is executed 5 times for each matrix size, ensuring consistency and minimizing fluctuations in performance.

4.3 Analysis

The results will be analyzed to compare the performance of Python, Java, and. We will calculate the average execution time and memory usage for each matrix size in each language, and present the results using tables and graphs for clarity. Furthermore, we will compute the **speedup** of Java and C implementations relative to Python, which will serve as a baseline.

The speedup is defined as:

$$\text{Speedup} = \frac{\text{Execution time of Python}}{\text{Execution time of language X (Java or C)}}$$

Finally, we will discuss how the performance scales with increasing matrix sizes and the potential bottlenecks or advantages each language presents in handling matrix multiplication.

5 Experiments

The following figure shows a comparison of the execution times for matrix multiplication in C, Java, and Python across different matrix sizes. The x-axis represents the matrix size, which varies from 10 to 1000, while the y-axis (logarithmic scale) represents the execution time in milliseconds.

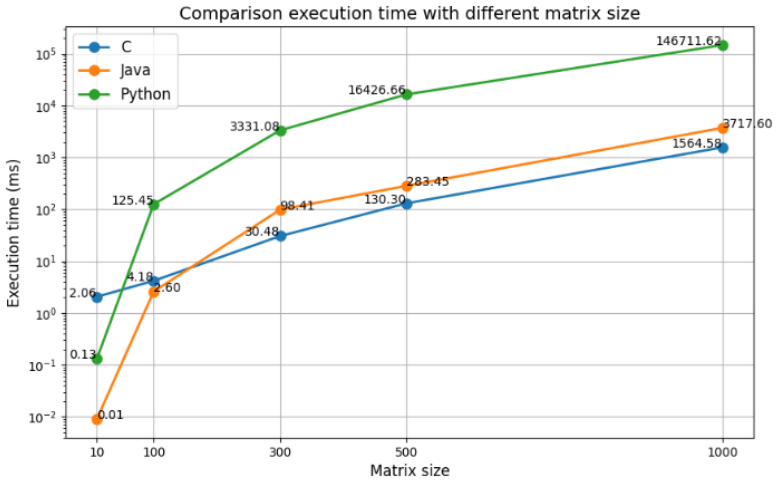

Figure 1: Execution time in different languages.

In the graph, we observe the following trends:

1. **C (blue line)**: C shows the highest execution time for small matrices (e.g., 10x10), where it takes 2.06 ms, compared to Java and Python, which are significantly faster. However, as the matrix size increases, C demonstrates better scaling, and its execution time grows more slowly compared to the other languages. For a matrix of size 1000x1000, C takes 1564.58 ms, which is the lowest among the three languages.

2. **Java (orange line)**: Java starts with the best performance for small matrices. For example, at a matrix size of 10x10, Java takes only 0.01 ms, which is faster than both C and Python. However, as the matrix size increases, Java's performance deteriorates relative to C, although it remains much better than Python. For a matrix of size 1000x1000, Java's execution time is 3717.60 ms.

3. **Python (green line)**: Python performs adequately for very small matrices, but it quickly becomes inefficient as the matrix size grows. For instance, Python takes 125.45 ms for a 100x100 matrix, which is considerably slower than C and Java. For larger matrices, such as 1000x1000, Python's execution time dramatically increases to 146711.62 ms (around 146.7 seconds), making it the worst performer for larger matrices.

The table below shows the SpeedUp comparison between Python and Java, as well as Python and C, across different matrix sizes. The SpeedUp is calculated as the ratio of Python's execution time to Java's or C's execution time. Higher values indicate that Python is significantly slower compared to the other language.

Size	**SpeedUp (Python/Java)**	**SpeedUp (Python/C)**
10	14.418	0.0630
100	48.342	30.0029
300	33.848	109.2872
500	57.953	126.0662
1000	39.464	93.7705

Table 1: SpeedUp comparison between Python, Java, and C for different matrix sizes.

From the table, we observe the following:

1. **SpeedUp (Python/Java):** For smaller matrix sizes (size 10), Python is 14.44 times slower than Java. As the matrix size increases, this inefficiency becomes more pronounced. For instance, at size 500, Python is 57.95 times slower than Java and for the largest matrix size (1000), Python is 39.46 times slower than Java. While the SpeedUp decreases slightly for larger matrices, Python remains considerably less efficient than Java as the matrix size increases.

2. **SpeedUp (Python/C):** For very small matrices (size 10), Python performs slightly better than C, with a SpeedUp of 0.0630, meaning that Python is slightly faster than C for very small matrices. This is unusual given C's expected performance. As matrix sizes increase, C significantly outperforms Python. For example, at matrix size 500, Python is 126.0662 times slower than C. Even for the largest matrix size (1000), Python is 93.7705 times slower than C, indicating that C remains the best performer as matrix sizes grow.

- Java consistently outperforms Python, though the difference is less pronounced for very small matrices. However, as the matrix size grows, the performance gap widens in favor of Java.

- C is the fastest overall, especially for large matrix sizes, where the performance difference between Python and C becomes extreme. Python struggles considerably with larger matrix sizes compared to both Java and C.

Figure 2 shows a comparison of memory usage in three programming languages: Python, Java, and C, when performing matrix multiplication for different matrix sizes. The following observations can be made:

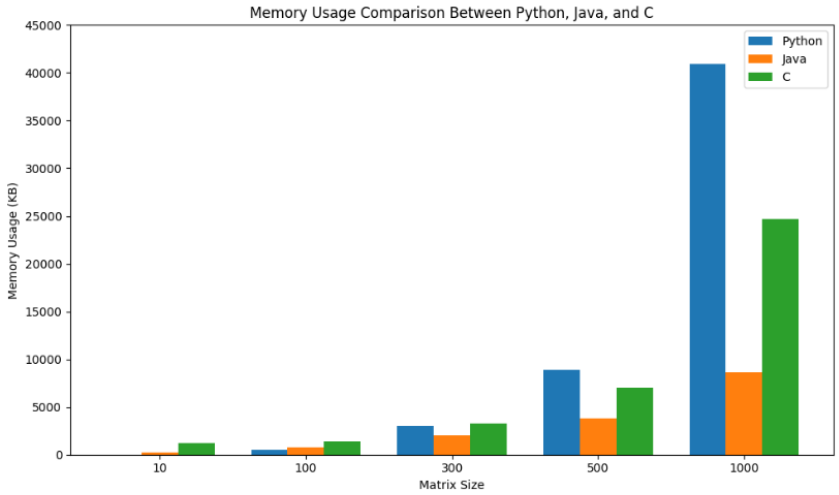

Figure 2: Memory Usage Comparison Between Python, Java, and C

- **Python's Memory Usage**: Python shows a drastic increase in memory usage as the matrix size grows. For small matrices (like 10×10), memory usage is relatively low, but for large matrices, such as 1000×1000, Python consumes over 40,000 KB, significantly more than both Java and C.

- **Java's Memory Usage**: Java demonstrates relatively consistent and efficient memory usage across all matrix sizes. While its memory consumption is slightly higher than Python for small matrices, it remains considerably lower than both Python and C as the matrix sizes increase, using just over 8,000 KB for a 1000×1000 matrix.

- **C's Memory Usage**: C starts with moderate memory consumption for small matrices. As matrix size increases, memory usage also grows, but at a much slower rate than Python. For large matrices 1000×1000, C uses around 25,000 KB, more than Java but significantly less than Python.

6 Conclusions

In summary, C stands out as the best choice for matrix multiplication when performance is critical. It delivers the fastest execution times across all matrix sizes, particularly for larger matrices, and shows efficient scaling as the problem size increases. While C consumes more memory than Java, it remains well within reasonable limits, making it a strong option for tasks where speed is the primary concern. Its close-to-hardware execution and low-level optimizations enable it to handle large matrix operations with impressive efficiency.

Although Java is slightly slower than C, it still offers strong performance, particularly for small to medium-sized matrices. Its key advantage is its exceptional memory efficiency, consistently using less memory than both C and Python, even as matrix sizes increase. This makes Java highly suitable for applications where minimizing memory usage is essential. While Java's execution times are slower than C's for larger matrices, they remain competitive and significantly faster than Python's, making it a reliable choice when balancing performance and memory considerations.

On the other hand, Python is the least efficient language in terms of both execution time and memory usage. While Python is useful for small-scale tasks or rapid prototyping due to its ease of use, it becomes highly inefficient for large matrix operations. Its execution time increases dramatically with matrix size, making it unsuitable for performance - sensitive applications. Moreover, its memory consumption grows disproportionately compared to C and Java, further reducing its viability for large-scale computations.

In conclusion, C is the optimal choice when raw performance and execution speed are paramount, particularly for large matrices. Java strikes a strong balance between performance and memory efficiency, making it ideal for applications where memory constraints are important but where reasonable execution times are still required. Python, while user-friendly and flexible, should be reserved for smaller tasks where performance and memory are not the primary concerns.

7 Future Work

The matrix multiplication algorithms discussed in this paper can be optimized further by exploring parallel processing techniques and algorithmic improvements. Future work may also involve comparing additional languages and using more advanced profiling tools to gather deeper insights into performance bottlenecks.

The source code for this project is available at the following GitHub repository: *BenchmarkMM.git*