Optimized Matrix Multiplication and Sparse Matrices

Luis Perera Pérez
Grado en Ciencia e Ingeniería de Datos

Academic year 2024/25

Abstract

This study investigates the performance of different matrix multiplication algorithms, specifically focusing on both dense and sparse matrices. As matrix size increases, traditional algorithms face significant challenges in terms of computational time and memory usage. In this paper, we evaluate and compare three matrix multiplication algorithms: the basic algorithm, blocked matrix multiplication, and loop unrolling. The performance is measured in terms of execution time, memory usage, and garbage collection events, with tests conducted on matrices of various sizes and sparsity levels. The results indicate that, for large matrices, sparse matrices can be processed more efficiently with specialized algorithms like the blocked matrix multiplication and loop unrolling, demonstrating improved performance as sparsity increases. The experiments also highlight the trade-offs in memory usage and garbage collection overhead in different scenarios.

1 Introduction

Matrix multiplication is a fundamental operation in numerous fields, including numerical analysis, computer graphics, machine learning, and scientific computing. However, as matrix sizes grow, traditional matrix multiplication algorithms face challenges in terms of computational complexity and memory consumption. The standard approach, which has a time complexity of $O(n^3)$ for $n \times n$ matrices, becomes inefficient as matrix size increases.

In real-world applications, matrices are often sparse, meaning they contain many zero elements. Sparse matrices offer the potential for optimization since most of their elements do not contribute to the final result. Traditional algorithms, however, do not take advantage of the sparsity, resulting in redundant computations and wasted memory resources.

This paper aims to explore more efficient algorithms for matrix multiplication, particularly for large and sparse matrices. Specifically, we investigate three approaches: the basic algorithm, blocked matrix multiplication, and loop unrolling. By conducting experiments on matrices of varying sizes and sparsity levels, we compare the performance of these algorithms in terms of execution time, memory usage, and garbage collection events. Our goal is to identify the most effective methods for optimizing matrix multiplication in both dense and sparse matrix scenarios.

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

This study evaluates three matrix multiplication algorithms: basic multiplication, blocked multiplication, and loop unrolling. We tested these algorithms using matrices ranging from 10×10 to 1000×1000 with sparsity levels of 75%, 95%, and 99%.

Performance was measured by:

- **Execution Time:** Time taken for matrix multiplication.
- **Memory Usage:** Memory allocated during execution.
- **GC Count:** Garbage collection events during execution.

Experiments were conducted in Java with a block size of 10% of the matrix dimension for the blocked algorithm. The sparsity was controlled by varying the percentage of non-zero elements.

4 Experiments

Experiments were conducted on matrices of sizes 10×10, 100×100, 500×500, and 1000×1000 using Java. For each matrix size, both dense and sparse matrices were tested, with sparsity levels of 99%, 95%, and 75%, to evaluate the performance of the different algorithms under various conditions. 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;

public class MatrixMultiplier {

    // Matrix multiplication considering sparse matrices
    public static double[][] matrixMultiply(double[][] A, double
        [][] B, boolean sparse) {
        int n = A.length;
        double[][] result = new double[n][n];

        // Matrix multiplication with sparsity consideration
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                double sum = 0;
                for (int k = 0; k < n; k++) {
                    // For sparse matrices, skip multiplication if
                        A[i][k] or B[k][j] is zero
                    if (sparse) {
                        if (A[i][k] != 0 && B[k][j] != 0) {
                            sum += A[i][k] * B[k][j];
                        }
                    } else {
                        // For dense matrices, just multiply
                            normally
                        sum += A[i][k] * B[k][j];
                    }
                }
                result[i][j] = sum;
            }
        }

        return result;
    }
}
```

4.1.2 Blocked Matrix Multiplication

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

```java
package com.example;

public class BlockedMatrixMultiplication {

    // Blocked matrix multiplication method
    public static double[][] multiply(double[][] A, double[][] B,
        boolean sparse) {
        int n = A.length;

        // Calculate block size as 10% of n
        int blockSize = Math.max(1, n / 10);

        // Result matrix C
        double[][] C = new double[n][n];

        // Blocked matrix multiplication
        for (int ii = 0; ii < n; ii += blockSize) {
            for (int jj = 0; jj < n; jj += blockSize) {
                for (int kk = 0; kk < n; kk += blockSize) {
                    // Iterate over the submatrices (blocks)
                    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;

                            // If the matrix is sparse, do not
                                multiply zero elements
                            if (sparse) {
                                for (int k = kk; k < Math.min(kk +
                                    blockSize, n); k++) {
                                    if (A[i][k] != 0 && B[k][j] !=
                                        0) {
                                        sum += A[i][k] * B[k][j];
                                    }
                                }
                            } else {
                                // For dense matrices, multiply all
                                    elements
                                for (int k = kk; k < Math.min(kk +
                                    blockSize, n); k++) {
                                    sum += A[i][k] * B[k][j];
                                }
                            }

                            // If the sum is not zero, add it to the result matrix
                            if (sum != 0) {
                                C[i][j] += sum;
                            }
                        }
                    }
                }
            }
        }

        return C;
    }
}
```

4.1.3 Multiplication With Loop Unrolling

A simple implementation of the Multiplication With Loop Unrolling algorithm for matrix multiplication in Java is shown below:

```java
package com.example;

public class MultiplyWithLoopUnrolling {

    // Matrix multiplication with loop unrolling and sparsity consideration
    public static double[][] multiply(double[][] A, double[][] B, boolean sparse) {
        int n = A.length;
        double[][] result = new double[n][n];

        // Matrix multiplication with sparsity and loop unrolling
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                double sum = 0;
                int k = 0;

                // Loop unrolling, processing 4 elements at a time
                for (; k <= n - 4; k += 4) {
                    if (sparse) {
                        // For sparse matrices, only multiply if neither element is zero
                        if (A[i][k] != 0 && B[k][j] != 0) sum += A[i][k] * B[k][j];
                        if (A[i][k + 1] != 0 && B[k + 1][j] != 0) sum += A[i][k + 1] * B[k + 1][j];
                        if (A[i][k + 2] != 0 && B[k + 2][j] != 0) sum += A[i][k + 2] * B[k + 2][j];
                        if (A[i][k + 3] != 0 && B[k + 3][j] != 0) sum += A[i][k + 3] * B[k + 3][j];
                    } else {
                        // For dense matrices, multiply directly
                        sum += A[i][k] * B[k][j];
                        sum += A[i][k + 1] * B[k + 1][j];
                        sum += A[i][k + 2] * B[k + 2][j];
                        sum += A[i][k + 3] * B[k + 3][j];
                    }
                }

                // Process remaining elements if n is not a
                    multiple of 4
                for (; k < n; k++) {
                    if (sparse) {
                        if (A[i][k] != 0 && B[k][j] != 0) {
                            sum += A[i][k] * B[k][j];
                        }
                    } else {
                        sum += A[i][k] * B[k][j];
                    }
                }

                result[i][j] = sum;
            }
        }

        return result;
    }
}
```

4.2 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 and Sparsity:** Change the matrix size by adjusting the variable n in the code to test different matrix sizes (10, 100, 500, 1000). Additionally, you can choose to use matrices with varying levels of sparsity or no sparsity at all, allowing for the evaluation of algorithm performance on both dense and sparse matrices.

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.

Each algorithm tested was optimized by setting the block size at 10% of the matrix dimensions, a configuration chosen to balance memory usage and computational efficiency in the Blocked Algorithm. This setting helps to optimize cache usage, minimizing memory access times by ensuring that each block fits well within the processor cache for improved processing speed.

5.1 Sparse Matrices

Sparse matrices are matrices in which a significant proportion of elements are zero. Handling sparse matrices efficiently is essential in applications where most of the matrix data is unused or unnecessary for calculations. By focusing only on non-zero values, matrix multiplication algorithms can reduce both computation time and memory usage, especially when matrices are large. This study examines three levels of sparsity: 99% Sparse, 95% and 75%.

These varying sparsity levels provide insight into how each algorithm adapts to sparse data and highlight the conditions under which they can optimize performance by avoiding unnecessary calculations.

5.2 Execution Time and Memory Usage for 10×10 Matrices

Table 1: Performance Results for 10×10 Matrix Multiplication Approaches

Algorithm	Execution Time (ns/op)	Memory Usage (MB/sec)	GC Count
Basic Algorithm	1535.554 ± 176.828	610.007 ± 49.341	1227
Basic Algorithm (Sparsity: 99%)	1104.571 ± 47.761	837.515 ± 35.197	1629
Basic Algorithm (Sparsity: 95%)	1263.284 ± 76.116	733.968 ± 40.080	1532
Basic Algorithm (Sparsity: 75%)	1327.588 ± 31.704	695.112 ± 15.817	1394
Blocked Matrix Multiply	8102.746 ± 173.817	113.910 ± 2.380	405
Blocked (Sparsity: 99%)	5404.016 ± 94.036	170.736 ± 2.771	607
Blocked (Sparsity: 95%)	6046.948 ± 449.847	153.803 ± 10.436	547
Blocked (Sparsity: 75%)	6432.190 ± 197.845	143.587 ± 4.177	512
Matrix Multiply with Loop Unrolling	1606.226 ± 68.846	575.734 ± 22.785	1157
Loop Unrolling (Sparsity: 99%)	1223.939 ± 77.426	758.230 ± 46.356	1479
Loop Unrolling (Sparsity: 95%)	1369.730 ± 45.220	674.216 ± 21.993	1345
Loop Unrolling (Sparsity: 75%)	1853.695 ± 65.939	498.386 ± 17.172	1216

For 10×10 matrices, the Basic Algorithm achieves relatively low execution time (1,535.554 ns/op) and high memory usage (610.007 MB/sec), indicating efficient processing of small matrices with a modest GC count of 1,227. When sparsity is increased to 99%, the Basic Algorithm becomes faster (1,104.571 ns/op) and uses more memory (837.515 MB/sec), though the GC count increases to 1,629, showing improved efficiency with higher sparsity at some garbage collection cost.

In the Blocked Matrix Multiply approach, the execution time is significantly longer (8,102.746 ns/op) with low memory usage (113.910 MB/sec), suggesting the overhead of managing blocks in small matrices. Sparsity helps somewhat, with the 99% sparse version reducing execution time to 5,404.016 ns/op and increasing memory usage to 170.736 MB/sec, though the GC count remains relatively low at 607, showing limited effectiveness of blocking at this scale.

The Loop Unrolling method improves on the Basic Algorithm's execution time (1,606.226 ns/op) with similar memory usage (575.734 MB/sec) and a slightly lower GC count of 1,157, which suggests optimized performance with less overhead. As sparsity increases to 99%, Loop Unrolling shows reduced execution time (1,223.939 ns/op) and improved memory usage (758.230 MB/sec) with a higher GC count (1,479), making it effective for sparse matrices but with variable garbage collection demands across sparsity levels.

5.3 Execution Time and Memory Usage for 100×100 Matrices

Table 2: Performance Results for 100×100 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)	GC Count
Basic Algorithm	1.118 ± 0.038	66.703 ± 2.166	239
Basic Algorithm (Sparsity: 99%)	0.840 ± 0.157	93.918 ± 16.872	334
Basic Algorithm (Sparsity: 95%)	0.732 ± 0.021	101.826 ± 2.849	365
Basic Algorithm (Sparsity: 75%)	2.467 ± 0.237	30.496 ± 1.974	108
Blocked Matrix Multiply	1.136 ± 0.019	65.595 ± 1.133	235
Blocked (Sparsity: 99%)	0.850 ± 0.053	88.152 ± 5.077	314
Blocked (Sparsity: 95%)	1.104 ± 0.081	68.027 ± 4.841	242
Blocked (Sparsity: 75%)	3.295 ± 0.060	22.606 ± 0.399	81
Matrix Multiply with Loop Unrolling	1.145 ± 0.024	65.032 ± 1.297	231
Loop Unrolling (Sparsity: 99%)	0.903 ± 0.023	82.554 ± 1.948	295
Loop Unrolling (Sparsity: 95%)	1.002 ± 0.038	74.457 ± 2.783	265
Loop Unrolling (Sparsity: 75%)	3.078 ± 0.255	24.457 ± 1.976	87

For 100×100 matrices, the Basic Algorithm provides a moderate execution time (1.118 ms/op) with memory usage at 66.703 MB/sec and a GC count of 239, balancing performance and resource consumption. With increased sparsity, this algorithm benefits notably, with the 99% sparse version achieving a faster execution time (0.840 ms/op) and higher memory usage (93.918 MB/sec), though GC count rises to 334, showing improved efficiency with sparse data despite additional garbage collection overhead.

The Blocked Matrix Multiply method achieves similar performance to the Basic Algorithm (1.136 ms/op, 65.595 MB/sec) with a comparable GC count of 235. For higher sparsity (99%), this approach also improves to an execution time of 0.850 ms/op and a memory usage of 88.152 MB/sec, while the GC count increases to 314, indicating that blocking is effective for sparse matrices at this scale but requires additional garbage collection.

The Matrix Multiply with Loop Unrolling technique performs similarly to the Basic Algorithm, with an execution time of 1.145 ms/op and memory usage at 65.032 MB/sec, accompanied by a slightly lower GC count of 231. As sparsity increases to 99%, Loop Unrolling sees a reduced execution time (0.903 ms/op) and improved memory usage (82.554 MB/sec), though GC count rises to 295, highlighting enhanced performance with sparse matrices but variable garbage collection demands across sparsity levels.

For all algorithms, a sparsity level of 75% increases execution time and reduces memory efficiency, suggesting limited performance benefits for matrices with moderate sparsity.

5.4 Execution Time and Memory Usage for 500×500 Matrices

Table 3: Performance Results for 500 × 500 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)	GC Count
Basic Algorithm	192.981 ± 10.631	9.501 ± 0.445	34
Basic Algorithm (Sparsity: 99%)	71.123 ± 1.042	25.670 ± 0.358	92
Basic Algorithm (Sparsity: 95%)	79.699 ± 2.390	22.936 ± 0.671	84
Basic Algorithm (Sparsity: 75%)	334.756 ± 7.372	5.460 ± 0.117	20
Blocked Matrix Multiply	156.709 ± 11.879	11.755 ± 0.812	42
Blocked (Sparsity: 99%)	99.699 ± 1.604	18.312 ± 0.281	66
Blocked (Sparsity: 95%)	106.311 ± 2.922	17.191 ± 0.444	63
Blocked (Sparsity: 75%)	357.352 ± 16.560	5.126 ± 0.218	17
Matrix Multiply with Loop Unrolling	186.112 ± 4.920	9.817 ± 0.235	35
Loop Unrolling (Sparsity: 99%)	109.153 ± 2.004	16.732 ± 0.293	60
Loop Unrolling (Sparsity: 95%)	116.332 ± 3.195	15.711 ± 0.419	58
Loop Unrolling (Sparsity: 75%)	379.522 ± 15.979	4.828 ± 0.201	18

For 500 × 500 matrices, the Basic Algorithm takes the longest time to execute in the non-sparse scenario, with an execution time of 192.981 ms/op, low memory usage of 9.501 MB/sec, and a GC count of 34. However, as sparsity increases, the Basic Algorithm shows significant performance gains, achieving its best time with 99% sparsity (71.123 ms/op) and an increase in memory usage to 25.670 MB/sec, though the GC count rises to 92, indicating that greater sparsity benefits execution but raises garbage collection demands.

The Blocked Matrix Multiply algorithm performs faster than the Basic Algorithm in non-sparse conditions (156.709 ms/op) and has a moderate memory usage of 11.755 MB/sec with a GC count of 42. With increased sparsity, the blocked approach shows notable performance improvements, achieving a minimum execution time of 99.699 ms/op at 99% sparsity, though with reduced memory efficiency (18.312 MB/sec) and an increase in GC count to 66, highlighting a balance of execution time and memory efficiency in sparse scenarios.

The Matrix Multiply with Loop Unrolling approach demonstrates a similar performance pattern, with a baseline execution time of 186.112 ms/op, memory usage of 9.817 MB/sec, and a GC count of 35. With higher sparsity, this method improves in speed, achieving 109.153 ms/op at 99% sparsity, along with higher memory usage at 16.732 MB/sec and a GC count of 60, illustrating the effectiveness of loop unrolling for sparse matrices.

Across all methods, a sparsity of 75% tends to lead to the highest execution times and lowest memory usage, particularly in the Basic Algorithm (334.756

ms/op) and Loop Unrolling (379.522 ms/op), suggesting that moderate sparsity offers limited performance benefits for larger matrices.

5.5 Execution Time and Memory Usage for 1000×1000 Matrices

Table 4: Performance Results for 1000×1000 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)	GC Count
Basic Algorithm	8720.195 ± 586.890	0.857 ± 0.062	8
Basic Algorithm (Sparsity: 99%)	552.592 ± 11.397	13.209 ± 0.254	52
Basic Algorithm (Sparsity: 95%)	706.175 ± 18.448	10.341 ± 0.254	41
Basic Algorithm (Sparsity: 75%)	3442.562 ± 113.192	2.128 ± 0.075	11
Blocked Matrix Multiply	1777.506 ± 516.248	4.611 ± 1.051	21
Blocked (Sparsity: 99%)	780.770 ± 66.030	9.466 ± 0.829	39
Blocked (Sparsity: 95%)	762.150 ± 30.268	9.597 ± 0.392	39
Blocked (Sparsity: 75%)	2416.217 ± 54.059	3.037 ± 0.070	14
Matrix Multiply with Loop Unrolling	8116.395 ± 494.737	0.920 ± 0.053	10
Loop Unrolling (Sparsity: 99%)	826.597 ± 19.947	8.829 ± 0.216	34
Loop Unrolling (Sparsity: 95%)	964.021 ± 21.031	7.576 ± 0.156	30
Loop Unrolling (Sparsity: 75%)	3844.577 ± 128.152	1.909 ± 0.056	10

At the 1000×1000 scale, the Basic Algorithm performs the slowest among all methods in the non-sparse case, with an execution time of 8720.195 ms/op and minimal memory usage at 0.857 MB/sec, reflecting significant overhead when handling large data without optimizations. Sparsity improves performance substantially for this algorithm, particularly at 99% sparsity, where the execution time drops to 552.592 ms/op with increased memory usage of 13.209 MB/sec, though it incurs a higher GC count of 52, indicating that memory demands grow with greater sparsity.

The Blocked Matrix Multiply approach performs more efficiently in both non-sparse and sparse cases, achieving a baseline execution time of 1777.506 ms/op with moderate memory usage of 4.611 MB/sec and a GC count of 21. With increased sparsity, the performance improves, with the 99% sparse version achieving an execution time of 780.770 ms/op and memory usage at 9.466 MB/sec, though GC count remains moderate at 39, showing that the blocked structure optimizes memory handling effectively in sparse matrices.

For the Matrix Multiply with Loop Unrolling approach, the non-sparse performance is slower, with an execution time of 8116.395 ms/op and low memory usage at 0.920 MB/sec. As sparsity increases, the performance improves, reaching 826.597 ms/op and a memory usage of 8.829 MB/sec at 99% sparsity, with a GC count of 34, indicating that loop unrolling gains a moderate efficiency boost with sparse matrices but continues to face memory limitations.

Overall, 99% sparsity results in the fastest execution times across all algorithms, while lower sparsity, such as 75%, leads to intermediate performance gains compared to the non-sparse cases but is still more costly in execution time and memory usage than higher sparsity levels. The Blocked and Loop Unrolling approaches perform well under sparse conditions, offering a balance between execution speed and memory management for large-scale matrix operations.

5.6 General Analysis

The performance of matrix multiplication algorithms varies significantly based on matrix size and sparsity levels. Overall, the choice of algorithm should consider both matrix density and size:

- **Basic Algorithm:** This algorithm is efficient for smaller matrices due to its straightforward computational approach and low memory demands. It also performs well with high sparsity, which reduces computation time and boosts memory efficiency, although this can lead to a higher garbage collection (GC) count due to increased object allocation.
- **Blocked Algorithm:** The blocked approach is most effective at larger matrix scales. Its structured memory access and block-wise processing result in balanced memory usage and a moderate GC count, particularly with higher sparsity (99% or 95%). It remains efficient by minimizing cache misses and optimizing memory bandwidth usage, making it suitable for large and sparse matrices where an appropriate block size is chosen.
- **Loop Unrolling Algorithm:** This method introduces optimizations through loop unrolling, improving performance for medium-sized matrices. However, as matrix size grows, it becomes less efficient due to its relatively high memory usage and increased GC events. Loop unrolling is beneficial in cases where the matrix has high sparsity, but for very large matrices, its memory demands can outweigh the computational benefits.

6 Conclusion

The results of this study demonstrate the significant impact of matrix sparsity on the performance of matrix multiplication algorithms. For dense matrices, the basic algorithm provides a straightforward approach but becomes inefficient as matrix size increases. On the other hand, the blocked matrix multiplication and loop unrolling algorithms perform better as the sparsity of the matrices increases, showing clear advantages in execution time and memory usage.

In particular, for sparse matrices with 99% sparsity, both blocked matrix multiplication and loop unrolling outperform the basic algorithm by reducing computational overhead and memory demands. However, the performance benefits of these optimized algorithms are less pronounced for matrices with lower sparsity levels, such as 75%, where traditional methods may still offer competitive performance for smaller matrices.

Furthermore, the experiments highlight the importance of memory management in large-scale matrix multiplication tasks. Both the blocked and loop unrolling approaches introduce additional memory overhead, which can be mitigated with careful tuning and optimization of block sizes and loop unrolling factors.

Overall, this study provides insights into the trade-offs between execution time, memory usage, and garbage collection for different matrix multiplication algorithms. These findings can be applied to optimize matrix multiplication in real-world applications, particularly in fields like machine learning and scientific computing, where large sparse matrices are common.

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.