Language Benchmark of matrix multiplication

Luna Yue Hernández Guerra
GitHub: github.com/lunahernandez/BigDataIndividualAssignments

October 2024

Abstract

This study investigates the performance of matrix multiplication across three programming languages: Python, Java, and C. The challenge lies in comparing execution times and memory usage of a basic algorithm and its optimized versions. Experiments were conducted using distinct implementations in each language, with Python leveraging the NumPy library, Java employing block and parallel multiplication, and C utilizing loop interchange techniques. The results indicate that before optimization, C provides the best execution time, while Java demonstrates superior memory efficiency. However, optimized Python significantly outperforms both C and Java in execution time for large matrices, such as those of size $n = 1000$. Meanwhile, Java continues to demonstrate better memory efficiency than C, highlighting the importance of selecting the appropriate programming language and optimization strategies based on specific application requirements. This study underscores the need for further exploration of data structures and memory management techniques to enhance performance in high-computational tasks.

Contents

List of Figures

1 Introduction

The performance of programming languages is a critical factor in software development, particularly in applications that demand high computational efficiency, such as matrix multiplication. This mathematical operation is fundamental in areas including artificial intelligence, computer graphics, scientific simulations, and data processing. Given that the basic matrix multiplication operation has a computational complexity of $O(n^3)$, it serves as an excellent benchmark for evaluating the performance of various programming languages.

High-level languages like Python provide ease of use and rapid development but may not always be optimal for performance-intensive tasks. In contrast, lower-level languages such as C allow for greater control over system resources, often resulting in improved execution times and optimized memory usage. Additionally, the problem is further complicated by the need to consider not just raw execution speed but also the memory management techniques utilized by each language.

Furthermore, optimizations such as blocking techniques and parallelization can significantly impact performance, yielding varying results based on the language implementation. This paper aims to compare the performance of a basic matrix multiplication algorithm implemented in Python, Java, and C, alongside various optimizations applied in each language. The objective is to measure and analyze execution time and memory usage relative to matrix size. This study's contribution lies in providing a quantitative evaluation that considers both performance and the context of development, assisting developers in making informed choices when selecting a programming language for computationally intensive applications.

2 Problem Statement

Given the increasing demand for efficient computation in various domains, the problem addressed in this study is to determine how different programming languages (Python, Java, and C) perform in executing matrix multiplication algorithms. This includes analyzing execution time and memory usage while considering the impact of language characteristics and optimization techniques. Understanding these factors will aid developers in selecting the most appropriate language for their specific computational needs.

3 Methodology

To efficiently address the matrix multiplication problem, a systematic approach is crucial for implementing and evaluating various methods. The first step involves separating the production code from the test code, enabling independent unit tests without interfering with the main logic. Additionally, it is essential to parameterize the functions to ensure adaptability to different array sizes and configurations.

This experiment will be conducted using three programming languages: Python, Java, and C. A base version of the matrix multiplication algorithm will be implemented for each language as a reference. Subsequently, optimized methods will be developed and evaluated to improve the performance of the operation.

The comparative analysis will focus on two key performance indicators: execution time in milliseconds and memory usage in MiB (Mebibytes). This will help identify the optimal language and the most effective optimization strategy for various scenarios.

For Python, we developed scripts to automate data handling by reading the JSON files generated by `pytest` during the benchmarking process. These scripts extract relevant data and save it in CSV format, as implemented in `file_converter.py`. The plotting process was also automated using the code in `plotter.py`.

In Java, data visualization begins with `BenchmarkResultProcessor.java`, which converts the JSON output from the benchmarking results into CSV files, extracting the necessary data. The subsequent plotting of results is handled by `BenchmarkPlotter.java` and `MemoryDataPlotter.java`, which create graphs categorized by the respective functions used. The IntelliJ JMH (Java Microbenchmark Harness) plugin was utilized for accurate and reliable performance measurements.

For the C implementation, `benchmark_matrix_multiplication.sh` extracts execution time and memory usage data, writing the results to a CSV file. Subsequently, `gnuplot` is used to visualize these results through graphs.

Overall, the experiments were conducted in different environments: Python was developed using PyCharm, Java was developed in IntelliJ, and C was implemented in a Linux-based virtual machine.

4 Experiments

As previously stated, a number of experiments will be conducted in each language. The results will then be analysed individually to allow for intralinguistic comparisons, before proceeding to interlinguistic comparisons.

4.1 Python

The following functions, located in the file `matrix_multiplication.py`, are utilized for matrix multiplication:

- **matrix_multiplication()** initializes an empty matrix c to store the result of the multiplication and performs matrix multiplication using three nested loops. This is a manual implementation of the algorithm.
- **matrix_multiplication_numpy_dot()** utilizes the `dot()` function from the NumPy library to compute the matrix multiplication. This method is efficient and leverages NumPy's optimized routines.
- **matrix_multiplication_numpy_matmul()** uses the `matmul()` function from the NumPy library for matrix multiplication. This function is recommended over `dot()` for multiplying 2D arrays, as it is specifically optimized for that purpose.

The test section presents the following functions, which perform 100 iterations and 10 rounds. Furthermore, except for the first function, all the others utilize 5 warm-up rounds. According to the results obtained during the experiment, it was concluded that the time cost of the first two functions, which do not have optimized code, is high, as shown in figure 1. For this reason, the matrix sizes $n = 500$ and $n = 1000$ have been omitted. These functions can be found in the file `test_matrix_multiplication.py`.

- **test_matrix_multiplication_1** uses the function *matrix_multiplication()* and runs a benchmark without warm up rounds.
- **test_matrix_multiplication_2** employs the function *matrix_multiplication()* and runs a benchmark with 5 warm up rounds.
- **test_matrix_multiplication_3** works with the function *matrix_multiplication_numpy_dot()*
- **test_matrix_multiplication_4** makes use of the function *matrix_multiplication_numpy_matmul()*

Looking at the graph 1, it can be concluded that the basic matrix multiplication code has an average execution time of more than 80 ms for $n = 100$. On the other hand, the function using NumPy's `matmul` for $n = 1000$ does not exceed this time. Furthermore, as mentioned in the NumPy documentation, although the `dot` (1) and `matmul` (2)functions are intended to perform the same operation, `matmul` shows better performance when working with 2D arrays, as is the case here, so its use is recommended.

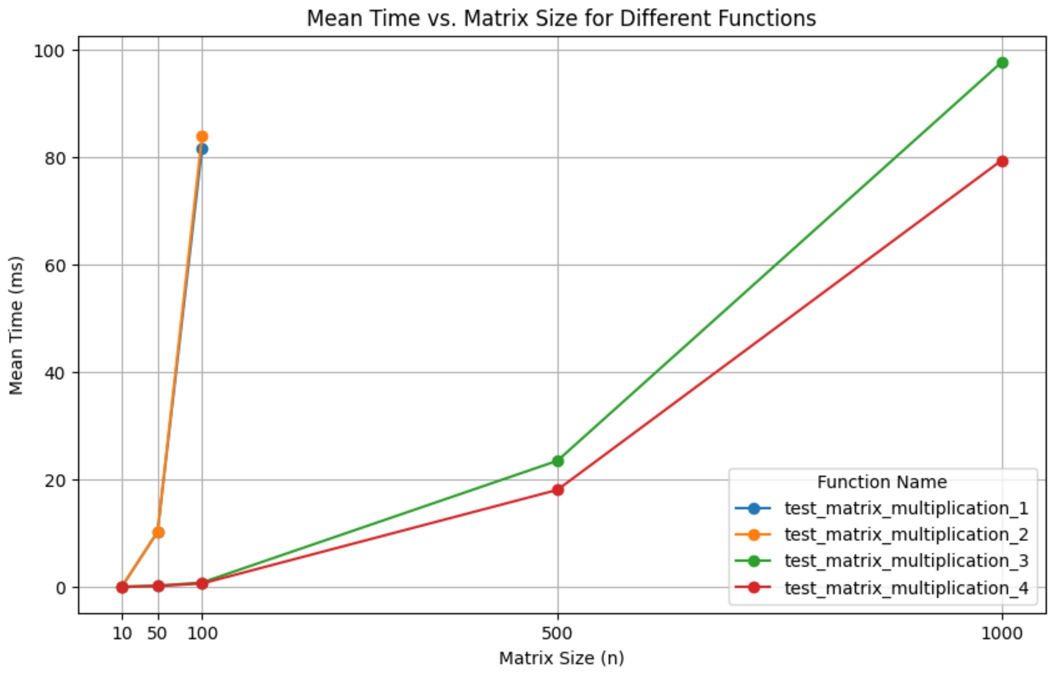

Figure 1: Time Execution Comparison in Python

In terms of memory usage, the same experiments as above are analyzed, specifically presenting data for $n = 500$ and $n = 1000$ in the multiplications performed with `dot` and `matmul`. The memory profiling was conducted using the `memory_profiler` library. By placing the `@profile` decorator above the functions of interest, the memory consumption was tracked line by line. The results were then manually extracted to a CSV file for further analysis. As shown in the graph 2, the three implementations of matrix multiplication for $n = 10, 50$, and 100 exhibit similar memory usage, as indicated by the overlapping lines in the graph. These functions can be found in the file `memory_profiler_matrix_multiplication.py`.

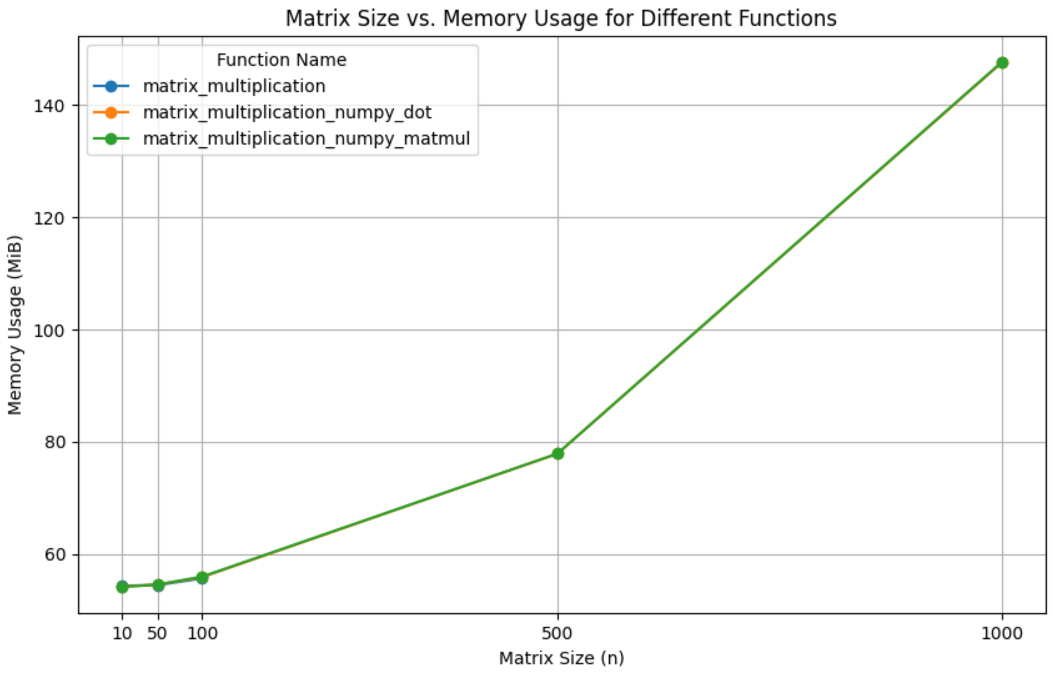

Figure 2: Memory Usage Comparison in Python

Therefore, it can be concluded that matrix multiplication using the `matmul` function from the NumPy library is the most efficient in terms of execution time, while its memory usage is nearly identical to that of the `dot` function.

4.2 Java

The following matrix multiplication functions are implemented in the `MatrixMultiplication.java` file:

- **execute()**: This method performs a standard matrix multiplication using three nested loops. It initializes an empty matrix c to store the result and asserts that the dimensions of matrices a and b are compatible.
- **blockMatrixMultiplication()**: This method implements block matrix multiplication, which divides the matrices into smaller blocks. This approach can improve cache performance. The method iterates over the matrix in blocks of size *block_size* and calls the `multiplyBlock()` method to multiply each block.
- **multiplyBlock()**: A helper method that multiplies a block of matrices a and b and adds the result to matrix c. It iterates through the specified block of indices and calculates the sum for each entry.
- **rowMajorMatrixMultiplication()**: This method performs matrix multiplication in row-major order. It initializes the result matrix c and iterates through the rows and columns of matrices a and b, calculating the sum for each element.
- **columnMajorMatrixMultiplication()**: Similar to the previous method, but it performs the multiplication in column-major order. This approach may yield different performance characteristics depending on the memory access patterns.

And parallel matrix multiplication is implemented in the `ParallelMatrixMultiplication.java` file:

- **multiplyMatricesParallel()** performs matrix multiplication using multiple threads. It divides the workload by assigning a specific range of rows from matrix A to each thread, which calculates the corresponding rows of the result matrix C. After initializing the result matrix, it launches the threads and waits for their completion before returning the final product.

The benchmarking process is implemented in the `MatrixMultiplicationBenchmarking.java` file. This file contains the following functions designed to evaluate different approaches to matrix multiplication:

- **multiplication1** implements the basic matrix multiplication algorithm, serving as the baseline for performance comparison.
- **multiplication2** utilizes block matrix multiplication, which divides the matrices into smaller submatrices, potentially improving cache efficiency and performance.
- **multiplication3** implements row-major matrix multiplication, which accesses elements in a row-wise manner, optimizing memory access patterns for certain systems.
- **multiplication4** uses column-major matrix multiplication, accessing elements in a column-wise manner, which may benefit specific data layouts.
- **multiplication5** performs parallel matrix multiplication using threads, enabling simultaneous computation and potentially reducing execution time for larger matrices.

For the benchmarking process, the average time mode was established, with the unit of measurement set to milliseconds and utilizing 1 fork.

As shown in Figure 3, the red line represents the basic multiplication method, positioned in the middle of the other results. Notably, the row-major (multiplication 3) and column-major (multiplication 4) approaches exhibit the highest average execution times in this context. Conversely, the block multiplication and parallel multiplication methods demonstrate the lowest execution times, indicating their efficiency in handling matrix operations.

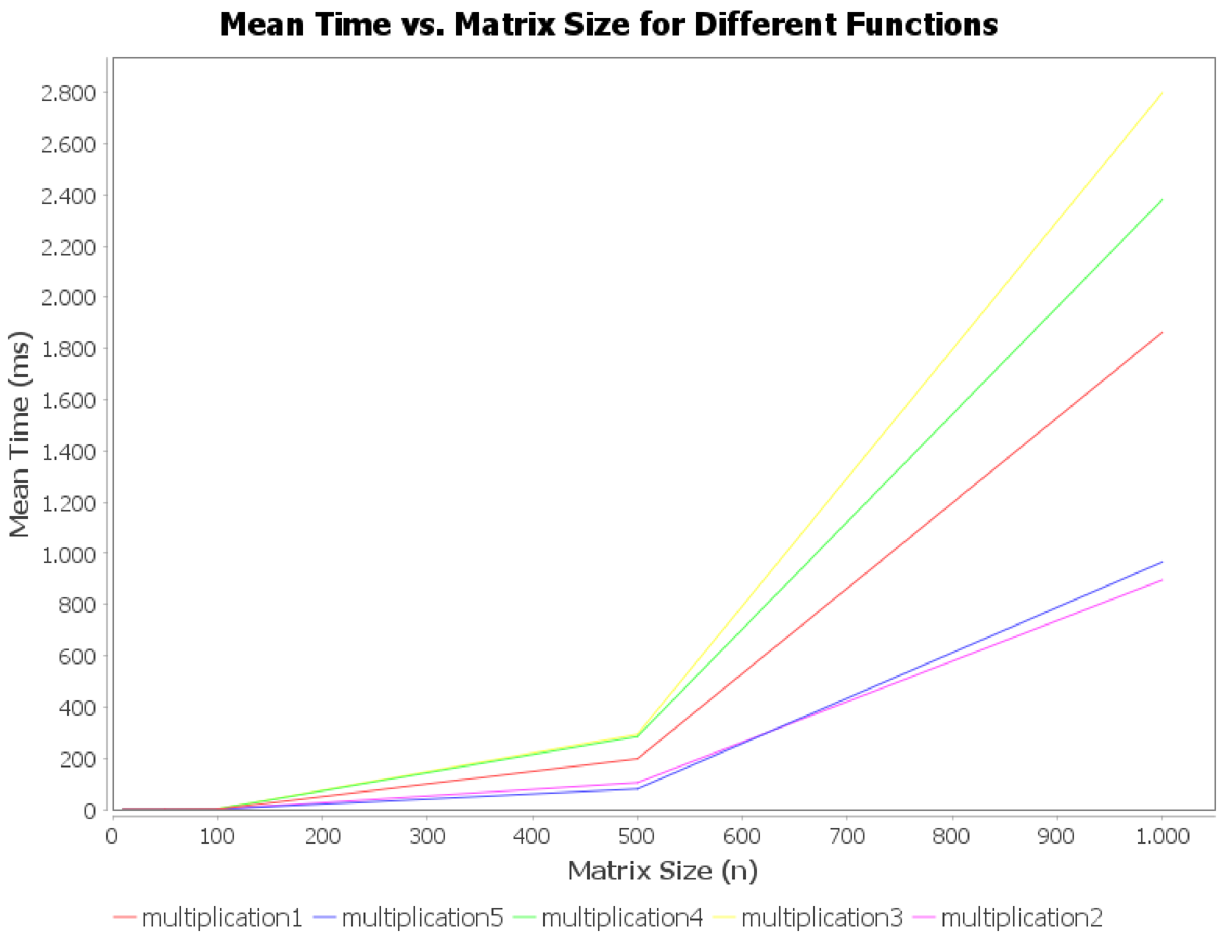

Figure 3: Time Execution Comparison in Java

In terms of memory usage, this time it has been integrated into the same code as the benchmarking process. The methodology involves calculating memory usage in MiB and storing the results in a CSV file for subsequent processing.

According to Figure 4, it is evident that, in terms of memory efficiency, the column-major multiplication outperforms the row-major approach significantly. Notably, at higher dimensions such as $n = 1000$, the column-major method yields the best results. Following closely are the block multiplication and the parallel multiplication methods, which also demonstrate commendable memory efficiency.

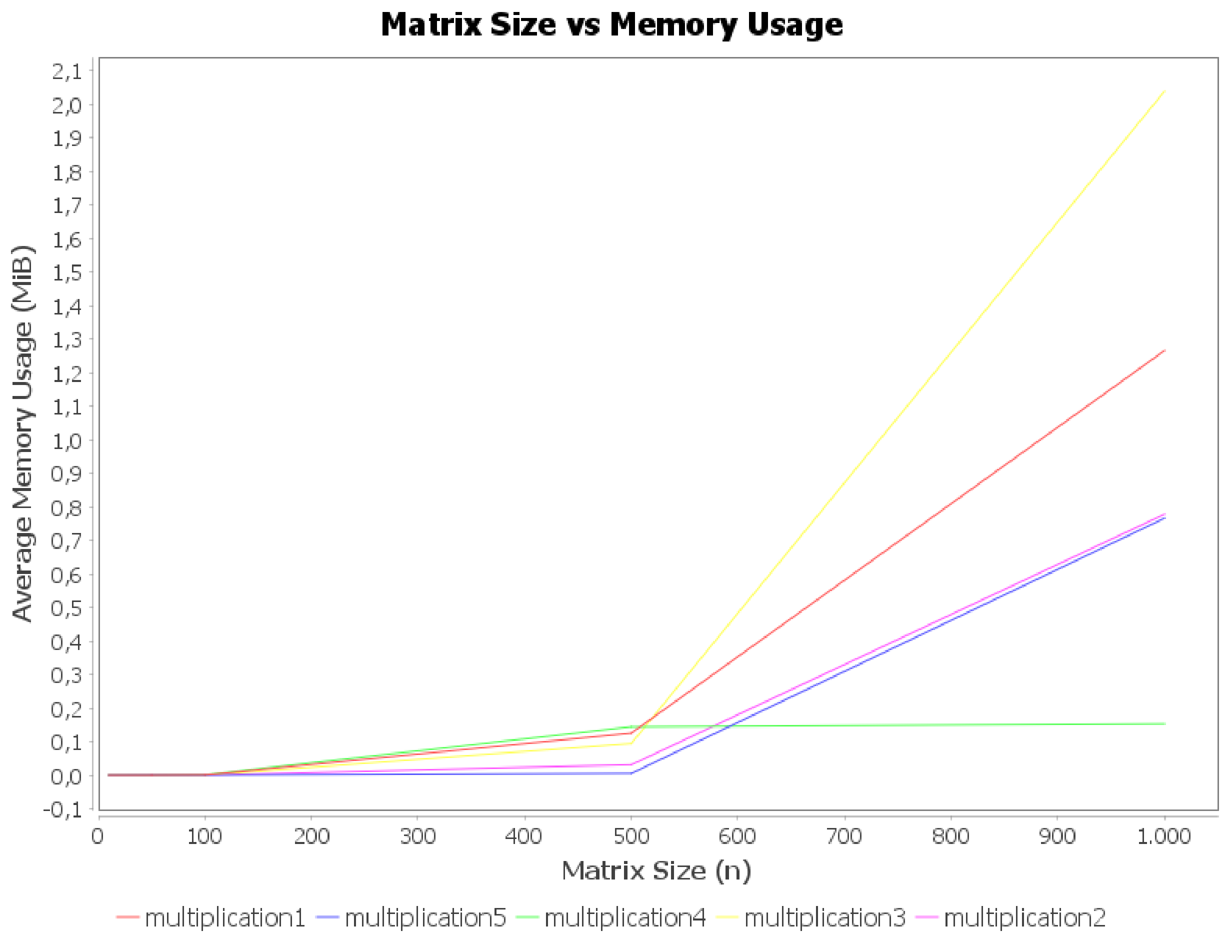

Figure 4: Memory Usage Comparison in Java

In conclusion, the most effective methods for balancing execution time and memory usage in Java are the block multiplication method and the parallel multiplication using threads. These approaches not only enhance performance but also optimize resource utilization, making them suitable choices for large-scale matrix operations.

4.3 C

The functions used for matrix multiplication in C are as follows:

- **matrix.c** serves as the code base for all matrix operations, but is not included in the evaluation.
- **matrix1_parametrization.c** is the basic matrix multiplication, which accepts parameters for matrix dimensions and values.
- **matrix2_parametrization.c** is a variation of the basic multiplication in which the loops are reordered. Specifically, the innermost loop is swapped with the middle loop, which can affect caching performance and execution speed.
- **matrix3_parameterization.c** implements block matrix multiplication, where the matrix is split into smaller sub-matrices (blocks) to optimize memory access patterns, improving performance on larger matrices.

It can be observed that the function `matrix2_multiplication` achieves better results in terms of execution time compared to other implementations, as illustrated in Figure 5. This improvement is attributed to the reorganization of the loops in the code, where the iteration is done over the rows of the matrix instead of the columns.

By changing the order of the loops, unnecessary access to entire columns of the matrix is avoided, which also contributes to a reduction in the overall execution time.

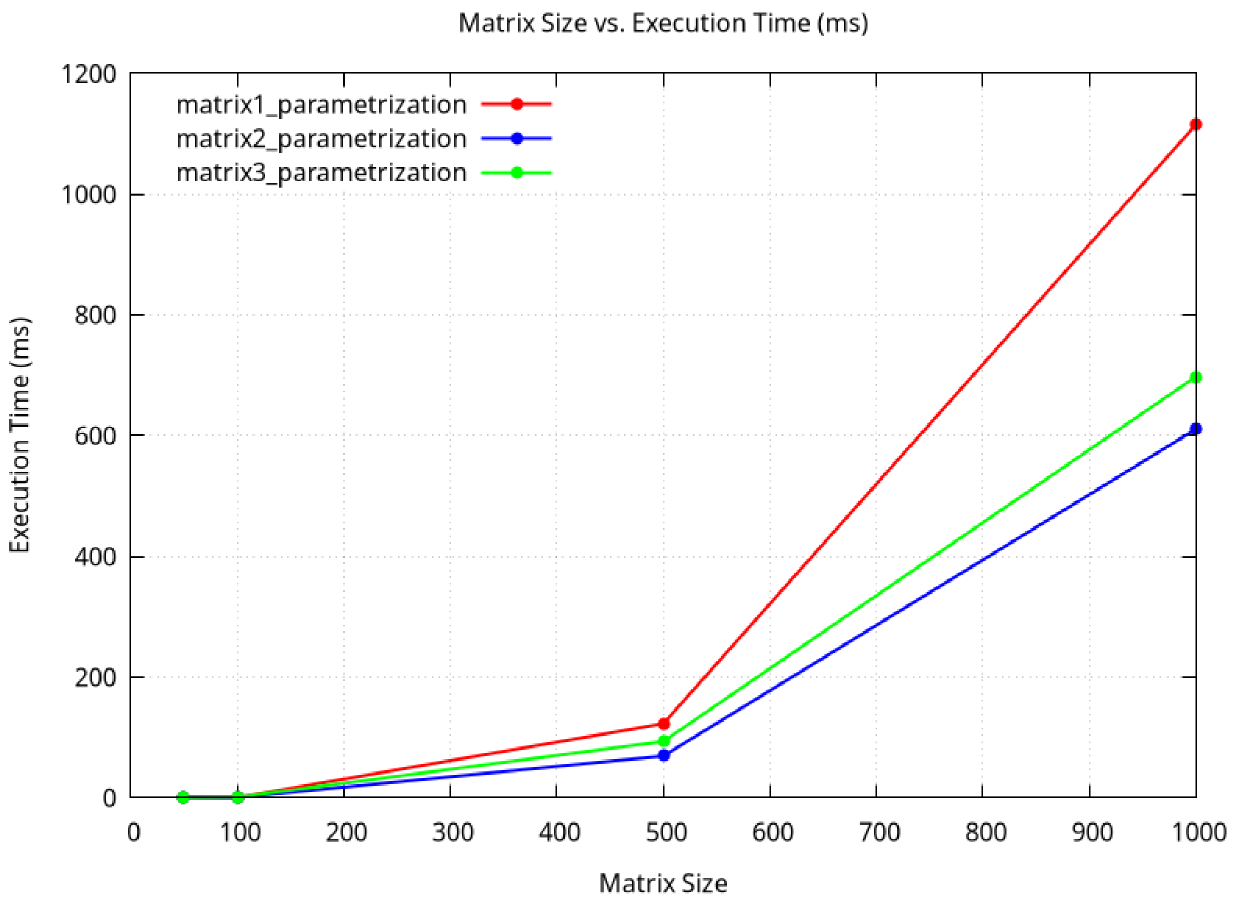

Figure 5: Time Execution Comparison in C

Furthermore, as shown in Figure 6, the memory usage across the three matrix multiplication approaches is quite similar. This similarity suggests that while the execution time varies significantly due to the different loop arrangements, the memory overhead remains consistent, indicating that all three methods operate within comparable memory constraints.

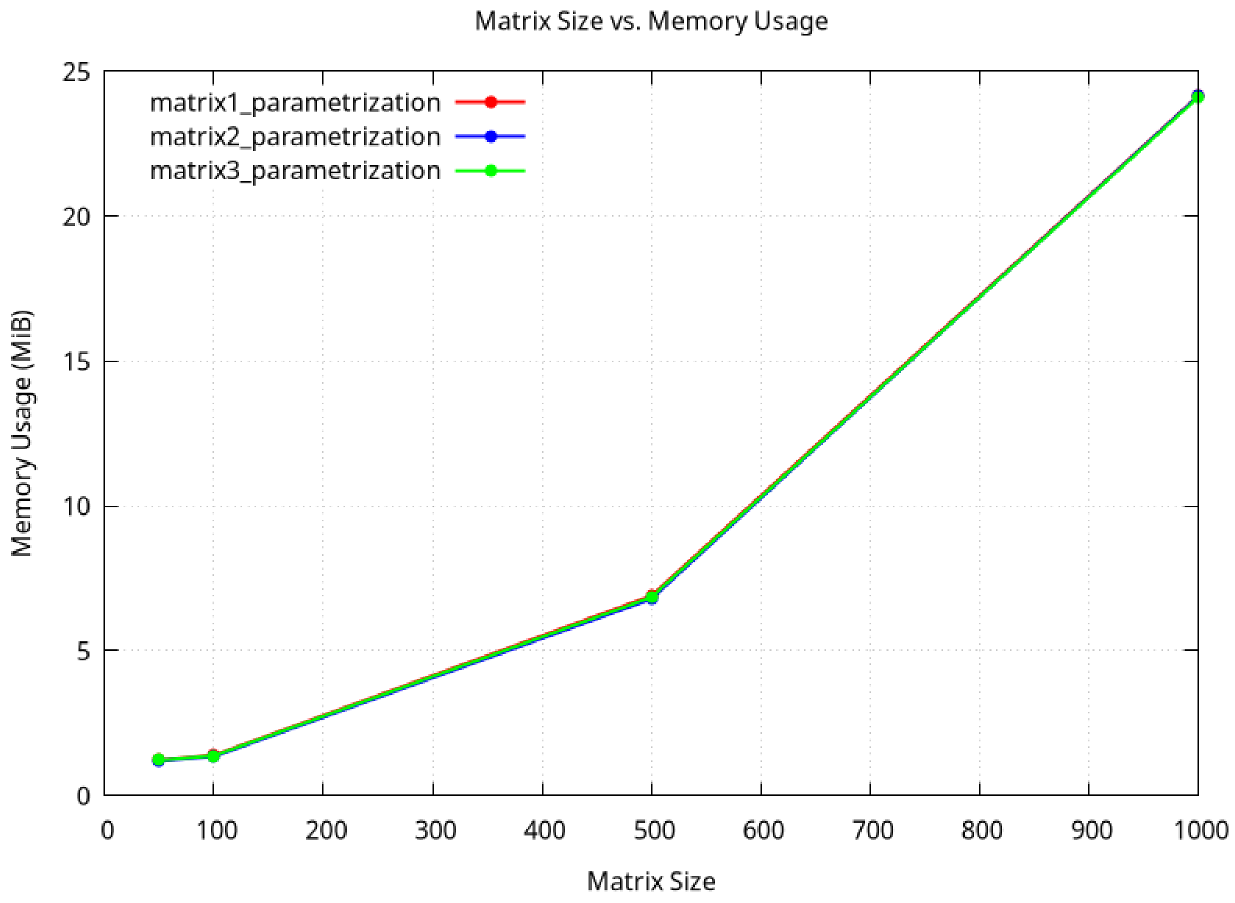

Figure 6: Memory Usage Comparison in C

In conclusion, among the three proposed methods for performing matrix multiplication in C, the most efficient approach is to interchange the order of the loops. By accessing an entire row first, this method minimizes unnecessary calculations and optimizes memory access patterns, ultimately leading to improved execution times.

4.4 Comparison Between Languages

The performance of matrix multiplication varies significantly between Python, Java, and C due to differences in language design, optimization strategies, and memory management.

4.4.1 Execution Time

In terms of execution time, prior to any optimizations, the results indicate that C offers the fastest performance. Java follows closely, while Python lags behind. However, when optimizations are applied, the performance landscape shifts significantly.

The C implementation benefits from loop interchange. Java utilizes parallel and block multiplication techniques. Meanwhile, Python leverages the efficiency of the NumPy library. As a result, the optimized Python implementation can outperform both C and Java in certain scenarios, particularly with larger matrix sizes. This trend is evident in Tables 1 and 2.

4.4.2 Memory Usage

The memory usage analysis before optimization shows that Java has the lowest consumption, starting at 0.0000 MiB for a matrix size of 10 and reaching 1.2652 MiB for size 1000, benefiting from efficient memory allocation. C follows with memory usage beginning at 0.00117 MiB for size 10 and rising to 24.1500 MiB for size 1000, as it incurs overhead with larger matrices. In contrast, Python exhibits the highest memory consumption, starting at 54.4 MiB for size 10 and increasing to 55.7 MiB for size 100.

After optimization, Java continues to demonstrate the least memory usage, with minimal consumption at 0.0000 MiB for size 10 and 0.7778 MiB for size 1000. Python, using the NumPy library, maintains stable usage, from 54.2 MiB for size 10 to 147.6 MiB for size 1000, reflecting its efficient handling of array operations. C's memory usage shows a significant increase, from 1.10 MiB for size 10 to 24.14 MiB for size 1000, indicating greater overhead. Overall, Java remains the most memory-efficient, while Python's usage stays stable, and C experiences a more substantial increase in consumption. These details can be found in Tables 1 and 2.

Table 1: Execution Time and Memory Usage for Matrix Multiplication in Python, Java, and C (before optimization)

Language	**Function**	**Matrix Size**	**Exec. Time (ms)**	**Mem. Used (MiB)**
Python	Basic Multiplication	10	0.0968	54.4
Python	Basic Multiplication	50	10.2312	54.5
Python	Basic Multiplication	100	81.7493	55.7
Java	Basic Multiplication	10	0.0012	0.0000
Java	Basic Multiplication	50	0.0983	0.0001
Java	Basic Multiplication	100	0.8944	0.0009
Java	Basic Multiplication	500	197.6877	0.1256
Java	Basic Multiplication	1000	1863.2659	1.2652
C	matrix1_parametrization	10	0.00019	0.00117
C	matrix1_parametrization	50	0.00020	0.00123
C	matrix1_parametrization	100	0.00027	0.00140
C	matrix1_parametrization	500	122.8363	6.9000
C	matrix1_parametrization	1000	1115.3391	24.1500

Table 2: Execution Time and Memory Usage for Matrix Multiplication in Python, Java, and C (after optimization)

Language	**Function**	**Matrix Size**	**Exec. Time (ms)**	**Mem. Used (MiB)**
Python	NumPy Matmul	10	0.0122	54.2
Python	NumPy Matmul	50	0.1783	54.6
Python	NumPy Matmul	100	0.6297	56.0
Python	NumPy Matmul	500	18.0659	77.9
Python	NumPy Matmul	1000	79.4340	147.6
Java	Block Multiplication	10	0.0010	0.0000
Java	Block Multiplication	50	0.0889	0.0001
Java	Block Multiplication	100	0.7872	0.0002
Java	Block Multiplication	500	104.3046	0.0314
Java	Block Multiplication	1000	896.1918	0.7778
Java	Parallel Multiplication	10	0.2790	0.0000
Java	Parallel Multiplication	50	0.3204	0.0003
Java	Parallel Multiplication	100	0.6783	0.0002
Java	Parallel Multiplication	500	80.2817	0.0055
Java	Parallel Multiplication	1000	966.6094	0.7659
C	matrix2_parametrization	10	0.00041	1.10

Language	Function	Matrix Size	Exec. Time (ms)	Mem. Used (MiB)
C	matrix2_parametrization	50	0.00023	1.22
C	matrix2_parametrization	100	0.00022	1.35
C	matrix2_parametrization	500	69.2721	6.79
C	matrix2_parametrization	1000	610.8760	24.14

5 Conclusion

This study addressed the challenge of evaluating the performance of matrix multiplication across three programming languages: Python, Java, and C. Matrix multiplication is a fundamental operation in many computational applications, making it essential for developers to understand the performance implications of different programming languages for efficient data processing solutions.

The methodology involved implementing a basic matrix multiplication algorithm in each language and applying various optimization techniques, such as blocking and parallelization. Execution time and memory usage were measured for different matrix sizes, allowing for an accurate assessment of each implementation's performance characteristics.

Initially, C demonstrated the fastest execution times, followed closely by Java, while Python lagged behind. However, after applying optimizations, the performance dynamics changed significantly. The optimized Python implementation, using the NumPy library, occasionally outperformed both C and Java, especially with larger matrix sizes.

In terms of memory usage, Java consistently exhibited the lowest consumption, while Python maintained stable usage, and C showed a noticeable increase with larger matrix sizes. Overall, the results underscore the importance of considering execution time and memory efficiency when selecting a programming language for matrix multiplication tasks. This analysis provides valuable insights for developers aiming to optimize performance in computationally intensive applications.

6 Future Work

Future research can explore alternative data structures, such as sparse matrices and block matrices, to assess their performance across different programming languages. This exploration could yield valuable insights into optimization strategies.

Additionally, a more detailed evaluation of memory management techniques and their effects on performance in real-world applications would deepen the understanding of effective optimization strategies for high-performance computing tasks.

7 Bibliography and References

References

[1] N. Developers. numpy.dot. Accessed: October 7, 2024. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.dot.html#numpy.dot

[2] ——. numpy.matmul. Accessed: October 7, 2024. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.matmul.html#numpy.matmul