Optimized Matrix Multiplication Approaches and Sparse Matrices

Lucía Afonso Medina

November 10, 2024

Contents

Abstract

Matrix multiplication is a fundamental operation in numerous computational fields, including machine learning, scientific simulations, and big data analytics. However, the naive matrix multiplication algorithm, with a time complexity of $O(n^3)$, becomes inefficient for large matrices due to high memory demands and poor cache utilization. This study investigates optimized approaches for matrix multiplication, focusing on techniques such as cache blocking and loop unrolling to improve computational efficiency. Additionally, we explore the use of sparse matrices, which contain a high proportion of zero elements, to further enhance performance by minimizing unnecessary computations. Experimental results compare the execution time, memory usage, and scalability of the naive and optimized algorithms across varying matrix sizes and sparsity levels.

1 Introduction

Matrix multiplication is a core operation in a wide range of applications, including data processing, machine learning, and scientific computing. Efficient matrix multiplication is crucial for handling large datasets and complex simulations, but the traditional naive algorithm has significant limitations. With a time complexity of $O(n^3)$, this approach becomes computationally expensive as the size of the matrices increases, leading to prolonged processing times and elevated memory requirements.

Given the need for more efficient matrix multiplication methods, various optimization techniques have been developed. These techniques aim to improve computation time and memory efficiency by leveraging hardware characteristics such as cache memory and parallel processing capabilities. In particular, cache blocking and loop unrolling are two well-established methods that restructure matrix multiplication to enhance data locality and reduce loop overhead, respectively.

Sparse matrices, which contain a high percentage of zero elements, offer another avenue for optimization. By focusing on non-zero elements, sparse matrix representations allow for reduced memory usage and faster computation, making them particularly useful in fields where data sparsity is common, such as network analysis, machine learning, and scientific simulations. This study aims to evaluate and compare the performance of the naive, optimized, and sparse matrix multiplication approaches, analyzing how these techniques affect execution time, memory usage, and scalability across varying matrix sizes and sparsity levels.

2 Problem Statement

The naive approach to matrix multiplication, with its $O(n^3)$ complexity, poses significant limitations when dealing with large matrices, both in terms of time and memory efficiency. As datasets grow in size and complexity, this computational cost becomes prohibitive, especially when high-performance and real-time processing are required. Furthermore, the inefficiencies of the naive algorithm are amplified in cases where matrices are sparse, as it performs unnecessary computations on zero elements, wasting valuable memory and processing power.

To address these issues, we seek to implement and evaluate optimized matrix multiplication algorithms that leverage cache blocking and loop unrolling to improve performance. Additionally, we aim to incorporate sparse matrix representations, such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, to reduce memory usage and accelerate computation by focusing only on non-zero elements. By comparing these optimized algorithms to the naive approach, this study aims to determine the extent to which optimized and sparse matrix multiplication techniques can improve computational efficiency, particularly for large matrices and varying levels of sparsity.

3 Methodology

This section outlines the methodology and experimental setup used to evaluate the performance of different matrix multiplication algorithms. The focus of this study is to compare the execution time, memory usage, and scalability of three approaches: the naive matrix multiplication, and two optimized methods using cache blocking and loop unrolling. Additionally, we examine the performance impact of sparse matrix representations, specifically in Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, across different sparsity levels.

3.1 Experimental Setup

All experiments were conducted on a machine with the following specifications:

- Processor: 13th Gen Intel(R) Core(TM) i5-1340P, 1.90 GHz
- RAM: 16.0 GB
- Operating System: Windows 11 Home, Version 23H2
- Compiler/Interpreter versions:
  - Java: Version 21.0.1

3.2 Optimized Matrix Multiplication Approaches

In this section, we present three approaches for matrix multiplication: the naive algorithm, an optimized version using cache blocking, and another optimization through loop unrolling. Each approach and its impact on performance are explained below.

3.2.1 Naive Matrix Multiplication (O(n^3) Complexity)

Algorithm 1 Naive Matrix Multiplication

```
1: for $i = 0$ to $n - 1$ do
2:     for $j = 0$ to $n - 1$ do
3:         $C[i][j] \leftarrow 0$
4:         for $k = 0$ to $n - 1$ do
5:             $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
6:         end for
7:     end for
8: end for
```

The naive matrix multiplication algorithm is a straightforward approach with a time complexity of $O(n^3)$, where n is the size of the matrix (assuming square matrices of size $n \times n$). This method consists of three nested loops that iterate over each row and column of the input matrices A and B. For each element in the resulting matrix C, it performs a sum of products from the corresponding row in A and column in B.

The main advantage of the naive algorithm is its simplicity and ease of implementation. However, it has significant limitations when it comes to performance. Since the algorithm does not take advantage of memory locality, it tends to have frequent cache misses, especially when dealing with large matrices. This is because each element access in A and B may require fetching data from non-contiguous memory locations, which increases memory access time.

In summary, while the naive approach provides a baseline for matrix multiplication, its inefficiency in terms of cache usage and high computational complexity make it less suitable for large-scale problems. This motivates the use of optimized algorithms, such as cache blocking and loop unrolling, to better utilize hardware resources and improve computation time.

3.2.2 Matrix Multiplication with Cache Blocking

Algorithm 2 Blocked Matrix Multiplication

```
1:  for $i\_block = 0$ to $n - 1$ step $B\_size$ do
2:      for $j\_block = 0$ to $n - 1$ step $B\_size$ do
3:          for $k\_block = 0$ to $n - 1$ step $B\_size$ do
4:              for $i = i\_block$ to $\min(i\_block + B\_size, n) - 1$ do
5:                  for $j = j\_block$ to $\min(j\_block + B\_size, n) - 1$ do
6:                      for $k = k\_block$ to $\min(k\_block + B\_size, n) - 1$ do
7:                          $C[i][j] \leftarrow C[i][j] + A[i][k] \times B[k][j]$
8:                      end for
9:                  end for
10:             end for
11:         end for
12:     end for
13: end for
```

The cache blocking optimization improves the naive algorithm by taking advantage of the CPU cache. Instead of processing entire rows and columns of the matrices, cache blocking divides the matrices into smaller submatrices (or blocks) of size $B \times B$. Each block fits into the cache, allowing the CPU to access data more efficiently and reduce cache misses.

3.2.3 Matrix Multiplication with Loop Unrolling

Algorithm 3 Unrolled Matrix Multiplication

```
1:  for $i = 0$ to $n - 1$ do
2:      for $j = 0$ to $n - 1$ do
3:          $sum \leftarrow 0$
4:          for $k = 0$ to $n - 1$ step 4 do                ▷ Unroll by a factor of 4
5:              $sum \leftarrow sum + A[i][k] \times B[k][j]$
6:              $sum \leftarrow sum + A[i][k+1] \times B[k+1][j]$
7:              $sum \leftarrow sum + A[i][k+2] \times B[k+2][j]$
8:              $sum \leftarrow sum + A[i][k+3] \times B[k+3][j]$
9:          end for
10:         $C[i][j] \leftarrow sum$
11:     end for
12: end for
```

Loop unrolling is an optimization that reduces the overhead associated with loop control structures. By unrolling the innermost loop, we increase the number of operations within a single loop iteration, which reduces the number of conditional checks and increments. In this case, the innermost loop is unrolled by a factor of 4, which decreases the number of iterations and allows the processor to perform more calculations in each cycle.

This technique improves performance by allowing the CPU to take advantage of its ability to execute multiple instructions simultaneously (instruction-level parallelism). Loop unrolling reduces control logic instructions and enables more efficient use of the CPU's pipeline. It is most effective when the matrix size is a multiple of the unrolling factor; otherwise, additional code is needed to handle any remaining elements.

3.3 Sparse Matrix Multiplication

Sparse matrix multiplication is an optimization approach used when dealing with matrices that contain a high percentage of zero elements.Directly applying standard matrix multiplication to sparse matrices is inefficient, as it performs redundant calculations on zero values, wasting both memory and computational resources. In sparse matrix multiplication, only the non-zero elements are considered, reducing the number of operations and memory usage. When multiplying two sparse matrices, the algorithm focuses on aligning and multiplying only those rows and columns that contain non-zero elements, bypassing unnecessary operations.

3.4 Representation of Sparse Matrices

To efficiently store and manipulate sparse matrices, specialized formats are used that only record non-zero elements and their locations. Two common formats for sparse matrix representation are Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC), which are explained below:

- **Compressed Sparse Row (CSR)**: This format is well-suited for row-wise operations. It uses three arrays:
  - **Values**: Contains the non-zero elements of the matrix in row-major order.
  - **Column Indices**: Stores the column indices corresponding to each element in the *Values* array.
  - **Row Pointers**: This array indicates the starting position of each row in the *Values* array, allowing quick access to each row's non-zero elements.

  The CSR format reduces memory usage by eliminating the need to store zero elements and enables efficient access to row data, making it suitable for operations where rows of the matrix are processed sequentially.

- **Compressed Sparse Column (CSC)**: This format is analogous to CSR but is optimized for column-wise access. It also uses three arrays:
  - **Values**: Stores non-zero elements column by column.
  - **Row Indices**: Holds the row indices for each non-zero element in the *Values* array.

  - **Column Pointers**: Points to the start of each column in the *Values* array, allowing quick access to each column's non-zero elements.

  The CSC format is ideal for column-wise operations, making it advantageous in scenarios where matrix multiplication or transformations require accessing data by columns.

Both CSR and CSC formats dramatically reduce memory consumption and avoid redundant calculations on zero elements, enhancing the overall efficiency of sparse matrix operations.

In our study, we evaluated the performance of sparse matrix multiplication under various sparsity levels to understand the impact on execution time and memory efficiency.

3.5 Experimental Procedure

To ensure reliable results, each algorithm was executed multiple times for various matrix sizes and sparsity levels. Execution time was measured using the Java Microbenchmark Harness (JMH), which provides high-resolution and precise timing specifically designed for benchmarking in Java. Additionally, memory usage was tracked to assess each algorithm's efficiency in handling larger matrices. The results were then analyzed to determine the maximum matrix size that each algorithm could handle efficiently and to evaluate the advantages of sparse matrix representations across different sparsity levels.

3.6 Metrics for Evaluation

The primary metrics used to evaluate the performance of each approach include:

- **Execution Time**: The time required for each matrix multiplication operation, measured in miliseconds.
- **Memory Usage**: The amount of memory utilized during each operation, providing insight into the efficiency of each algorithm.
- **Scalability**: The maximum matrix size that each approach can handle efficiently, reflecting the impact of optimizations and sparse representations.

This methodology allows us to compare the effectiveness of different optimization techniques and sparse representations in improving the efficiency of matrix multiplication for various matrix sizes and sparsity levels.

4 Experiments

In the graph in Figure 1 , we observe the execution time for matrix multiplication as matrix size increases, comparing three different optimization techniques: **Cache Blocking**, **Loop Unrolling**, and **Naive Multiplication**. The y-axis uses a logarithmic scale to capture the wide range of execution times across various matrix sizes, from 10 to 2000.

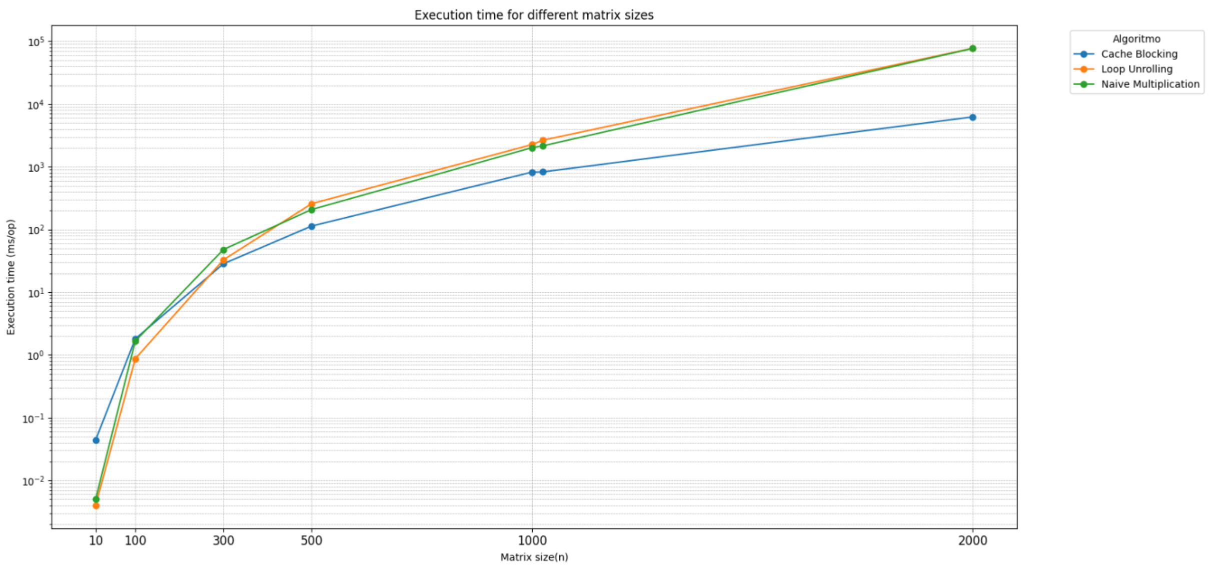


Figure 1: Execution time comparison for different matrix sizes and optimization algorithms.

Initially, for small matrix sizes (up to about 100), the execution times of the three methods are very similar, indicating that optimization techniques do not significantly impact performance for smaller matrices. This could be because, at smaller sizes, the total computation time remains low, and the differences between optimization techniques are less apparent.

As matrix size increases, differences between the methods become more pronounced. **Cache Blocking** (shown in blue) maintains a more gradual increase in execution time compared to **Loop Unrolling** and **Naive Multiplication**, making it more efficient at larger sizes.

Loop Unrolling (orange) initially performs well, but as matrix size grows beyond 500, its execution time starts to align more closely with **Naive Multiplication** (green). This suggests that the advantages of **Loop Unrolling** decrease with larger matrices, where it becomes less effective than **Cache Blocking**.

Overall, the graph shows that **Cache Blocking** scales more efficiently for large matrices, while **Loop Unrolling** and **Naive Multiplication** exhibit less effective scaling as matrix size increases. This trend highlights the suitability of **Cache Blocking** for handling larger datasets in matrix multiplication tasks.

In Figure 2, we observe the memory usage across different matrix sizes for three optimization techniques: **Cache Blocking**, **Loop Unrolling**, and **Naive Multiplication**. The y-axis uses a logarithmic scale to accommodate the wide range of memory usage values, from smaller matrices (size 10) to larger matrices (size 2000). This allows us to clearly see the differences in memory consumption among the algorithms as matrix size increases.

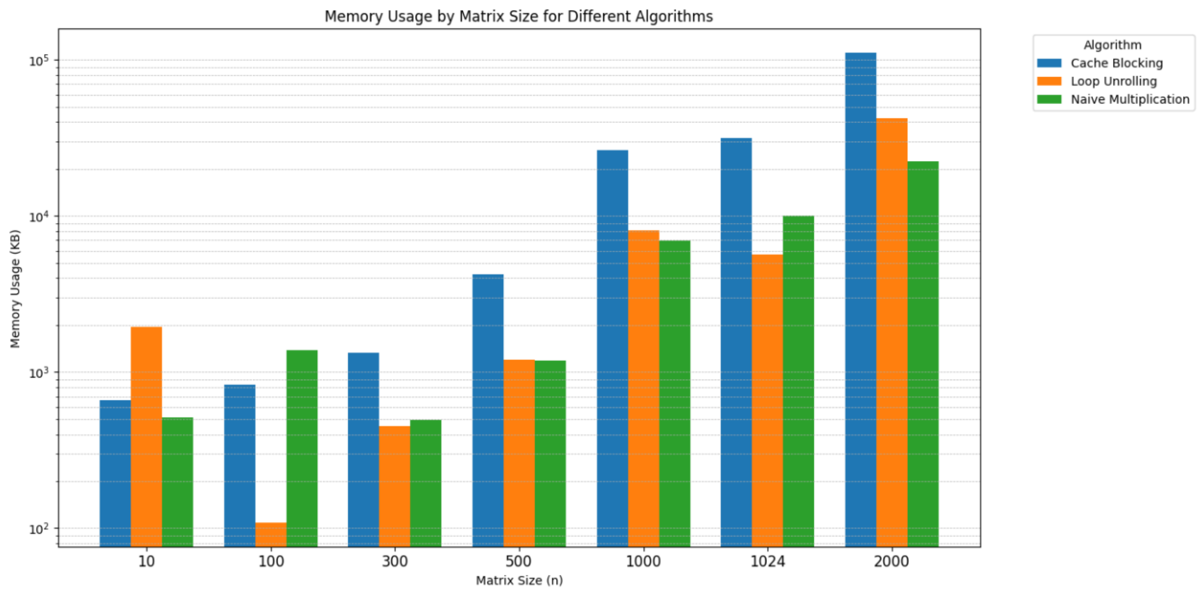
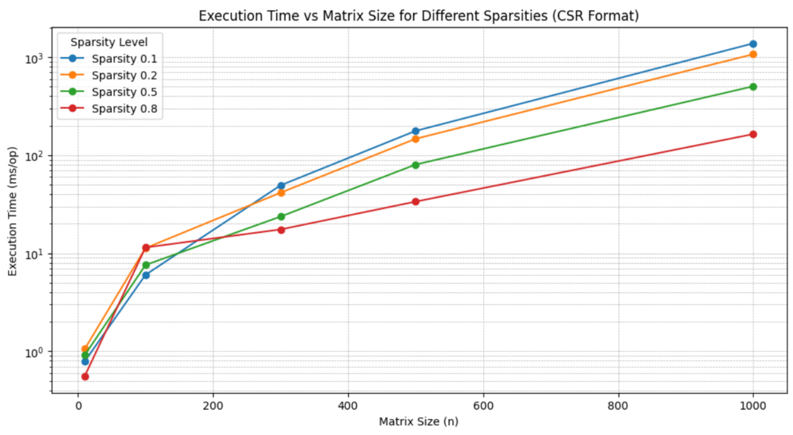


Figure 2: Memory usage comparison for different matrix sizes and optimization algorithms.

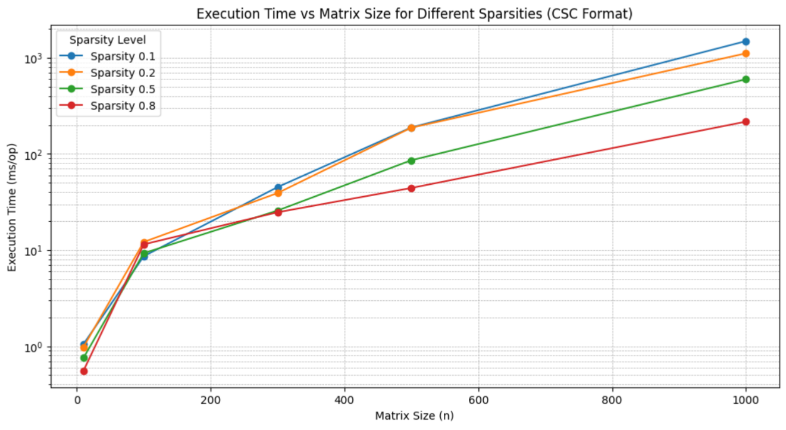
For smaller matrices (size 10), **Loop Unrolling** exhibits the highest memory usage compared to **Cache Blocking** and **Naive Multiplication**. This suggests that **Loop Unrolling** introduces additional overhead in terms of memory usage at very small matrix sizes. However, as the matrix size grows, **Cache Blocking** consistently shows the highest memory usage among the three techniques, especially for matrices of sizes 1000 and above. This trend is expected since **Cache Blocking** divides the matrix into smaller blocks, potentially increasing memory consumption due to additional storage requirements for the blocks.

On the other hand, **Naive Multiplication** generally maintains lower memory usage across all matrix sizes, highlighting its simplicity and lack of additional memory-intensive optimizations. For medium-sized matrices (around 300 to 500), the memory usage of **Loop Unrolling** and **Naive Multiplication** is comparable, while **Cache Blocking** begins to diverge with significantly higher memory requirements.

The graphs in Figure 3 illustrate the relationship between execution time and matrix size across varying sparsity levels for the **CSR** and **CSC** formats. The y-axis, set on a logarithmic scale, captures execution time (ms/op) for different sparsity levels (0.1, 0.2, 0.5, and 0.8) across matrix sizes from 10 to 1000. This logarithmic scale helps to highlight how execution time scales as matrix dimensions increase.



(a) Execution Time vs Matrix Size for Different Sparsities (CSR Format)



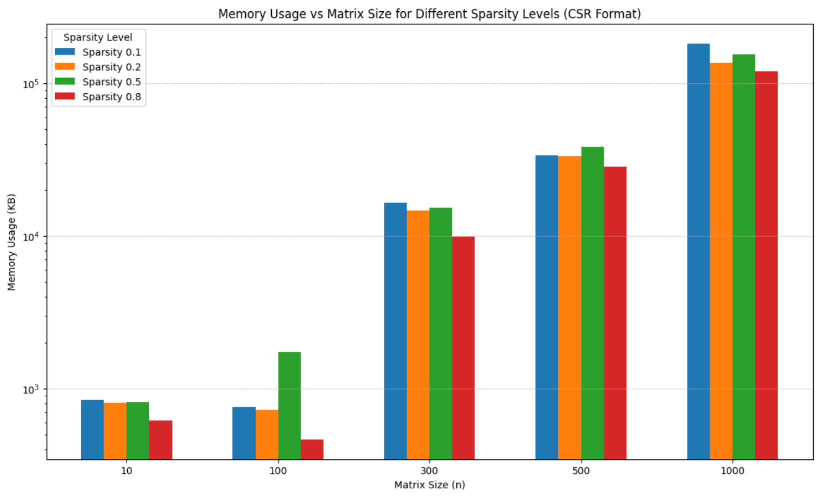
(b) Execution Time vs Matrix Size for Different Sparsities (CSC Format)

Figure 3: Comparison of Execution Time for Different Sparsities in CSR and CSC Formats

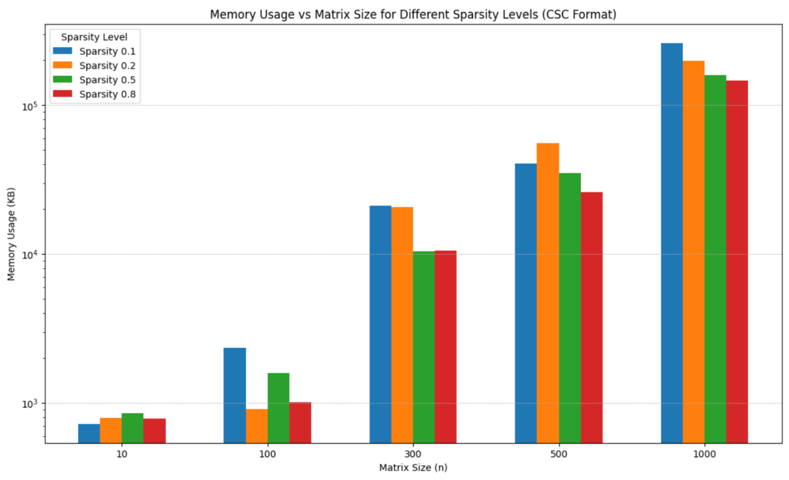
For both CSR and CSC formats, higher sparsity levels (e.g., 0.8) consistently result in lower execution times. This outcome is expected, as matrices with more sparsity (i.e., a higher percentage of zero elements) reduce the number of operations required, leading to faster processing. Conversely, lower sparsity levels, such as 0.1, involve more computations due to a higher density of non-zero elements, resulting in longer execution times.

When comparing **CSR** and **CSC** formats, the execution times follow similar trends across all sparsity levels, although the CSR format generally shows slightly lower execution times. This difference suggests that the CSR format may be marginally more efficient for handling sparse matrices in terms of execution time. However, both formats demonstrate that as matrix size grows, the impact of sparsity becomes more pronounced, with higher sparsity levels showing much slower increases in execution time compared to lower sparsity levels.

The two graphs in Figure 4 show the memory usage across different sparsity levels (0.1, 0.2, 0.5, and 0.8) for increasing matrix sizes, using CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats.



(a) Memory Usage vs Matrix Size for Different Sparsities (CSR Format)



(b) Memory Usage vs Matrix Size for Different Sparsities (CSC Format)

Figure 4: Comparison of Memory Usage for Different Sparsities in CSR and CSC Formats

As the sparsity level increases (meaning more zero elements), the memory usage generally decreases in both CSR and CSC formats. This trend is because higher sparsity levels involve storing fewer non-zero elements, which reduces the overall memory requirement. For matrices with high sparsity (e.g., 80% zeros), memory usage is consistently lower across all matrix sizes compared to matrices with lower sparsity (e.g., 10% zeros). Memory usage increases with matrix size across all sparsity levels, which aligns with the expectation that larger matrices require more storage. At lower sparsity levels (e.g., 0.1 or 10% zeros), the memory usage grows more significantly with matrix size, reflecting the storage cost associated with denser data. Both CSR and CSC formats follow similar patterns in memory usage, with slight variations based on the matrix sparsity and size. In general, CSR and CSC formats exhibit comparable memory efficiency for matrices of similar sizes and sparsity levels, although there may be minor differences depending on the specific configuration.

In Figure 5, we compare all algorithms using dense matrices (0% sparsity) to understand their performance with fully populated data. The x-axis represents matrix size, ranging from 10 to 1000, and the y-axis, on a logarithmic scale, shows the execution time (ms/op) for each algorithm.

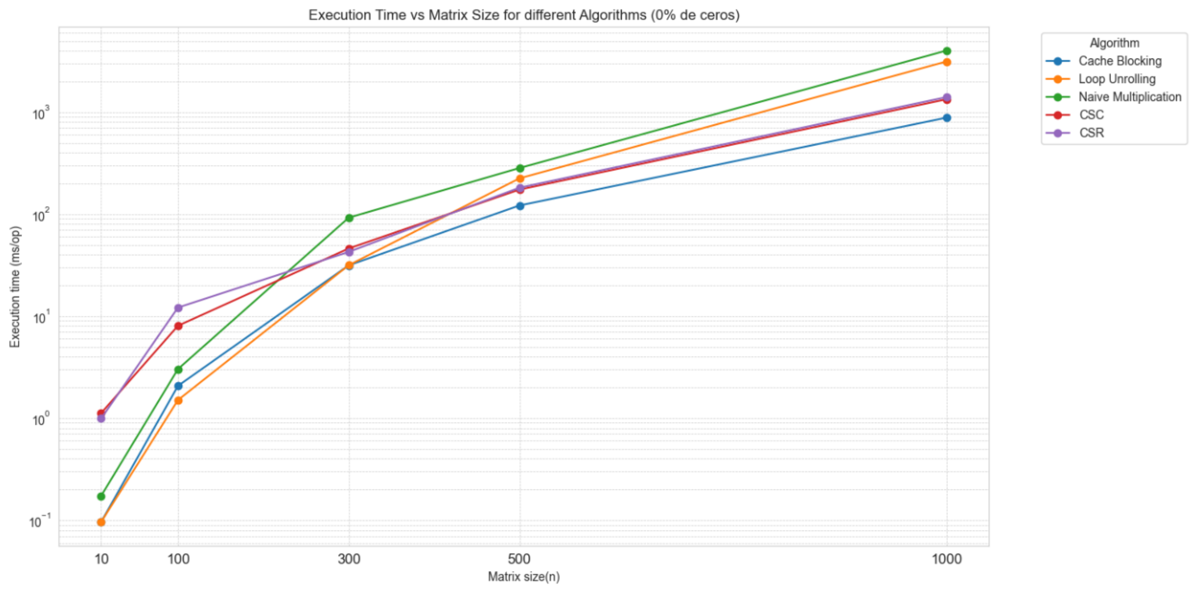


Figure 5: Execution Time comparison for different matrix sizes and optimization algorithms.

For smaller matrices (10 - 100), **CSR** and **CSC** formats exhibit the highest execution times among the algorithms. This suggests that these formats are less efficient with small dense matrices, possibly due to the overhead involved in handling sparse data structures even when the matrix is fully populated. However, as the matrix size increases, the **Loop Unrolling** and **Naive Multiplication** algorithms begin to show much higher execution times, surpassing **CSR** and **CSC**. This indicates that **Loop Unrolling** and **Naive Multiplication** are less scalable and become inefficient for larger matrices.

Cache Blocking demonstrates the best scalability, maintaining the lowest execution times as the matrix size grows. This efficiency is due to its ability to optimize cache usage effectively, making it the preferred choice for large dense matrices. Overall, while **CSR** and **CSC** formats perform slower on small dense matrices, they manage to stabilize with larger matrices, and **Cache Blocking** remains the most efficient algorithm for dense matrix multiplication.

This Figure 6 illustrates the execution time of various optimization algorithms across different matrix sizes and sparsity levels. The x-axis represents matrix size, while the y-axis, on a logarithmic scale, shows the execution time in milliseconds per operation (ms/op).

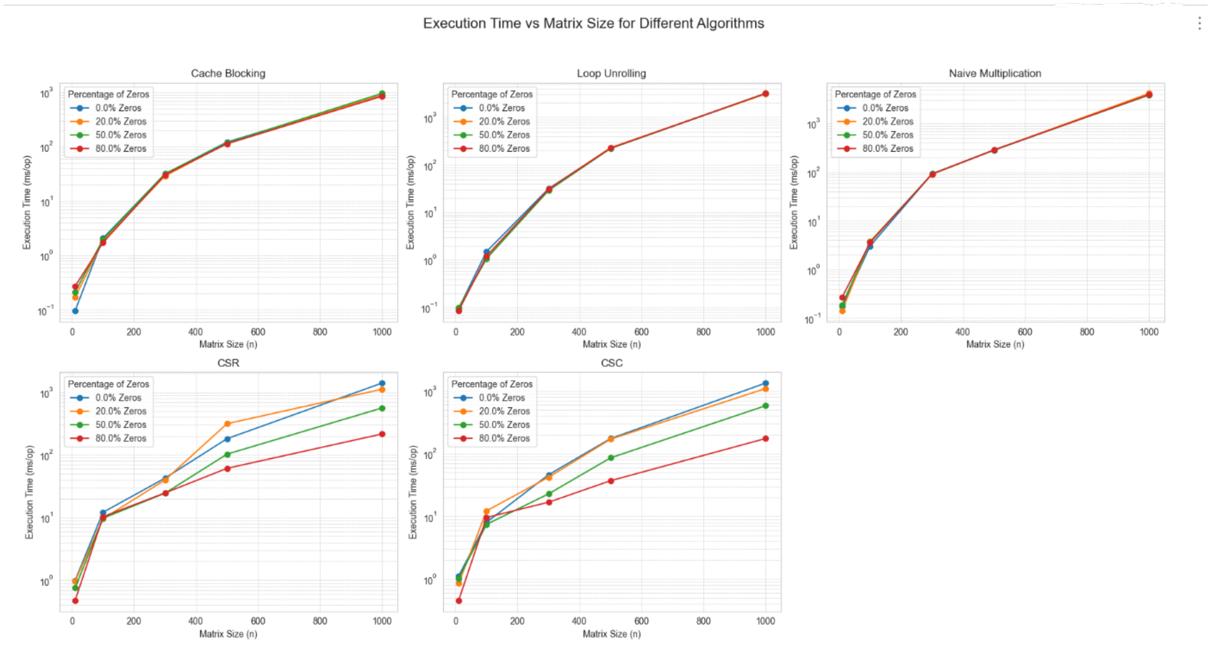


Figure 6: Execution Time comparison for different matrix sizes and optimization algorithms.

In general, for algorithms like **Cache Blocking**, **Loop Unrolling**, and **Naive Multiplication**, the execution time increases consistently with matrix size, regardless of sparsity. However, for sparse matrix formats like **CSR** and **CSC**, the execution time is significantly affected by the sparsity level. Higher sparsity levels (e.g., 80% zeros) show noticeably lower execution times, as the reduced number of non-zero elements leads to fewer computations.

For smaller matrices, the impact of sparsity is less pronounced, especially in algorithms optimized for dense matrices, such as **Cache Blocking** and **Loop Unrolling**. However, as matrix size increases, higher sparsity levels (50% and 80%) help to improve performance in sparse formats (**CSR** and **CSC**), as these formats can leverage the structure more effectively.

Cache Blocking remains the most efficient for larger matrices regardless of sparsity due to its cache optimization strategy. On the other hand, **Loop Unrolling** and **Naive Multiplication** become less efficient as matrix size grows, especially in denser configurations. This demonstrates the effectiveness of choosing an algorithm based on both matrix size and sparsity requirements.

5 Conclusions

In conclusion, the findings of this study underscore the importance of selecting appropriate algorithms and data structures for matrix multiplication based on the matrix's size and sparsity characteristics. For large, dense matrices, **Cache Blocking** is the most efficient approach, as it optimizes cache usage, leading to faster execution times and improved scalability. However, this efficiency comes at the cost of increased memory usage due to the need for managing smaller blocks within the cache. **Loop Unrolling** is also beneficial but primarily for smaller matrices, as its efficiency diminishes with increasing matrix size, where it becomes less effective than **Cache Blocking**. These results suggest that while dense matrix multiplication techniques offer speed benefits, they may be memory-intensive, especially for very large matrices.

For high-sparsity matrices, **CSR** and **CSC** formats provide substantial advantages in terms of both memory usage and computational efficiency. By storing only the non-zero elements, these formats reduce memory requirements significantly, making them ideal for large, sparse datasets where a dense representation would be inefficient. Sparse matrix formats also improve computational efficiency by focusing only on meaningful data, avoiding redundant calculations on zero elements, and thus resulting in faster execution times.

The choice between dense and sparse matrix multiplication methods should align with the specific requirements of the task, including matrix size, sparsity, and memory limitations. Dense approaches like **Cache Blocking** are preferable for fully populated matrices due to their speed, while sparse formats like **CSR** and **CSC** are essential for handling high-sparsity matrices where memory efficiency is paramount. These results highlight the critical role of memory management and algorithm selection in achieving optimal performance and scalability, providing valuable insights for applications that involve large-scale matrix operations across diverse fields, from scientific computing to machine learning.

6 Future Work

Future work on optimized matrix multiplication should focus on parallel processing techniques to improve scalability and efficiency. Implementing parallel matrix multiplication on multi-core CPUs and GPUs could significantly reduce execution times for large matrices. Techniques such as OpenMP for CPU parallelism and CUDA for GPU acceleration can help distribute computation across multiple cores, enhancing performance and making larger datasets feasible to process.

Additionally, integrating parallelism with optimizations like cache blocking and loop unrolling may further improve data locality and reduce memory access times within each parallelized task. Exploring distributed computing frameworks, such as MPI, could also enable processing of extremely large matrices by distributing the workload across multiple machines.

The source code for this project is available at the following GitHub repository: *OptimizedMatrixMultiplication.git*