Paralellization Benchmark of matrix multiplication

Jorge Lorenzo Lorenzo

November 23, 2024

Abstract

This task focuses on optimizing matrix multiplication through parallel computing and vectorized approaches. Three implementations were developed: a basic sequential algorithm, a parallel version using multi-threading, and a vectorized version based on the Apache Commons Math library. Experiments with large matrices evaluated metrics such as execution time, speedup, and resource usage. Results showed that the vectorized approach achieved the best performance with a speedup of 8.73, followed by the parallel implementation with 4.75 and an efficiency of 59%. Although vectorization did not directly utilize SIMD instructions, the findings demonstrate that both parallel and library-optimized approaches significantly outperform the sequential algorithm.

Contents

1 Introduction

Matrix multiplication is an essential operation in various computational fields, such as machine learning, scientific simulations, and data analysis. Due to its computational complexity $O(n^3)$, optimizing this operation is crucial for improving the performance of applications that rely on intensive calculations. In this context, techniques like parallel computing and the use of optimized libraries have proven to be effective tools for accelerating this process.

Parallel computing divides computational tasks into multiple threads that execute simultaneously on different processor cores, maximizing the utilization of available hardware. Additionally, the use of specialized libraries, such as Apache Commons Math, simplifies the implementation of optimized algorithms by leveraging internal improvements to enhance matrix operations' efficiency. While these approaches have been extensively studied, their practical implementation and direct comparison in a uniform context remain an interesting challenge.

This work implements and compares three approaches to matrix multiplication: a basic sequential algorithm, a parallel implementation using multiple threads, and a vectorized version employing Apache Commons Math. The results provide a clear understanding of the advantages and limitations of each approach, emphasizing the importance of modern techniques for computational performance optimization.

2 Methodology

This work introduces four main classes designed to implement and analyze different approaches to matrix multiplication. Three of these classes focus on the algorithms: the base (sequential) algorithm, the parallel (multi-threaded) algorithm, and the vectorized algorithm using the Apache Commons Math library. The fourth class, named `Main`, acts as a controller to generate the input matrices, execute the algorithms, and display their results. This modular structure allows for a clear evaluation of the differences between the implementations and facilitates the replication of experiments by the reader.

2.1 Matrix Multiplication Algorithms

2.1.1 Sequential Approach (Base)

The sequential approach, implemented in the `MatrixMultiplicationBase` class, performs matrix multiplication following the standard method, using three nested loops. In this algorithm, each element of the resulting matrix `c` is calculated as the dot product of the corresponding row from `a` and the corresponding column from `b`. This method is straightforward but computationally intensive, with a time complexity of $O(n^3)$.

Algorithm Flow:

- Initialize the matrix `c` with zero values.
- Iterate over each row `i` of matrix `a`.
- For each row, iterate over each column `j` of matrix `b`.
- Compute each element of `c[i][j]` by summing the product of corresponding elements:

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot b[k][j]$$

Java Implementation: The following code snippet demonstrates the base approach implementation in the `MatrixMultiplicationBase` class:

```java
package software.ulpgc;

public class MatrixMultiplicationBase {
    private final double[][] a;
    private final double[][] b;
    private final double[][] c;

    public MatrixMultiplicationBase(double[][] a, double[][] b) {
        this.a = a;
        this.b = b;
        this.c = new double[a.length][b[0].length];
    }

    public long execute() {
        Runtime runtime = Runtime.getRuntime();
        long initialMemory = runtime.totalMemory() - runtime.freeMemory();

```

```
        long start = System.currentTimeMillis();
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < b[0].length; j++) {
                for (int k = 0; k < b.length; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        long stop = System.currentTimeMillis();

        long finalMemory = runtime.totalMemory() - runtime.freeMemory();
        System.out.printf("Memory used (MB): %d\n", (finalMemory -
            initialMemory) / (1024 * 1024));
        System.out.printf("Execution time (s): %.3f\n", (stop - start) * 1e-3)
            ;

        return stop - start; // Returns time in milliseconds
    }
}
```

Listing 1: Sequential Matrix Multiplication

Observations: This algorithm serves as a baseline for evaluating optimized approaches. Its simplicity makes it easy to implement, but it does not take advantage of modern hardware resources such as multiple cores or vectorization optimizations.

2.1.2 Parallel Approach (Multi-threaded)

The parallel approach, implemented in the `MatrixMultiplicationParallel` class, uses multiple threads to divide the workload of matrix multiplication across processor cores. Each thread is responsible for processing a specific range of rows from the input matrix `a`, allowing the computation to be performed concurrently and efficiently.

Algorithm Flow:

- Divide the rows of the input matrix `a` evenly among a predefined number of threads (`numThreads`).
- Each thread computes the corresponding rows of the resulting matrix `c`, processing only the assigned rows.
- Once all threads complete their calculations, the final matrix is assembled automatically.

Thread Assignment: The rows of matrix `a` are divided among the threads as follows:

- Each thread processes a block of rows, determined by the variable `rowsPerThread`.
- The starting row (`startRow`) for thread `t` is calculated as:

$$\text{startRow} = t \cdot \text{rowsPerThread}$$

- The ending row (`endRow`) is:

$$\text{endRow} = \text{startRow} + \text{rowsPerThread}$$

- For the last thread, `endRow` is adjusted to ensure all rows are processed, even if `a.length` is not perfectly divisible by `numThreads`.

Java Implementation: The following code snippet demonstrates the parallel matrix multiplication algorithm:

```java
package software.ulpgc;

public class MatrixMultiplicationParallel {
    private final double[][] a;
    private final double[][] b;
    private final double[][] c;
    private final int numThreads;

    public MatrixMultiplicationParallel(double[][] a, double[][] b, int
        numThreads) {
        this.a = a;
        this.b = b;
        this.c = new double[a.length][b[0].length];
        this.numThreads = numThreads;
    }

    public long execute() {
        Runtime runtime = Runtime.getRuntime();
        long initialMemory = runtime.totalMemory() - runtime.freeMemory();

        Thread[] threads = new Thread[numThreads];
        int rowsPerThread = a.length / numThreads;

        long start = System.currentTimeMillis();
        for (int t = 0; t < numThreads; t++) {
            final int startRow = t * rowsPerThread;
            final int endRow = (t == numThreads - 1) ? a.length : startRow +
                rowsPerThread;
            threads[t] = new Thread(() -> {
                for (int i = startRow; i < endRow; i++) {
                    for (int j = 0; j < b[0].length; j++) {
                        for (int k = 0; k < b.length; k++) {
                            c[i][j] += a[i][k] * b[k][j];
                        }
                    }
                }
            });
            threads[t].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long stop = System.currentTimeMillis();

        long finalMemory = runtime.totalMemory() - runtime.freeMemory();
        System.out.printf("Memory used (MB): %d\n", (finalMemory -
            initialMemory) / (1024 * 1024));
        System.out.printf("Execution time (s): %.3f\n", (stop - start) * 1e-3)
            ;

        return stop - start;
    }
}
```

Listing 2: Parallel Matrix Multiplication with Threads

Observations: This algorithm significantly reduces execution time by leveraging multiple threads. The division of rows ensures an even workload distribution, with adjustments for the last thread to avoid missing rows. The method highlights the benefits of parallelization for computationally intensive tasks.

2.1.3 Vectorized Approach

The vectorized approach, implemented in the `MatrixMultiplicationVectorized` class, utilizes the Apache Commons Math library to simplify and optimize matrix multiplication. Instead of implementing the multiplication algorithm manually, this approach relies on the internal optimizations provided by the library's `RealMatrix` class. This significantly reduces the code complexity while improving execution efficiency.

Algorithm Flow:

- The input matrices (`a` and `b`) are converted into `RealMatrix` objects using the `MatrixUtils.createRealMatrix()` method from the Apache Commons Math library.
- The multiplication is performed using the `multiply()` method, which computes the resulting matrix.
- Execution time and memory usage are measured during the process.

Required Dependency: To use the Apache Commons Math library, the following Maven dependency must be added to the project's `pom.xml` file:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-math3</artifactId>
    <version>3.6.1</version>
</dependency>
```

Listing 3: Apache Commons Math Maven Dependency

Java Implementation: The following code demonstrates the implementation of the vectorized matrix multiplication:

```
package software.ulpgc;

import org.apache.commons.math3.linear.MatrixUtils;
import org.apache.commons.math3.linear.RealMatrix;

public class MatrixMultiplicationVectorized {
    private final RealMatrix a;
    private final RealMatrix b;

    public MatrixMultiplicationVectorized(double[][] a, double[][] b) {
        this.a = MatrixUtils.createRealMatrix(a);
        this.b = MatrixUtils.createRealMatrix(b);
    }

    public long execute() {
        Runtime runtime = Runtime.getRuntime();
        long initialMemory = runtime.totalMemory() - runtime.freeMemory();

        long start = System.currentTimeMillis();
```

```
        RealMatrix result = a.multiply(b); // Vectorized operation
        long stop = System.currentTimeMillis();

        long finalMemory = runtime.totalMemory() - runtime.freeMemory();
        System.out.printf("Memory used (MB): %d\n", (finalMemory -
            initialMemory) / (1024 * 1024));
        System.out.printf("Execution time (s): %.3f\n", (stop - start) * 1e-3)
            ;

        return stop - start;
    }
}
```

Listing 4: Vectorized Matrix Multiplication with Apache Commons Math

Observations: The vectorized approach benefits from the internal optimizations of the Apache Commons Math library, allowing for efficient matrix operations with minimal implementation effort. However, its performance depends on the library's implementation and may not fully utilize all available CPU cores compared to a custom multi-threaded solution. This method is particularly useful for reducing development time while achieving competitive performance for large matrix operations.

2.2 Matrix Multiplication Execution and Metrics

2.2.1 Main Class: Comparing Algorithms and Evaluating Performance

The `Main` class serves as the entry point for the program and is responsible for orchestrating the execution of the three matrix multiplication algorithms: sequential (base), parallel, and vectorized. Additionally, it measures and reports key metrics such as execution time, speedup, and parallel efficiency.

Algorithm Flow: The `Main` class follows these steps:

1. **Generate Random Matrices:** Two square matrices, `matrixA` and `matrixB`, of size $n \times n$ are created using the `generateRandomMatrix()` method, which fills them with random double values.
2. **Execute Sequential Algorithm:** The sequential algorithm (`MatrixMultiplicationBase`) is executed first to establish a baseline for performance comparison. The execution time is stored in `baseTime`.
3. **Execute Parallel Algorithm:** The parallel algorithm (`MatrixMultiplicationParallel`) is executed with a predefined number of threads (8 in this case). Metrics calculated:
   - `speedupParallel`: Ratio of the sequential execution time to the parallel execution time:
   $$\text{speedupParallel} = \frac{\text{baseTime}}{\text{parallelTime}}$$
   - `efficiencyParallel`: Speedup divided by the number of threads:
   $$\text{efficiencyParallel} = \frac{\text{speedupParallel}}{8}$$
4. **Execute Vectorized Algorithm:** The vectorized algorithm (`MatrixMultiplicationVectorized`) is executed. Its speedup relative to the sequential algorithm is calculated as:
   $$\text{speedupVectorized} = \frac{\text{baseTime}}{\text{vectorizedTime}}$$
5. **Print Metrics:** For each algorithm, the execution time, memory usage, and performance metrics are displayed in the console.

Java Implementation: The following code shows the implementation of the `Main` class:

```java
package software.ulpgc;

import java.util.Random;

public class Main {
    static int n = 1024; // Matrix size
    static double[][] matrixA;
    static double[][] matrixB;

    public static void main(String[] args) {
        // Generate random matrices
        matrixA = generateRandomMatrix(n, n);
        matrixB = generateRandomMatrix(n, n);

        // Sequential execution (Base)
        System.out.println("\n--- Base (Sequential) ---");
        MatrixMultiplicationBase base = new MatrixMultiplicationBase(matrixA,
            matrixB);
        long baseTime = base.execute();

        // Parallel execution
        System.out.println("\n--- Parallel (Multi-threaded) ---");
        MatrixMultiplicationParallel parallel = new
            MatrixMultiplicationParallel(matrixA, matrixB, 8); // 8 threads
        long parallelTime = parallel.execute();
        double speedupParallel = (double) baseTime / parallelTime;
        System.out.printf("Speedup (Parallel): %.2f\n", speedupParallel);
        System.out.printf("Efficiency (Parallel): %.2f\n", speedupParallel /
            8);

        // Vectorized execution
        System.out.println("\n--- Vectorized ---");
        MatrixMultiplicationVectorized vectorized = new
            MatrixMultiplicationVectorized(matrixA, matrixB);
        long vectorizedTime = vectorized.execute();
        double speedupVectorized = (double) baseTime / vectorizedTime;
        System.out.printf("Speedup (Vectorized): %.2f\n", speedupVectorized);
    }

    public static double[][] generateRandomMatrix(int rows, int cols) {
        Random random = new Random();
        double[][] matrix = new double[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = random.nextDouble();
            }
        }
        return matrix;
    }
}
```

Listing 5: Main Class for Algorithm Execution and Metric Evaluation

Observations: The `Main` class effectively integrates the three algorithms and automates the metric evaluation process:

- It ensures fair comparisons by using the same input matrices (`matrixA` and `matrixB`) for all algorithms.

- The calculated metrics, such as speedup and efficiency, provide insights into the performance gains achieved by parallelization and vectorization.
- The modular design allows easy modification of parameters, such as the matrix size or the number of threads, for further experimentation.

3 Experiments

The experiments conducted are based on the execution of the three developed algorithms (sequential, parallel, and vectorized) under various parameter configurations. For the parallel algorithm, tests will be performed by varying the number of threads to 2, 4, and 8. Additionally, different sizes of square matrices will be tested: 256×256, 512×512, 1024×1024, and 2048×2048.

In the experiments described, several key metrics will be analyzed to evaluate the performance and efficiency of the three algorithms under the parameter configurations mentioned. These metrics include:

- **Memory Used (MB):** Indicates the amount of memory consumed during the execution of the algorithms. This metric is particularly useful for assessing resource efficiency, especially in algorithms requiring more complex structures, such as the vectorized approach.
- **Execution Time (s):** Represents the total time required by the algorithm to complete the matrix multiplication. Comparing this value across configurations helps identify which approaches are faster.
- **Speedup (Parallel and Vectorized):** Measures the performance improvement by comparing execution times against the sequential algorithm. It is calculated as:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Algorithm Time}}$$

  This metric is crucial to understanding the impact of parallelization and vectorization.
- **Parallel Efficiency:** Evaluates how system resources are utilized in the parallel algorithm. It is defined as:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

  High efficiency indicates that threads are working effectively, while low efficiency may point to bottlenecks or synchronization overhead.

These metrics will be represented in tables and charts to provide a clear view of the performance and resource usage of the algorithms under different configurations.

3.1 Analysis of Results with Number of Threads = 2

The following analysis examines the results obtained for each matrix size with the number of threads fixed at 2. The parallel efficiency has been recalculated, and key observations are highlighted for each experiment.

Results by Matrix Size

1. Matrix Size: 256×256

Table 1: Metrics for 256×256 Matrix (Threads = 2)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.080	0.022	-	-
Parallel	0.121	0.018	1.22	0.61
Vectorized	0.507	0.017	1.29	-

Observations:

- The parallel algorithm achieves a small improvement in execution time with a speedup of 1.22 and an adjusted efficiency of 0.61.
- The vectorized approach shows a better execution time than the parallel algorithm but at the cost of higher memory usage (0.507 MB vs. 0.121 MB).

2. Matrix Size: 512×512

Table 2: Metrics for 512×512 Matrix (Threads = 2)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.160	0.173	-	-
Parallel	0.320	0.111	1.56	0.78
Vectorized	2.011	0.064	2.70	-

Observations:

- The improvement in the parallel algorithm is more evident with a speedup of 1.56 and an efficiency of 0.78.
- The vectorized approach significantly reduces execution time (0.064 s) but uses more memory (2.011 MB vs. 0.320 MB for the parallel algorithm).

3. Matrix Size: 1024×1024

Table 3: Metrics for 1024×1024 Matrix (Threads = 2)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.260	3.296	-	-
Parallel	0.223	1.533	2.15	1.08
Vectorized	8.084	0.453	7.28	-

Observations:

- The parallel algorithm starts to show a greater advantage with a speedup of 2.15 and an efficiency of 1.08, indicating that parallelism is effective for this size.
- The vectorized approach excels with a speedup of 7.28, being highly efficient for shorter execution times, although it uses significantly more memory (8.084 MB).

4. Matrix Size: 2048×2048

Table 4: Metrics for 2048×2048 Matrix (Threads = 2)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.800	43.398	-	-
Parallel	0.456	21.537	2.02	1.01
Vectorized	33.008	3.217	13.49	-

Observations:

- The parallel algorithm shows a speedup of 2.02 and an efficiency of 1.01, demonstrating a significant improvement in execution time.
- The vectorized approach achieves a remarkable reduction in execution time (3.217 s) with a speedup of 13.49 but uses 33.008 MB of memory, making it less resource-efficient.

Graph: Execution Time The graph below illustrates the execution time for the three algorithms (sequential, parallel, and vectorized) for varying matrix sizes with the number of threads fixed at 2. This visualization highlights the comparative performance of the algorithms as the problem size increases.

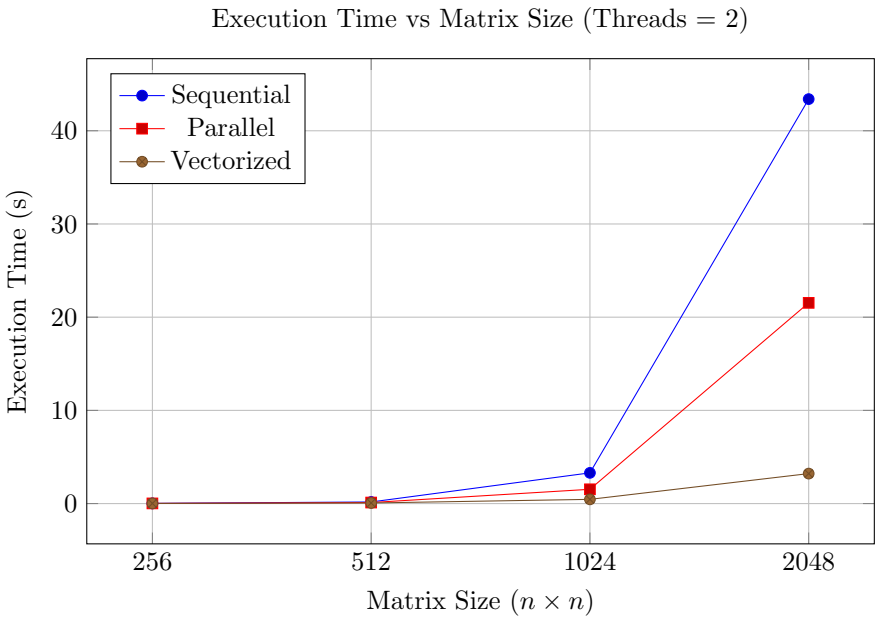

Figure 1: Execution Time for Different Algorithms (Threads = 2)

Graph Observations

- **Sequential:** The execution time increases exponentially with the matrix size due to the sequential nature of the operations. This makes the sequential algorithm inefficient for large matrices.
- **Parallel:** The parallel algorithm significantly reduces execution time compared to the sequential version, especially for larger matrix sizes, such as 1024×1024 and 2048×2048. This demonstrates the effectiveness of multithreading.
- **Vectorized:** The vectorized algorithm consistently achieves the shortest execution time across all matrix sizes. However, the performance improvement is more noticeable for larger matrices, while for smaller ones, the overhead of vectorization may slightly offset the gain.
- **Overall:** As the matrix size increases, the performance gap between the three algorithms widens, with the vectorized approach excelling at larger scales. The parallel algorithm offers a balance between performance and resource usage.

Conclusion The graph clearly demonstrates the advantages of parallelization and vectorization over sequential execution. While vectorization is the most efficient in terms of execution time, it requires significantly more memory compared to the parallel algorithm. This trade-off highlights the importance of selecting the appropriate approach based on the specific requirements of the application.

3.2 Analysis of Results with Number of Threads = 4

The following analysis examines the results obtained for each matrix size with the number of threads fixed at 4. The parallel efficiency has been recalculated, and key observations are highlighted for each experiment.

Results by Matrix Size

1. Matrix Size: 256×256

Table 5: Metrics for 256×256 Matrix (Threads = 4)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.080	0.024	-	-
Parallel	0.121	0.028	0.86	0.22
Vectorized	0.507	0.018	1.33	-

Observations:

- The parallel algorithm shows limited improvement with a speedup of 0.86 and an efficiency of 0.22, likely due to the small matrix size not fully utilizing multithreading.
- The vectorized approach is more efficient, achieving a better execution time (0.018 s) with higher memory usage (0.507 MB).

2. Matrix Size: 512×512

Table 6: Metrics for 512×512 Matrix (Threads = 4)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.160	0.186	-	-
Parallel	0.320	0.074	2.51	0.63
Vectorized	2.011	0.071	2.62	-

Observations:

- The parallel algorithm improves significantly with a speedup of 2.51 and an efficiency of 0.63, showing better multithreading utilization.
- The vectorized approach achieves a slightly better execution time (0.071 s) compared to parallel, but with higher memory consumption (2.011 MB).

3. Matrix Size: 1024×1024

Table 7: Metrics for 1024×1024 Matrix (Threads = 4)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.260	3.245	-	-
Parallel	0.185	0.785	4.13	1.03
Vectorized	8.049	0.463	7.01	-

Observations:

- The parallel algorithm achieves a speedup of 4.13 and an efficiency of 1.03, demonstrating excellent multithreading performance for larger matrix sizes.
- The vectorized approach continues to excel with a speedup of 7.01, but at the cost of higher memory usage (8.049 MB).

4. Matrix Size: 2048×2048

Table 8: Metrics for 2048×2048 Matrix (Threads = 4)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.800	43.520	-	-
Parallel	0.494	11.622	3.74	0.93
Vectorized	33.008	3.583	12.15	-

Observations:

- The parallel algorithm achieves a speedup of 3.74 and an efficiency of 0.93, showing significant improvements in execution time for large matrices.
- The vectorized approach achieves the fastest execution time (3.583 s) with a speedup of 12.15, though its memory usage remains considerably higher (33.008 MB).

Graph: Execution Time The graph below illustrates the execution time for the three algorithms (sequential, parallel, and vectorized) for varying matrix sizes with the number of threads fixed at 4. This visualization highlights the comparative performance of the algorithms as the problem size increases.

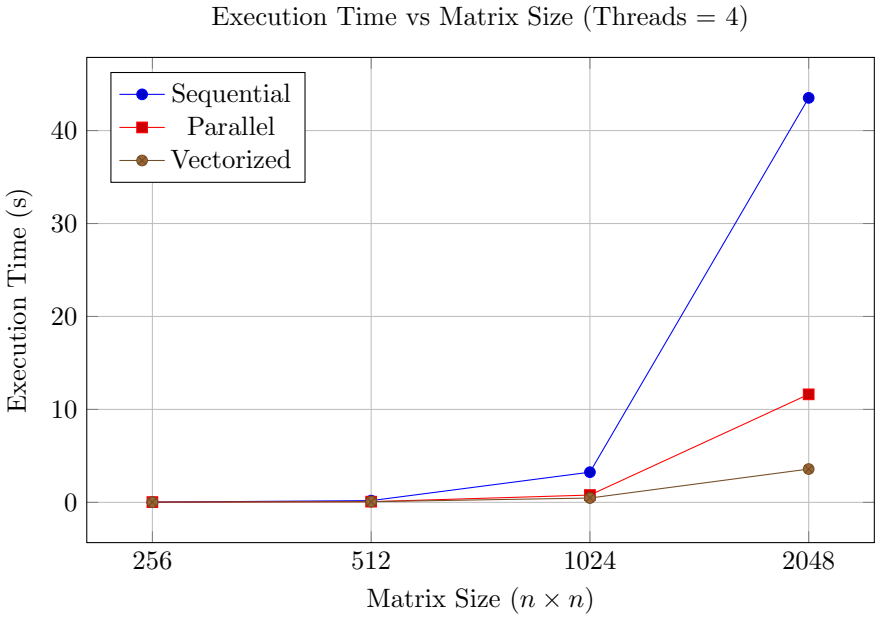

Figure 2: Execution Time for Different Algorithms (Threads = 4)

Graph Observations

- **Sequential:** The execution time increases exponentially with the matrix size due to the sequential nature of the operations. For larger matrices (2048×2048), the execution time reaches 43.52 seconds.
- **Parallel:** The parallel algorithm shows significant improvement, particularly for medium to large matrix sizes. The execution time for 2048×2048 is reduced to 11.622 seconds, demonstrating effective multithreading.
- **Vectorized:** The vectorized algorithm achieves the shortest execution times for all matrix sizes, with a time of just 3.583 seconds for 2048×2048. However, this comes at the cost of higher memory usage compared to the parallel algorithm.

Conclusion The graph clearly demonstrates the advantages of both parallelization and vectorization over the sequential approach. The parallel algorithm strikes a balance between performance and memory usage, making it suitable for large-scale problems. However, the vectorized algorithm excels in reducing execution time significantly, albeit with higher memory requirements. The choice of algorithm should depend on the specific constraints and goals of the application.

3.3 Analysis of Results with Number of Threads = 8

The following analysis examines the results obtained for each matrix size with the number of threads fixed at 8. Key observations are highlighted for each experiment.

Results by Matrix Size

1. Matrix Size: 256×256

Table 9: Metrics for 256×256 Matrix (Threads = 8)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.080	0.023	-	-
Parallel	0.121	0.035	0.66	0.08
Vectorized	0.507	0.017	1.35	-

Observations:

- The parallel algorithm shows limited improvement with a speedup of 0.66 and an efficiency of 0.08, suggesting that the matrix size is too small to benefit from 8 threads.
- The vectorized algorithm outperforms the parallel approach with a better execution time (0.017 s) but at a cost of higher memory usage (0.507 MB).

2. Matrix Size: 512×512

Table 10: Metrics for 512×512 Matrix (Threads = 8)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	0.160	0.184	-	-
Parallel	0.320	0.069	2.67	0.33
Vectorized	2.011	0.070	2.63	-

Observations:

- The parallel algorithm improves significantly with a speedup of 2.67 and an efficiency of 0.33, indicating better utilization of threads for this matrix size.
- The vectorized algorithm achieves a similar execution time (0.070 s) but uses more memory (2.011 MB).

3. Matrix Size: 1024×1024

Table 11: Metrics for 1024×1024 Matrix (Threads = 8)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.260	3.386	-	-
Parallel	0.223	0.756	4.48	0.56
Vectorized	8.122	0.456	7.43	-

Observations:

- The parallel algorithm achieves a speedup of 4.48 and an efficiency of 0.56, demonstrating effective use of 8 threads.
- The vectorized algorithm excels with a speedup of 7.43, but it uses significantly more memory (8.122 MB).

4. Matrix Size: 2048×2048

Table 12: Metrics for 2048×2048 Matrix (Threads = 8)

Algorithm	Memory Used (MB)	Execution Time (s)	Speedup	Parallel Efficiency
Sequential	1.800	44.653	-	-
Parallel	0.494	9.245	4.83	0.60
Vectorized	33.008	3.648	12.24	-

Observations:

- The parallel algorithm achieves a speedup of 4.83 and an efficiency of 0.60, highlighting significant time reduction for large matrix sizes.
- The vectorized algorithm achieves the fastest execution time (3.648 s) with a speedup of 12.24, though it consumes much more memory (33.008 MB).

Graph: Execution Time The graph below illustrates the execution time for the three algorithms (sequential, parallel, and vectorized) for varying matrix sizes with the number of threads fixed at 8.

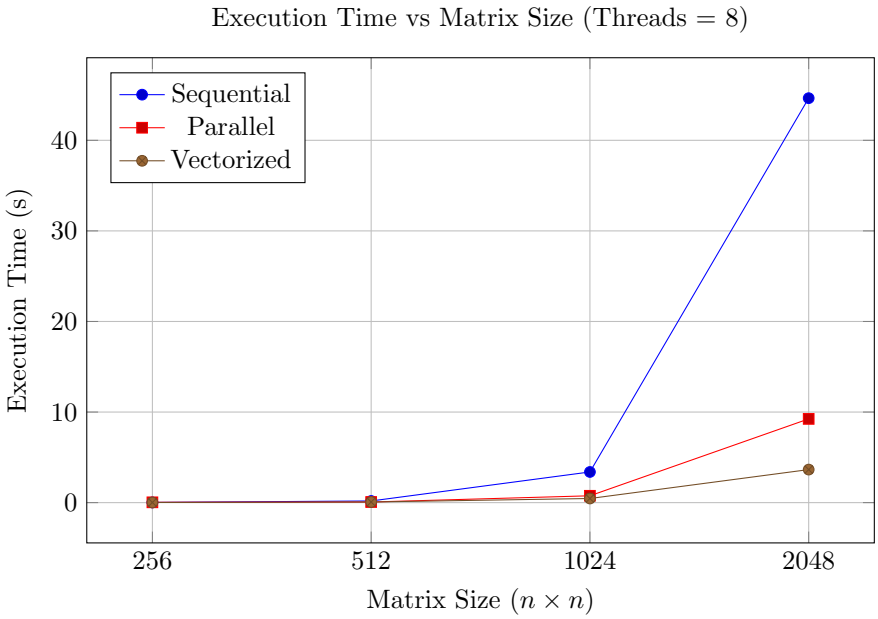

Figure 3: Execution Time for Different Algorithms (Threads = 8)

Graph Observations

- **Sequential:** The execution time increases exponentially with the matrix size, reaching 44.653 seconds for 2048×2048.
- **Parallel:** The parallel algorithm significantly reduces execution time for larger matrices, with the highest reduction for 2048×2048 (9.245 seconds).
- **Vectorized:** The vectorized algorithm consistently achieves the shortest execution times, with the best performance at larger matrix sizes, though it uses significantly more memory.

Conclusion The results demonstrate that while the vectorized algorithm provides the fastest execution times, the parallel algorithm offers a good balance of speed and memory efficiency, particularly for larger matrices. As the matrix size increases, the performance gap between these approaches becomes more pronounced, emphasizing the importance of selecting an appropriate method based on system constraints.

4 Conclusion

This research has evaluated the efficiency and performance of three matrix multiplication algorithms: sequential, parallel, and vectorized. The results highlight that, although the vectorized approach consumes significantly more memory compared to the other methods, it delivers the best execution times, particularly for large matrix sizes.

On the other hand, the parallel algorithm demonstrates steady progress as the number of threads increases, achieving a good balance between execution time and resource usage. While it does not match the vectorized algorithm in terms of speed, its lower memory consumption makes it more efficient for systems with resource constraints.

Compared to the sequential algorithm, both the parallel and vectorized approaches represent substantial improvements in execution time. The choice between these methods depends on the system's priorities, whether maximizing performance or minimizing memory usage. These conclusions emphasize the importance of carefully selecting the parallelization strategy based on the specific requirements of the problem at hand.

5 Future Work

This study has established a solid foundation for evaluating matrix multiplication using sequential, parallel, and vectorized approaches. However, future research could focus on:

- **Optimization:** Explore techniques like OpenMP or CUDA and test other vectorization libraries to enhance performance.
- **Matrix Variety:** Experiment with rectangular matrices to analyze how their shape affects performance.
- **Advanced Metrics:** Use tools to measure energy consumption and cache usage in addition to memory and time.
- **Comparison:** Compare performance with algorithms like Strassen or Winograd.
- **Real-World Applications:** Test these algorithms in practical contexts such as neural networks or simulations.

These directions provide opportunities to optimize and broaden the applicability of the algorithms.

6 GitHub

For more information about the code and implementation details, please visit the GitHub repository: GitHub Repository - Task 3