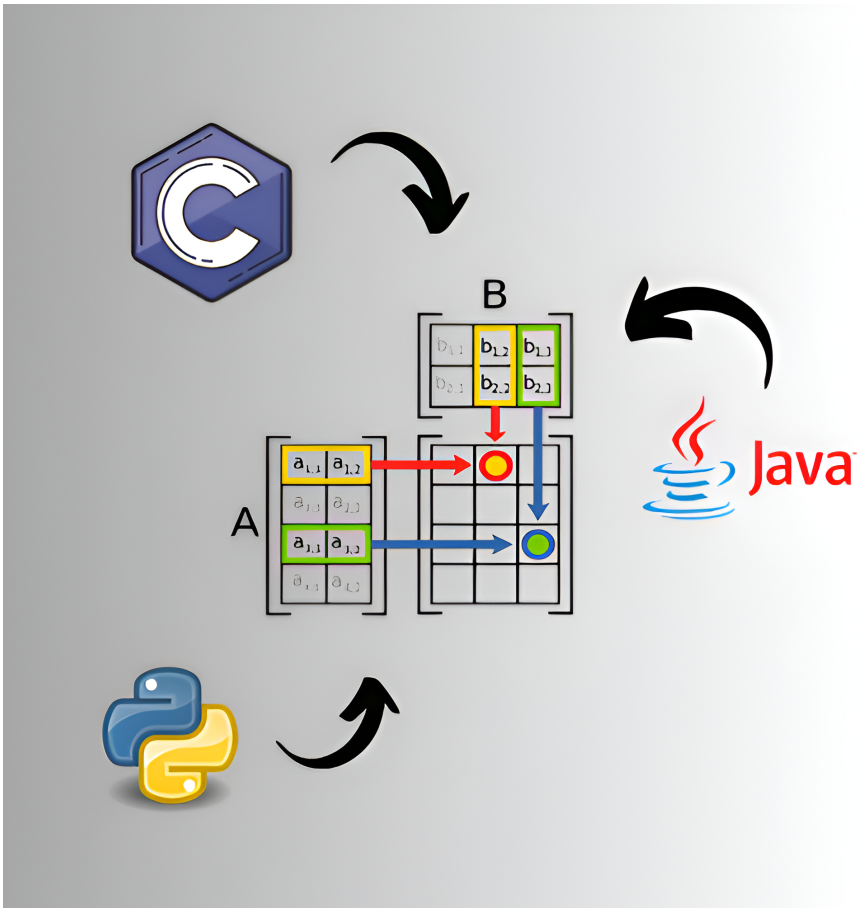
Matrix Multiplication in Different Programming Languages

José Luis Perdomo de Vega

Universidad de Las Palmas de Gran Canaria

October 20, 2024



1 Abstract

This project explores the implementation of matrix multiplication across three programming languages: Python, Java, and C. The challenge lies in assessing the performance differences inherent to each language while executing the same algorithm. To conduct the experimentation, I implemented naive time measurements directly in the code, complemented by professional benchmarking tools such as pytest for Python or JMH for Java. Each implementation was tested with various matrix sizes to ensure a comprehensive analysis of execution time and efficiency.

The most significant results revealed that C consistently outperformed both Python and Java, demonstrating its superior execution speed and memory management capabilities. While Python provided ease of implementation, it lagged considerably in performance, and Java offered a middle ground but still fell short compared to C. This study confirms that for computationally intensive tasks like matrix multiplication, C is the most efficient choice among the evaluated languages, highlighting the trade-offs between ease of use and execution speed.

2 Introduction

Benchmarking programming languages is a critical aspect of software development, particularly when optimizing algorithms for computational efficiency. In tasks such as matrix multiplication, the choice of programming language can significantly affect execution time and resource utilization. As the size of the matrices increases, the demand on CPU and memory resources grows, making it essential to choose the right language for the task.

In recent years, various studies have focused on benchmarking programming languages, with an emphasis on comparing execution times of computationally expensive algorithms. These comparisons often explore popular languages such as Python, Java, and C, each of which offers distinct advantages. Python is known for its simplicity and ease of implementation, Java for its balance between performance and portability, and C for its low-level memory management and high-speed execution. The differences between these languages are particularly evident in matrix multiplication tasks, where the trade-off between ease of use and execution speed becomes clear.

Several researchers have contributed to this area of study. For example, Abdullah Almurayh [1] and others have explored language performance by focusing primarily on execution time as the sole metric. While this approach provides valuable insights, it leaves out important factors such as memory usage and CPU efficiency, which can be crucial in real-world applications. Moreover, these studies often rely on basic time measurement techniques, which may lack the precision needed for a thorough evaluation.

In this paper, I extend previous research by incorporating more sophisticated benchmarking tools alongside traditional execution time measurements. Python's pytest, Java's JMH, the command perf for C and the PerfMon tool were employed to offer a deeper analysis of performance across multiple matrix sizes. This allows for a more comprehensive comparison that not only evaluates raw execution speed but also considers factors such as memory management, CPU utilization (% of processor time), available memory (bytes available), and disk access performance (physical disk bytes read). The added value of this study lies in the use of advanced profiling tools, providing a nuanced understanding of how each language handles matrix multiplication, and offering a clear recommendation for developers facing similar computational challenges.

3 Problem Statement

Matrix multiplication is a computationally intensive task, especially as the size of the matrices increases. The naive algorithm for multiplying two matrices follows a time complexity of $O(n^3)$ where represents

the dimension of the matrices. This cubic growth means that for each additional row or column in the matrix, the number of operations required to compute the product grows significantly. For instance, doubling the size of the matrices results in an eightfold increase in the number of computations. This exponential increase in computational demands leads to a steep rise in execution time, particularly for large matrices.

Algorithm 1 Matrix Multiplication Algorithm

Input: Two matrices A and B of size $n \times n$
Output: Matrix C which is the product of A and B
Initialize matrix C as a zero matrix of size $n \times n$
for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    for $k = 1$ to n **do**
      $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
    end for
  end for
end for
return C

Figure 1: Cubic Complexity Algorithm

The reason behind this increase lies in the structure of the matrix multiplication algorithm. For each element in the resulting matrix, the algorithm must compute the dot product of a row from the first matrix and a column from the second matrix, a process that involves iterating over all elements in the row and column. As the size of the matrices grows, the number of these operations multiplies, leading to an inevitable rise in computational cost.

Furthermore, larger matrices impose additional strain on system resources, such as memory and cache, which can exacerbate the execution time. With small matrices, data can often be held in cache memory, allowing for faster access and computation. However, as matrix size increases, data overflows into main memory, resulting in more frequent memory access, which is slower and less efficient.

4 Methodology

To thoroughly evaluate the performance of matrix multiplication across different programming languages, two distinct types of experiments were designed for each language: a naive approach and a professional benchmarking approach. These complementary methods allowed me to capture both a high-level overview of execution times and a more nuanced analysis of how system resources were utilized during the computation.

In the naive experiment, the focus was on measuring the execution time directly within the code itself, using each language's built-in timing functions, such as time.time() for Python, System.nanoTime() in Java and the clock() function in C. While these methods are straightforward and provide an informative baseline, they do not capture deeper system-level metrics such as memory usage or CPU efficiency. Nevertheless, they offer a simple way to understand raw performance differences between the languages under test.

For a more comprehensive analysis, professional benchmarking tools were employed, each tailored to the language being evaluated. In Python, the pytest framework was utilized, specifically with time profiling enabled through the pytest-benchmark plugin. This tool allows for precise benchmarking of

code, automatically handling multiple iterations and providing detailed statistical outputs such as mean execution time, standard deviation, and percentiles.

For Java, the Java Microbenchmark Harness (JMH) was the tool of choice. JMH is a robust framework specifically designed for measuring the performance of Java programs. It handles common pitfalls in benchmarking, such as warm-up phases and just-in-time (JIT) compiler optimizations, which can significantly affect the accuracy of raw timing data.

In C, I used the Linux command 'perf'. By incorporat- ing it, I was able to move beyond simple execution time measurements to understand how C's low-level optimizations affect hardware performance with variables such as context switches or page faults (K/sec)

Also, the PerfMon tool from Windows was leveraged to gather detailed performance data. PerfMon allows for the monitoring of various system-level metrics, including CPU utilization (% processor time), available memory (measured in bytes), and physical disk access (bytes read and written). This tool offers a system-wide perspective on resource consumption, capturing the underlying impact of matrix multiplication on system components such as the memory hierarchy and I/O operations.

All experiments were conducted on a dedicated machine, with hardware specifications detailed in Table 1, ensuring consistency and accuracy across the benchmarking process.

System Information	**Details**
Operating System Name:	Microsoft Windows 11 Pro
Operating System Version:	10.0.22631
OS Build Type:	Multiprocessor Free
System Manufacturer:	Acer
System Model:	Nitro AN515-58
System Type:	x64-based PC
Processor:	12th Gen Intel(R) Core(TM) i7-12700H, 2.70 GHz
Total Physical Memory:	16.088 GB
Virtual Memory: Maximum Size:	23.512 GB
SSD Storage:	500 GB

Table 1: System Specifications

5 Experiments

To comprehensively assess the performance of matrix multiplication across different programming languages, a series of experiments were conducted using matrices of varying sizes. These experiments ranged from small matrices (e.g., 10x10) to significantly larger ones (e.g., 1000x1000), allowing us to evaluate how execution time, memory usage, and system resource utilization scale with increasing computational demands. By testing multiple matrix sizes, we aimed to provide a thorough understanding of each language's strengths and weaknesses in handling matrix multiplication tasks.

This is the link to my Git Repository of the Project with all the code implemented in the three programming languages.

5.1 Python

The benchmarking of two matrix multiplication functions in Python, one with efficient matrix access and the other with inefficient access, revealed notable performance differences. The experiments were

conducted using relatively small matrix sizes (100, 200, 300, 400, and 500) due to the fact that for matrix size 1000, the inefficient function took too long to complete, making it impractical for these benchmarks.

The efficient function, as expected, consistently outperformed the inefficient one across all tested matrix sizes. The primary difference between the two functions lies in the way they access and manipulate matrix elements. The efficient function leverages contiguous memory access, which improves cache utilization and reduces memory latency, thus speeding up computations. In contrast, the inefficient function exhibits poor memory access patterns, resulting in more cache misses and slower overall performance.

The results from both functions are summarized in Table 2. The efficient function has a mean execution time of approximately 13.35 seconds with a very small standard deviation, highlighting its consistent performance. On the other hand, the inefficient function not only has a higher mean execution time of about 14.73 seconds but also displays greater variability, with a significantly higher standard deviation. These results confirm that memory access patterns can have a profound impact on the performance of computationally expensive tasks like matrix multiplication.

Table 2: Benchmark Results for Efficient and Inefficient Matrix Multiplication Functions

Metric	**Efficient Function**	**Inefficient Function**
Min Time (s)	13.28	13.60
Max Time (s)	13.44	18.84
Mean Time (s)	13.35	14.73
Std. Deviation (s)	0.07	2.30
Median Time (s)	13.35	13.71
Total Time (s)	66.77	73.66
Operations per Second	0.0749	0.0679

One of the key factors that explain the performance difference between the two algorithms is the organization of the inner loops and how they access the matrix elements. In the case of the matrix_multiplication_inefficient function, the loops are organized so that the elements of matrix `b` are accessed column by column in the second loop (`j`), which does not leverage memory locality efficiently. This type of access is costly because the elements of the columns are not contiguous in memory, increasing cache misses and forcing more frequent access to RAM. The pseudocode for the inefficient algorithm is shown below:

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            c[i][j] += a[i][k] * b[k][k]
```

In contrast, the matrix_multiplication_efficient function reorganizes the loops so that the access pattern to matrix elements is more efficient, accessing the elements of `b` row by row via the `k` index before `j`. This approach significantly reduces cache misses, improving memory utilization by the CPU. The pseudocode for the efficient algorithm is shown below:

```
for i in range(n):
    for k in range(n):
        for j in range(n):
            c[i][j] += a[i][k] * b[k][k]
```

For the experiments in Java and C, efficient versions of the matrix multiplication algorithms will be implemented, following the same scheme used in matrix_multiplication_efficient. Since Java and C allow more direct control over memory, these efficient versions will fully leverage data locality, optimizing memory access and reducing execution times. These optimizations are critical to avoid the negative impact that inefficient matrix access would have in lower-level languages like C, where performance is crucial in compute-intensive tasks.

5.2 Java

In this section, I present the results of a naive implementation of the matrix multiplication algorithm, where execution time serves as my primary metric for performance evaluation. Unlike more sophisticated benchmarking approaches, I focused on the straightforward measurement of the time taken to perform matrix multiplications of varying sizes. For this analysis, I chose matrix sizes of 100, 500, 1000, and 1200, which allowed me to clearly observe how execution time escalates with increasing matrix dimensions.

Table 3 summarizes the execution times I observed for each matrix size tested, highlighting the growth in processing time as the matrices become larger. This data offers valuable insight into the computational demands of the naive algorithm, underscoring the challenges associated with matrix operations as their size increases.

Matrix Size	**Time Taken (seconds)**
100	0.004
500	0.125
1000	1.254
1200	4.01

Table 3: Execution time for naive matrix multiplication algorithm.

The benchmarks conducted on the matrix multiplication algorithm in Java reveal significant performance variations based on the size of the matrices being processed. For this analysis, I used the same matrix sizes of 100, 500, 1000, and 1200. The smaller sizes allowed for a practical assessment of performance without excessively long wait times.

The results of the benchmarks are summarized in Table 4, which presents the average execution time per operation across the various matrix sizes tested. The data indicates a clear trend: as the size of the matrices increases, the time per operation also rises steeply. For example, the average execution time for a matrix size of 100 was only 0.674 ms/op, whereas for size 1200, it increased to an average of 6390.616 ms/op. This exponential growth in execution time can be attributed to the inherent computational complexity of the matrix multiplication algorithm, which is $\mathcal{O}(n^3)$.

Matrix Size	**Avg Time (ms/op)**	**Min**	**Max**	**Stdev**	**CI (99.9%)**
100	0.674 ± 0.059	0.657	0.692	0.015	[0.615, 0.732]
500	170.014 ± 28.674	161.346	181.277	7.446	[141.341, 198.688]
1000	1505.572 ± 94.026	1477.226	1534.124	24.418	[1411.546, 1599.598]
1200	6390.616 ± 510.286	6252.981	6583.968	132.520	[5880.329, 6900.902]

Table 4: Benchmark results for matrix multiplication in Java.

5.3 C

In this section, I present the results of a naive benchmark for the implementation of the matrix multiplication algorithm in C, where execution time is used as the key metric for performance evaluation. The inherent efficiency of C, along with its direct memory management, allows for significantly faster execution times compared to the other languages tested.

The analysis focused on matrix sizes of 100, 500, 1000, and 5000, enabling me to observe the rapid increase in execution time as the matrix dimensions grow. The execution time follows the expected $O(n^3)$ pattern due to the complexity of the algorithm, confirming the mathematical foundation of matrix multiplication.

Table 5 summarizes the execution times observed for each matrix size. The results clearly illustrate that the C implementation is significantly faster than both Java and Python. For example, the execution time for a matrix size of 100 is a mere 0.002 seconds, showcasing the superior performance of C for computationally intensive tasks.

Matrix Size	**Time Taken (seconds)**
100	0.002
500	0.270
1000	2.144
5000	293.058

Table 5: Execution time for naive matrix multiplication algorithm in C.

Matrix Size (n)	Task Clock (msec)	CPUs Utilized	Context Switches	Page Faults (K/sec)
500	60.46	0.752	9	25.272
1000	490.52	0.974	40	12.089
2000	5534.05	0.985	400	4.251
5000	95403.61	0.995	7075	1.537

Table 6: Performance metrics for matrix multiplication in C using the ‘perf‘ tool in WSL.

The measurements were obtained using the ‘perf‘ command on Linux via WSL (Windows Subsystem for Linux). This tool allows for the monitoring of various performance metrics, including task execution time (task clock), CPU utilization, context switches, and page faults.

As matrix size increases, we observe the following trends:

1. Task Clock (Execution Time): The execution time scales exponentially with matrix size. For example, a 500x500 matrix takes 60.46 milliseconds, while a 5000x5000 matrix takes 95.4 seconds, reflecting the $O(n^3)$ complexity of matrix multiplication.

2. Context Switches: The number of context switches grows significantly with larger matrix sizes. The smallest matrix (500x500) has only 9 context switches, while the largest (5000x5000) has 7075. This increase suggests higher system overhead due to more frequent task switching and resource management as the matrix size grows.

3. Page Faults: The rate of page faults decreases as the matrix size increases. For a 500x500 matrix, the page fault rate is 25.272 K/sec, while for a 5000x5000 matrix, it drops to 1.537 K/sec. This decline may indicate more efficient memory management by the system, possibly due to better caching or optimizations that reduce the need for frequent memory accesses as the workload increases.

5.4 PerfMon

The use of the PerfMon tool in Windows for benchmarking provided a deeper insight into the performance characteristics of the matrix multiplication algorithm. By collecting metrics such as disk read bytes per second, available memory bytes, and CPU time percentage, I gained a comprehensive understanding of how the algorithm operates under different conditions.

5.4.1 Python

Memory Access	Matrix Size (n)	Disk Read Bytes/sec	Available Memory (Bytes)	CPU Time Percentage
Inefficient	600	162.595,862	4.422.467.584	1.999,144
Inefficient	800	24.987,254	3.942.394.539	1.998,684
Inefficient	900	147.754,328	3.928.561.664	1.998,844
Efficient	600	183.035,157	3.924.727.808	1.998,290
Efficient	800	87.422,534	3.962.191.872	1.998,821
Efficient	900	141.395,501	3.960.506.368	1.999,756

Table 7: Benchmarking Results for Python Matrix Multiplication

5.4.2 Java

Matrix Size (n)	Disk Read Bytes/sec	Available Memory (Bytes)	CPU Time Percentage
1500	143.903,343	4.922.437.632	1.995,700
1750	147.747,599	4.842.574.248	1.996,775
2000	198.248,092	4.850.043.563	1.968,455

Table 8: Benchmarking Results for Java Matrix Multiplication

5.4.3 C

The results obtained for various matrix sizes are summarized in Table 9:

Matrix Size (n)	Disk Read Bytes/sec	Available Memory (Bytes)	CPU Time Percentage
100	155,555.692	6,639,771,648	1.997
500	77,576.447	6,661,339,136	1.997
1000	143,408.465	7,144,732,672	1.924
5000	155,911.316	6,971,872	1.999
10000	4,333,807.674	7,676,071,701	1.944

Table 9: Performance metrics collected using PerfMon for matrix multiplication algorithm.

The data reveals several key observations. First, the disk read throughput fluctuates significantly with matrix size, suggesting that larger matrices may lead to increased disk activity. The available memory consistently remains high, which indicates that the algorithm does not exhaust system memory resources, even for larger matrix sizes.

Additionally, the percentage of CPU time used during the execution remains relatively low across all matrix sizes, with values around 1.9% to 2.0%. This suggests that the algorithm is not fully utilizing the CPU resources available to it.

Given the abundant memory and the low CPU usage, one potential optimization approach would be to adjust the operating system settings to allocate more resources to the C implementation of the algorithm.

This could help to enhance performance by ensuring that the algorithm can take full advantage of the CPU capabilities, potentially leading to significant reductions in execution time.

6 Conclusion

The benchmarking of matrix multiplication across multiple programming languages has provided significant insights into the performance characteristics of Python, Java, and C. The experiments conducted reveal that C exhibits superior execution times, largely due to its low-level memory management and efficient use of system resources. As observed, the naive implementations in both Java and Python performed significantly slower compared to C, particularly for larger matrix sizes, confirming the computational complexity of the naive algorithm which scales with $O(n^3)$.

The use of professional benchmarking tools, such as PerfMon, further highlighted the importance of not only measuring execution time but also understanding resource utilization. The metrics collected allowed for a nuanced analysis of disk activity, available memory, and CPU efficiency, revealing areas where optimizations can be applied. Notably, the low percentage of CPU utilization across all matrix sizes suggests that optimizing resource allocation could lead to performance enhancements.

Although the experiments provided valuable data, they also opened avenues for future research. The implementation of parallel processing and advanced algorithms, such as Strassen's algorithm, could further reduce execution times. Additionally, investigating efficient storage methods for matrices may enhance performance in terms of memory usage.

In summary, the choice of programming language and optimization strategies plays a crucial role in the efficiency of computational tasks such as matrix multiplication. This study lays the groundwork for ongoing exploration in the field of performance benchmarking and optimization, providing a reference point for developers seeking to make informed decisions in algorithm implementation.

7 Future Work

Future work will focus on several key areas to enhance and extend the findings of this experimentation. One potential direction is the implementation of parallelization techniques, such as leveraging multi-threading and GPU acceleration, which could significantly reduce execution time for large matrix multiplications and benefit from all the CPU time execution and memory that is not currently using.

Further improvements could also be achieved by optimizing the matrix multiplication algorithm itself, including techniques to reduce the number of operations, such as Strassen's algorithm. Investigating more efficient ways to store matrices, like sparse matrix representations, could also enhance performance by minimizing memory usage.

References

[1] Abdullah Almurayh. *Improved Matrix Multiplication by Changing Loop Order*, volume 4. Imam Abdulrahman Bin Faisal University, 2022.