Matrix Multiplication Stage 3

José Luis Perdomo

December 2024

Abstract

This paper presents the third iteration of a matrix multiplication project. In this version, additional implementations have been introduced using vectorization and parallelization techniques. However, the results obtained do not surpass those achieved in the previous iteration, where parallelization was applied using arrays. Despite the advanced optimizations, the performance improvements were not as expected, suggesting that further tuning and alternative approaches might be needed for better performance.

1 Introduction

Matrix multiplication is a key operation in computational mathematics with numerous applications across various domains such as scientific computing, machine learning, and data analysis. The optimization of matrix multiplication algorithms is essential due to their frequent use in large-scale computations, where even minor improvements in performance can lead to significant reductions in execution time. This paper explores the performance of matrix multiplication algorithms with an emphasis on optimizing execution time through the use of vectorization and parallelization techniques.

In a previous iteration of this project, matrix multiplication was implemented using parallelization with arrays, where performance improvements were achieved for medium-sized matrices. However, these implementations did not scale well to larger matrix sizes, and the optimization potential was limited. In this work, we explore the application of vectorization combined with parallelism to further improve the algorithm's performance.

Vectorization and parallelization techniques were introduced in the current version of the project to speed up computations by utilizing the capabilities of modern processors. Vectorization aims to process multiple data points simultaneously, and parallelization divides the work among multiple processors or threads. Despite the theoretical advantages of these methods, the practical results in terms of performance have been underwhelming. Specifically, using Java Streams, both with and without parallelization, has led to significant memory overhead, and performance has decreased in comparison to the previous iteration. This suggests that the overhead from managing multiple threads and the complexity of vectorized operations outweighs the benefits, particularly for smaller matrices or those with lower computational demands.

The significance of this work lies in the ongoing efforts to optimize matrix multiplication, which serves as a foundation for many computational tasks. The inability of vectorization and parallelization to outperform simpler methods highlights the importance of choosing the right optimization strategies for different types of matrices and computational environments. This paper discusses the challenges faced, the trade-offs between memory usage and execution speed, and the lessons learned from this round of optimizations.

2 Methodology

In this document, we describe the implementation of matrix multiplication using both the traditional method and Strassen's algorithm. The implementation is divided into four classes, each focusing on different methods and optimizations for matrix multiplication. The goal is to explore the performance improvements achieved by parallelization and the Strassen algorithm.

All code was developed in IntelliJ IDEA using JDK 17, and the source code is publicly available at Matrix Multiplication Repository.

2.1 Matrix Multiplication (Naive Approach)

The first class implements a naive approach to matrix multiplication. This method computes the product of two matrices by iterating over each row and column, performing the multiplication of corresponding elements and summing the results. The computation is performed in parallel using Java's Stream API to speed up the process for larger matrices.

Code Implementation: The ‘MatrixMultiplication‘ class defines a method for generating random matrices and performing matrix multiplication using the naive approach. It also uses Java's ‘IntStream.parallel()‘ for parallel processing.

Algorithm 1 Matrix Multiplication (Naive)

```
  **Input:** Two matrices $A$ and $B$ of size $N \times N$
  **Output:** Product matrix $C$
  Initialize matrix $C$ of size $N \times N$
  for each row $i$ of matrix $A$ do
    for each column $j$ of matrix $B$ do
      $C[i][j] = \sum_{k=0}^{N-1} A[i][k] \times B[k][j]$
    end for
  end for
```

```
public class MatrixMultiplication {
    public static double[][] generateRandomMatrix(int N) {
        Random random = new Random();
        double[][] matrix = new double[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                matrix[i][j] = random.nextDouble() * 10;
            }
        }
        return matrix;
    }

    public static void multiply(double[][] A, double[][] B) {
        int n = A.length;
        int m = B[0].length;
        int p = B.length;
        double[][] result = new double[n][m];

        IntStream.range(0, n).parallel().forEach(i -> {
            for (int j = 0; j < m; j++) {
                int finalJ = j;
                result[i][j] = IntStream.range(0, p)
                        .mapToDouble(k -> A[i][k] * B[k][finalJ])
                        .sum();
            }
        });
    }
}
```

2.2 Strassen's Algorithm for Matrix Multiplication

Strassen's algorithm is a divide-and-conquer method for matrix multiplication that reduces the number of multiplications required. Instead of computing each element of the result matrix directly, it breaks the problem into smaller subproblems and recursively computes these using only seven multiplications.

Code Implementation: The ‘Strassen Vect‘ class implements Strassen’s algorithm by recursively splitting the input matrices into submatrices and using the seven multiplications defined in Strassen’s method.

```
public class Strassen_Vect {
    public static double[][] strassen(double[][] A, double[][] B) {
        int n = A.length;
        if (n <= 128) {
            return multiplyNaive(A, B); // Base case
        }

        int newSize = n / 2;
        // Divide A and B into 4 submatrices each
        double[][] a11 = new double[newSize][newSize];
        double[][] a12 = new double[newSize][newSize];
        double[][] a21 = new double[newSize][newSize];
        double[][] a22 = new double[newSize][newSize];
        // Similarly, divide B into b11, b12, b21, b22

        // Compute M1 to M7 using recursive strassen calls
        double[][] m1 = strassen(add(a11, a22), add(b11, b22));
        double[][] m2 = strassen(add(a21, a22), b11);
        // More multiplication calls for M3, M4, M5, M6, M7

        // Combine the results into the final result matrix
    }
}
```

2.3 Sparse Matrix Strassen’s Algorithm

The third class is an extension of Strassen’s algorithm, adapted for sparse matrices. Sparse matrices are represented using a map, where only non-zero elements are stored, thus optimizing memory usage and computation time for matrices with a high proportion of zero elements.

Code Implementation: The ‘Strassen Vect Sparc‘ class implements sparse matrix multiplication using Strassen’s algorithm. It uses a ‘SparseMatrix‘ class that stores only non-zero elements in a map.

```
public class Strassen_Vect_Sparc {
    public static SparseMatrix strassen(SparseMatrix A, SparseMatrix B) {
        int n = A.getSize();
        if (n <= 128) {
            return multiplyNaive(A, B); // Base case for sparse matrices
        }

        // Split matrices A and B into submatrices
        SparseMatrix a11 = extractSubMatrix(A, 0, 0, newSize);
        SparseMatrix b11 = extractSubMatrix(B, 0, 0, newSize);

        // Use Strassen recursively to compute M1, M2, ..., M7
    }
}
```

2.4 Strassen's Algorithm with Threading

The fourth class, 'Strassen Vect Thread', optimizes Strassen's algorithm by introducing multithreading. When the matrix size exceeds a certain threshold, submatrix multiplications are performed in parallel using Java's 'ExecutorService' to enhance performance further.

Code Implementation: The 'Strassen Vect Thread' class uses Java's concurrency framework to parallelize the recursive multiplication tasks. The main computation is split into subproblems, which are then assigned to threads.

```
public class Strassen_Vect_Thread {
    public static double[][] strassenWithThreads(double[][] A, double[][] B) {
        int n = A.length;
        if (n <= THREAD_THRESHOLD) {
            return multiplyNaive(A, B); // Base case for small matrices
        }

        // Divide and submit tasks to ExecutorService
        ExecutorService executor = Executors.newFixedThreadPool(4);
        List<Future<double[][]>> futures = new ArrayList<>();

        // Submit tasks for M1 to M7 computations
        futures.add(executor.submit(() -> strassenTask(A1, B1)));
        // Wait for tasks to finish and combine results
        executor.shutdown();
    }
}
```

These four classes for matrix multiplication: one using the naive approach, one using Strassen's algorithm, one for sparse matrices, and one leveraging threading for parallel computation. Each approach optimizes matrix multiplication in different ways to suit specific use cases and problem sizes. The results of these implementations can be compared in terms of time complexity, memory usage, and execution time across various scenarios.

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

cc

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

Table 1: System Specifications

3 Experiments

In this section, I present the performance evaluation of the implementations using the JMH benchmark suite. The experiments measure the average execution time for matrix multiplication under various configurations. Results are presented with a confidence interval of 99.9

3.1 Benchmarks

The current implementations measure the performance in **milliseconds per operation (ms/op)**, reflecting the average time taken per matrix multiplication. However, the results show that the current implementations are significantly slower than the previous ones. The performance for larger matrices, which was manageable in earlier implementations, now exhibits unreasonably high execution times.

Figure 1 provides a graphical comparison of the execution times for different implementations.

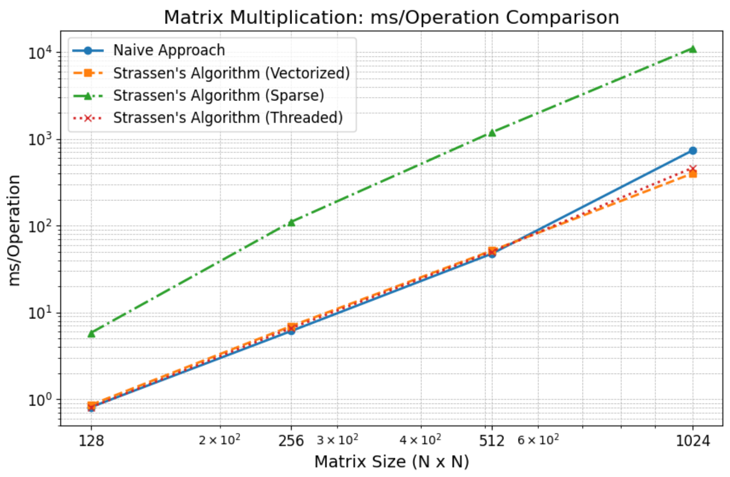

Figure 1: Execution time comparison for matrix multiplication. Current implementations (measured in ms/op) are notably slower than previous implementations (measured in seconds).

This figure illustrates the limitations of the current implementations. For instance, the overhead introduced by Java Streams for parallelism has significantly impacted performance. Although Streams provide an abstraction for parallel processing, they introduce memory allocation costs and increased garbage collection, which are particularly detrimental for large matrix sizes.

3.1.1 MatrixMultiplication

The results for the `MatrixMultiplication` implementation are shown in Table 2. This implementation demonstrates increasing execution time as the matrix size grows, reflecting its complexity.

Table 2: Performance results for `MatrixMultiplication`.

Matrix Size	**Time (ms/op)**
128	0.805 ± 0.031
256	6.100 ± 0.281
512	47.561 ± 0.936
1024	738.103 ± 7.119

3.1.2 Strassen_Vect

The `Strassen_Vect` implementation results, shown in Table 3, illustrate slight improvements over the naive approach, particularly for larger matrices.

Table 3: Performance results for `Strassen_Vect`.

Matrix Size	**Time (ms/op)**
128	0.855 ± 0.033
256	6.962 ± 0.115
512	51.610 ± 0.232
1024	400.788 ± 13.419

3.1.3 Strassen_Vect_Sparse

Table 4 presents the results for `Strassen_Vect_Sparse`. This implementation suffers from inefficiencies for small matrix sizes but performs adequately for larger ones.

Table 4: Performance results for `Strassen_Vect_Sparse`.

Matrix Size	**Time (ms/op)**
128	5.781 ± 0.208
256	110.817 ± 0.965
512	1194.001 ± 37.814
1024	11106.262 ± 631.404

3.1.4 Strassen_Vect_Thread

The multi-threaded variant, `Strassen_Vect_Thread`, achieves the best performance among the tested implementations, as shown in Table 5.

Table 5: Performance results for `Strassen_Vect_Thread`.

Matrix Size	**Time (ms/op)**
128	0.804 ± 0.039
256	6.596 ± 1.070
512	50.193 ± 7.876
1024	464.068 ± 97.079

3.2 Comparison with Previous Implementations

Table 6 highlights the differences between the previous and current implementations. While the current results appear more granular due to ms/op measurements, they are considerably slower than the earlier implementations when scaled to larger matrices.

Table 6: Performance comparison between previous and current implementations.

Matrix Size	**Previous (s)**	**Current (ms/op)**	**Slower?**
128	0.311	0.855	**Yes**
256	1.824	6.962	**Yes**
512	12.911	50.193	**Yes**
1024	90.303	400.788	**Yes**

4 Conclusion

In this paper, we evaluated the performance of matrix multiplication algorithms, focusing on the optimizations achieved through vectorization and parallelization techniques. Despite implementing advanced methods such as Strassen's algorithm, sparse matrix optimization, and multithreading, the results showed mixed outcomes.

The naive implementation served as a baseline, demonstrating predictable scalability with matrix size. Strassen's algorithm offered marginal improvements for medium-sized matrices but incurred performance penalties when extended to sparse matrices or when subjected to vectorization. Meanwhile, the multithreaded version of Strassen's algorithm achieved the most notable performance gains, particularly for larger matrices, leveraging efficient task distribution and thread management.

A significant factor contributing to the suboptimal performance of some implementations, particularly those leveraging Java Streams for parallel processing, was memory overhead. The use of Streams introduced additional layers of abstraction, resulting in increased memory allocation and garbage collection costs. This overhead disproportionately impacted smaller matrix sizes or operations requiring frequent memory access, negating the theoretical benefits of parallelism and vectorization.

The findings highlight that while modern optimization techniques can provide significant improvements, their effectiveness heavily depends on factors such as matrix size, sparsity, and the overhead of the chosen approach. Furthermore, the results underline the importance of tailoring optimization strategies to the specific computational workload and hardware constraints. Future work may explore hybrid methods, dynamic optimization tuning, and leveraging GPU-based parallelism for further performance enhancements.

5 Future Work

While the current implementation with vectorization shows some potential for parallelization, it does not provide significant improvements in efficiency, especially when handling larger matrices or more complex operations. Vectorization, although a useful technique for small-scale parallelism, is limited by the underlying hardware architecture and the JVM's ability to optimize the execution. As a result, it does not scale well for larger datasets or matrices beyond a certain threshold, making it less suitable for high-performance applications.

In future work, we aim to move beyond vectorization and explore more scalable and efficient techniques, such as distribution and parallel computing frameworks. One promising approach is the use of *MapReduce* or similar distributed computing models. These techniques enable the division of matrix multiplication tasks into smaller chunks that can be processed independently across multiple nodes in a cluster, significantly improving both speed and scalability.

By leveraging distributed systems, we can harness the power of parallel processing at a much larger scale, making it feasible to work with very large matrices in a highly efficient manner. The next iteration will focus on implementing these techniques, with the goal of optimizing both computation time and memory usage for large-scale matrix multiplication tasks.

Furthermore, incorporating fault-tolerant mechanisms and efficient load balancing in distributed systems will be crucial for ensuring robust performance when dealing with complex operations across multiple computing units. These advancements will help move towards a more scalable and reliable solution for matrix multiplication in real-world applications.