# Optimized Matrix Multiplication and Sparse Matrices

Luis Perera Pérez
Bachelor's Degree in Data Science and Engineering

Academic year 2024/25

## Abstract

This paper investigates the optimization of matrix multiplication using algorithmic techniques such as Strassen's algorithm and block matrix multiplication, in addition to the use of sparse matrices for improving computational efficiency. Experiments were conducted on matrices of varying sizes (10x10, 100x100, 500x500, and 1000x1000), comparing basic matrix multiplication with optimized versions in terms of execution time and memory usage. Key results indicate significant performance improvements in optimized approaches, especially with sparse matrices where sparsity reduced computational load substantially. These findings emphasize the importance of algorithm selection based on matrix density and problem size.

## 1 Introduction

Matrix multiplication is a fundamental operation in various fields, including computer graphics, machine learning, and scientific computing. Optimizing this operation is critical due to its computational intensity, particularly for large matrices. Several optimization techniques, such as Strassen's algorithm and blocked matrix multiplication, aim to reduce the computational load and improve efficiency. Sparse matrices, which have a high percentage of zero elements, offer additional optimization potential by focusing calculations only on non-zero elements.

This paper examines the performance of different optimized matrix multiplication approaches, analyzing how these methods perform under varying matrix sizes and sparsity levels. We aim to provide insights into the computational benefits of each approach, highlighting scenarios in which specific techniques are most advantageous.

## 2 Problem Statement

Efficiently multiplying large matrices poses a significant computational challenge due to the high complexity of traditional matrix multiplication algorithms. As matrices scale, both time and memory requirements increase dramatically, leading to bottlenecks in performance. Furthermore, in sparse matrices, where most elements are zero, applying conventional algorithms results in redundant computations. The need for efficient matrix multiplication methods becomes crucial, especially in applications with limited computational resources.

## 3 Methodology

To address this challenge, we implemented and benchmarked three matrix multiplication algorithms: the basic (naive) approach, blocked matrix multiplication, and Strassen's algorithm. Additionally, a sparse matrix multiplication approach was implemented to evaluate the impact of sparsity on performance.

## 4 Experiments

Experiments were conducted on matrices of sizes $10 \times 10$, $100 \times 100$, $500 \times 500$, and $1000 \times 1000$ using Java. A brief explanation of each algorithm is detailed below.

### 4.1 Matrix Multiplication Algorithms

The implementations of matrix multiplication methods are presented here:

#### 4.1.1 Basic Matrix Multiplication

A simple implementation of the basic matrix multiplication algorithm in Java is provided below:

```java
package com.example;

import java.util.Random;

public class MatrixMultiplier {

    public static double[][] matrixMultiply(int n) {
        double[][] a = new double[n][n];
        double[][] b = new double[n][n];
        double[][] c = new double[n][n];

        Random random = new Random();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                a[i][j] = random.nextDouble();
                b[i][j] = random.nextDouble();
                c[i][j] = 0;
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
        return c;
    }
}
```

### 4.1.2 Blocked Matrix Multiplication

A simple implementation of the blocked matrix multiplication algorithm in Java is provided below:

```java
package org.example;

public class BlockedMatrixMultiplication {
    public static double[][] multiply(double[][] A, double[][] B,
        int blockSize) {
        int n = A.length;
        double[][] C = new double[n][n];

        for (int ii = 0; ii < n; ii += blockSize) {
            for (int jj = 0; jj < n; jj += blockSize) {
                for (int kk = 0; kk < n; kk += blockSize) {
                    for (int i = ii; i < Math.min(ii + blockSize, n
                        ); i++) {
                        for (int j = jj; j < Math.min(jj +
                            blockSize, n); j++) {
                            double sum = 0;
                            for (int k = kk; k < Math.min(kk +
                                blockSize, n); k++) {
```

```java
                                sum += A[i][k] * B[k][j];
                            }
                            C[i][j] += sum;
                        }
                    }
                }
            }
        }

        return C;
    }
}
```

### 4.1.3 Strassen Matrix Multiplication

A simple implementation of the Strassen algorithm for matrix multiplication in Java is shown below:

```java
package org.example;

public class StrassenMatrixMultiplication {
    public static double[][] multiply(double[][] A, double[][] B) {
        int n = A.length;
        if (n == 1) {
            double[][] C = {{A[0][0] * B[0][0]}};
            return C;
        }

        int newSize = n / 2;
        double[][] A11 = new double[newSize][newSize];
        double[][] A12 = new double[newSize][newSize];
        double[][] A21 = new double[newSize][newSize];
        double[][] A22 = new double[newSize][newSize];
        double[][] B11 = new double[newSize][newSize];
        double[][] B12 = new double[newSize][newSize];
        double[][] B21 = new double[newSize][newSize];
        double[][] B22 = new double[newSize][newSize];

        split(A, A11, 0, 0);
        split(A, A12, 0, newSize);
        split(A, A21, newSize, 0);
        split(A, A22, newSize, newSize);
        split(B, B11, 0, 0);
        split(B, B12, 0, newSize);
        split(B, B21, newSize, 0);
        split(B, B22, newSize, newSize);

        double[][] M1 = multiply(add(A11, A22), add(B11, B22));
        double[][] M2 = multiply(add(A21, A22), B11);
        double[][] M3 = multiply(A11, subtract(B12, B22));
        double[][] M4 = multiply(A22, subtract(B21, B11));
        double[][] M5 = multiply(add(A11, A12), B22);
        double[][] M6 = multiply(subtract(A21, A11), add(B11, B12))
            ;
        double[][] M7 = multiply(subtract(A12, A22), add(B21, B22))
            ;

        double[][] C11 = add(subtract(add(M1, M4), M5), M7);
        double[][] C12 = add(M3, M5);
        double[][] C21 = add(M2, M4);
        double[][] C22 = add(subtract(add(M1, M3), M2), M6);

        double[][] C = new double[n][n];
        join(C11, C, 0, 0);
        join(C12, C, 0, newSize);
        join(C21, C, newSize, 0);
        join(C22, C, newSize, newSize);

        return C;
    }

    private static void split(double[][] P, double[][] C, int iB,
        int jB) {
        for (int i = 0, i2 = iB; i < C.length; i++, i2++)
            for (int j = 0, j2 = jB; j < C.length; j++, j2++)
                C[i][j] = P[i2][j2];
    }

    private static void join(double[][] C, double[][] P, int iB,
        int jB) {
        for (int i = 0, i2 = iB; i < C.length; i++, i2++)
            for (int j = 0, j2 = jB; j < C.length; j++, j2++)
                P[i2][j2] = C[i][j];
    }

    private static double[][] add(double[][] A, double[][] B) {
        int n = A.length;
        double[][] C = new double[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                C[i][j] = A[i][j] + B[i][j];
        return C;
    }

    private static double[][] subtract(double[][] A, double[][] B)
        {
        int n = A.length;
        double[][] C = new double[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                C[i][j] = A[i][j] - B[i][j];
        return C;
    }
}
```

### 4.1.4 Sparse Matrix

A simple implementation of a sparse matrix class in Java is as follows:

```java
package org.example;

import java.util.ArrayList;
import java.util.List;

public class SparseMatrix {
    private final int rows;
    private final int cols;
    private final List<List<Double>> values;

    // Constructor
    public SparseMatrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.values = new ArrayList<>(rows);
        for (int i = 0; i < rows; i++) {
            values.add(new ArrayList<>());
        }
    }

    // Method to add a value in the matrix
    public void addValue(int row, int col, double value) {
        // Check that the position is valid
        if (row < 0 || row >= rows || col < 0 || col >= cols) {
            throw new IndexOutOfBoundsException("Index␣out␣of␣bounds");
        }

        // Store the value at the specific position
        if (col >= values.get(row).size()) {
            for (int i = values.get(row).size(); i <= col; i++) {
                values.get(row).add(0.0); // Add zeros until the required index
            }
        }
        values.get(row).set(col, value); // Assign the value
    }

    // Method to get a value in the matrix
    public double getValue(int row, int col) {
        if (row < 0 || row >= rows || col < 0 || col >= cols) {
            throw new IndexOutOfBoundsException("Index␣out␣of␣bounds");
        }
        return col < values.get(row).size() ? values.get(row).get(col) : 0.0;
    }

    // Method to multiply matrices (only if needed)
    public SparseMatrix multiply(SparseMatrix other) {
        if (this.cols != other.rows) {
            throw new IllegalArgumentException("Incompatible␣dimensions␣for␣multiplication");
        }
        SparseMatrix result = new SparseMatrix(this.rows, other.cols);
        for (int i = 0; i < this.rows; i++) {
            for (int j = 0; j < other.cols; j++) {
                double sum = 0;
                for (int k = 0; k < this.cols; k++) {
                    sum += this.getValue(i, k) * other.getValue(k,
                        j);
                }
                if (sum != 0) {
                    result.addValue(i, j, sum);
                }
            }
        }
        return result;
    }
}
```

### 4.2 Benchmark Code

Aquí se presentan las implementaciones de los benchmarks utilizados para medir el rendimiento de los algoritmos de multiplicación de matrices:

#### 4.2.1 Benchmark for Basic Algorithm

```
package com.example;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
public class MyBenchmark {

    @Benchmark
    public double[][] testMatrixMultiply() {
        int n = 10;
        return MatrixMultiplier.matrixMultiply(n);
    }
}
```

#### 4.2.2 Benchmark for Blocked Matrix

```
package org.example;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
```

```java
@State(Scope.Thread)
public class BlockedMatrixBenchmark {

    private double[][] matrixA;
    private double[][] matrixB;

    @Setup(Level.Trial)
    public void setUp() {
        int n = 1000;  // Size of the matrices
        matrixA = generateRandomMatrix(n, n);
        matrixB = generateRandomMatrix(n, n);
    }

    @Benchmark
    public double[][] testBlockedMatrixMultiply() {
        int blockSize = 50; // Adjust block size for optimization
        return BlockedMatrixMultiplication.multiply(matrixA,
            matrixB, blockSize);
    }

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

### 4.2.3 Benchmark for Sparse Matrix

```java
package org.example;

import org.openjdk.jmh.annotations.*;

import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
public class SparseMatrixBenchmark {

    private SparseMatrix sparseMatrix;

    @Setup(Level.Trial)
    public void setUp() {
        int size = 1000; // Size of the matrix
        sparseMatrix = new SparseMatrix(size, size);
        fillSparseMatrix();
    }

    // Method to fill the sparse matrix
    private void fillSparseMatrix() {
        for (int i = 0; i < 1000; i++) {
            for (int j = 0; j < 1000; j++) {
                // Add sparse values
                if (Math.random() < 0.01) { // 1% probability to
                    add a value
                    sparseMatrix.addValue(i, j, Math.random());
                }
            }
        }
    }

    @Benchmark
    public void testSparseMatrixMultiply() {
        SparseMatrix otherMatrix = new SparseMatrix(1000, 1000); //
             Create another matrix
        fillSparseMatrix(otherMatrix); // Fill the other matrix
        sparseMatrix.multiply(otherMatrix);
    }

    private void fillSparseMatrix(SparseMatrix matrix) {
        for (int i = 0; i < 1000; i++) {
            for (int j = 0; j < 1000; j++) {
                // Add sparse values
                if (Math.random() < 0.01) { // 1% probability to
                    add a value
                    matrix.addValue(i, j, Math.random());
                }
            }
        }
    }
}
```

### 4.2.4 Benchmark for Strassen Matrix

```java
package org.example;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Thread)
public class StrassenMatrixBenchmark {

    private double[][] matrixA;
    private double[][] matrixB;

    @Setup(Level.Trial)
    public void setUp() {
        int n = 1000;  // Size of the matrices
        matrixA = generateRandomMatrix(n, n);
        matrixB = generateRandomMatrix(n, n);
    }

    @Benchmark
    public double[][] testStrassenMatrixMultiply() {
        return StrassenMatrixMultiplication.multiply(matrixA,
            matrixB);
    }

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

## 4.3 How to Run the Code from GitHub

To replicate these experiments, the code for matrix multiplication in Java is available in the following GitHub repository: `https://github.com/LuisPereraPerez/BigData`

Follow these steps to download and execute the code:

- **Clone the repository:** Clone the repository by running the following command:

  ```
  git clone https://github.com/LuisPereraPerez/BigData
  ```

- **Requirements:**
  - For Java, ensure JDK 8 or higher is installed.
  - Install the JMH (Java Microbenchmark Harness) plugin in IntelliJ.

- **Compilation:** Run the following command to compile the project and its dependencies:

  ```
  mvn clean install
  ```

- **Execution:** Once the compilation is complete, run the program with the following command:

  ```
  java -jar target/benchmarks-1.0.jar -prof gc
  ```

- **Varying Matrix Size:** Change the variable $n$ in the code to test different matrix sizes (10, 100, 500, 1000).

## 5 Results

This section presents the execution time, memory usage, and garbage collection (GC) counts for the different matrix multiplication algorithms implemented. Each metric is essential for evaluating performance and is described in detail below:

- **Execution Time (ms/op):** Represents the average time, in milliseconds per operation, that each algorithm takes to complete the matrix multiplication. A lower execution time indicates faster performance.
- **Memory Usage (MB/sec):** Refers to the rate at which memory is allocated during execution, measured in megabytes per second. Higher memory usage rates can signal increased memory demands, which may affect performance if the usage becomes excessive.
- **GC Count:** Denotes the number of garbage collection events triggered during execution. A higher GC count can indicate frequent memory allocation and deallocation, which can introduce performance overhead due to the resources required for memory management.

### 5.1 Execution Time and Memory Usage for $10 \times 10$ Matrices

Table 1: Performance Results for $10 \times 10$ Matrix Multiplication Approaches

| Algorithm | Execution Time (ms/op) | Memory Usage (MB/sec) | GC Count |
|---|---|---|---|
| Basic Algorithm | 0.008 | 366.498 | 322 |
| Blocked (Block Size: 1) | 0.012 | 80.422 | 286 |
| Strassen | 0.169 | 589.604 | 517 |
| Sparse (Sparsity: 99%) | 0.005 | 169.975 | 149 |
| Sparse (Sparsity: 95%) | 0.015 | 148.470 | 473 |
| Sparse (Sparsity: 75%) | 0.011 | 584.633 | 512 |

For $10 \times 10$ matrices, the Basic Algorithm achieves low execution time (0.008 ms/op) and high memory usage (366.498 MB/sec), reflecting efficient handling of minimal data. The Blocked algorithm with a block size of 1 is similarly fast (0.012 ms/op) and shows reduced memory usage (80.422 MB/sec), though it incurs a slightly higher GC count. Strassen's algorithm, designed for larger matrices, shows relatively high memory usage (589.604 MB/sec) and GC count (517), indicating inefficiencies at this scale. Sparse matrix algorithms are efficient for small, sparse matrices, with the 99% sparse version achieving the fastest execution (0.005 ms/op) and moderate memory usage (169.975 MB/sec), though GC counts remain variable across sparsity levels.

## 5.2 Execution Time and Memory Usage for $100 \times 100$ Matrices

Table 2: Performance Results for $100 \times 100$ Matrix Multiplication Approaches

| Algorithm | Execution Time (ms/op) | Memory Usage (MB/sec) | GC Count |
|---|---|---|---|
| Basic Algorithm | 1.861 | 120.664 | 105 |
| Blocked (Block Size: 10) | 1.984 | 40.941 | 146 |
| Strassen | 80.596 | 538.422 | 475 |
| Sparse (Sparsity: 99%) | 3.288 | 74.910 | 66 |
| Sparse (Sparsity: 95%) | 4.860 | 140.546 | 123 |
| Sparse (Sparsity: 75%) | 5.358 | 178.472 | 159 |

For $100 \times 100$ matrices, the Basic Algorithm continues to perform efficiently in execution time (1.861 ms/op) and memory usage (120.664 MB/sec), with minimal GC overhead (105). The Blocked algorithm, using a block size of 10, performs similarly, with slight increases in execution time (1.984 ms/op) and a lower memory allocation rate (40.941 MB/sec). Strassen's algorithm, optimized for larger matrices, shows high execution time (80.596 ms/op) and memory usage (538.422 MB/sec), reflecting significant intermediate data handling. Sparse algorithms show a trade-off between execution time and sparsity; the 99% sparse configuration achieves a low GC count (66) and moderate memory usage, while less sparse configurations see increased execution time and GC activity.

## 5.3 Execution Time and Memory Usage for $500 \times 500$ Matrices

For $500 \times 500$ matrices, the Basic Algorithm's execution time increases significantly (278.024 ms/op), but memory usage remains low (19.763 MB/sec) with minimal GC events. The Blocked algorithm (block size 50) further reduces execution time (161.1 ms/op) and memory usage (11.3 MB/sec) with a low GC count (10), demonstrating its efficiency at larger scales. Strassen's algorithm performs poorly, with very high execution time (2972.8 ms/op), memory usage (788.7 MB/sec), and GC count (830), indicating inefficiency in handling large

Table 3: Performance Results for $500 \times 500$ Matrix Multiplication Approaches

| Algorithm | Execution Time (ms/op) | Memory Usage (MB/sec) | GC Count |
|---|---|---|---|
| Basic Algorithm | 278.024 | 19.763 | 16 |
| Blocked (Block Size: 50) | 161.1 | 11.3 | 10 |
| Strassen | 2972.8 | 788.7 | 830 |
| Sparse (Sparsity: 99%) | 1016.3 | 15.4 | 11 |
| Sparse (Sparsity: 95%) | 1349.5 | 15.6 | 15 |
| Sparse (Sparsity: 75%) | 1599.3 | 15.0 | 15 |

intermediate data structures. Sparse algorithms also experience higher execution times as sparsity decreases, but they remain more efficient than Strassen's, especially at higher sparsity.

## 5.4 Execution Time and Memory Usage for $1000 \times 1000$ Matrices

Table 4: Performance Results for $1000 \times 1000$ Matrix Multiplication Approaches

| Algorithm | Execution Time (ms/op) | Memory Usage (MB/sec) | GC Count |
|---|---|---|---|
| Basic Algorithm | 12976.7 | 1.7 | 10 |
| Blocked (Block Size: 100) | 1339.6 | 5.5 | 5 |
| Strassen | 21494.2 | 781.5 | 730 |
| Sparse (Sparsity: 99%) | 28823.5 | 2.5 | 19 |
| Sparse (Sparsity: 95%) | 30179.1 | 3.2 | 11 |
| Sparse (Sparsity: 75%) | 24909.9 | 4.2 | 6 |

At the $1000 \times 1000$ scale, the Basic Algorithm has a sharp increase in execution time (12976.7 ms/op) and minimal memory usage (1.7 MB/sec), indicating that it is not efficient for very large matrices. The Blocked algorithm (block size 100) achieves a lower execution time (1339.6 ms/op) with controlled memory demands (5.5 MB/sec) and minimal GC activity, showing its scalability. Strassen's algorithm, though designed for large matrices, performs poorly with an execution time of 21494.2 ms/op and memory usage of 781.5 MB/sec, with a high GC count (730). Sparse algorithms show the best results at higher sparsity levels, with the 99% sparse version achieving manageable memory usage and GC events, though execution times remain high due to density.

## 5.5 General Analysis

Across all matrix sizes, the Basic Algorithm is efficient for smaller matrices but scales poorly in memory usage and execution time as matrix size increases. The Blocked algorithm, especially with well-chosen block sizes, achieves balanced performance and proves efficient for larger matrices due to lower memory and GC overhead. Strassen's algorithm, while optimized for specific large matrices, incurs high memory and processing costs due to intermediate data structures, especially at larger sizes. Sparse matrix algorithms perform well at high sparsity levels but see increased execution times and memory demands as density grows.

Overall, the choice of algorithm should consider both matrix density and size:

- **Basic Algorithm:** Suitable for small matrices where computational and memory demands are minimal.
- **Blocked Algorithm:** Performs most efficiently at larger scales due to balanced memory usage and manageable garbage collection counts, especially with an appropriate block size.
- **Strassen:** While optimized for some large matrix operations, it tends to be inefficient for very large matrices due to its high memory requirements and frequent garbage collection events.
- **Sparse Algorithms:** Effective for large, sparse matrices, particularly at higher sparsity levels. Higher sparsity reduces memory usage and garbage collection overhead, making these algorithms well-suited for applications with sparse data.

# 6 Conclusion

This study evaluated various matrix multiplication algorithms, highlighting their performance in terms of execution time, memory usage, and garbage collection (GC) counts across different matrix sizes and sparsity levels. The results demonstrated that:

1. **Basic Algorithm**: This algorithm is efficient for smaller matrices, providing low execution times and acceptable memory usage. However, it scales poorly with larger matrices, leading to increased execution times and higher memory demands.

2. **Blocked Algorithm**: The Blocked algorithm consistently performed well, particularly with appropriately chosen block sizes. It effectively balances memory usage and garbage collection overhead, making it the most suitable option for larger matrices.

3. **Strassen's Algorithm**: Although designed for efficient multiplication of large matrices, Strassen's algorithm exhibited significant inefficiencies at larger scales due to high memory requirements and frequent garbage collection events.

4. **Sparse Algorithms**: These algorithms performed optimally with higher sparsity levels, effectively managing memory usage and execution times. They are particularly advantageous for applications dealing with sparse data structures.

Overall, the selection of an algorithm should consider both the size and density of the matrices involved. Future work could explore additional optimization techniques, such as hardware-specific parallelism, and the tuning of algorithms for various sparsity patterns. Furthermore, extending these experiments to other matrix sizes and real-world data distributions could enhance our understanding of the optimal conditions for each algorithm.

## 7 Future Work

Future research can explore additional optimization techniques, such as hardware-specific parallelism and tuning algorithms for different sparsity patterns. Furthermore, extending these experiments to other matrix sizes and real-world data distributions could enhance our understanding of the optimal conditions for each algorithm.