Paralellization and Vectorized Benchmarks of Matrix Multiplication

Alejandro Alemán Alemán

December 1, 2024

Contents

Abstract

Matrix multiplication is a critical operation in computational mathematics, but its $O(n^3)$ complexity limits its performance for large-scale problems. This study explores parallel computing techniques, including multithreading and synchronization mechanisms, to optimize matrix multiplication. In addition, an optional SIMD-based vectorized approach was evaluated. Experiments on large matrices demonstrate significant speedup and efficiency gains with parallelization. Resource usage analysis highlights the trade-offs in computational and memory demands, showcasing the potential of parallel techniques for scalable matrix multiplication.

1 Introduction

Matrix multiplication is a fundamental operation in computational mathematics, widely used in fields such as machine learning, computer graphics, and scientific simulations. Despite its simplicity, the algorithm's cubic time complexity ($O(n^3)$) becomes a limiting factor for large matrices, leading to high computational costs and increased execution times.

With the growing demand for faster and more efficient computations, parallel computing has emerged as a practical solution to overcome these limitations. By dividing tasks among multiple processing units, parallel computing allows for significant reductions in execution time, making it a critical area of research for optimizing matrix operations. Additionally, advancements in hardware technologies, such as multi-core processors and vectorized instructions (e.g., SIMD), provide new opportunities to further accelerate computations.

This study focuses on implementing and analyzing parallel techniques for matrix multiplication using multithreading and synchronization mechanisms. Furthermore, an optional exploration of vectorized approaches is included to compare performance gains. The main objectives are to evaluate the speedup achieved through parallelization, assess parallel efficiency, and analyze resource usage, such as memory. By comparing these approaches to a basic sequential implementation, this work aims to provide insights into the scalability and practicality of parallel and vectorized techniques for matrix multiplication.

2 Problem Statement

Matrix multiplication is a key computational operation where the product of two $n \times n$ matrices is computed. The most basic algorithm for this task involves iterating through rows of the first matrix and columns of the second matrix, performing n multiplications for each entry, and summing the results. This leads to a time complexity of $O(n^3)$. While straightforward, this complexity becomes a significant limitation for large matrices due to the exponential growth in the number of operations required as n increases. For instance, doubling the size of the matrices results in an eightfold increase in computational cost, making the approach unsuitable for high-performance applications or real-time scenarios.

2.1 Challenges in Parallelization

Optimizing matrix multiplication through parallel computing introduces new challenges:

- **Thread Contention:** When multiple threads access shared resources, such as parts of the matrix or memory buffers, contention can occur, leading to performance degradation. Efficient thread management is essential to minimize such bottlenecks.
- **Load Balancing:** Ensuring that all threads perform an equal amount of work is critical for achieving high efficiency. Poor load balancing may leave some threads idle while others are overburdened, reducing overall speedup.
- **Synchronization Overhead:** Synchronization mechanisms, such as locks or atomic operations, are often required to maintain data consistency in multithreaded environments. However, excessive synchronization can negate the benefits of parallelism by introducing delays.

3 Methodology

3.1 Implementation

3.1.1 Basic Algorithm (Naive)

This is the basic matrix multiplication algorithm follows the traditional and most widely known method for multiplying two matrices, often referred to as the classical algorithm. This method operates with a time complexity of $O(n^3)$, where n represents the size of the matrix (for an n x n matrix). The algorithm involves performing three nested loops to calculate the resulting matrix.

Given two square matrices, A and B, of size n x n, the algorithm aims to compute the result matrix, C, also of size n x n. For each element in the result matrix C, the algorithm computes the dot product of the corresponding row from matrix A and the corresponding column from matrix B. The process can be described as follows:

1. Outer Loop: Iterates over each row i of matrix A.
2. Middle Loop: Iterates over each column j of matrix B.
3. Inner Loop: Performs the dot product calculation by iterating through the elements of the row from A and the column from B. For each index k, the algorithm multiplies the elements $A[i][k]$ and $B[k][j]$, then adds the result to the current value of $C[i][j]$.

The resulting matrix element $C[i][j]$ is computed using the formula:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

This requires n multiplications and n additions for each element in the result matrix. Since there are n^2 elements in the result matrix, the overall complexity of the algorithm is $O(n^3)$.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Figure 1: Pseudo code of matrix multiplication algorithm implemented.

3.1.2 Parallel Algorithms

Parallel computing techniques were employed to optimize the basic algorithm, leveraging multi-threading mechanisms. The following approaches were implemented:

Fixed Thread Pool (Executors): The Fixed Thread Pool (Executors) implementation utilises a fixed number of threads to parallelise matrix multiplication by assigning each thread to compute a single row of the resulting matrix. The ExecutorService efficiently manages threads, minimising overhead by reusing them from the pool. Each task calculates the dot product of a row from the first matrix and a column from the second, ensuring no synchronisation is needed as each thread operates independently. After submitting all tasks, the program waits for their completion before returning the result. This method balances simplicity, scalability, and efficient resource utilisation for large matrices.

Parallel Streams: The Parallel Streams algorithm leverages Java's IntStream and its parallel() method to distribute matrix multiplication tasks across multiple threads using the ForkJoin framework. Each row of the resulting matrix is computed independently in parallel, with threads calculating the dot product of the corresponding row from matrix A and the columns of matrix B. The parallelism level is configurable by setting the ForkJoin common pool's thread count, enabling efficient utilization of multicore processors. This approach simplifies parallelism, as the Java Streams API automatically manages thread allocation and workload distribution, making it a concise yet powerful solution for parallel matrix multiplication.

Atomic Variables: The Atomic Matrix Multiplication implementation uses an AtomicInteger to coordinate work among multiple threads, ensuring thread-safe operations without explicit synchronisation locks. The AtomicInteger manages the assignment of rows to threads dynamically. Each thread retrieves a row index using the getAndIncrement() method, computes the corresponding row of the result matrix, and moves to the next available row. This approach eliminates contention and ensures efficient workload distribution among threads while maintaining data consistency.

3.1.3 Synchronization Mechanisms

Semaphore: The Semaphore Matrix Multiplication implementation uses a Semaphore to control the number of threads executing concurrently. Each thread is responsible for computing one row of the result matrix, and the semaphore ensures that only a fixed number of threads (defined by numThreads) are active at any given time. This mechanism limits resource contention and prevents the system from being overwhelmed by too many threads. Each thread acquires a permit from the semaphore before starting its computation and releases it upon completion, ensuring controlled parallelism.

Synchronization Blocking: This implementation uses Java's synchronized keyword to protect critical sections of code that modify shared resources. Each thread computes a specific range of rows from the result matrix, ensuring efficient parallelism. Within the critical section, threads synchronise when updating the result matrix, ensuring thread-safe operations and avoiding race conditions. The workload is divided equally among threads based on the number of rows, with the last thread handling any extra rows.

3.1.4 Vectorized algorithm

The Vectorized Matrix Multiplication implementation leverages the Aparapi library to execute computations on GPU-like parallel processors using SIMD (Single Instruction, Multiple Data) principles. The matrices are flattened into 1D arrays to optimise data access patterns, enabling efficient use of hardware vectorisation. A custom kernel is defined to calculate the dot product for each element of the result matrix, with parallel threads (global IDs) independently processing individual matrix cells. This approach maximises throughput by utilising hardware acceleration.

3.2 Testing Methodology

The performance tests were carried out using a benchmarking framework (JMH) in Java, designed to gather accurate metrics on execution time and memory usage for each matrix multiplication approach. The testing methodology includes the following components:

- **Evaluated Metrics:** The primary metrics assessed in this benchmarking process are:
  - **Execution Time:** The average time taken to complete each matrix multiplication operation, measured in milliseconds.
  - **Memory Usage:** The memory consumed during matrix multiplication, calculated by comparing initial and final memory states, measured in kilobytes.

All tests were conducted in a controlled execution environment to ensure consistent results. The benchmarking framework JMH was configured with parameters such as warm-up iterations (2), measurement iterations (3), and standardized time units for each run. Java classes, such as `Runtime`, were employed to measure memory usage accurately.

For all algorithms, we tested square matrices of varying sizes ($n \times n$), ranging from 10 to 3000. Additionally, for algorithms that utilise threads, tests were conducted using between 1 and 16 threads to evaluate parallelization performance.

3.2.1 Speedup and Efficiency Calculations

In addition to execution time, we calculated the **speedup** and **efficiency** for the parallel algorithms to assess their scalability and performance relative to the baseline sequential implementation.

- **Speedup (S):** Speedup measures how much faster a parallel algorithm performs compared to a sequential algorithm. It is defined as:
$$S = \frac{T_s}{T_p}$$
where T_s is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm.

- **Efficiency (E):** Efficiency evaluates the utilization of processing resources in the parallel implementation. It is calculated as:
$$E = \frac{S}{P}$$
where S is the speedup, and P is the number of processing threads used.

3.2.2 Methodology Overview

This methodology allowed us to systematically assess the performance, scalability, and efficiency of various matrix multiplication techniques across different scenarios. By comparing the calculated speedup and efficiency, we gained valuable insights into the strengths and limitations of each approach, particularly for different matrix sizes and thread configurations.

3.3 Test Environment

The majority of benchmarks were performed on a system with the following specifications:

- **Operating System:** macOS Sequoia 15.0
- **Processor (CPU):** Apple M1
- **Number of Cores:** 8
- **Memory (RAM):** 8.00 GB
- **Storage:** 512GB SSD

However, while implementing the vectorization algorithm, the computer experienced issues that limited our ability to conduct tests on the original system. Consequently, the vectorized testing was carried out on a Windows computer under comparable conditions. For consistency, the basic algorithm was also tested on this system to allow for a direct comparison.

- **Operating System:** Windows 11 Home, Version 23H2, Build 22631.4317
- **Processor (CPU):** 11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz
- **Number of Cores:** 8
- **Memory (RAM):** 8.00 GB
- **Storage:** 475 GB
- **System Type:** 64-bit Operating System, x64-based processor

The Java implementation was executed using IntelliJ IDEA on both systems.

4 Experiments and Results

4.1 Parallel and synchronized algorithms

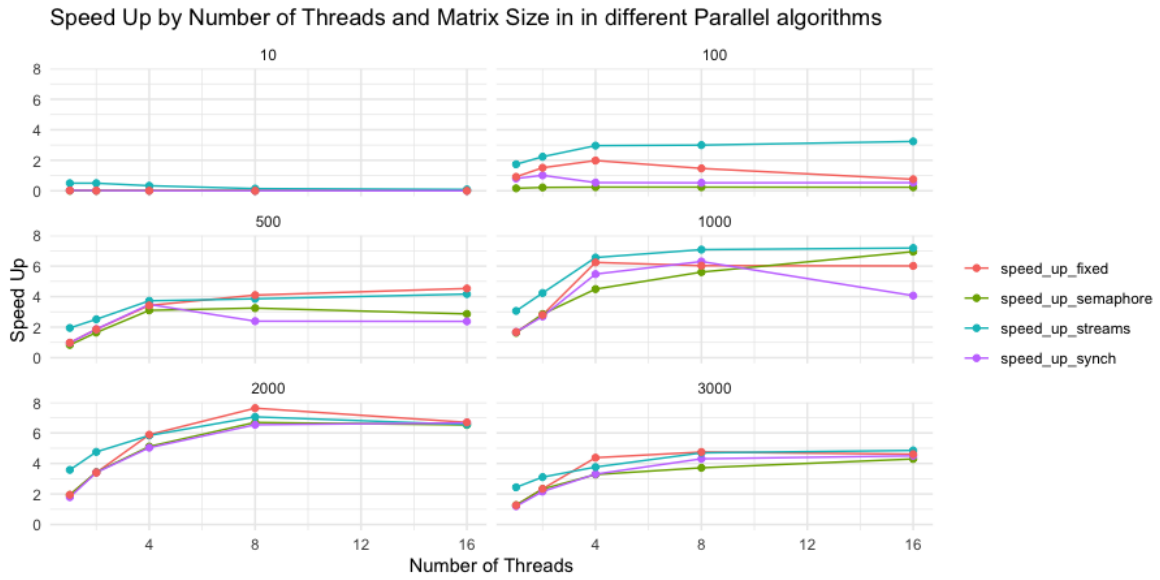

Figure 2: Speed Up in parallel algorithms

The graph demonstrates how the performance of different parallel matrix multiplication algorithms scales with the number of threads and varying matrix sizes. For small matrices (e.g., sizes 10 and 100), all algorithms exhibit minimal speedup regardless of the number of threads. This is because the computational workload is too small to benefit from parallelism, and the overhead of creating and managing threads outweighs any potential performance gains.

As the matrix size increases to medium levels (e.g., 500 and 1000), the speedup becomes more noticeable. Here, we observe that algorithms such as FixedThreads and Streams perform better, achieving substantial speedup as threads increase up to 8. However, beyond 8 threads, the speedup begins to peak, likely due to the limitations of available hardware resources or the increased overhead of managing more threads. The synchronized blocking algorithm shows lower performance compared to others, which can be attributed to the cost of synchronisation mechanisms, particularly as the number of threads increases.

For large matrices (e.g., sizes 2000 and 3000), all algorithms achieve significant speedup, indicating that parallelisation becomes increasingly beneficial for computationally intensive tasks. Among the algorithms, Streams consistently performs the best, leveraging Java's optimised parallel stream framework to manage threads efficiently. In contrast, synchronized blocking continues to underperform due to the overhead introduced by synchronisation. Additionally, while Semaphore and FixedThreads deliver strong results, they are slightly less efficient than Streams, particularly at higher thread counts.

A notable trend across all matrix sizes is that the speedup slightly declines beyond 8 threads. This suggests that the optimal number of threads aligns with the number of physical or virtual CPU cores available on the system. Increasing threads beyond this limit leads to contention for resources and diminishing returns, highlighting the importance of balancing thread count and hardware capabilities for optimal performance.

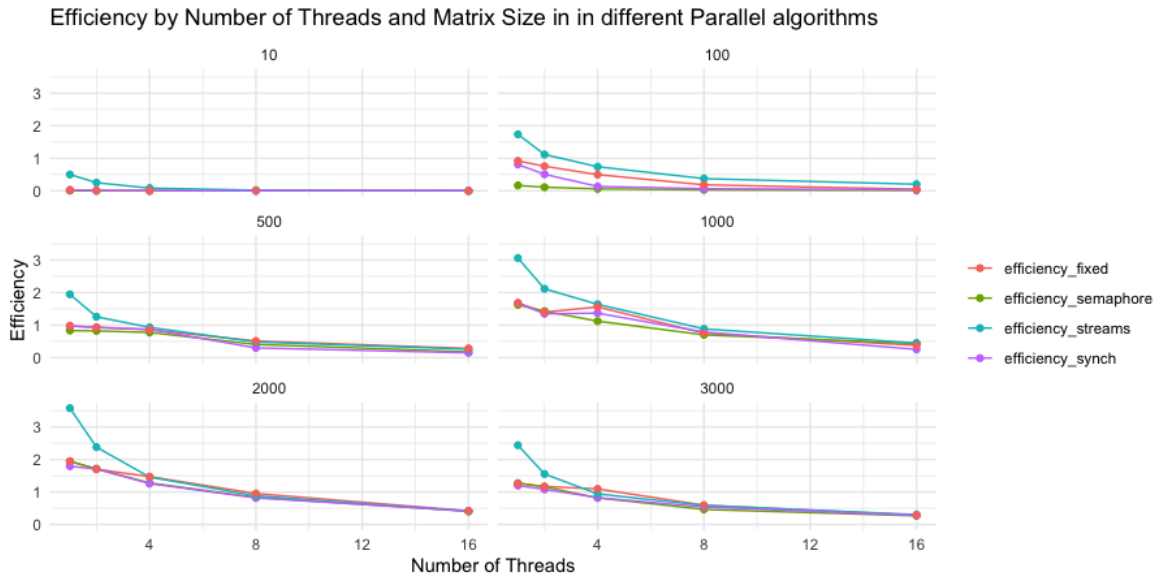

Figure 3: Efficiency in parallel algorithms

The graph illustrates the efficiency of different parallel algorithms for matrix multiplication across varying numbers of threads and matrix sizes. For small matrices (e.g., sizes 10 and 100), all algorithms exhibit extremely low efficiency, regardless of the number of threads. This is because the computational workload is too small to benefit from parallelism, and the overhead associated with thread management dominates the performance. Consequently, none of the algorithms show significant differences in efficiency at these matrix sizes.

As matrix sizes increase to medium levels (e.g., sizes 500 and 1000), efficiency improves, particularly when using a smaller number of threads, such as 4 or 8. Among the algorithms, Streams (Parallel Streams) demonstrate slightly better efficiency compared to others, highlighting the benefits of Java's optimised parallel thread management. However, as the number of threads increases beyond 8, all algorithms show a decline in efficiency due to factors such as thread contention for shared hardware resources and the increasing overhead of managing additional threads. In this range, Synch (Synchronized Blocking) performs the worst, indicating that the synchronisation mechanisms impose significant overhead and reduce the ability to scale effectively.

For large matrices (e.g., sizes 2000 and 3000), the efficiency is noticeably higher across all algorithms when fewer threads are used, peaking at around 4 to 8 threads. Streams continue to outperform the other algorithms, demonstrating the best scalability and thread utilisation for these larger computational tasks. However, efficiency drops sharply as the number of threads increases to 12 or 16. This is likely due to hardware limitations, where adding more threads exceeds the physical or virtual core capacity, leading to diminishing returns and increased resource contention.

Overall, the graph highlights that small matrices do not benefit from parallelism, medium matrices achieve moderate gains, and large matrices make the most of parallel execution. Streams consistently emerge as the most efficient algorithm, while Synch struggles with the added overhead of synchronisation. The efficiency trends suggest that the optimal number of threads for these tasks is around 4 to 8, aligning with the system's hardware capabilities.

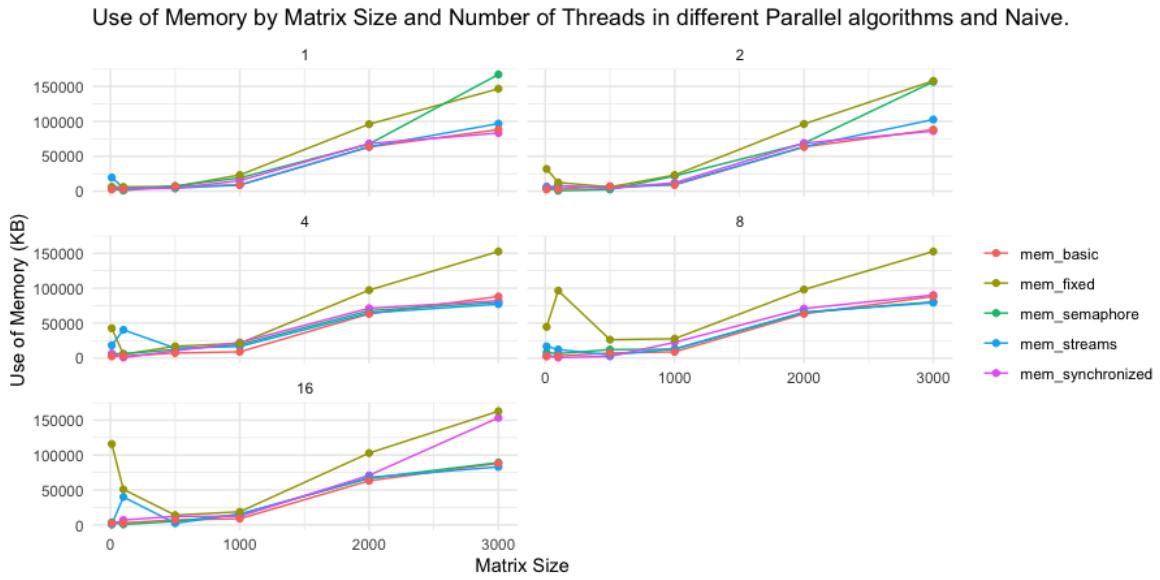

Figure 4: Memory use in parallel algorithms and naive.

The graph illustrates the memory usage of different parallel algorithms for matrix multiplication, including the naive (basic) algorithm, across varying matrix sizes and thread counts. For a single thread, memory usage remains nearly identical across all algorithms for small matrix sizes, as the parallel mechanisms are not actively contributing to additional overhead. However, as the matrix size increases, the Semaphore algorithm begins to consume more memory than the others, likely due to the additional data structures required for managing semaphores.

Memory usage increases proportionally with matrix size for all algorithms, but the impact of parallelism and thread synchronization mechanisms varies significantly. Streams stands out as the most memory-efficient algorithm across all thread counts, while Semaphore and Fixed Threads demonstrate the highest memory overhead, particularly for larger matrices and higher thread counts. These results highlight the importance of balancing memory usage and parallelism to optimise performance for large-scale matrix computations.

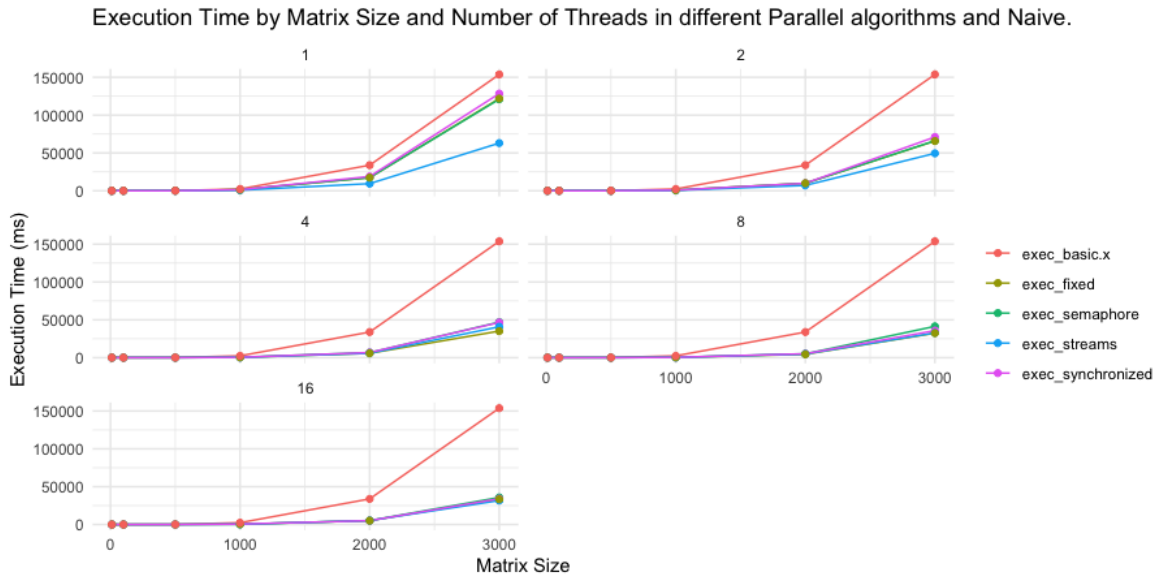

Figure 5: Execution time in parallel algorithms and naive.

The graph illustrates the execution times of various parallel algorithms for matrix multiplication, including the naive (basic) approach, across different matrix sizes and thread counts. For a single thread, the Basic Algorithm demonstrates significantly higher execution times than the parallel algorithms, particularly for larger

matrix sizes. This is expected since the basic algorithm lacks parallelization, leading to a direct and sequential computation for all elements in the matrices. The parallel algorithms, however, show comparable execution times for small matrices, as the overhead of thread management and coordination outweighs the benefits of parallelism in such cases.

As the number of threads increases to 2, parallel algorithms begin to outperform the Basic Algorithm more noticeably, especially for medium and large matrices. At this stage, the benefits of workload distribution across threads start to become evident. Among the parallel methods, Streams (Parallel Streams) and Fixed Threads demonstrate slightly better execution times than Semaphore and Synchronized Blocking, highlighting the efficiency of Java's parallel stream optimisations and thread pool management.

When the thread count increases further to 4 and 8, execution times for all parallel algorithms drop significantly compared to the basic approach. Streams consistently achieves the best performance across all matrix sizes, leveraging its highly optimised thread management. Fixed Threads and Semaphore follow closely, while Synchronized Blocking begins to show higher execution times, particularly for larger matrices, due to the additional overhead imposed by synchronisation mechanisms.

At the highest thread count (16), all parallel algorithms achieve their best execution times for larger matrices, with a dramatic reduction compared to the Basic Algorithm. However, the performance gains from adding more threads diminish slightly beyond 8 threads, likely due to thread contention and system resource limits. Even at 16 threads, Streams maintains the lowest execution times, showcasing its excellent scalability. Semaphore and Fixed Threads perform similarly but are marginally slower, reflecting the overhead of managing a larger number of threads.

Overall, the graph highlights the clear advantages of parallel algorithms for large-scale matrix computations. Streams (Parallel Streams) emerges as the most efficient and scalable algorithm, while Synchronized Blocking struggles with the added synchronisation overhead, particularly as the thread count increases. These results underscore the importance of selecting an appropriate parallelisation approach to minimise execution time, especially when dealing with larger computational workloads.

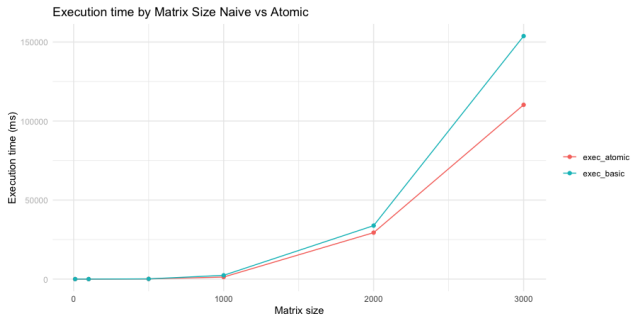

Figure 6: Execution time of Atomic comparing to algorithm with Naive.

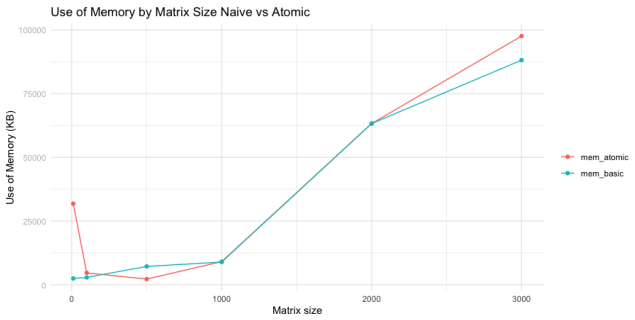

Figure 7: Memory use of Atomic comparing to algorithm with Naive.

The first graph compares the execution time of the Basic (Naive) algorithm and the Atomic algorithm for matrix multiplication across different matrix sizes. For small matrices (e.g., sizes 100 to 1000), the execution times for both algorithms are nearly identical. This is because the computational workload is too small to benefit significantly from the overhead introduced by atomic operations, resulting in comparable performance. However, as the matrix size increases, the Atomic algorithm starts to outperform the Basic algorithm, with a noticeable reduction in execution time for medium matrices (1000–2000). This is due to the better coordination and partial parallelism provided by atomic operations, which improve resource utilisation. For large matrices (2000–3000), the gap in performance becomes significant, with the Atomic algorithm achieving far lower execution times. This highlights the scalability of the Atomic algorithm, which becomes increasingly advantageous as the computational workload grows.

The second graph illustrates the memory usage of the Basic and Atomic algorithms across varying matrix sizes. For small matrices, the Atomic algorithm initially consumes slightly more memory than the Basic algorithm, reflecting the overhead of maintaining atomic variables and synchronisation structures. However, as the matrix size increases, the memory usage of both algorithms stabilises and becomes similar, particularly for medium-sized matrices (1000–2000). For large matrices (2000–3000), the Atomic algorithm uses marginally more memory than the Basic algorithm. This is expected due to the additional requirements for managing atomic operations and thread synchronisation, but the increase is relatively minor compared to the overall memory usage of the matrix data.

In summary, the Atomic algorithm demonstrates clear advantages over the Basic algorithm in terms of execution time, especially for medium to large matrices. While it incurs slightly higher memory usage, the difference is minimal and does not outweigh the substantial performance improvements in execution time. These results suggest that the Atomic algorithm is a better choice for large-scale matrix computations where reducing execution time is a priority.

4.2 Vectorized algorithm

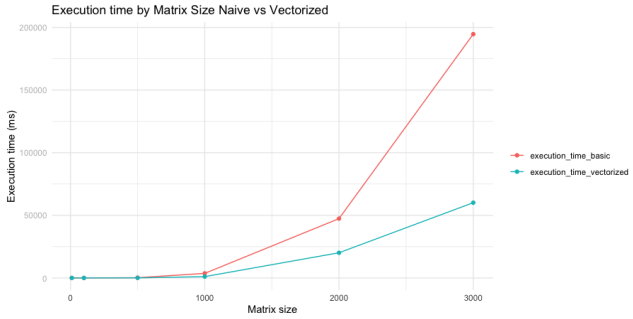

Figure 8: Execution time of Vectorized comparing to algorithm with Naive.

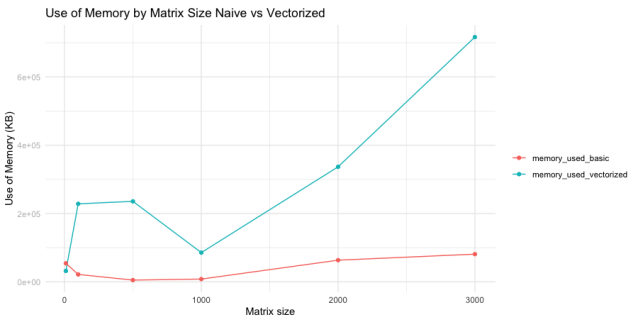

Figure 9: Memory use of Vectorized comparing to algorithm with Naive.

The fisrt graph highlights the execution time differences between the Basic and Vectorized algorithms. For small matrices, both algorithms have similar execution times, as the setup costs of vectorization (e.g., data preparation and coordination of vectorized operations) negate any potential performance gains. This suggests that vectorization is not advantageous for small workloads, where the computational demands are too low to offset its overhead.

As the matrix size increases to medium levels (1000 to 2000), the Vectorized algorithm begins to outperform the Basic algorithm, with its execution time growing at a slower rate. The benefits of vectorization become more apparent as the workload scales, leveraging parallelism and SIMD (Single Instruction, Multiple Data) operations to optimise computations. For large matrices (2000 to 3000), the performance gap widens significantly. The Basic algorithm scales poorly due to its sequential nature, resulting in much longer execution times. In contrast, the Vectorized algorithm handles the increased workload efficiently, showcasing its scalability and speed.

The second graph compares the memory usage of the Basic (Naive) algorithm and the Vectorized algorithm for matrix multiplication across varying matrix sizes. For small matrices (e.g., sizes 100 to 1000), the Vectorized algorithm exhibits significantly higher memory usage compared to the Basic algorithm. This is primarily due to the overhead associated with vectorization, such as restructuring matrices and allocating additional resources for intermediate computations. As the matrix size increases slightly, the memory usage of the Vectorized algorithm stabilises, but it remains consistently higher than that of the Basic algorithm, which only uses memory for the input and output matrices.

For medium-sized matrices (1000 to 2000), the memory usage of the Vectorized algorithm continues to grow proportionally with the matrix size. This scaling reflects the increasing demands of vectorization as more data must be handled simultaneously. The Basic algorithm, on the other hand, maintains relatively low and stable memory usage, making it a more memory-efficient option. When working with large matrices (2000 to 3000), the gap in memory usage widens even further. The Vectorized algorithm shows a steep increase in memory consumption due to the additional resources required for vectorized and parallelised operations. Meanwhile, the Basic algorithm remains much more memory-efficient but lacks the performance advantages of vectorization.

In summary, the Vectorized algorithm excels in reducing execution time for medium to large matrices, making it ideal for computationally intensive tasks. However, this performance improvement comes at the cost of significantly higher memory usage, particularly for large matrices. The Basic algorithm, while slower, remains more memory-efficient, making it suitable for smaller workloads or situations with limited memory resources. These results underscore the trade-off between memory consumption and execution time, highlighting the importance of selecting the appropriate algorithm based on the specific requirements and constraints of the computation.

4.3 Overall comparison

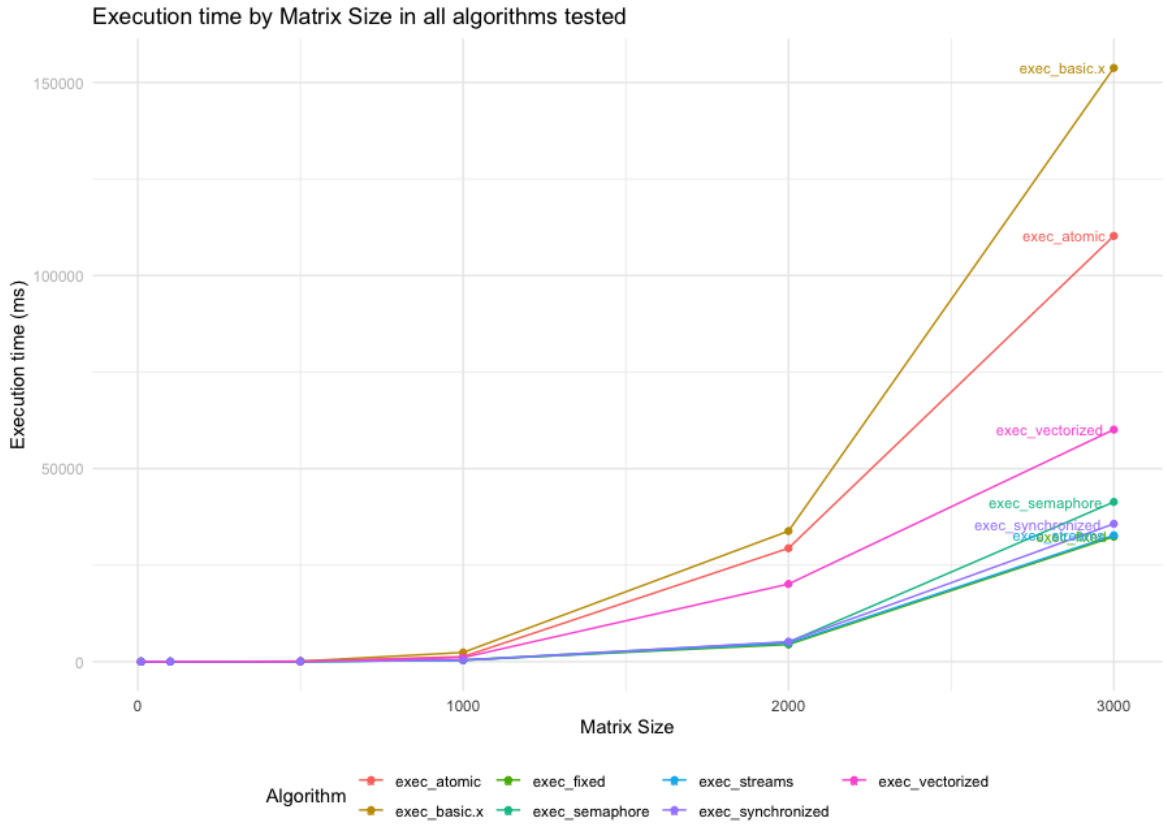

Figure 10: Execution time for all algorithms tested

This graph compares the execution times of all tested matrix multiplication algorithms across various matrix sizes using 8 threads for the parallel approaches. For small matrices (100–1000), the Basic algorithm exhibits the highest execution times due to its sequential nature. Parallel algorithms, such as Streams, Fixed Threads, and Semaphore, show similar execution times for small matrices, as the workload is too small to fully utilise parallelism, and threading overhead diminishes their advantage. The Vectorized algorithm performs similarly to these parallel approaches, as the setup costs for vectorisation offset its benefits at smaller scales.

As the matrix size increases to medium levels (1000–2000), the performance gap between the Basic algorithm and parallel approaches widens significantly. Parallel algorithms like Streams and Fixed Threads achieve much lower execution times by distributing the workload effectively across threads. The Atomic algorithm also benefits from better resource coordination but begins to fall behind optimised approaches like Streams. The Vectorized algorithm starts to demonstrate its advantages, outperforming other approaches due to its ability to process data more efficiently through SIMD operations.

For large matrices (2000–3000), the Basic algorithm exhibits a steep increase in execution time, making it unsuitable for large-scale computations. The Atomic algorithm performs better but still lags behind the more advanced parallel and vectorised techniques. Among the parallel approaches, Streams and Fixed Threads deliver similar execution times, reflecting their efficient thread management. The Vectorized algorithm achieves the lowest execution time for large matrices, showcasing its scalability and efficiency in handling computationally intensive tasks.

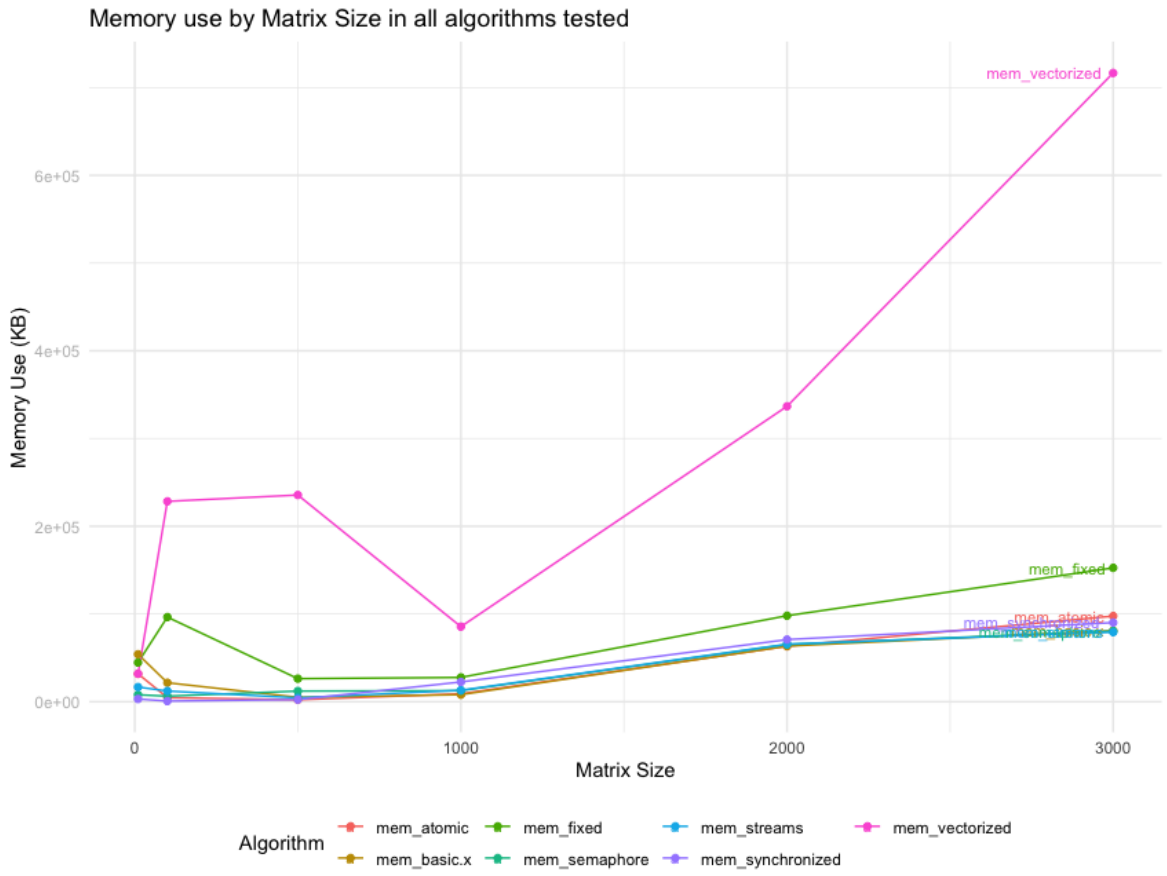

Figure 11: Memory use for all algorithms tested

This second graph compares the memory usage of the algorithms across the same matrix sizes. For small matrices (100–1000), the Vectorized algorithm consumes significantly more memory than all other approaches due to the overhead of preparing and managing data for vectorisation. Parallel algorithms, including Streams, Fixed Threads, and Semaphore, use comparable and relatively low amounts of memory, as the workload does not demand extensive additional resources. The Basic algorithm remains the most memory-efficient, as it lacks any parallel or vectorisation overhead.

As the matrix size grows to medium levels (1000–2000), the Vectorized algorithm continues to dominate in memory consumption, with its overhead becoming more apparent. Among the parallel approaches, Fixed Threads begins to show slightly higher memory usage due to the management of thread pools. The Basic algorithm maintains low and stable memory usage, though it sacrifices execution time efficiency.

For large matrices (2000–3000), the Vectorized algorithm shows a sharp increase in memory usage, significantly outpacing all other algorithms. This reflects the trade-off between its execution time advantages and its high memory overhead. The Fixed Threads approach consumes moderate memory, slightly more than Streams or Semaphore, due to its thread management structures. Meanwhile, Streams, Semaphore, and Synchronized maintain relatively low memory usage, making them more balanced options compared to the Vectorized algorithm.

In summary, the Vectorized algorithm demonstrates superior execution time performance for large matrices but at the cost of significantly higher memory usage, making it suitable for environments where memory resources are not constrained. Parallel approaches such as Streams and Fixed Threads strike a balance between execution time and memory efficiency, making them versatile for various workloads. The Basic algorithm, while highly memory-efficient, is impractical for large-scale computations due to its poor execution time performance. These results underscore the trade-offs between memory consumption and execution speed, highlighting the importance of choosing the appropriate algorithm based on system constraints and workload requirements.

5 Conclusion

5.1 Summary of Key Findings

This study explored various approaches to optimising matrix multiplication, including parallelisation techniques such as Fixed Threads, Streams, and Atomic variables, as well as vectorised computation. The results reveal clear trade-offs between execution time and memory usage. The Vectorized algorithm consistently outperformed other methods for large matrices, demonstrating its scalability and efficiency through SIMD operations. Among the parallel techniques, Streams emerged as the most balanced approach, delivering significant speedups with minimal overhead for medium and large matrices. In contrast, methods reliant on synchronisation, such as Synchronized Blocking, showed higher execution times due to thread contention and additional overhead.

Memory usage varied significantly across algorithms. The Vectorized algorithm exhibited the highest memory consumption, reflecting the cost of managing additional data structures for vectorisation. Parallel approaches like Streams and Fixed Threads demonstrated moderate memory usage, balancing resource efficiency and performance. The Basic algorithm, while the most memory-efficient, was unsuitable for large-scale computations due to its poor scalability and sequential nature.

Parallelisation benefits were most evident for medium to large matrices, where task granularity and workload distribution aligned well with system capabilities. However, efficiency declined when thread counts exceeded the physical core limit of the hardware, highlighting the importance of balancing parallelism with available system resources.

5.2 Future Work

Future work can focus on several areas to further enhance the understanding and performance of matrix multiplication. Advanced optimisation techniques, such as GPU-accelerated computation using frameworks like CUDA or OpenCL, present an opportunity to extend the benefits of parallelisation. Additionally, hybrid approaches that combine parallel and vectorised techniques could unlock new performance gains.

Energy efficiency analysis is another critical area for future study. Evaluating the energy consumption of different approaches would provide insights into their suitability for energy-constrained environments. Lastly, applying these optimised methods to real-world applications, such as deep learning or scientific simulations, would help assess their practical impact on performance and scalability.

In conclusion, by combining parallelisation and vectorisation techniques, this study establishes a foundation for scalable and efficient matrix multiplication. The results highlight the importance of selecting algorithms based on workload characteristics and hardware constraints, paving the way for future innovations in high-performance computing.

A Link to GitHub repository

https://github.com/alejandroalemanaleman/ParallelAndVectorizedMatrixMultiplication.git