Parallel and Vectorized Matrix Multiplication

Alonso León, María

Data Science and Engineering

Big Data - ULPGC

GitHub Repository

November 29, 2024

Abstract: Matrix multiplication is a critical operation in computational science, underpinning fields such as artificial intelligence, image processing, and scientific simulations. However, its computational intensity, particularly for large matrices, poses significant challenges. This study addresses these challenges by implementing and benchmarking various optimization techniques, including parallelization and vectorization, across different matrix sizes and thread configurations.

The experimentation was conducted using Java, leveraging multithreading, vectorization with SIMD instructions, and a modular architecture, alongside comparative analyses with Python implementations. Metrics such as speedup, efficiency, memory usage, and CPU utilization were evaluated, ensuring a comprehensive performance assessment. Notably, the VectorizedSIMD algorithm emerged as the superior approach, delivering the highest speedup and lowest execution time per operation across all scenarios. Parallel algorithms, while effective for larger matrices, exhibited diminishing returns due to thread synchronization and memory contention issues.

These findings underscore the efficacy of hardware-level optimizations, particularly SIMD instructions, in overcoming the computational demands of matrix multiplication. This work provides valuable insights into the design and application of optimized algorithms, contributing to improved performance in computationally intensive domains.

Key words: Matrix multiplication, optimization, parallelization, vectorization, SIMD, Java, Python, benchmarking, computational efficiency, multithreading.

1 Introduction

In recent years, the exponential growth in data generation has driven the need for advanced Big Data techniques to process and analyze vast datasets efficiently. Among these techniques, matrix multiplication has emerged as a fundamental operation, crucial in numerous domains such as scientific simulations, artificial intelligence, and even video game development. The computational demand of this operation, especially for large-scale matrices, necessitates the development and adoption of optimized algorithms. This study focuses on two significant approaches to enhance matrix multiplication performance: parallelization and vectorized algorithms.

Matrix multiplication is computationally intensive due to its inherent complexity, involving a large number of floating-point operations. To address this, several optimization strategies have been proposed in recent decades. These include leveraging parallel computing frameworks such as multi-threading or libraries like OpenMP, as well as vectorization techniques that utilize SIMD (Single Instruction, Multiple Data) instructions to process multiple data points simultaneously.

Parallelization and vectorization present distinct theoretical advantages. Parallelization exploits the computational power of multi-core processors, dividing tasks into smaller units that can be executed concurrently. On the other hand, vectorization takes advantage of hardware-level capabilities to process data in bulk, potentially reducing execution times significantly. The combination of these approaches is particularly relevant in image processing, where matrix multiplication forms the backbone of various filtering, transformation, and feature extraction techniques.

This paper is based on a university-level assignment that explores the implementation and comparison of different optimization strategies for matrix multiplication. The primary analysis is conducted using Java, leveraging its threading and vectorization capabilities. Additionally, a comparative analysis with Python is performed to assess cross-language performance differences. A range of metrics is used to quantify the results, including speedup, efficiency, memory usage, CPU utilization, and operations per millisecond. Experiments are conducted using matrices of various sizes and thread counts to ensure a comprehensive evaluation.

By offering a rich comparative analysis of parallel and vectorized methods, this work not only highlights the practical implications of these optimization techniques but also aims to contribute valuable insights into their applicability in real-world

scenarios. It builds upon existing knowledge while addressing gaps such as the lack of detailed experimental evaluations in the literature. The findings underscore the importance of tailored optimization strategies in computationally demanding fields, emphasizing the relevance of both theoretical and practical considerations in algorithm design.

2 Problem Statement

Matrix multiplication, a fundamental operation in computational science, faces critical challenges when performed at scale. The inefficiency of basic algorithms becomes apparent as matrix sizes grow, demanding substantial computational resources and time. In real-world applications such as image processing, where matrices often represent pixel data or transformations, the need for fast and efficient algorithms is paramount. Basic methods struggle to meet these demands due to limitations in speed and resource usage, especially in environments requiring immediate results.

From a theoretical perspective, the computational complexity of matrix multiplication using the naive algorithm is $\mathcal{O}(n^3)$, where n is the dimension of the square matrices. This approach involves calculating each element of the resultant matrix as the dot product of a row from the first matrix and a column from the second. For large matrices, this quickly becomes impractical due to the exponential growth in the number of operations required. Optimizations, therefore, are essential to mitigate these limitations.

Parallelization and vectorization provide two prominent strategies to enhance performance. Parallel algorithms distribute the computation across multiple processor cores, theoretically reducing execution time proportional to the number of threads. However, challenges such as thread synchronization, memory contention, and diminishing returns with increasing threads pose significant obstacles. Similarly, vectorized algorithms leverage SIMD instructions to process multiple elements simultaneously, offering a hardware-level optimization. Nonetheless, implementing vectorized methods requires careful alignment of data and a deep understanding of hardware-specific constraints.

Conducting a fair comparison between these optimization approaches presents an additional layer of complexity. Differences in hardware configurations, memory bandwidth, and implementation details can significantly influence the results, making it challenging to draw generalizable conclusions. Moreover, resource limitations such as the number of available cores or memory capacity further constrain the scope of experimentation, particularly in academic or low-budget environments.

This problem is especially relevant in fields like Big Data and image processing, where matrix multiplication is ubiquitous. Addressing it is not only vital for improving the performance of individual applications but also for contributing to broader optimization strategies in computational science.

The naive matrix multiplication algorithm calculates each element of the result matrix C as:

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \cdot B[k][j]$$

where A, B, and C are $n \times n$ matrices. This involves n^3 scalar multiplications and a similar number of additions, leading to its $\mathcal{O}(n^3)$ complexity.

Optimized approaches attempt to reduce this complexity or distribute the workload. For instance:

- **Strassen's Algorithm**: Reduces the complexity to approximately $\mathcal{O}(n^{2.81})$ using divide-and-conquer techniques.
- **Sparse Matrix Multiplication**: Leverages the sparsity of matrices (many zero entries) to skip unnecessary computations.
- **Parallelism**: Divides the computation into independent tasks, which are executed concurrently, with an ideal speedup of T_1/T_p, where T_1 is the time taken by a single thread and T_p by p threads.

This study aims to explore these optimizations, focusing on practical implementations of parallel and vectorized methods, and to provide a detailed analysis of their performance, resource usage, and scalability. By addressing the outlined challenges, this work contributes to advancing efficient matrix computation techniques, particularly in scenarios requiring immediate and reliable results.

3 Methodology

This section outlines the methodology employed to implement, execute, and analyze various matrix multiplication algorithms, including their parallel and vectorized versions. The methodology is designed to ensure a comprehensive evaluation of performance across different matrix sizes, thread counts, and optimization strategies.

The experiments were primarily conducted in Java, leveraging a modular architecture built with the Maven framework. The implementation is structured as follows:

- In the main/java directory, two interfaces were created to define the operations for different types of algorithms:

  - **Parallel Algorithms Interface:** Parallel Algorithms Interface: This interface includes a method multiply(double matrixA, double matrixB, int numThreads) that accommodates the parallelization aspect by specifying the number of threads.
  - **Sequential and Vectorized Algorithms Interface:** In this interface, the method multiply(double matrixA, double matrixB) is provided, which excludes the thread parameter as it is not relevant for sequential or vectorized operations.
  - Multiple implementations of these interfaces, covering both parallel and vectorized algorithms, were developed. These will be discussed in greater detail in the following subsection.

- Additionally, in the test directory, rigorous benchmarking and resource monitoring tests were implemented:
  - The Java Microbenchmark Harness (JMH) library was used to ensure accurate and reliable benchmarking of the algorithms.
  - Resource usage metrics, such as CPU and memory consumption, were tracked using specialized libraries like oshi.hardware.CentralProcessor and oshi.SystemInfo.

To complement the Java implementation, an optional Python version was also developed. This implementation mirrors the structure of the Java solution, with analogous algorithms to facilitate cross-language comparisons. Furthermore, additional algorithms specific to Python were included to enrich the analysis, as will be detailed in the next subsection.

This modular and systematic design ensures scalability and maintainability of the codebase while enabling precise performance measurement and reproducibility of results.

3.1 Implementation of Algorithms

As it was mentioned before, a wide range of algorithms were implemented in Java. These include both sequential and optimized implementations, such as parallel algorithms leveraging multi-threading and vectorized algorithms utilizing SIMD capabilities. Each algorithm was designed with specific optimization techniques to address different computational challenges. The following list summarizes their implementations.

- **Basic Matrix Multiplication:** Simple and straightforward implementation of matrix multiplication using a sequential approach. Each element of the result matrix is calculated by multiplying the rows of the first matrix with the columns of the second and summing the products. This algorithm serves as a base for comparing the efficiency of other, more advanced implementations.
- **Parallel Algorithms:**
  - AtomicMatrixMultiplication: This algorithm uses atomic operations to ensure safe synchronization when multiple threads attempt to access and modify the shared variable row concurrently. Despite using a concurrency control technique, it follows the basic matrix multiplication pattern but with the particularity of handling thread access atomically.
  - FixedThreadsMatrixMultiplication: This implementation divides the multiplication task among a fixed number of threads using an ExecutorService. Each thread calculates a set of rows of the result matrix. It ensures that all threads complete their work before returning the result using the awaitTermination function.
  - MatrixMultiplicationThreads: In this approach, the work is manually distributed among a configurable number of threads. Each thread is responsible for calculating a portion of the rows of the result matrix, dividing the rows evenly. It ensures that all threads complete their task before finishing.
  - SynchronizedMatrixMultiplication: This algorithm manages access to shared resources using a semaphore and an atomic counter. Threads request permission to operate on rows of the matrix in a controlled manner, preventing multiple threads from modifying the same element simultaneously. This ensures correct execution even when multiple concurrent operations are performed.
  - ExecutorMatrixMultiplication: Uses Java's Executor framework to manage a fixed thread pool that calculates the rows of the result matrix in parallel. Each task is assigned to a thread within the pool, and synchronization of operations is automatically handled through this mechanism. Using an ExecutorService allows for efficient thread management without the need to create and control threads manually.
  - ParallelOpenMPMatrixMultiplication: Implements matrix multiplication in parallel using Java's ForkJoinPool, emulating the behavior of the OpenMP model in C/C++. It uses recursive tasks to divide the work into smaller blocks, enabling efficient parallel processing. The threshold determines when to recursively divide the work to optimize performance.

- **Vectorized and Sequential Algorithms:**
  - VectorizedMatrixMultiplication: This algorithm uses vectorized techniques to multiply matrices. Instead of performing multiplications sequentially, the multiplication and summation operations are vectorized, allowing multiple data to be processed simultaneously and increasing execution speed.
  - OptimizedVectorizedMatrixMultiplication: This algorithm optimizes multiplication by using a ”vectorized” version of the operation. Instead of operating on a single value during each iteration, the concept of vectors is leveraged, where multiple operations are performed simultaneously. This improves computational efficiency and is particularly useful when SIMD instructions are available.
  - StreamMatrixMultiplication: Leverages Java streams to perform matrix multiplication in parallel. Each thread handles a row of the matrix, and the system's parallel processing capability is used to perform the calculations more efficiently, improving performance on multi-core systems. Although this algorithm is parallel, it is included for the analysis in the sequential section since the threads are managed automatically, it is something to keep in mind when viewing the analysis.
  - VectorizedSIMDMatrixMultiplication: An advanced version of matrix multiplication that uses SIMD (Single Instruction, Multiple Data) instructions to perform operations in parallel. It uses the jdk.incubator.vector API to leverage the hardware's vector capabilities, significantly improving performance by operating on multiple data simultaneously.

The Python implementation includes similar parallel and vectorized approaches, along with unique additions, such as GPU-based matrix multiplication using CUDA. This diversity of algorithms allows for a comprehensive and exhaustive comparison across languages, optimization strategies, and computational paradigms. The inclusion of GPU-based techniques further extends the scope of the analysis, demonstrating the potential benefits of hardware acceleration. These are the algorithms implemented in python:

- **Basic Matrix Multiplication**: Implements standard matrix multiplication using three nested loops. This is the classic, simple-to-understand approach, but generally inefficient for large matrices.
- **Parallel Algorithms**:
  - SynchronizedMatrixMultiplication: Uses a semaphore to control thread access to resources and synchronizes access to the matrix rows. This is useful to prevent threads from interfering with each other when modifying the result.
  - ExecutorMatrixMultiplication: This class uses the ThreadPoolExecutor module from concurrent.futures to parallelize matrix multiplication across multiple threads. Each thread computes a row of the result matrix.
  - CudaMatrixMultiplication: Uses the GPU to accelerate multiplication through CUDA. It is optimized for large matrices and takes advantage of the massive parallelism offered by GPUs.
  - MatrixMultiplicationThreads: Uses manual thread creation (threading.Thread) to divide the matrix multiplication task into multiple threads, controlling the range of rows processed by each thread.
  - ParallelMatrixMultiplicationOpenMP: Implements a parallelization approach that mimics the concept of OpenMP, using threads to divide the multiplication task into smaller blocks and processing them in parallel.
  - AtomicMatrixMultiplication: This class uses an approach based on sequential processing, it handles rows atomically.
- **Vectorized and Sequential Algorithms**:
  - OptimizedVectorizedMatrixMultiplication: Optimizes the multiplication process using Numba to compile matrix multiplication functions and perform low-level loop optimizations.
  - VectorizedMatrixMultiplicationSIMD: Uses NumPy vectorized operations to efficiently compute the matrix product. By using the transposition of the second matrix and vectorized matrix multiplication operations, this is one of the most efficient implementations.
  - StreamMatrixMultiplication: Uses an approach similar to Java's streams, where the rows of the matrix are distributed among multiple threads to perform multiplication concurrently.

3.2 System Configuration

The experiments were conducted on a Windows 11 machine equipped with:

- 32 GB RAM
- 10 physical cores and 16 logical processors
- Execution environment: IntelliJ IDEA with Java's built-in benchmarking tools.
- Additionally, Python experiments were conducted using PyCharm as the development environment.

3.3 Experimental Design

Experiments were performed using square matrices of varying sizes: 64, 128, 256, 512 and 1024. For parallel algorithms, thread counts of 2, 4, 8, and 16 were tested.

Performance metrics metrics for Java were evaluated using the Java Microbenchmark Harness (JMH), configured with two warm-up iterations and five execution iterations for statistical robustness. In Python, analogous tests were conducted using specific libraries such as os, platform, psutil, json, threading, and time to monitor performance and resource utilization.

The evaluation criteria included:

- **Speedup:** The ratio of execution time between the baseline (sequential, in this case the basic algorithm) and optimized (parallel) algorithms, defined as:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}}$$

  where $T_{\text{sequential}}$ is the execution time of the baseline algorithm and T_{parallel} is the execution time of the parallel algorithm.

- **Efficiency:** The ratio of speedup to the number of threads, defined as:

$$\text{Efficiency} = \frac{\text{Speedup}}{N_{\text{threads}}} = \frac{T_{\text{sequential}}}{T_{\text{parallel}} \cdot N_{\text{threads}}}$$

  where N_{threads} is the number of threads used in the parallel algorithm.

- **Milliseconds per operation (ms/ops):** A measure of the time per matrix operation, calculated as the total execution time divided by the number of operations performed.
- **Memory and CPU Usage:** Evaluated separately through single-run experiments.

3.4 Comparison of Algorithms

For each algorithm, the best-performing configuration (e.g., thread count for parallel methods) was selected for comparative analysis. Separate comparisons were made between:

- Different parallel algorithms.
- Different vectorized algorithms.
- The overall performance of parallel versus vectorized approaches.

To ensure fair comparisons, JMH automatically manages the isolation of experiments and mitigates environmental interference, such as background processes.

3.5 Visualization and Analysis

The results are primarily presented through graphs, detailing the performance metrics across varying matrix sizes and thread counts. Patterns such as scalability, memory efficiency, and CPU utilization are analyzed to derive meaningful insights. These visualizations will be discussed further in the *Experiments* section.

3.6 Reproducibility

To ensure reproducibility, the full implementation is available on GitHub. The repository includes:

- Source code for all algorithms.
- Instructions for setting up the benchmarking environment.
- JSON files containing the results obtained during the tests.

By leveraging rigorous benchmarking tools and making the code publicly available, this methodology ensures both transparency and the ability to replicate findings, making it a robust foundation for evaluating matrix multiplication optimizations.

4 Experiments

4.1 SpeedUp and Efficiency - Parallel Algorithms

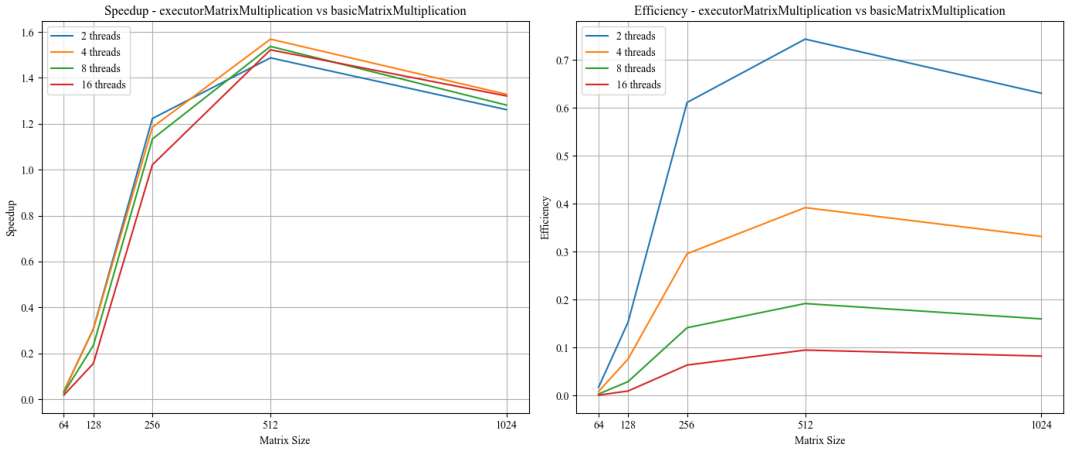

Figure 1: Speedup and Efficiency for ExecutorMatrixMultiplication Algorithm

The goal of this experiment was to evaluate the performance of the ExecutorMatrixMultiplication algorithm by measuring its speedup and efficiency across different matrix sizes and thread counts (2, 4, 8, and 16).

The results are presented in two line graphs. The first graph shows speedup for varying thread counts, while the second displays the efficiency.

On the one hand, the initial exponential growth in speedup suggests that increasing the number of threads has a significant impact on performance for smaller matrices (up to 256). However, as the matrix size increases, the rate of speedup growth decreases, and a slight decline is observed at larger sizes (1024). This can be attributed to factors such as thread contention, memory access latency, and diminishing returns from adding more threads as the matrix becomes large enough to saturate the available CPU resources. The closely grouped lines suggest that the algorithm benefits similarly from increasing the number of threads.

On the other hand, the efficiency behaves inversely to speedup, as expected. The highest efficiency is achieved with fewer threads, indicating that the overhead of managing more threads may outweigh the benefits when using a high thread count. The noticeable separation between the lines in the efficiency graph suggests that the parallel execution is less efficient with increasing threads, which is likely due to increased contention for resources and overheads associated with synchronization and context switching.

To sum up, the ExecutorMatrixMultiplication algorithm shows promising speedup with increased threads up to a point, with diminishing returns for larger matrix sizes. Efficiency decreases as the number of threads increases, with the lowest efficiency observed when using 16 threads. This indicates that, for this particular algorithm and hardware configuration, increasing the number of threads beyond a certain point introduces significant overhead that limits overall efficiency.

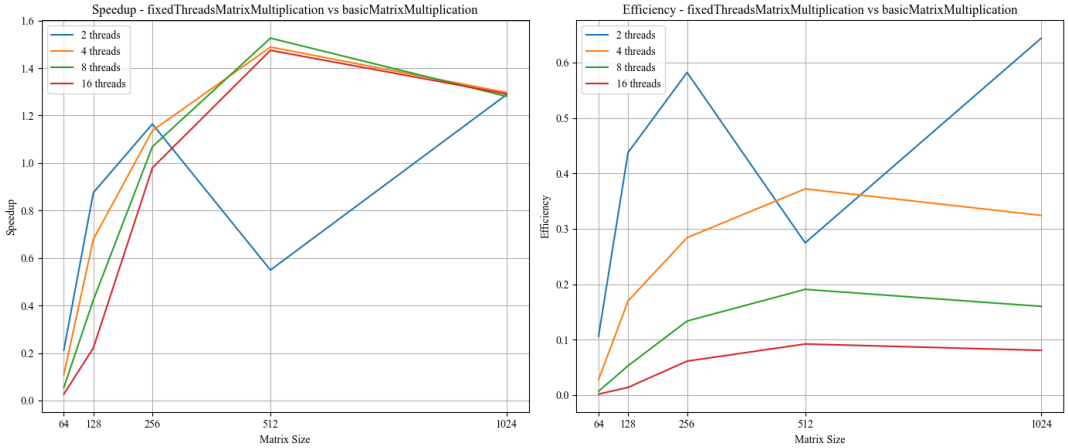

Figure 2: Speedup and Efficiency for Executor Algorithm

The aim of this experiment was to evaluate the performance of the FixedThreadsMatrixMultiplication algorithm, particularly focusing on its speedup and efficiency as matrix sizes increase and different thread counts (2, 4, 8, 16) are tested.

The FixedThreadsMatrixMultiplication algorithm exhibits strong speedup with increasing matrix sizes, especially for smaller matrices. However, from 512 to 1024, there is a noticeable drop in speedup, particularly for configurations with 2 threads. The convergence of speedup values at 1024 matrices suggests that the fixed thread model has limits in how well it scales with larger matrix sizes, possibly due to thread contention or overheads associated with maintaining a fixed number of threads. The radical drop in speedup at 512 matrices with 2 threads might indicate that for small-scale problems, the fixed thread approach introduces more synchronization overhead or inefficiency when only two threads are used, whereas with more threads, the workload distribution improves.

Efficiency behaves similarly to speedup, with the most efficient configuration being 2 threads in general, except at matrix size 512 where it exhibits a drastic drop before improving at larger sizes. 16 threads show the lowest efficiency, followed by 8 threads, with 4 threads being generally more efficient for the larger matrix sizes. This suggests that the algorithm may have an optimal point for thread configurations, beyond which the additional threads do not significantly improve performance and instead contribute to overheads. The stabilization in efficiency for larger matrices (512 and beyond) implies that for higher thread configurations, the algorithm is better able to balance workload distribution and synchronization.

Consequently, these observations can be summarized as followed:

- The FixedThreadsMatrixMultiplication algorithm demonstrates strong speedup with increasing matrix sizes, though this speedup begins to decline from matrix size 512 to 1024. This decline is likely due to increased synchronization or overheads when scaling to larger matrices. Additionally, it is important to note that for two threads, the behavior is somewhat different, as we observe a clear drop in performance for the 512 matrix. This is significant because the tests were performed with multiple iterations, which mitigates environmental factors.
- The efficiency is higher with 2 threads, except at matrix size 512, where efficiency drops significantly before recovering. The higher thread configurations (8 and 16 threads) show lower efficiency, which suggests diminishing returns when using more threads for this algorithm.

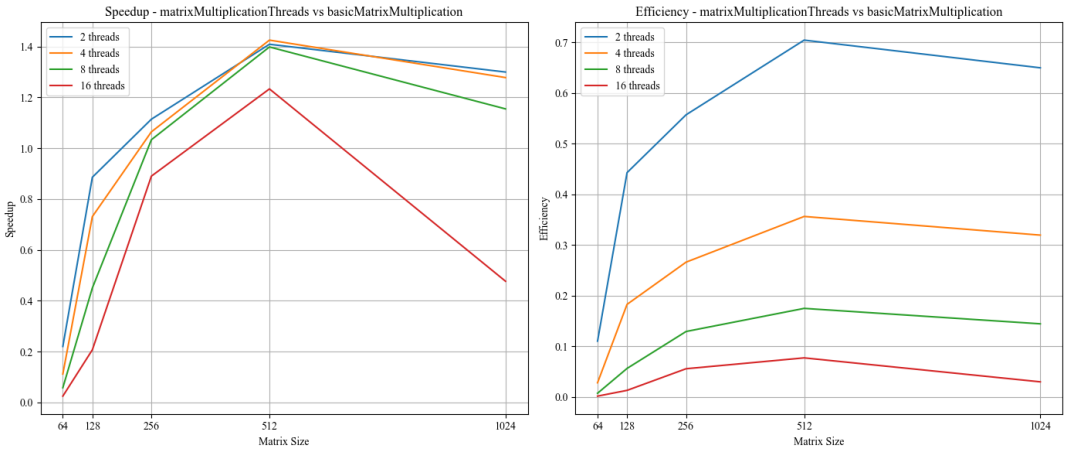

Figure 3: Speedup and Efficiency for Threads Algorithm

This experiment aims to evaluate the performance of the MatrixMultiplicationThreads algorithm by analyzing its speedup and efficiency across different matrix sizes and thread counts (2, 4, 8, and 16).

The results are displayed in two line graphs. The first graph shows speedup for different thread counts, while the second graph shows efficiency.

The MatrixMultiplicationThreads algorithm follows a similar trend to the previous experiment, showing exponential speedup for smaller matrices (up to 256). The sharp drop in speedup for 16 threads from matrix size 512 to 1024 suggests that the overhead of managing 16 threads becomes especially detrimental for larger matrices. This drop may result from factors such as thread contention, memory bottlenecks, or increased synchronization overhead as the number of threads grows too large relative to the problem size.

The efficiency pattern remains consistent with the first experiment, where the efficiency is highest with fewer threads. As with the previous experiment, the efficiency decreases with an increasing number of threads, with 2 threads maintaining the best performance in terms of efficiency. This suggests that, for this algorithm, the overhead associated with parallel execution at higher thread counts outweighs the performance benefits.

To conclude, the MatrixMultiplicationThreads algorithm demonstrates exponential speedup for smaller matrices, with a significant drop in speedup for 16 threads for matrix sizes 512 to 1024. This highlights a performance bottleneck when using a higher number of threads, especially as the matrix size increases. Efficiency continues to decrease with an increasing number of threads, with the most efficient execution observed when using 2 threads. This suggests that while the algorithm benefits from parallelization, the overheads at higher thread counts diminish the overall performance gain. As with the previous experiments, the drop in speedup for larger matrices and higher thread counts suggests the need for further optimization techniques to improve efficiency for larger matrix sizes and more threads.

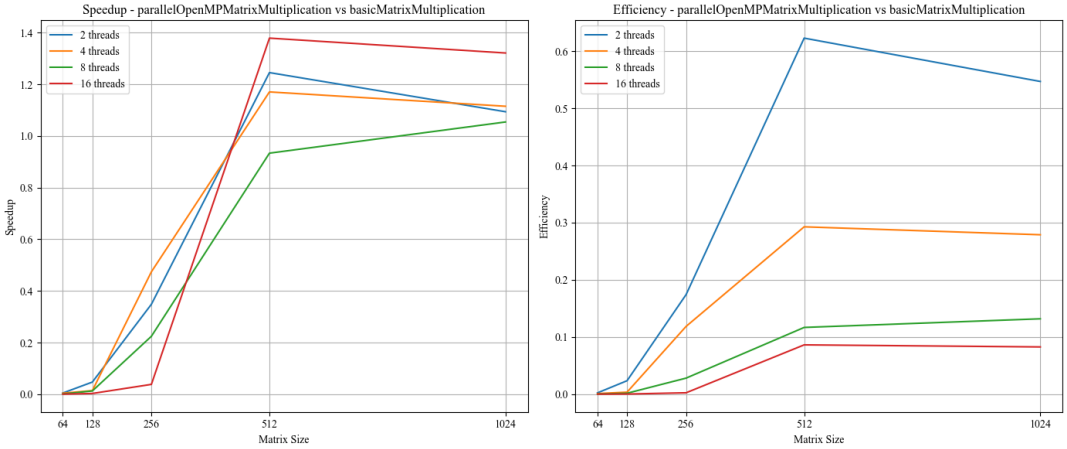

Figure 4: Speedup and Efficiency for ParallelOpenMP Algorithm

The goal of this experiment was to assess the performance of the ParallelOpenMPMatrixMultiplication algorithm, which uses ForkJoinPool to simulate OpenMP behavior, by evaluating its speedup and efficiency for various matrix sizes and thread counts (2, 4, 8, and 16).

As before, the results are shown in two line graphs: one for speedup and one for efficiency.

The ParallelOpenMPMatrixMultiplication algorithm demonstrates an interesting behavior. For 16 threads, the speedup is initially poor for smaller matrices, but it outperforms the other thread counts as the matrix size increases (from 512 upwards). This is likely due to the algorithm's ability to better leverage available threads as the matrix size becomes large enough to reduce contention. For 8 threads, the speedup is lower for most matrix sizes, indicating that it may not fully benefit from parallelization in the middle range of matrix sizes. In addition, 4 threads show the best speedup for smaller matrices, but their performance does not improve as significantly for larger matrices. Finally, 2 threads exhibit a similar trend to the previous experiments, with generally stable but not as high speedup across different matrix sizes.

Another notable observation is that from matrix size 512 to 1024, the speedup becomes relatively stable, suggesting that the algorithm reaches a point where the performance gains from increasing the matrix size or the number of threads start to plateau. This is a sign that the parallel overheads are being better managed, or that the computational load is being distributed more efficiently.

The efficiency of the ParallelOpenMPMatrixMultiplication algorithm follows a similar trend to the speedup graph. The efficiency is highest with 2 threads and decreases with higher thread counts. However, the efficiency drop for matrix size 1024 is much less pronounced than in the previous experiments. In some cases, the efficiency remains stable, which indicates that the algorithm is maintaining a relatively balanced workload and not suffering as much from the overheads associated with higher thread counts.

To sum up, the ParallelOpenMPMatrixMultiplication algorithm shows improved performance with 16 threads for larger matrices (from 512 upwards), despite initial lower speedup for small matrices. This suggests that the algorithm benefits from higher thread counts for larger problem sizes. Additionally, there is a noticeable stabilization in both speedup and efficiency from matrices of 512 to 1024. This is a significant difference compared to the previous experiments.

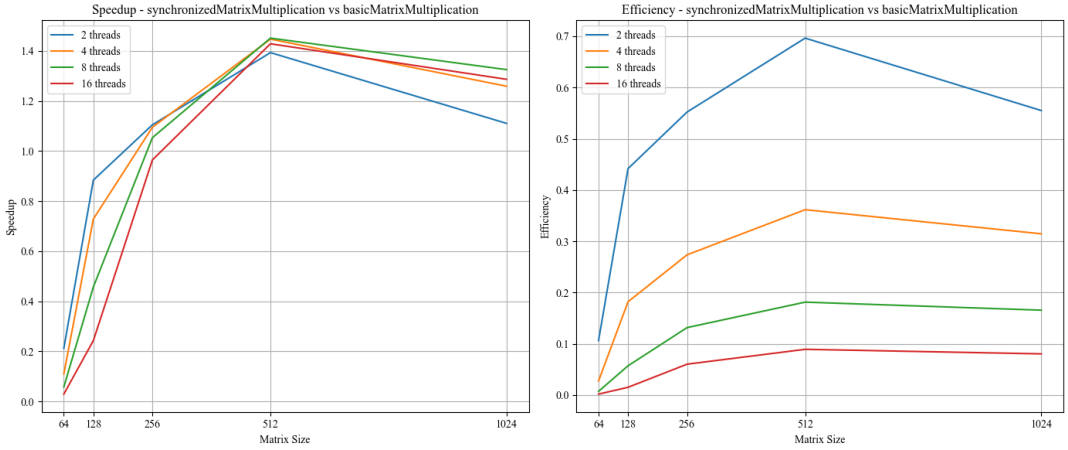

Figure 5: Speedup and Efficiency for Synchronized Algorithm

This experiment evaluates the performance of the SynchronizedMatrixMultiplication algorithm by analyzing its speedup and efficiency across varying matrix sizes and thread counts.

The speedup pattern is consistent with expectations and similar to previous algorithms. However, a clear drop is observed for two threads from matrices of 512 elements to 1024. This is surprising because for smaller matrices, the speedup values were higher.

The efficiency graph highlights a critical observation: the efficiency with 2 threads drops significantly between matrix sizes 512 and 1024. This drop may be attributed to synchronization overheads becoming disproportionately impactful when fewer threads are used, reducing the algorithm's overall efficiency.

For 4, 8, and 16 threads, the efficiency stabilizes for larger matrices, indicating that the algorithm effectively balances synchronization costs with computational gains. This stability suggests that synchronization overheads are better distributed across more threads when dealing with larger workloads.

In conclusion, while the speedup pattern aligns with expectations, a significant drop for two threads between matrix sizes 512 and 1024 was observed, which is unexpected given the higher speedup values for smaller matrices. The efficiency analysis

reveals a notable decrease for 2 threads in larger matrices, likely due to synchronization overhead. For 4, 8, and 16 threads, efficiency stabilizes, suggesting that synchronization costs are better balanced as the number of threads increases.

4.2 SpeedUp - Sequential and Vectorized Algorithms

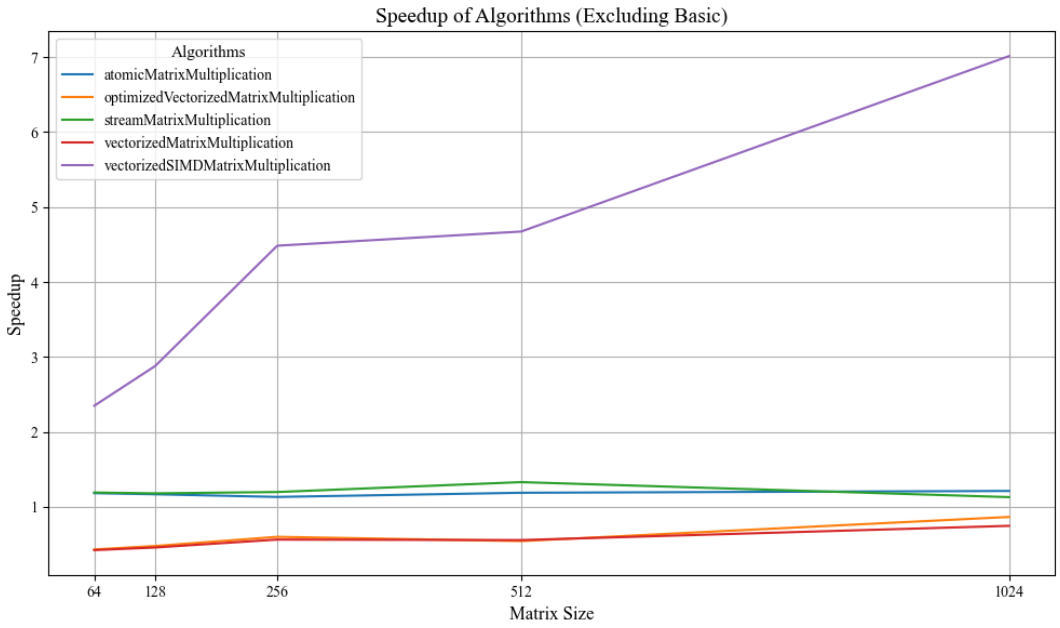

Figure 6: SpeedUp for Sequential and Vectorized Algorithms

This experiment analyzes the speedup performance of different sequential algorithms for matrix multiplication. The goal is to compare their computational efficiency across various matrix sizes.

The line graph showcases the speedup for each algorithm across varying matrix sizes:

- **VectorizedSIMD:** This algorithm demonstrates the highest speedup, significantly outperforming the others across all matrix sizes. Its advantage is most noticeable as the matrix size increases, highlighting the power of SIMD instructions in efficiently handling computational workloads.
- **Vectorized and OptimizedVectorized:** These two algorithms show minimal speedup and are closely aligned throughout the matrix sizes. Their performance is so similar that their lines frequently overlap, indicating negligible differences in their optimization levels.
- **Stream and Atomic:** These algorithms exhibit a moderate speedup, falling between the high-performance Vectorized-SIMD and the low-performance Vectorized/OptimizedVectorized. The performance of Stream and Atomic algorithms is very similar, with their lines following a nearly parallel trajectory across matrix sizes.

To sum up, VectorizedSIMD clearly outperforms the other algorithms, demonstrating that SIMD instructions are a powerful tool for optimizing matrix multiplication. Vectorized and OptimizedVectorized fail to achieve meaningful speedup, suggesting the need for alternative optimization strategies or improvements in their implementation. Finally, Stream and Atomic provide a reasonable balance, achieving moderate speedup while maintaining simplicity in implementation.

4.3 Speedup -General Comparison between algorithm

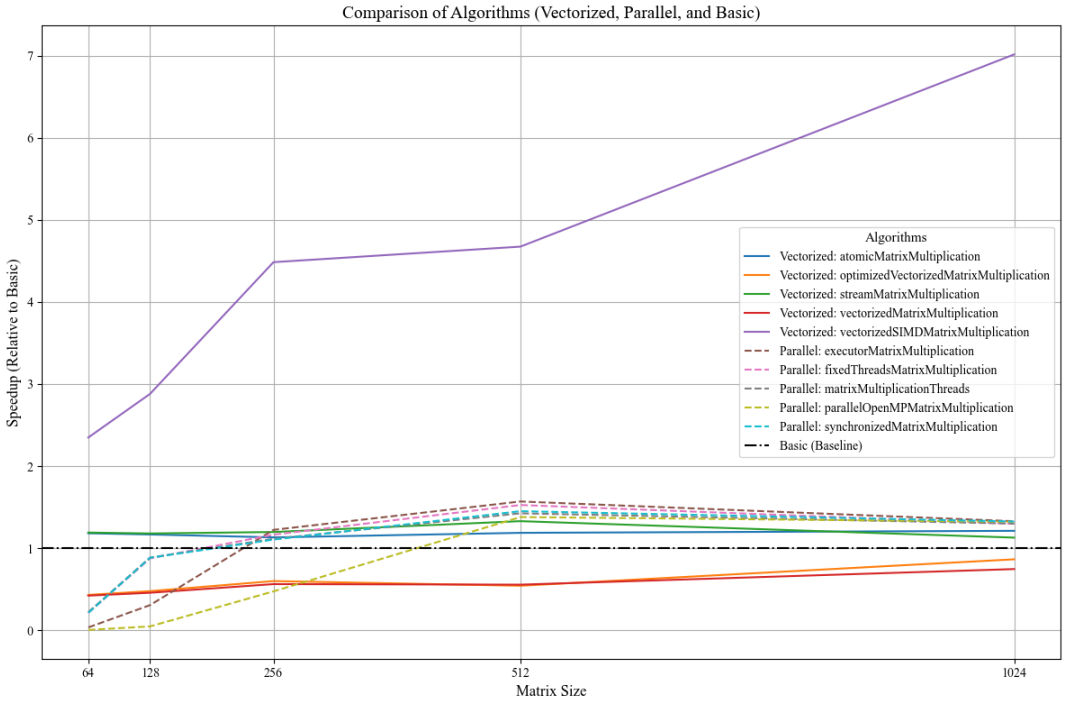

Figure 7: Speedup General Comparison

This experiment presents a comprehensive comparison of the best results from both parallel and sequential algorithms, focusing on their speedup performance relative to the baseline (basic algorithm). In the case of the parallel algorithms, since there were four results for each, it was decided to take the best one, meaning the one with the highest speedup, to make a fairer comparison. The comparison highlights which approaches deliver the highest computational speedup across varying matrix sizes.

The line graph illustrates the speedup of all algorithms. The baseline is represented by a horizontal line at **speedup = 1**. By observing the graph, these are the conclusions that can be drawn:

- **Squential:**
  - **VectorizedSIMD:** This algorithm dominates across all matrix sizes, maintaining a consistent significant lead over both parallel and other sequential algorithms. It never falls below the baseline, even for smaller matrices.
  - **Atomic and Stream:** These algorithms consistently perform just above the baseline for all matrix sizes, with similar trajectories. They maintain a steady advantage over the basic algorithm but do not approach the performance of
  - **Vectorized and OptimizedVectorized:** These algorithms remain below the baseline throughout all matrix sizes. Their underperformance relative to the basic algorithm highlights inefficiencies in their implementations.
- **Parallel:**
  - For small matrices (64, 128), the parallel algorithms tend to perform below the baseline, likely due to the overhead of managing threads.
  - As matrix sizes increase (from 256 and 512 onwards), the performance improves, surpassing the baseline. However, their speedup remains relatively modest, often staying close to the baseline.
  - None of the parallel algorithms rivals the performance of **VectorizedSIMD**, even at their best configurations.

Consequently, the conclusions for these graphs can be summarized as follows:

- **VectorizedSIMD** outperforms all other approaches, underscoring the effectiveness of SIMD instructions for matrix multiplication.

- **Atomic and Stream** provide moderate improvements over the basic algorithm, while **Vectorized and OptimizedVectorized** fail to deliver competitive results.
- **Parallel algorithms**, while showing promise for larger matrices, remain close to the baseline and do not leverage their full potential compared to the efficiency of SIMD-based solutions.

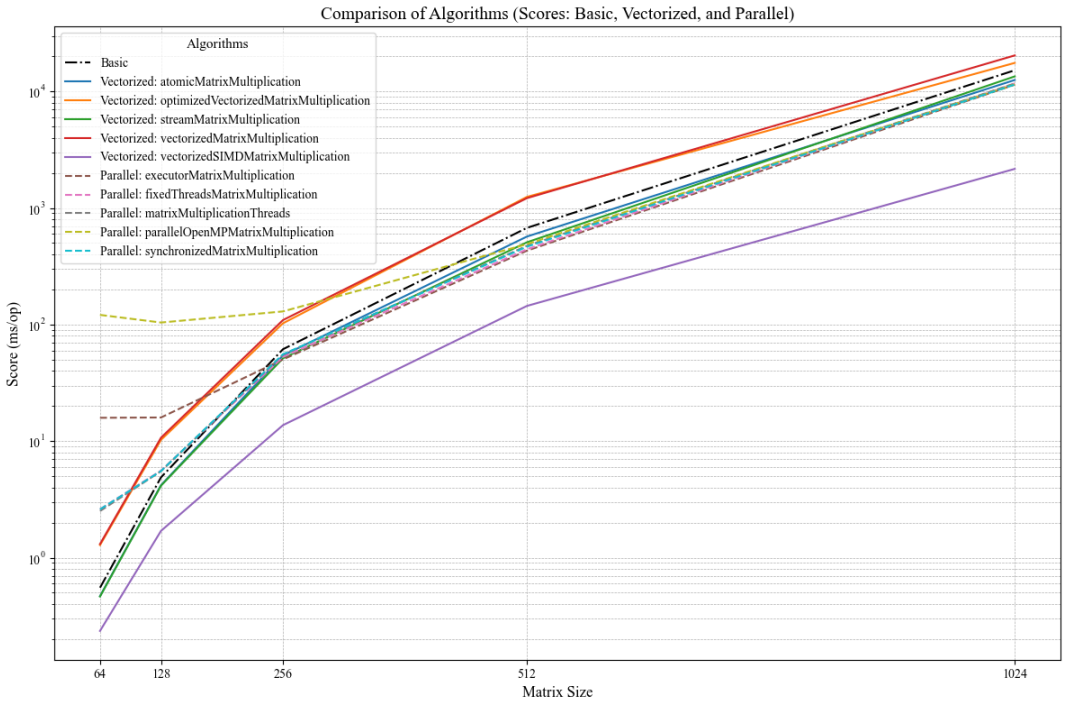

Figure 8: Performance General Comparison

This experiment evaluates the execution time per operation (**ms/ops**) for both the best-performing parallel and sequential algorithms, providing insight into their computational efficiency at the finest granularity.

The graph displays the **ms/ops** performance for each algorithm, with lower values indicating better perfomance.

- **Sequential:**
  - **VectorizedSIMD:** This algorithm consistently achieves the lowest **ms/ops** values, demonstrating its superior computational perfomance. Its performance remains outstanding across all matrix sizes, underscoring its dominance in reducing execution time.
  - **Vectorized and OptimizedVectorized:** These algorithms exhibit the highest **ms/ops** values overall, reflecting their inefficiency. Their lines are closely aligned, further emphasizing the negligible difference in performance between the two implementations.
  - **Basic Algorithm:** The basic algorithm ranks as the third-highest in **ms/ops** values, showing better efficiency than **Vectorized** and **OptimizedVectorized** but trailing far behind the best-performing algorithms.
- **Parallel:**
  - For small matrices (64, 128), the parallel algorithms have the highest **ms/ops** values, particularly in the following order: OpenMP, Executor, Synchronized.
  - For larger matrices (256 and above), the **ms/ops** values of the parallel algorithms significantly improve and begin to cluster together, indicating similar performance among them. However, they still do not match the efficiency of **VectorizedSIMD**.

From this graph, the following conclusions can be drawn:

- **VectorizedSIMD** excels as the most efficient algorithm, achieving the lowest **ms/ops** values across all matrix sizes.
- The high **ms/ops** values of **Vectorized** and **OptimizedVectorized** reinforce their inadequacy for efficient computation.
- **Parallel algorithms**, while improving for larger matrices, are hindered by high overheads for smaller inputs and fail to surpass the efficiency of SIMD-based techniques.

4.4 Resource Comparison

In this section, we will take it a step further and analyze the resource usage for the different algorithms. First, we will begin with memory usage and then move on to CPU usage.

4.4.1 Memory Usage

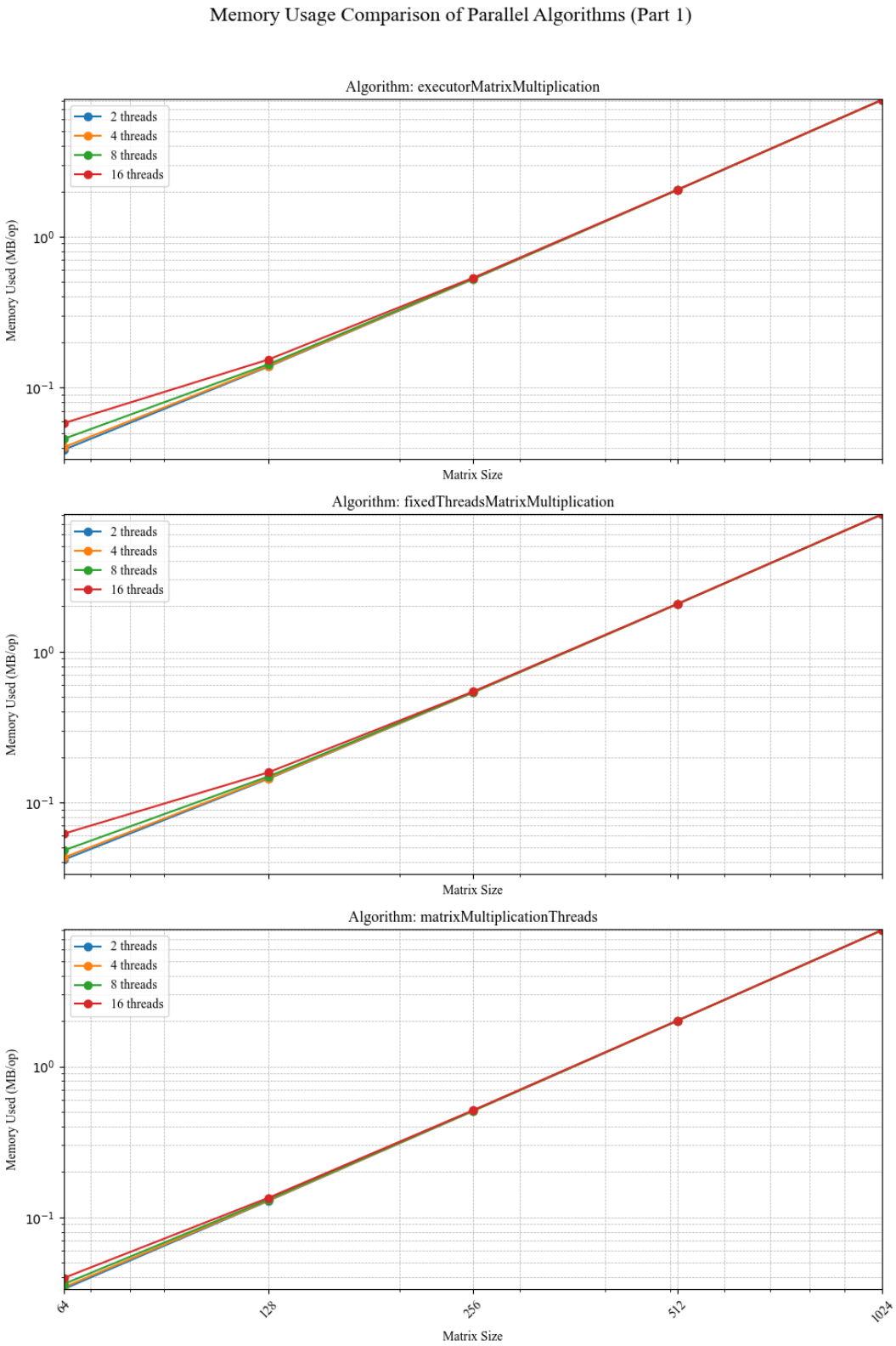

Figure 9: Memory Usage Comparison of Parallel Algorithms (Part1)

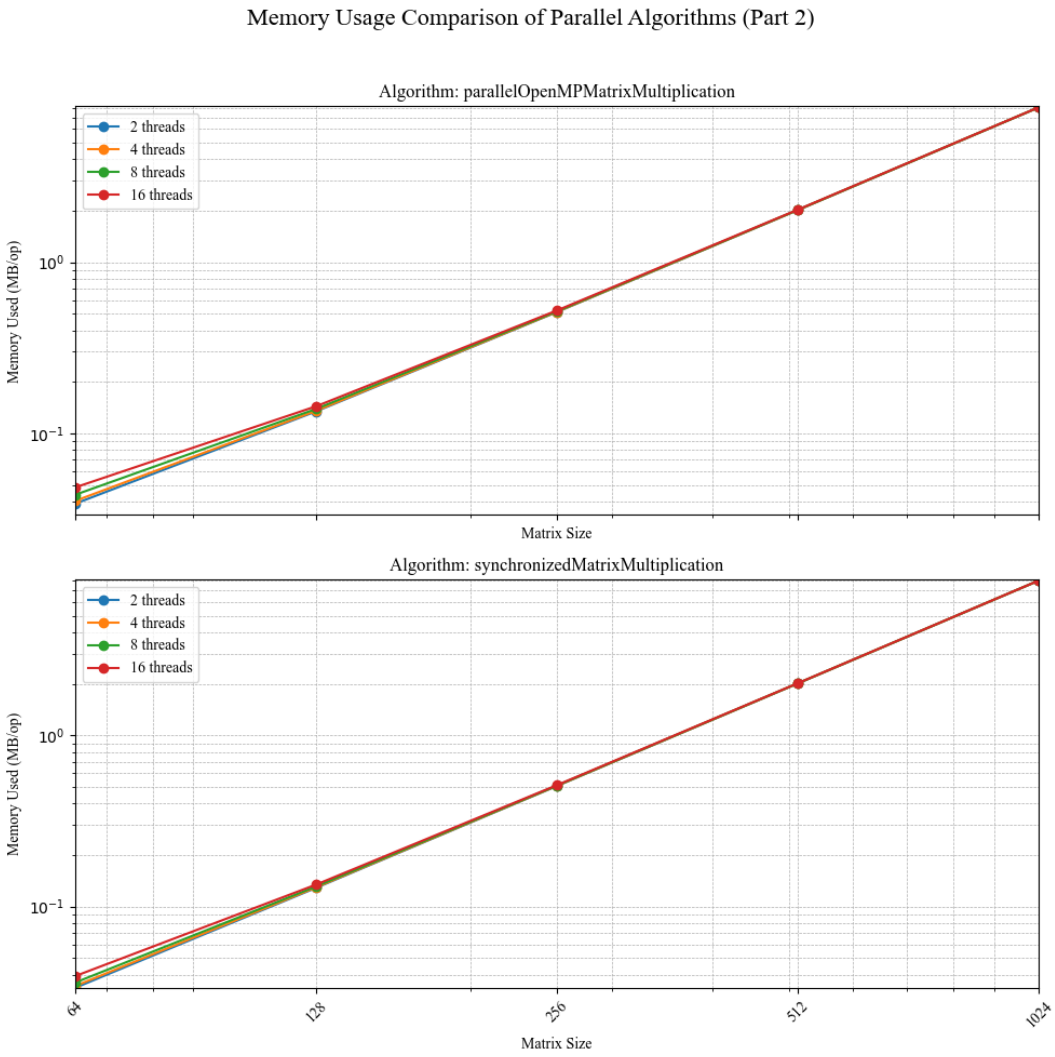

Figure 10: Memory Usage Comparison of Parallel Algorithms (Part2)

These graphs are composed of several subplots, each corresponding to a parallel algorithm, and its lines, like those for other experiments, represent the MB/operation for the algorithm using different numbers of threads. We can see that they all exhibit very similar behavior; in fact, there is no noticeable dominance in memory usage. On the other hand, the low variability across the different thread counts stands out. The difference is only noticeable with small matrices, but nothing particularly significant.

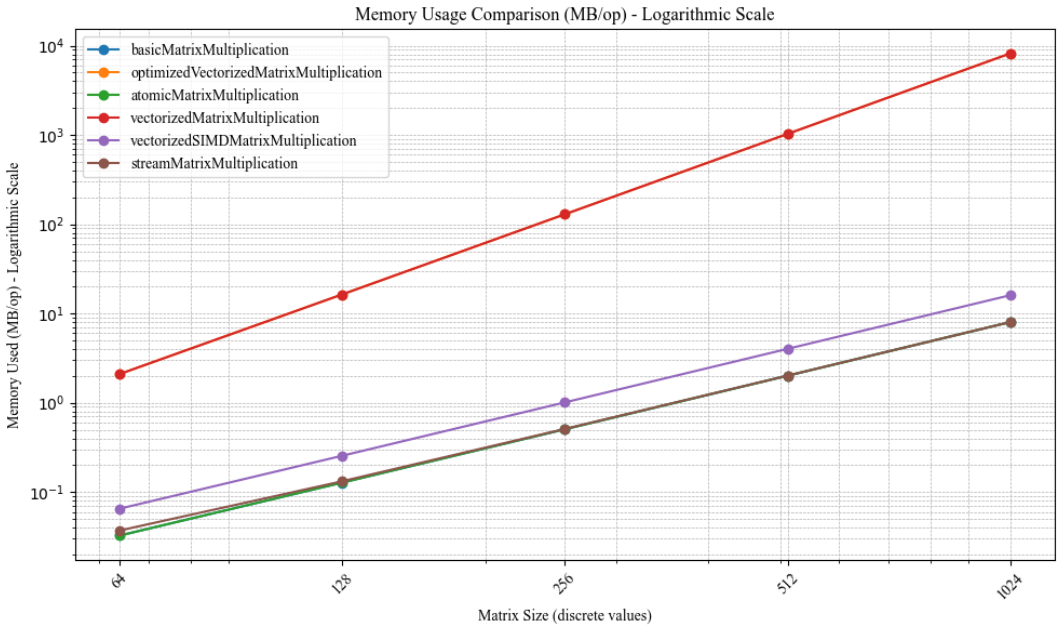

Figure 11: Memory Usage Comparison for Sequential's Algorithms

This graph shows a logarithmic scale comparison of memory usage for different matrix sizes with various algorithms. In this case, we focus on sequential algorithms. As can be seen, there is a clear upward trend, meaning that as the matrix size increases, all algorithms show higher memory usage, specifically measured in MB/operation on the y-axis. On the other hand, it is noticeable that the simple vectorized algorithm has by far the highest values, followed by the vectorized algorithm that uses SIMD instructions. The other algorithms show smaller values and are fairly close to each other. In relation to the previous comparison, it is clear that these have higher memory usage. Additionally, it is noticeably higher.

4.4.2 CPU Usage

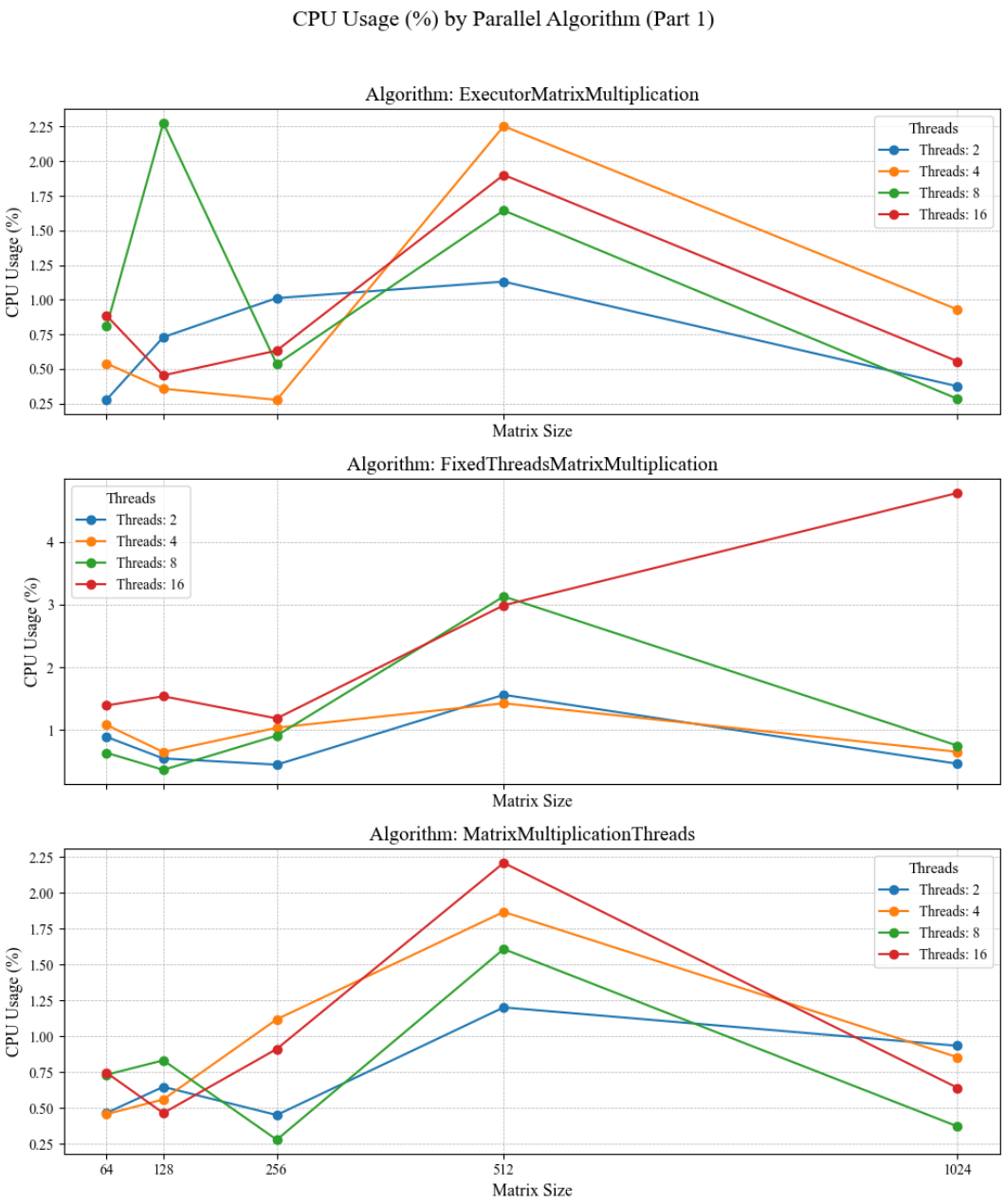

Figure 12: CPU Usage by Parallel Algorithms (Part1)

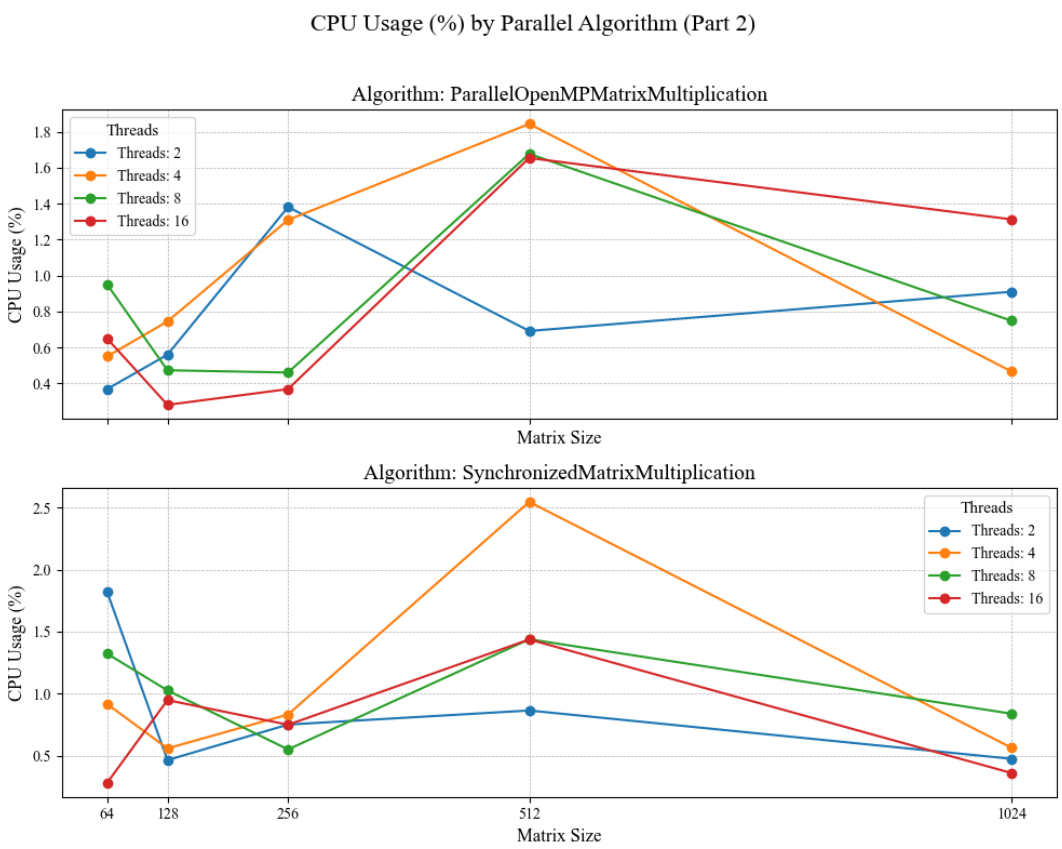

Figure 13: CPU Usage by Parallel Algorithms (Part2)

These graphs aim to show CPU usage as a percentage for the different parallel algorithms considered in this study. The main conclusions are presented below:

- **Executor:** The CPU usage of the Executor algorithm exhibits no clear pattern, with fluctuations being observed across different thread counts and matrix sizes. The 8-thread configuration in particular stands out due to high fluctuations in CPU usage, especially for smaller matrices. The overall trend of decreasing CPU usage followed by increases and eventual drops in larger matrices is consistent, though not entirely predictable. This suggests potential inefficiencies or scaling issues when using this algorithm.
- **FixedThread:** The increase in CPU usage for the 16-thread configuration with larger matrices may indicate potential bottlenecks or inefficiencies. In addition, it is important to note the decline between the 512 and 1024 element matrices.
- **Threads:** With small matrices, the algorithm shows significant fluctuations in CPU usage, making it difficult to draw any solid conclusions. When the matrix size reaches 512, it can be observed that the 16-thread version shows the highest usage, followed by 4, 8, and 2 threads. However, when the size increases to 1024, the order changes to 2, 4, 8, and 16 threads. This reveals many irregularities, similar to what was observed with previous algorithms.
- **OpenMP:** This is undoubtedly the algorithm that shows the most erratic behavior; no clear pattern can be established, as CPU usage fluctuates throughout the graph. The only noteworthy observation is that for 8 and 16 threads, the results are relatively close, except for matrices of 1024.
- **Synchronized:** The CPU usage fluctuates across different configurations, with the 4-thread configuration standing out for matrix size 512. This suggests potential inefficiencies or contention issues when using this thread count, particularly for this matrix size. The 2-thread configuration sees a reduction in CPU usage with larger matrices, indicating that fewer threads may perform more efficiently for larger matrix multiplications. The fluctuations in CPU usage highlight potential scaling issues with this algorithm when the number of threads increases.

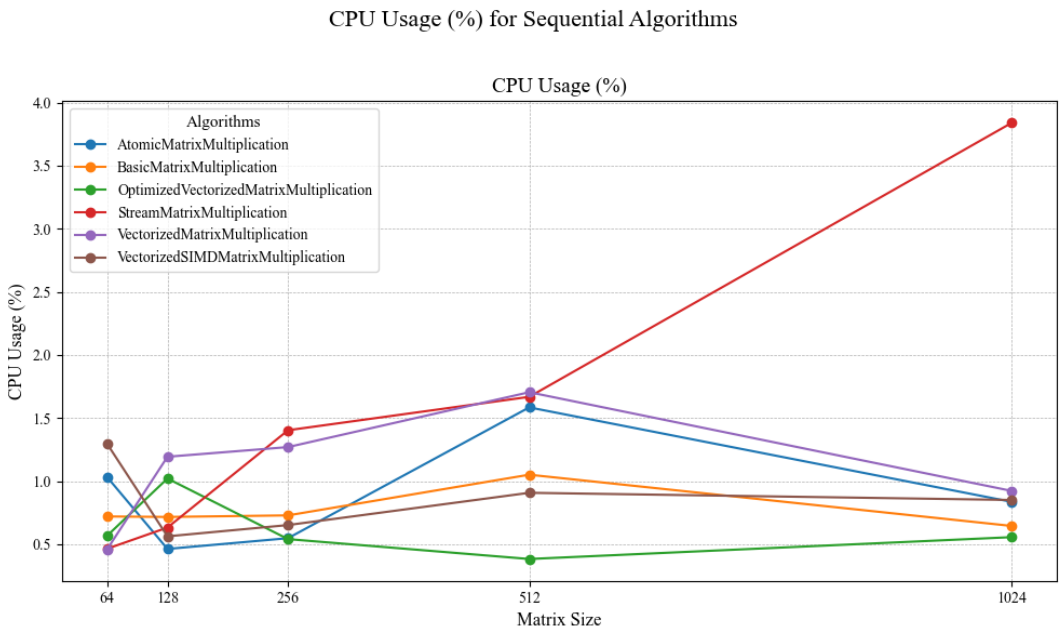

Figure 14: CPU Usage for Sequential Algorithms

This other graph, similar to the previous ones, focuses on CPU usage for the sequential and vectorized algorithms. In this case, although there are still quite a few fluctuations, it is easier to identify patterns:

- It is observed that, in general, the variant using streams stands out as having the highest CPU usage in several cases, especially for the 1024 matrix size, where its value is clearly separated from the rest. This is logical because it is not actually a sequential algorithm but a parallel one, and it serves in this graph to highlight the notable difference between these two types of algorithms.
- On the other hand, the optimizedVectorizedMatrixMultiplication, except for matrix sizes of 64 and 128, shows the lowest results.
- The basic and vectorized algorithms following SIMD instructions exhibit a relatively constant progression as the matrix size increases, with only a slight rise.
- The rest of the algorithms, such as the simple vectorized and atomic algorithms, show more fluctuations across the different matrix sizes.

4.4.3 Comparison between Java and Python

After thoroughly analyzing different matrix multiplication algorithms using Java, this section proceeds to compare the previously analyzed algorithms with other equivalent ones using Python. Additionally, other matrix multiplication algorithms, such as those utilizing CUDA, have been included.

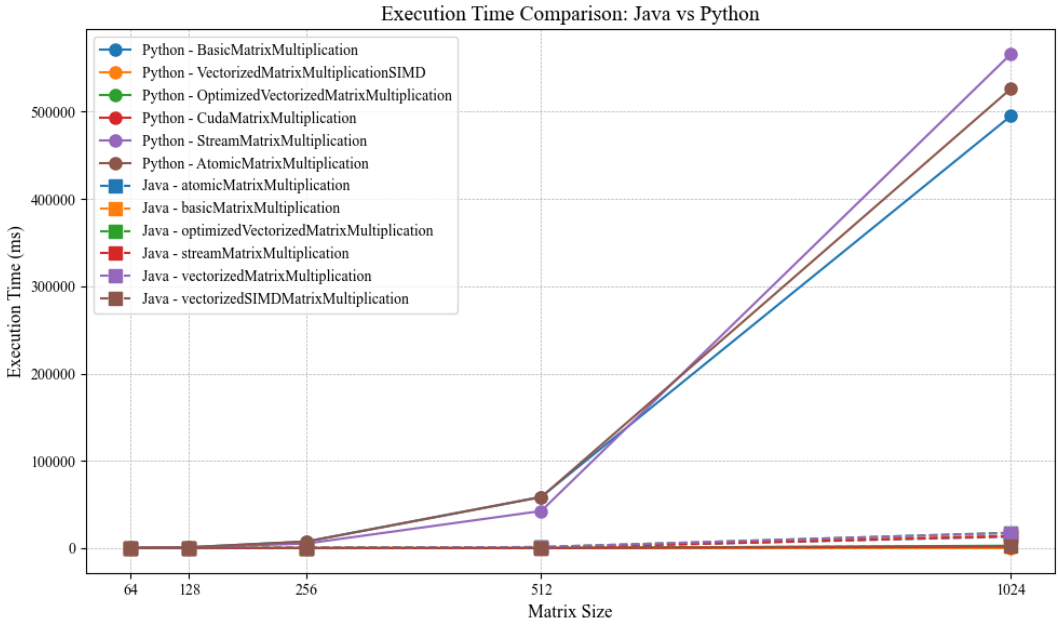

Figure 15: Execution Time Comparison: Java vs Python

This graph shows the execution time of the different algorithms used during the project, considering both those implemented in Python and Java. It can be observed that all algorithms exhibit an upward trend as the matrix size increases. On the other hand, it is also noticeable that the algorithms with the longest execution times are those implemented in Python: the basic, atomic, and stream-based algorithms. The rest of the algorithms generally show very similar behavior.

Regarding resource usage, after conducting a thorough comparison by examining the JSON files, no conclusion could be drawn that one programming language was better than another; on the contrary, it heavily depended on the algorithm. These results can be verified in the JSON files in the GitHub repository.

5 Conclusions

This study investigated various optimization strategies for matrix multiplication, focusing on parallelization and vectorization techniques, and conducted a thorough benchmarking of their performance across different matrix sizes and hardware configurations. The findings underline the critical role of tailored algorithmic approaches in enhancing computational efficiency.

The results revealed that among all tested algorithms, the VectorizedSIMD approach consistently delivered the best performance across all evaluated metrics. Its significant speedup and minimal execution time per operation demonstrate the effectiveness of leveraging SIMD instructions for parallel data processing. These findings emphasize the potential of hardware-level optimizations in addressing the computational challenges posed by large-scale matrix operations.

In contrast, while parallel algorithms showed promise, their efficiency was notably impacted by thread synchronization overheads and memory contention, particularly for smaller matrices or when utilizing higher thread counts. The performance of these algorithms improved with larger matrices but did not surpass that of the SIMD-based methods. Similarly, other sequential and vectorized implementations, such as the basic vectorized or optimized vectorized algorithms, exhibited limited improvements, highlighting inefficiencies in their designs or execution.

The comparative analysis between Java and Python implementations showed that the choice of programming language had little impact on performance; rather, the algorithm design was the key factor.

In summary, this study reaffirms the importance of selecting and optimizing algorithms based on specific use cases and hardware characteristics. The dominance of VectorizedSIMD underscores the value of advanced vectorization techniques in achieving superior computational efficiency, making it a preferred choice for large-scale matrix multiplication tasks.

6 Future Work

Building on the solid foundation established in this study, the next phase of the project will explore distributed matrix multiplication to address the challenges posed by extremely large matrices that exceed the memory capacity of a single

machine. This phase will shift focus to scalability and the unique advantages and limitations of distributed systems for handling such datasets.

The implementation will involve leveraging a distributed computing framework to design and test matrix multiplication algorithms. These implementations will be rigorously compared against the basic and parallel approaches discussed in this study. Experiments will target extremely large matrices to evaluate how well the distributed approach scales and to identify potential bottlenecks in performance.

The subsequent report will delve into key metrics such as scalability—how performance evolves with increasing matrix size—and the overhead introduced by network communication and data transfers. Additionally, the analysis will include detailed insights into resource utilization, focusing on the number of nodes employed and memory allocation per node.

This phase will provide a deeper understanding of distributed computing's potential to tackle large-scale matrix operations, offering valuable perspectives on its practical application and limitations in real-world scenarios.