Optimized Matrix Multiplication Approaches and Sparse Matrices

Alejandro Alemán Alemán

November 10, 2024

Contents

Abstract

Matrix multiplication is a fundamental operation in various computational fields, yet it is computationally intensive, especially for large-scale matrices. This work explores several optimization techniques to improve matrix multiplication efficiency, including loop unrolling, and cache optimization. Additionally, we investigate the use of sparse matrices, represented in Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats, which allow for significant reductions in storage and computational requirements when matrices contain a high percentage of zero elements. Our study examines the effects of different sparsity levels on performance, comparing dense and sparse representations across multiple matrix sizes. Performance metrics, such as execution time, memory usage, and maximum manageable matrix size, are analyzed to evaluate the effectiveness of each optimization. Findings indicate that these optimizations can significantly enhance performance, with sparse matrix formats offering notable efficiency improvements in high-sparsity scenarios. This work provides insights into selecting and implementing matrix multiplication techniques for both dense and sparse matrices, based on specific computational needs and data characteristics.

1 Introduction

1.1 Context

Matrix multiplication is a critical operation across many computationally intensive fields, including artificial intelligence, computational physics, and data analysis. In applications like deep learning, simulations, and large-scale data processing, efficient matrix multiplication is essential for handling large datasets and performing complex calculations in a reasonable time frame. However, as matrix size grows, so do the computational and memory requirements, which makes optimization techniques for matrix multiplication crucial.

1.2 Objective

The objective of this work is to investigate various optimization techniques for matrix multiplication and evaluate their impact on performance. By implementing optimized algorithms and using sparse matrix representations, we aim to reduce execution time and memory usage. Additionally, we analyze how different sparsity levels affect the efficiency of matrix multiplication, exploring the benefits of using compressed formats when matrices contain a high percentage of zero elements.

1.3 Scope

This study includes the implementation and evaluation of several optimization methods for matrix multiplication, such as loop unrolling, and cache optimization techniques. We also examine the use of sparse matrices, employing Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats to enhance performance in cases where the majority of matrix elements are zero. Performance analyses focus on metrics like execution time, memory usage, and the maximum manageable matrix size, comparing dense and sparse matrices across various levels of sparsity. This work aims to provide insights into selecting appropriate matrix multiplication techniques based on specific computational demands and data characteristics.

2 Optimized Approaches in Matrix Multiplication

2.1 Basic Algorithm

This is the basic matrix multiplication algorithm follows the traditional and most widely known method for multiplying two matrices, often referred to as the classical algorithm. This method operates with a time complexity of O(n^3), where n represents the size of the matrix (for an n x n matrix). The algorithm involves performing three nested loops to calculate the resulting matrix.

Given two square matrices, A and B, of size n x n, the algorithm aims to compute the result matrix, C, also of size n x n. For each element in the result matrix C, the algorithm computes the dot product of the corresponding row from matrix A and the corresponding column from matrix B. The process can be described as follows:

1. Outer Loop: Iterates over each row i of matrix A.

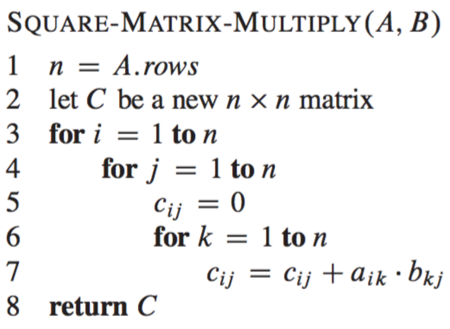
2. Middle Loop: Iterates over each column j of matrix B.

3. Inner Loop: Performs the dot product calculation by iterating through the elements of the row from A and the column from B. For each index k, the algorithm multiplies the elements $A[i][k]$ and $B[k][j]$, then adds the result to the current value of $C[i][j]$.

The resulting matrix element $C[i][j]$ is computed using the formula:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

This requires n multiplications and n additions for each element in the result matrix. Since there are n^2 elements in the result matrix, the overall complexity of the algorithm is O(n^3).



```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        $c_{ij} = 0$
        for k = 1 to n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C
```

Figure 1: Pseudo code of matrix multiplication algorithm implemented.

2.2 Loop unrolling

Loop unrolling is an optimization technique that reduces the overhead of loop control by ”unrolling” the loop, meaning that multiple iterations of the loop body are executed within a single iteration. This approach reduces the number of iterations needed and allows the compiler to better optimize for parallel execution. By performing multiple multiplications in one loop pass, loop unrolling increases data throughput, making it possible to use available processing resources more efficiently. Additionally, this technique minimizes the overhead of branching instructions, improving overall execution speed.

Algorithm 1 Matrix Multiplication with Loop Unrolling

```
Require: Matrices $A$ and $B$ of size $n \times n$
Ensure: Matrix $C = A \times B$
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $C[i][j] \leftarrow 0$
    end for
end for
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        Initialize $sum \leftarrow 0$
        for $k = 1$ to $n$ by 4 do
            $sum \leftarrow sum + A[i][k] \times B[k][j]$
            $sum \leftarrow sum + A[i][k+1] \times B[k+1][j]$
            $sum \leftarrow sum + A[i][k+2] \times B[k+2][j]$
            $sum \leftarrow sum + A[i][k+3] \times B[k+3][j]$
        end for
        $C[i][j] \leftarrow C[i][j] + sum$
    end for
end for
return $C$
```

2.3 Blocking (Cache optimization)

Cache optimization aims to improve data locality, reducing the time spent accessing data from slower memory. In matrix multiplication, poor cache utilization can result in frequent cache misses, slowing down execution. By carefully reordering operations and managing data layout, cache optimization ensures that data is loaded into cache efficiently, minimizing the need to access main memory. Techniques such as blocking (or tiling) are often used, where matrices are divided into smaller blocks that fit into the cache. This way, when a block of data is loaded, multiple operations can be performed on it before it is replaced, maximizing the cache's utility and speeding up computation.

Algorithm 2 Matrix Multiplication with Cache Blocking

```
Require: Matrices $A$ and $B$ of size $n \times n$, block size $B$
Ensure: Matrix $C = A \times B$
 1: for $i = 1$ to $n$ by $B$ do
 2:   for $j = 1$ to $n$ by $B$ do
 3:     for $k = 1$ to $n$ by $B$ do
 4:       for $ii = i$ to $\min(i + B - 1, n)$ do
 5:         for $jj = j$ to $\min(j + B - 1, n)$ do
 6:           $C[ii][jj] \leftarrow 0$
 7:         end for
 8:       end for
 9:       for $ii = i$ to $\min(i + B - 1, n)$ do
10:         for $jj = j$ to $\min(j + B - 1, n)$ do
11:           for $kk = k$ to $\min(k + B - 1, n)$ do
12:             $C[ii][jj] \leftarrow C[ii][jj] + A[ii][kk] \times B[kk][jj]$
13:           end for
14:         end for
15:       end for
16:     end for
17:   end for
18: end for
19: return $C$
```

2.4 Sparsed matrices and alogorithms

2.4.1 What Are Sparse Matrices?

Sparse matrices are matrices in which most elements are zero. Instead of storing all elements, sparse matrix representations focus only on the non-zero elements and their positions, significantly reducing memory usage and computational costs. This efficiency becomes particularly useful in fields like machine learning, data mining, and scientific computing, where matrices can be extremely large, and storage or computation on dense matrices would be infeasible.

2.4.2 Sparse Matrix Representations: CSR and CSC

To handle sparse matrices efficiently, we use specialized data structures like CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats:

-CSR (Compressed Sparse Row): In CSR format, a sparse matrix is stored row by row, compressing each row by only saving its non-zero values and their column indices. Additionally, an index pointer tracks the start of each row. CSR is particularly efficient for row-wise matrix operations, such as matrix-vector multiplication.

-CSC (Compressed Sparse Column): In CSC format, the matrix is stored column by column, saving only non-zero values and their row indices. Like CSR, it uses an index pointer to mark the start of each column. CSC format is more efficient for column-wise operations, which can be beneficial when the matrix multiplication involves transpositions or column access patterns.

Algorithm 3 CSR Sparse Matrix Multiplication

```
Require: Matrices $A$ and $B$ in CSR format with $A$ having dimensions $m \times k$ and $B$ having dimensions $k \times n$
Ensure: Matrix $C = A \times B$ in CSR format
 1: Initialize empty lists for $C_{\text{values}}$, $C_{\text{columnIndices}}$, and $C_{\text{rowPointers}}$
 2: Set $C_{\text{rowPointers}}[0] = 0$
 3: for each row $i$ in $A$ do
 4:   Initialize temporary array $rowResult$ of size $n$ with all elements set to 0
 5:   for each non-zero element in row $i$ of $A$ do
 6:     $colA \leftarrow A_{\text{columnIndices}}[j]$ {Get column index in $A$}
 7:     $valA \leftarrow A_{\text{values}}[j]$ {Get value of $A$ at $(i, colA)$}
 8:     for each non-zero element in row $colA$ of $B$ do
 9:       $colB \leftarrow B_{\text{columnIndices}}[k]$ {Get column index in $B$}
10:       $valB \leftarrow B_{\text{values}}[k]$ {Get value of $B$ at $(colA, colB)$}
11:       $rowResult[colB] \leftarrow rowResult[colB] + valA \times valB$ {Accumulate product in $rowResult$}
12:     end for
13:   end for
14:   Initialize $nonZeroCount \leftarrow 0$
15:   for each element $j$ in $rowResult$ do
16:     if $rowResult[j] \neq 0$ then
17:       Append $rowResult[j]$ to $C_{\text{values}}$
18:       Append $j$ to $C_{\text{columnIndices}}$
19:       $nonZeroCount \leftarrow nonZeroCount + 1$
20:     end if
21:   end for
22:   Append $(C_{\text{rowPointers}}[\text{last}] + nonZeroCount)$ to $C_{\text{rowPointers}}$
23: end for
24: return CSR matrix $C$ with $C_{\text{values}}$, $C_{\text{columnIndices}}$, and $C_{\text{rowPointers}}$
```

2.4.3 Why Use Sparse Matrices?

Storing only the non-zero elements saves memory and speeds up operations by avoiding calculations involving zero elements. When matrices have a high proportion of zero elements, the performance gains can be substantial.

2.4.4 Sparsity Level and Performance Impact

The sparsity level is the percentage of zero elements in a matrix, often denoted as a percentage. High sparsity levels (more zeros) allow greater optimization in storage and computation. However, as sparsity decreases (i.e., more non-zero elements), the benefits of sparse matrix representation diminish, and the overhead of managing a compressed format may even slow down computations compared to dense formats.

-High Sparsity: When sparsity is high, CSR and CSC formats significantly reduce memory usage and computational effort, as operations skip over zero elements.

-Low Sparsity: With lower sparsity, the overhead of tracking indices and values in sparse formats may outweigh the benefits, potentially making dense matrix representations more efficient.

In summary, the choice to use sparse matrix representations like CSR and CSC depends on the sparsity level, with greater performance benefits seen as sparsity increases.

3 Requirements and Methodology

3.1 Requirements

The specific requirements for the implementation and comparison of matrix multiplication methods in this study are as follows:

- Implementation of Optimized Algorithms: Develop at least two optimized versions of the matrix multiplication algorithm, such as block matrix multiplication and loop unrolling. Additionally, implement sparse matrix multiplication using CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats to handle matrices with high sparsity levels efficiently.
- Performance Comparison: Measure and compare the performance of the basic matrix multiplication approach against the optimized versions. This comparison includes execution time, memory usage, and CPU usage, providing insights into which optimizations are most effective for different matrix sizes and sparsity levels.
- Evaluation of Sparse Matrices Based on Sparsity Level: Implement and analyze sparse matrix multiplication by varying the sparsity level. This involves testing the performance at different sparsity levels (e.g., 0%, 10%, 20%, 50%, 70%, 90%) to understand how the number of non-zero elements affects computation time and memory usage.
- Testing with Increasingly Larger Matrices: Conduct tests with matrices of increasing sizes (e.g., 10x10, 100x100, 500x500, 1000x1000) to determine the maximum matrix size that each algorithm can handle efficiently. This will help assess the scalability of each optimization approach in terms of memory and computational overhead.

3.2 Testing Methodology

The performance tests were carried out using a benchmarking framework (JMH) in Java, designed to gather accurate metrics on execution time, memory usage, and CPU usage for each matrix multiplication approach. The testing methodology includes the following components:

- Evaluated Metrics: The primary metrics assessed in this benchmarking process are:
  - Execution Time: The average time taken to complete each matrix multiplication operation, measured in milliseconds.
  - Memory Usage: The memory consumed during matrix multiplication, calculated by comparing initial and final memory states.
- Comparative Procedures for Density and Sparsity: The tests included comparisons between dense and sparse matrices, examining how sparsity levels affect each algorithm's performance. For sparse matrices, both CSR and CSC representations were tested to evaluate which format performs better under varying conditions of sparsity.
- Hardware and Software: All tests were conducted on a specified execution environment to ensure consistent results. The benchmarking framework JMH was configured with parameters such as warm-up iterations, measurement iterations, and time units to standardize each run. Java classes, such as ThreadMXBean, were used to gather CPU usage data, while Runtime was employed to measure memory usage accurately.

This methodology allowed us to systematically assess the performance and efficiency of various matrix multiplication techniques across different scenarios, providing valuable insights into the strengths and limitations of each approach.

3.3 Test Environment

The benchmarks were conducted on a system with the following specifications:
- Processor: Apple M1 (8-core CPU)
- RAM: 8GB Unified Memory
- Operating System: macOS Sequoia 15.0
- Storage: 512GB SSD

The Java implementation was executed using IntelliJ IDEA on the host system.

4 Experiments and Results

4.1 Execution Time

4.1.1 Performance on Non-Sparse Matrices

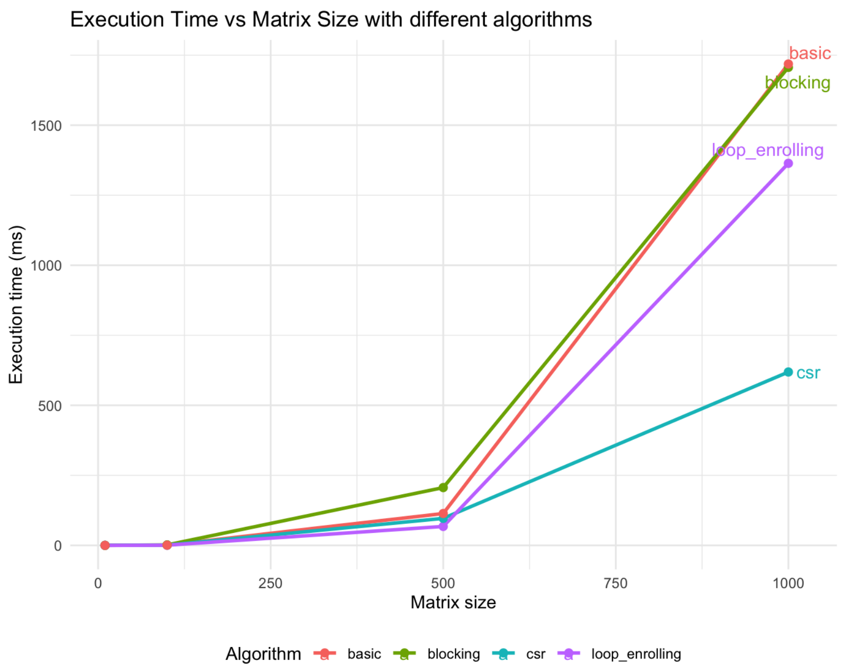


Figure 2: Execution time variation across different matrix sizes for each matrix multiplication algorithm on a dense matrix with a sparsity level of 0.1.

This graph shows the Execution Time vs. Matrix Size for different matrix multiplication algorithms on a dense matrix with a sparsity level of 0.1 (meaning only 10% of the elements are zero). The x-axis represents the Matrix Size (presumably the number of rows/columns in a square matrix), while the y-axis represents the Execution Time in milliseconds (ms). Each line corresponds to a different algorithm:

- Basic Algorithm (red): This line shows that the execution time increases steadily with matrix size. For larger matrices, the basic algorithm has high execution times, similar to other non-sparse optimized algorithms.
- Blocking Algorithm (green): This line follows a similar pattern to the basic algorithm but shows slightly better performance in some matrix sizes. However, it still has a steep increase in execution time for larger matrices.
- Loop Unrolling Algorithm (purple): The purple line indicates a improvement in performance over the basic and blocking algorithms. While the execution time still increases with matrix size, it is noticeably lower compared to the basic and blocking algorithms as the matrix size grows. This suggests that loop unrolling provides some efficiency gains for larger matrices
- CSR (Compressed Sparse Row) Algorithm (cyan): The CSR algorithm has a notably different performance profile, maintaining lower execution times than the other methods, especially for larger matrices (e.g., 1000x1000). This suggests that CSR, as a sparse matrix representation, is more efficient when handling larger matrices, likely due to its ability to skip zero elements, even with a sparsity level as low as 0.1.

4.1.2 Performance on Sparse Matrices

Both graphs illustrate the Execution Time vs. Matrix Size at various sparsity levels using two different sparse matrix representations: CSC (Compressed Sparse Column) in the first graph and CSR (Compressed Sparse Row) in the second graph.

- X-Axis (Matrix Size): Shows the size of the matrices used, increasing from smaller sizes up to 1000x1000.

- Y-Axis (Execution Time in ms): Indicates the time taken to complete the matrix multiplication for each matrix size.
- Lines for Each Sparsity Level: Each line represents a different sparsity level, ranging from 0 (dense matrix) to 0.9 (highly sparse matrix).

CSC

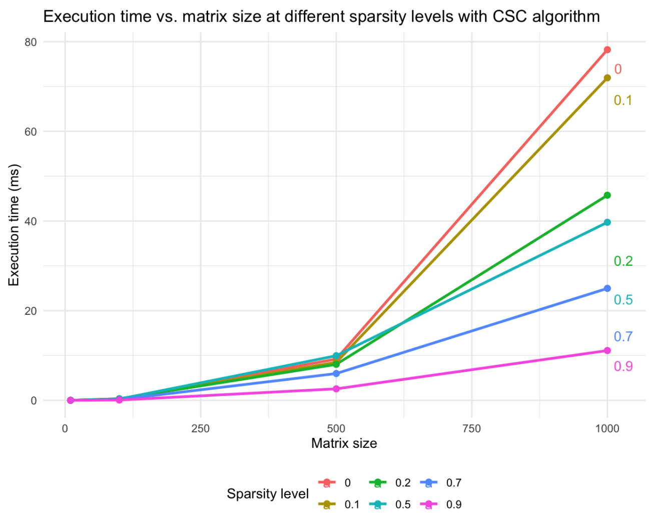


Figure 3: Execution time vs. matrix size at different sparsity levels using the CSC (Compressed Sparse Column) algorithm.

The graph shows that as the sparsity level increases (higher percentage of zero elements), the execution time decreases for all matrix sizes. The line representing a sparsity level of 0 (dense matrix) has the highest execution time, increasing sharply with larger matrix sizes. As the sparsity level increases to 0.9 (90% of elements are zero), the execution time is significantly lower, particularly for larger matrices.

CSR

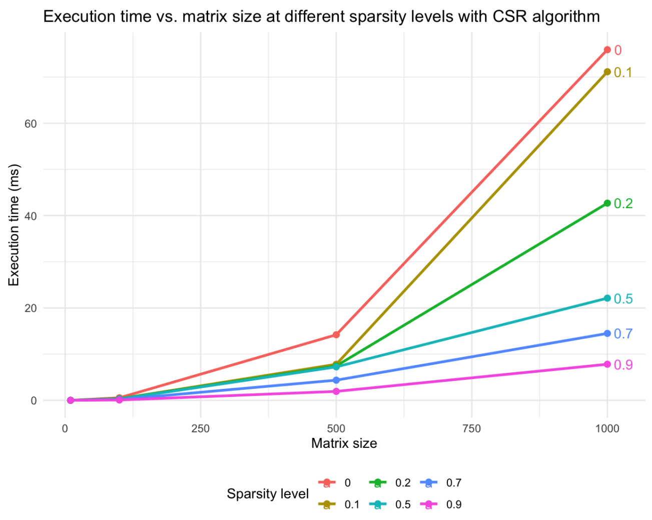


Figure 4: Execution time vs. matrix size at different sparsity levels using the CSR (Compressed Sparse Row) algorithm.

Similar to the CSC graph, the CSR graph shows that higher sparsity levels result in lower execution times for all matrix sizes. The line for sparsity level 0 (dense matrix) has the highest execution time, while higher sparsity levels like 0.7 and 0.9 show lower execution times. For very sparse matrices (e.g., sparsity level 0.9), the CSR format provides faster execution times, especially noticeable as matrix size increases.

4.2 Memory usage

4.2.1 Performance on Non-Sparse Matrices

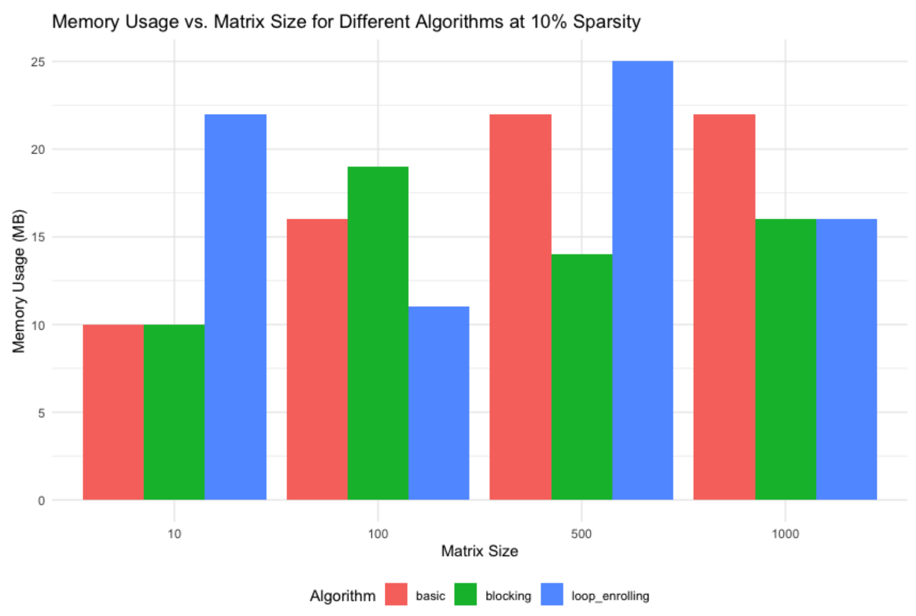


Figure 5: Memory usage vs. matrix size using different algorithms with dense matrices.

This bar graph shows Memory Usage vs. Matrix Size for three different matrix multiplication algorithms (Basic, Blocking, and Loop Unrolling) at a sparsity level of 10% (10% of elements are zero).

Each group of bars represents a different matrix size, with each color corresponding to a specific algorithm:

- Red (Basic Algorithm): The memory usage of the basic algorithm remains relatively stable across all matrix sizes, showing a slight increase as matrix size grows.
- Green (Blocking Algorithm): The blocking algorithm shows a similar pattern to the basic algorithm, with a small decrease in memory usage as the matrix size increases.
- Blue (Loop Unrolling Algorithm): The loop unrolling algorithm has varying memory usage depending on the matrix size, with a noticeable peak at the 500x500 matrix size, where memory usage is highest among the three algorithms.

The basic and blocking algorithms maintain relatively stable memory usage across different matrix sizes, suggesting efficient memory management. The blocking cache algorithm demonstrates similar memory usage to the basic algorithm. However, starting from a 500x500 matrix, it shows greater efficiency compared to the other methods. The loop unrolling algorithm shows a peak in memory usage at the 500x500 matrix size, indicating that this optimization might require additional memory under certain conditions or sizes.

This graph illustrates that while the basic and blocking algorithms have consistent memory usage across sizes, the loop unrolling algorithm may demand additional memory depending on the matrix size.

4.2.2 Performance of Sparse matrices

These two bar graphs show Memory Usage vs. Matrix Size for two different sparse matrix representations, CSC (Compressed Sparse Column) and CSR (Compressed Sparse Row), at two distinct sparsity levels: 10% and 90%.

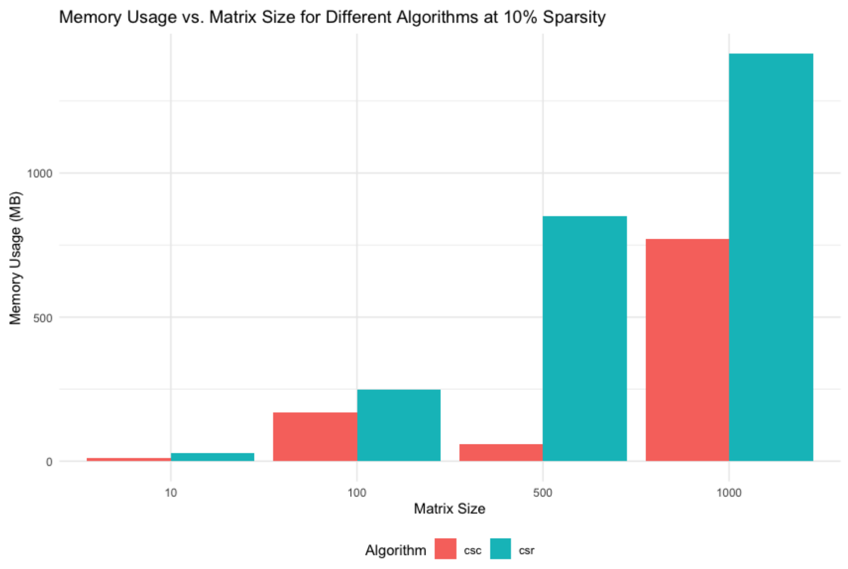


Figure 6: Memory usage vs. matrix size for CSC and CSR algorithms at 10% sparsity

Each bar represents the memory usage for either the CSC (red) or CSR (teal) algorithm at each matrix size.

Both CSC and CSR show increased memory usage as the matrix size grows. This is expected, as larger matrices require more storage.

For smaller matrix sizes (10 and 100), CSC and CSR have similar memory usage. For larger matrix sizes (500 and 1000), CSR generally requires more memory than CSC, which may be due to the overhead of handling sparse structures in the CSR format at a low sparsity level (10% zeros). CSR's structure adds more memory overhead for storing non-zero elements, particularly as the matrix size increases.

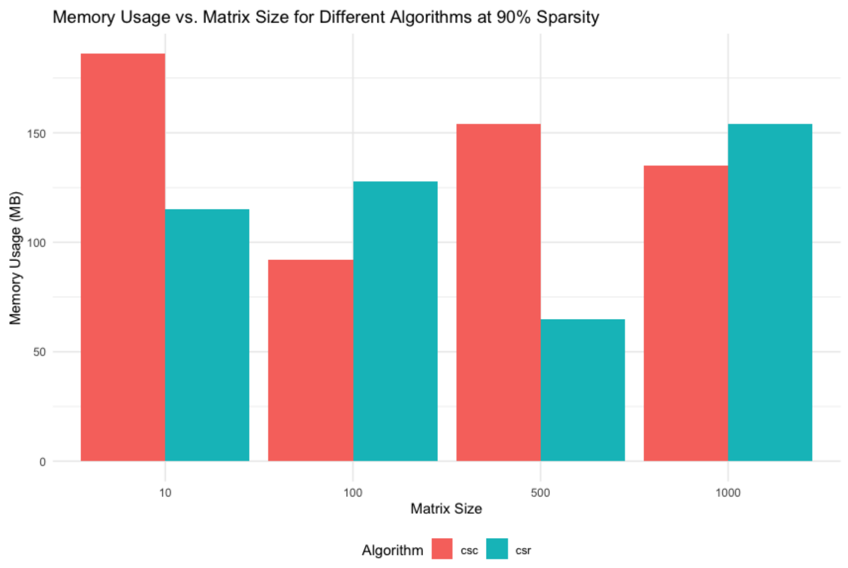


Figure 7: Memory usage vs. matrix size for CSC and CSR algorithms at 90% sparsity

In this graph, the sparsity level is much higher (90%), meaning 90% of the elements in the matrix are zero.

With 90% sparsity, CSR has noticeably lower memory usage for larger matrices, especially at sizes 500 and 1000. CSC shows higher memory usage than CSR at this sparsity level, particularly for smaller matrix sizes (10 and 500), indicating that CSC might not be as efficient at very high sparsity levels.

The CSR algorithm demonstrates significant memory efficiency at high sparsity levels, particularly with larger matrices, making it more advantageous for handling highly sparse data. In contrast, CSC's memory usage remains relatively high and consistent across different sparsity levels, indicating it may be less effective for very sparse matrices. While CSR offers clear memory savings at 90% sparsity, both algorithms show similar memory usage at lower sparsity (10%), though CSR tends to use slightly more memory for large matrices. Overall, CSR is better suited for highly sparse matrices, while CSC maintains a steady memory footprint regardless of sparsity.

4.3 Maximum Matrix Size Handled

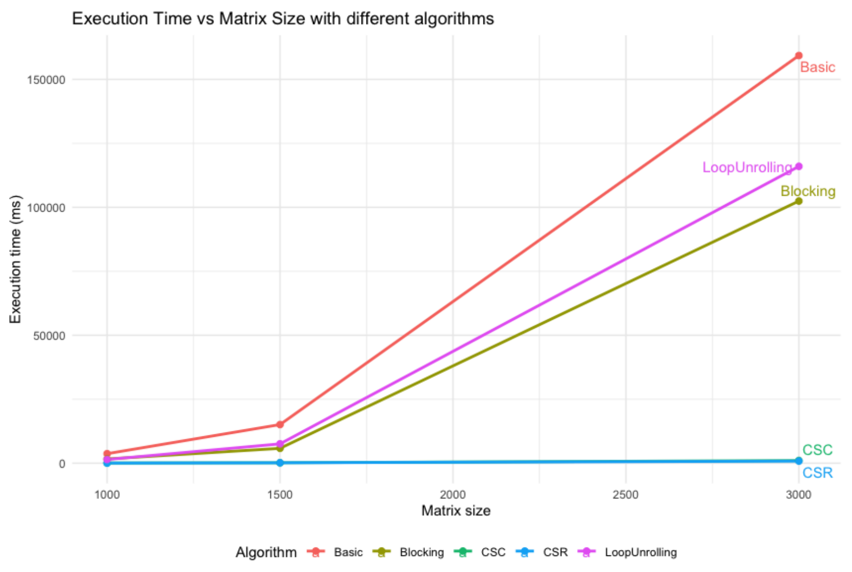


Figure 8: Execution time vs. matrix size for different matrix multiplication algorithms, showing scalability and performance limits

The graph provides insights into the maximum matrix size each algorithm can handle efficiently, highlighting significant differences in scalability among the methods tested.

- Basic Algorithm: The basic algorithm quickly reaches its limit in terms of efficiency as matrix size increases. For matrices around 3000x3000, the execution time becomes exceedingly high (over 150,000 ms), indicating that this method is impractical for very large matrices. Its steep growth in execution time shows that it cannot efficiently manage matrices beyond 1500x1500 without a substantial performance cost.

- Loop Unrolling and Blocking Algorithms: Both the loop unrolling and blocking algorithms perform somewhat better than the basic method, but they still face scalability challenges. At matrix sizes above 1500x1500, their execution times increase sharply, suggesting that these optimizations are insufficient to handle very large matrices (e.g., 3000x3000) efficiently. Although blocking offers marginal improvement due to better memory access patterns, both approaches reach their practical limit at around 2000x2000.

- CSC and CSR (Sparse Matrix Formats): The CSC and CSR algorithms, designed for sparse matrices, show exceptional scalability, with minimal increases in execution time even for matrices as large as 3000x3000. Unlike the other algorithms, CSC and CSR handle larger matrices efficiently due to their ability to skip computations involving zero elements. This allows them to manage very large matrices without a drastic increase in execution time, making them the optimal choice for large-scale matrix operations, especially when sparsity can be leveraged.

In general, the maximum practical matrix size that each algorithm can handle efficiently varies significantly:

Basic, Loop Unrolling, and Blocking: Efficient up to around 1000x1000 to 2000x2000, but they become impractical for larger sizes due to exponentially growing execution times. CSC and CSR: Efficiently handle matrix sizes up to 3000x3000 and beyond, maintaining low execution times due to the ability to optimize for sparsity. This analysis highlights that CSC and CSR are the only viable options for very large matrices (3000x3000 and larger), while the basic, loop unrolling, and blocking algorithms are best suited for smaller matrices.

5 Conclusion

5.1 Summary of Key Findings

This study demonstrates the effectiveness of various optimization techniques for matrix multiplication, including loop unrolling, cache optimization, and sparse matrix representations (CSR and CSC). Our results show that while traditional algorithms like the basic, blocking, and loop unrolling approaches can handle moderate matrix sizes efficiently, they face limitations when matrix dimensions grow beyond 1500x1500, resulting in substantial increases in execution time and memory usage. In contrast, the CSR and CSC formats show remarkable scalability, handling larger matrices (up to 3000x3000) with minimal increases in execution time, especially in cases of high sparsity (90%). These sparse matrix formats effectively reduce memory usage and computation time by skipping zero elements, proving to be highly advantageous for handling large, sparse datasets. Overall, this study highlights the importance of selecting appropriate matrix multiplication techniques based on matrix size and sparsity, with CSR and CSC being the preferred methods for very large and highly sparse matrices.

5.2 Future work

Future research could explore additional optimization strategies, such as parallelization and GPU acceleration, which may further improve the performance of matrix multiplication, particularly for very large matrices. Investigating hybrid approaches that dynamically select between dense and sparse representations based on the matrix's sparsity level could also yield more efficient results across varying sparsity levels. Additionally, testing these optimization methods on different hardware architectures and extending them to non-square matrices could provide further insights into their adaptability and scalability. Finally, exploring more advanced sparse matrix formats, such as Block CSR or CSR5, could offer further performance improvements for high-dimensional data applications.

A Link to GitHub repository

https://github.com/alejandroalemanaleman/OptimizedMatrixMultiplication.git