COMPARATIVE PERFORMANCE ANALYSIS OF MATRIX MULTIPLICATION: DENSE AND SPARSE MATRIX APPROACHES

Raquel Almeida Quesada

Big Data, Grado en Ciencia e Ingeniería de Datos

Universidad de Las Palmas de Gran Canaria

<raquel.almeida105@alu.ulpgc.es>

GitHub Repository

November, 2024

ABSTRACT

Matrix multiplication is a fundamental operation in various fields, such as signal processing, machine learning, and computational physics, where optimizing its performance is crucial due to its impact on the execution time of data-intensive systems.

This project focuses on optimizing matrix multiplication, exploring both dense and sparse matrices. Dense matrices, where most elements are non-zero, are efficient for traditional multiplication algorithms, though they present performance limitations for large-scale matrices. In contrast, sparse matrices mainly contain zero elements, enabling optimization by skipping these values, resulting in considerable savings in computation time and memory.

This study implements and compares several optimization techniques: Strassen's algorithm, loop unrolling, and cache optimization, as well as an optimized multiplication approach for sparse matrices with different sparsity levels. The analysis of each approach provides a comprehensive evaluation of the benefits and limitations of each technique, offering insights into the conditions under which each optimization is most effective in enhancing efficiency when processing large data volumes.

1. INTRODUCTION

In standard matrix multiplication, each element of the resulting matrix is calculated as the sum of products between rows and columns of the input matrices. This operation, for matrices of size nxn has a complexity of $O(n^3)$, which becomes costly when matrices are large. To improve its performance, this project explores optimization techniques for dense and sparse matrix multiplication.

Dense matrices contain few or no zero elements, allowing for the use of conventional algorithms such as the basic algorithm or Strassen's algorithm, which reduces complexity to approximately $O(n^{2.81})$. However, for very large matrices, these algorithms can be inefficient in terms of memory and processing.

$$\mathbf{A}_{\text{dense}} = \begin{bmatrix} 4 & 1 & 7 & 3 \\ 2 & 8 & 6 & 5 \\ 9 & 3 & 4 & 2 \\ 7 & 5 & 8 & 6 \end{bmatrix}$$

On the other hand, sparse matrices are mostly composed of zero elements. This enables a reduction in the number of operations by ignoring zeros, yielding potential savings in both time and space. In this context, a sparse matrix can be represented and multiplied using lists of indices of non-zero elements, simplifying the operation to a sum of products only in the relevant positions. Given a certain sparsity percentage (or level of zeros), the computation can be significantly reduced, making optimizations targeted at sparse matrices particularly useful in high-performance applications. This study implements and evaluates various optimization techniques, such as loop unrolling and cache optimization, providing an analysis of their effectiveness based on matrix density and problem size.

$$\mathbf{A}_{\text{sparse}} = \begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 8 \\ 4 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}$$

We were asked to use a specific sparse matrix but, due to the capacity of my computer, I was not able

to use such a large matrix for this experiment.

2. PROBLEM STATEMENT

In this project, various optimization techniques for matrix multiplication will be tested to identify and compare the efficiency of different approaches in enhancing computational performance. By examining both dense and sparse matrices, we aim to highlight the strengths and limitations of each technique in diverse scenarios. Specifically, we will implement advanced methods like Strassen's algorithm, loop unrolling, and cache optimization, along with a specialized approach for sparse matrices that reduces unnecessary operations by ignoring zero elements. Through rigorous benchmarking, we will determine the most effective matrix multiplication method, evaluating each technique's impact on execution time, memory usage, and overall efficiency in data-intensive applications.

3. METHODOLOGY

To achieve the comparison of different optimization techniques for matrix, multiplication, several optimized versions of the basic matrix multiplication algorithm were implemented, and performance tests were conducted to analyze the benefits and limitations of each approach. The main tasks and steps of the methodology are outlined below.

3.1. Implementation of Optimization Techniques

To optimize dense matrix multiplication, three main optimization techniques were implemented: Strassen's algorithm, loop unrolling, and cache optimization.

Sparse matrices contain many zero elements, allowing for optimization by skipping these elements in computation. Several methods were implemented to handle sparse matrices in this project.

Each of these implementations will be explained in the Experiments section.

3.2. Environment Setup

For the execution of the experiments, IntelliJ IDEA has been employed for the implementation in Java. This environment allows programmers to develop their codes in Java language.

3.3. Performance Measurements

Measurements have been carried out in three dimensions:

1. **Execution Time:** for each run, the total time taken to execute the matrix multiplication algorithm has been recorded. This has been done using the `System.nanoTime()` class.
2. **CPU usage:** CPU usage during the execution of the algorithm has been measured using `getSystemCpuLoad` class from `OsBean`.
3. **Memory Usage:** Memory usage has been evaluated using the `Runtime` class before and after the execution of the algorithm.

3.4. Execution of Experiments

The experiments have been conducted on a machine with the following specifications:

- **Processor:** Intel Core i7 (8 cores, 3.2 GHz)
- **RAM:** 16 GB
- **Operating System:** Windows 10

Multiple runs have been performed for each algorithm in each language, varying the size of the matrices (e.g., 100x100, 500x500, 1000x1000) to observe how they behave in terms of performance as the workload increases.

This methodological approach provides a clear framework for measuring and comparing the efficiency of matrix multiplication in different languages, allowing other researchers to reproduce the experiments and validate the obtained results.

4. EXPERIMENTS

To address the problem of optimizing matrix multiplication across in Java, I conducted a series of experiments using different optimization techniques. All results from the experiments were stored in a CSV file for easy manipulation and analysis.

4.1. Implementation of Optimization Techniques

- **Strassen's Algorithm:** This algorithm divides matrices into submatrices and reduces the number of required multiplications, lowering the computational complexity from $O(n^3)$ to approximately $O(n^{2.81})$. he implementation uses recursion to multiply the submatrices, saving time for large matrices. A specific

class, `StrassenMatrixMultiplication`, was created, where the `strassenMultiply` method breaks the matrix into submatrices, computes intermediate multiplications, and combines the results.

- **Loop Unrolling:** This technique speeds up computation by reducing the jump instructions in the inner multiplication loop. Instead of iterating element by element, the loop is divided into blocks that process multiple elements simultaneously. In the `UnrolledMatrixMultiplication` class, the `unrolledMatrixMultiplication` method implements this technique by iterating four elements at a time, accumulating products in a variable to avoid unnecessary memory accesses and reduce the number of jumps.
- **Cache Optimization:** This technique rearranges the loops to improve cache memory access, minimizing cache misses and increasing efficiency. The matrix is divided into smaller blocks to maximize cache use, so that each block can be retained in the cache during processing. In the `CacheOptimizedMatrixMultiplication` class, the `cacheOptimizedMultiply` method divides the matrix into fixed-size blocks and multiplies each block to better utilize cached data.

4.2. Implementation of Sparse Matrices

- **Sparse Matrix Generation:** The `generateSparseMatrix` method in the `SparseMatrixGenerator` class generates a sparse matrix with a specified percentage of zero elements (sparsity level). This sparsity level allows adjustment of the non-zero element percentage, enabling analysis of performance variation with different sparsity levels.
- **Optimized Multiplication for Sparse Matrices:** A multiplication method that operates only on the non-zero positions of the matrix was implemented. In the `SparseMatrixMultiplication` class, the `multiplySparseMatrices` method uses a list of non-zero element indices to compute the product of sparse matrices, avoiding positions that contain zeros. This reduces the number of operations and improves efficiency when sparsity levels are high.

4.3. Step-by-Step Implementation

1. Algorithmic Optimization

- **Strassen's Algorithm** was implemented in a specific method (`strassenMultiply`) to break down the matrix and reduce the number of operations.
- For **Loop Unrolling and Cache Optimization**, the basic algorithm was modified, creating optimized versions of the inner loops to minimize jump time and maximize cache usage, respectively.

2. Sparse Matrix Implementation

- `generateSparseMatrix` was implemented to create matrices with varying sparsity levels, where a certain percentage of elements are zero.
- Sparse matrix multiplication was optimized using `multiplySparseMatrices`, a method that skips zero positions, reducing operations and computation time.

3. Threshold-Based Stopping Algorithm

- To avoid excessive computation times for large matrices, a threshold-based stopping algorithm was introduced. This approach defines an execution time threshold beyond which a given algorithm is deemed non-optimal, thus skipping further calculations for larger matrix sizes with that algorithm.
- In this implementation, the threshold was set at 5 seconds (5000 milliseconds), based on the observation that any execution time significantly above this value results in inefficient use of resources and is not practical for real-time or large-scale applications. This threshold helps identify the maximum matrix size each algorithm can handle efficiently, especially under high sparsity levels where some optimizations may be less effective.
- The threshold-based stopping mechanism is implemented within the `Main` class, where each algorithm's execution time is measured. If the execution time exceeds the threshold for a specific matrix size, the algorithm is stopped, and the program proceeds to the next size or sparsity level as applicable. This approach minimizes wasted computational resources while

ensuring that the performance remains within acceptable limits.

4. Testing and Measurements

- Exhaustive tests were conducted in the `Main` class, running each algorithm version for different matrix sizes and sparsity levels. Performance metrics (time, memory, and CPU) were recorded for each execution and stored in a CSV file, facilitating analysis of the results.

This methodology allows for the evaluation of each optimization technique's performance and effectiveness across different matrix configurations, providing a solid foundation for determining best practices in large and sparse matrix multiplication. The inclusion of a threshold-based stopping mechanism enhances this approach by ensuring that algorithms only operate within efficient and practical limits, preventing prolonged execution times for configurations where performance would not be optimal.

Results

In this section you will find the results of each experiment for every optimization method used:

Execution Time Comparison of Matrix Multiplication Algorithms

This graph (Figure 1) presents the comparison of execution time in milliseconds for each matrix multiplication algorithms as a function of matrix size.

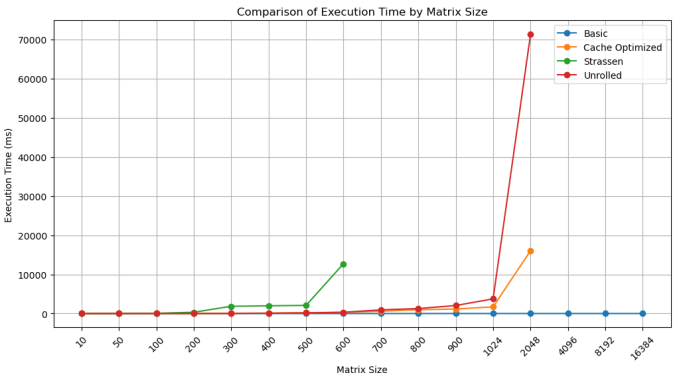

Figure 1. Execution Time Comparison

For smaller matrix sizes (e.g., from 10 to 900), all algorithms show relatively low and nearly constant execution times, suggesting that for small matrices, algorithmic complexity does not significantly affect execution time. As matrix size increases, performance differences between algorithms become apparent.

The basic algorithm maintains a consistently low execution time, remaining below the 5-second threshold even for large sizes (up to 16,384). This may indicate that the basic algorithm provides stable execution times for larger matrices, or that its optimizations are less sensitive to matrix size.

Both Strassen and Cache optimization algorithms show increases in execution time as matrix size grows, though they remain manageable for most sizes. However, Strassen's algorithm reaches a notable peak at size 600, likely due to its recursive nature, which becomes less efficient for certain matrix sizes because of its subdivision into submatrices. Cache optimization, while generally effective for larger matrices, shows an execution time increase at size 2048, likely due to cache memory limits reached with larger matrices.

The loop unrolling algorithm is fast for small matrix sizes, but its execution time increases significantly for matrix sizes of 1024 and above. This suggests that unrolling loses effectiveness with large matrices, possibly due to increased memory access and cache management overhead. At size 2048, execution time exceeds 70,000 ms, indicating that this algorithm is no longer optimal and has surpassed the defined efficiency threshold, explaining why larger sizes were not tested with loop unrolling.

Each algorithm reaches a different maximum matrix size due to a set execution time threshold of 5 seconds (5,000 ms). This threshold stops an algorithm's execution when runtime becomes excessive, indicating that it is not optimal for larger sizes. The threshold is particularly evident for the loop unrolling algorithm, which halts at 2048 due to execution time surpassing the threshold. Cache optimization also shows increased time at this size, potentially triggering an early stop at even larger sizes.

The graph reveals that while each algorithm has advantages at small or medium matrix sizes, not all are equally effective for large matrices. The applied stop threshold ensures that unnecessary calculations are avoided for configurations that are inefficient, preventing excessive execution times and allowing focus on matrix sizes where each algorithm is truly effective.

Memory Usage of Matrix Multiplication Algorithms

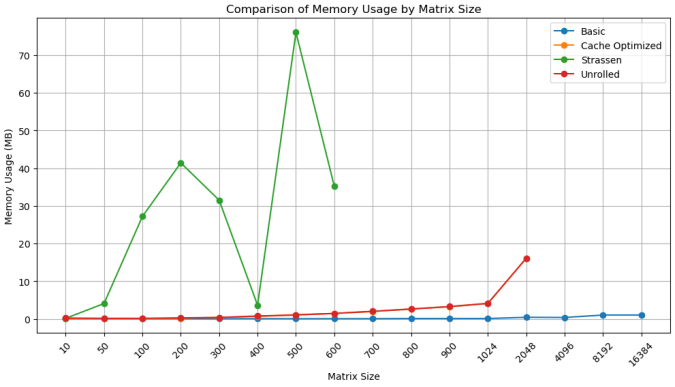

Figure 2. Memory Usage Comparison

As seen in Figure 2, Strassen's algorithm shows memory usage peaks for medium-sized matrices, especially at sizes 200 and 500, reaching up to 75 MB. This is due to the recursive nature of the algorithm, which requires dividing the matrix into multiple submatrices, temporarily increasing memory usage.

The loop unrolling algorithm maintains low, consistent memory usage for small sizes but begins to increase from matrix size 1024 onwards. This suggests that its memory efficiency decreases for larger sizes.

Both basic and Cache Optimization algorithms maintain consistently low memory usage compared to the other two, indicating that they have no significant memory overhead, making them more efficient in this aspect for all tested sizes.

As seen in the Figure 1, some algorithms do not reach larger matrix sizes due to the stop threshold based on execution time. This explains the absence of data for larger sizes for certain algorithms.

Memory Usage of Matrix Multiplication Algorithms

As seen in Figure 3, the cache optimized algorithm shows relatively high CPU usage compared to the others, with peaks at matrix sizes 200 and 500. The high CPU demand is expected due to memory access optimization, which aims to maximize cache utilization and may increase CPU load.

Strassen's algorithm has fluctuating CPU usage, with high values for smaller sizes (up to 200) that decrease as matrix size grows. This may be due to

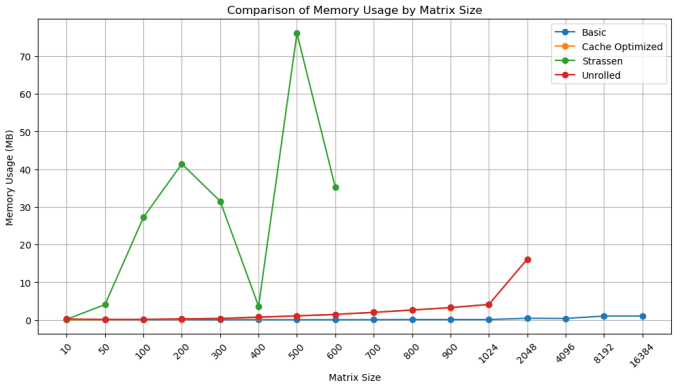

Figure 3. CPU Usage Comparison

the recursive nature of the algorithm, which can vary in CPU load depending on matrix subdivision.

The loop unrolling algorithm maintains relatively low and consistent CPU usage, indicating that this optimization is not CPU-intensive and is efficient in this regard.

Finally, the basic algorithm has the lowest and most consistent CPU usage across all matrix sizes, suggesting it is the least CPU-demanding, making it more suitable for applications where CPU usage needs to be limited.

Again, some algorithms do not reach larger matrix sizes due to the execution time threshold.

Sparse Matrix Analysis

This analysis examines the behavior of sparse matrix multiplication in relation to matrix size and sparsity level, which represents the percentage of zero elements in the matrix (with levels of 50%, 80%, and 90%). The same metrics were evaluated.

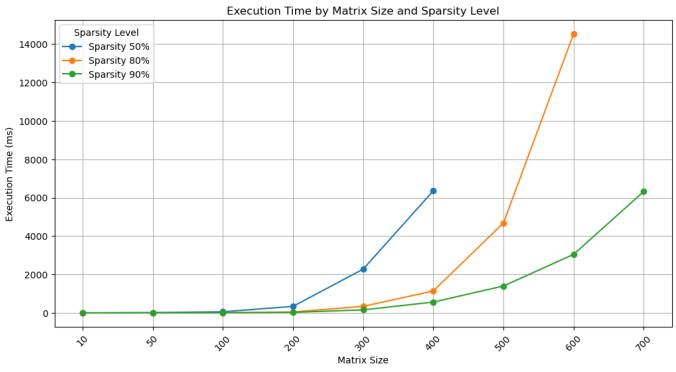

Figure 4. Execution Time by Matrix Size and Sparsity Level

Figure 4 shows how execution time increases with matrix size and how sparsity level affects this time:

- **50% Sparsity (blue):** This level shows the highest execution times, as only half of the elements are zeros, requiring a greater number of operations.
- **80% Sparsity (orange):** With 80% zeros, execution time is significantly lower than at 50%, though it still increases with matrix size.
- **90% Sparsity (green):** With 90% zeros, this level displays the lowest execution times due to the reduced number of operations.

In conclusion, as sparsity increases, execution time decreases, confirming that high sparsity levels are more efficient for large matrices.

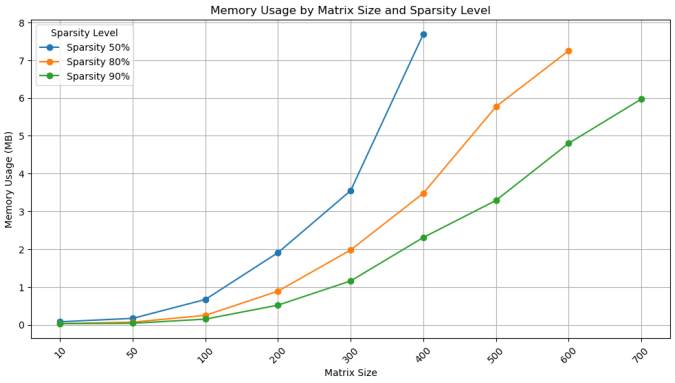

Figure 5. Memory Usage by Matrix Size and Sparsity Level

Figure 5 illustrates memory usage in relation to matrix size and sparsity level: 50% Sparsity uses the most memory, followed by 80% Sparsity and finally 90% Sparsity.

As matrix size grows, memory usage also increases across all three sparsity levels, though this increase is more pronounced in matrices with lower sparsity (50%) due to a higher count of non-zero elements.

This suggests that memory usage is directly proportional to the number of non-zero elements, making highly sparse matrices (with more zeros) use less memory in multiplication.

Figure 6 presents CPU usage according to matrix size and sparsity level:

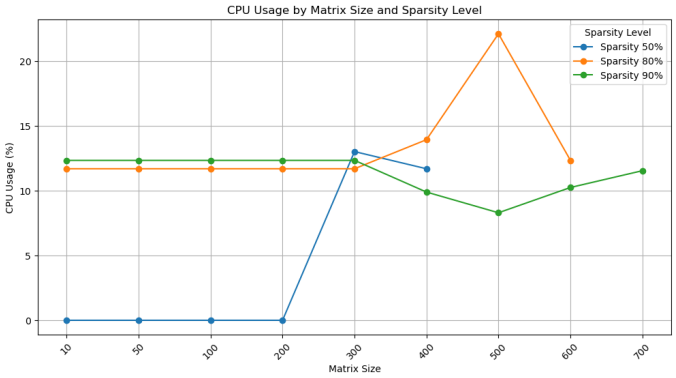

Figure 6. CPU Usage by Matrix Size and Sparsity Level

- 50% Sparsity shows an upward trend in CPU usage as matrix size increases, though moderately.
- 80% Sparsity and 90% Sparsity have more stable CPU values, with some occasional peaks. These peaks may result from the nature of sparsity and how the CPU manages operations in matrices with many zeros.

Overall, CPU usage is higher in less sparse matrices, indicating that a greater density of non-zero elements increases CPU load.

5. CONCLUSIONS

This project provides a comprehensive analysis of matrix multiplication optimization techniques, comparing the basic algorithm, Strassen's algorithm, cache optimization, and loop unrolling across various matrix sizes and sparsity levels. Each approach offers unique advantages depending on the matrix characteristics and computational requirements.

The **basic algorithm** demonstrated consistent efficiency across both small and large matrix sizes, proving to be the most stable in terms of memory and CPU usage. Its reliability makes it suitable for scenarios where predictability and low resource consumption are prioritized.

Strassen's algorithm and **cache optimization** were effective for certain matrix sizes but showed variable performance and efficiency, especially at larger sizes. Strassen's recursive nature led to increased memory and CPU usage, particularly in medium-sized matrices, which suggests it may be best suited for matrices that fit well within cache memory constraints. Cache optimization, while

generally effective for reducing memory access time, also experienced higher CPU demands at certain sizes, indicating that it works best in specific size ranges where cache utilization can be maximized without overloading the CPU.

Loop unrolling proved fast for small matrices, with low CPU usage, but quickly became inefficient as matrix size grew. The early execution stop observed in larger matrices underscores its limitations and highlights that this technique is better suited for applications involving smaller matrices.

For **sparse matrices**, high sparsity levels (e.g., 90%) consistently resulted in better performance across all metrics—execution time, memory usage, and CPU usage. By minimizing operations on zero elements, high sparsity levels allow substantial computational savings, making this approach ideal for applications that handle large, data-intensive matrices and require efficient resource usage.

In summary, the basic algorithm and cache optimization are the most memory-efficient techniques, while the basic algorithm and loop unrolling are the most effective in terms of CPU efficiency. For applications with large, sparse matrices, leveraging high sparsity levels is crucial for optimizing resource usage. This analysis provides a framework for selecting the appropriate matrix multiplication strategy based on specific performance needs, offering insights that can guide optimization in high-performance computing environments.