Optimized Matrix Multiplication Approaches and Sparse Matrices

Néstor Ortega Pérez

GitHub Repository (Click to Enter)

Abstract

Matrix multiplication, a fundamental operation in scientific and engineering applications, presents significant computational challenges, particularly in terms of execution time and memory efficiency for large datasets. The ability to optimize matrix multiplication for both dense and sparse matrices has important implications for fields that require processing large-scale data, such as physics, machine learning, and data analysis. This paper investigates and compares several optimized approaches to matrix multiplication, including Strassen's algorithm, Cache-Optimized techniques, and the Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) methods. Through experimental analysis across different matrix sizes and sparsity levels, we evaluate each algorithm's effectiveness in terms of execution time, memory usage, and scalability.This study provides a detailed comparison of the trade-offs associated with each method, offering a comprehensive guide for selecting the most efficient algorithm based on matrix characteristics and computational constraints. These findings underscore the importance of matching algorithm choice to matrix structure to optimize performance in large-scale applications.

1 Problem Statement

This research aims to optimize matrix multiplication through algorithmic and structural improvements and to evaluate their effectiveness with dense and sparse matrices. Specifically, the study addresses the following research questions:

1. How effective are CSC, CSR, Strassen and Cache Optimization representations in reducing computational and memory costs, especially at varying sparsity levels?

2. What scalability limitations exist for these techniques in terms of matrix size and density?

3. What performance bottlenecks or issues arise, and how can they be mitigated?

2 Equipment

The experiments were conducted on a MacBook Air (13 inches, 2017) with a 1.8 GHz dual-core Intel Core i5 processor, 8 GB of DDR3 RAM at 1600 MHz, and Intel HD Graphics 6000. The operating system is macOS High Sierra (version 10.13.6).

3 Experiments/Results

To evaluate the performance of the selected matrix multiplication algorithms, two experiments were conducted. These experiments aimed to compare execution time and memory usage across different algorithms and to test the scalability of a sparse algorithm with a large matrix.

3.1 Comparative Measurement of Execution Time and Memory Usage

In the first experiment, the execution time and memory usage of five algorithms—CSC, CSR, Strassen, Cache-Optimized, and Basic matrix multiplication—were measured and compared. Each algorithm was tested with matrices of varying sizes and sparsity levels to observe their performance under different conditions. The goal was to identify which algorithms offer the best balance of speed and efficiency, particularly in sparse vs. dense matrix scenarios. The metrics collected included the time taken to complete each multiplication and the peak memory consumption. These metrics were graphed to enable a more straightforward comparison of the computational

efficiency and scalability of each algorithm, allowing for better insight into their relative strengths and weaknesses.

3.2 Scalability Test with Sparse Algorithm on Large Matrices

The second experiment focused on testing the scalability of a sparse matrix multiplication algorithm when multiplying a large matrix by itself. A sparse algorithm (CSC) was chosen for this purpose due to its efficiency in handling matrices with a high percentage of zero elements. This experiment aimed to investigate the algorithm's ability to manage memory usage and maintain execution speed when applied to matrices with substantial dimensions, pushing the limits of matrix size.

4 Results and Discussion

4.1 First Experiment: Execution Time Comparison for CSC, CSR, Strassen, and CacheOptimized

- **Matrix 10x10:** All algorithms show high execution times in dense matrices (low sparsity levels), but execution time decreases considerably as sparsity increases. CSR and CSC offer slightly lower execution times at low sparsity levels, while CacheOptimized and Strassen do not show a significant advantage for this matrix size. Overall, for 10x10 matrices, the benefits of cache optimization and Strassen are minimal due to the small matrix size

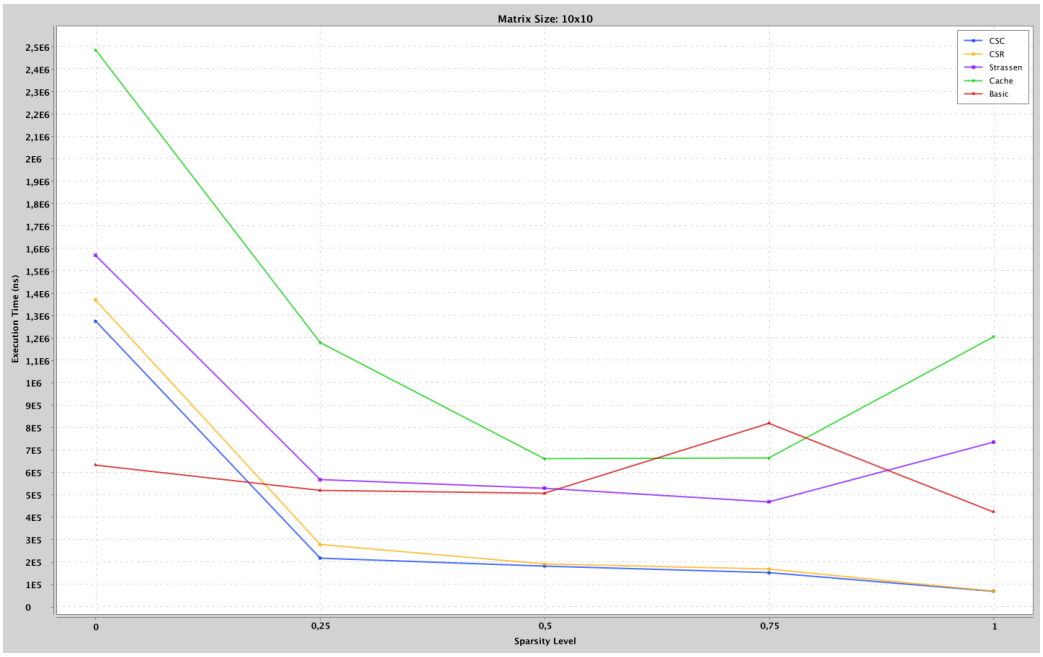

Figure 1: Execution Time 10*10

- **Matrix 50x50:** CSC and CSR prove more efficient than CacheOptimized and Strassen at low sparsity levels, maintaining lower execution times in dense matrices. However, as sparsity increases (more zeros in the matrix), all algorithms reduce their execution times. CacheOptimized shows a slight improvement at high sparsity levels, indicating it starts to benefit from blocks of non-zero elements in sparse matrices, though it still does not clearly outperform CSC and CSR.

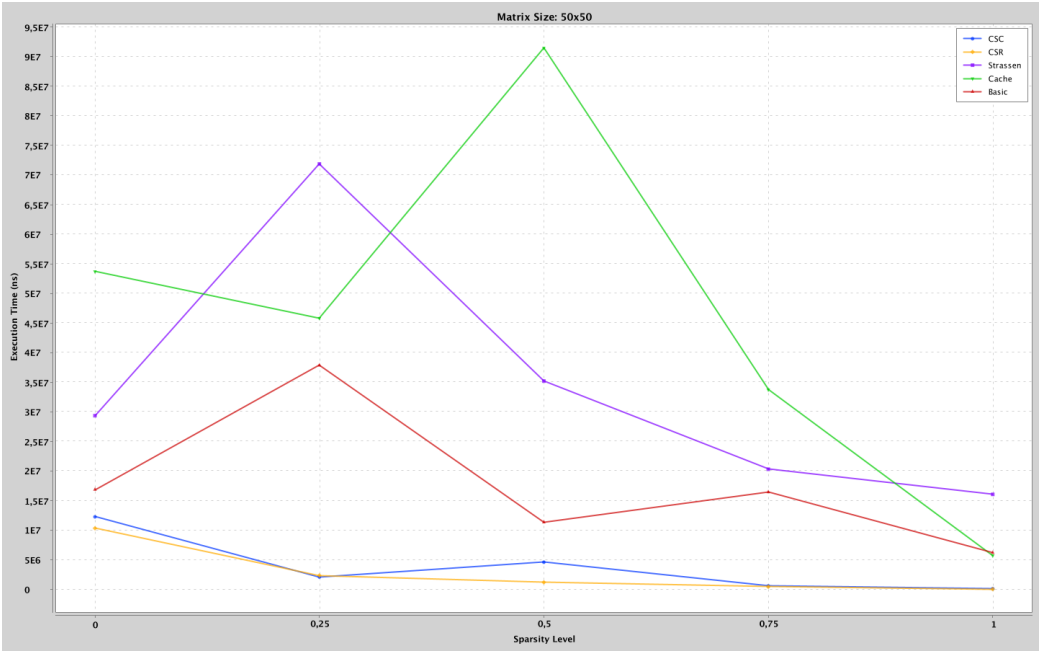

Figure 2: Execution Time 50*50

- **Matrix 100x100:** CSC and CSR continue to be the most efficient for dense matrices (low sparsity levels). However, at high sparsity levels, CacheOptimized significantly reduces its execution time, demonstrating its ability to leverage the sparse structure through block processing. Strassen shows some advantage in dense matrices, though its execution time is not as low as CSC or CSR.

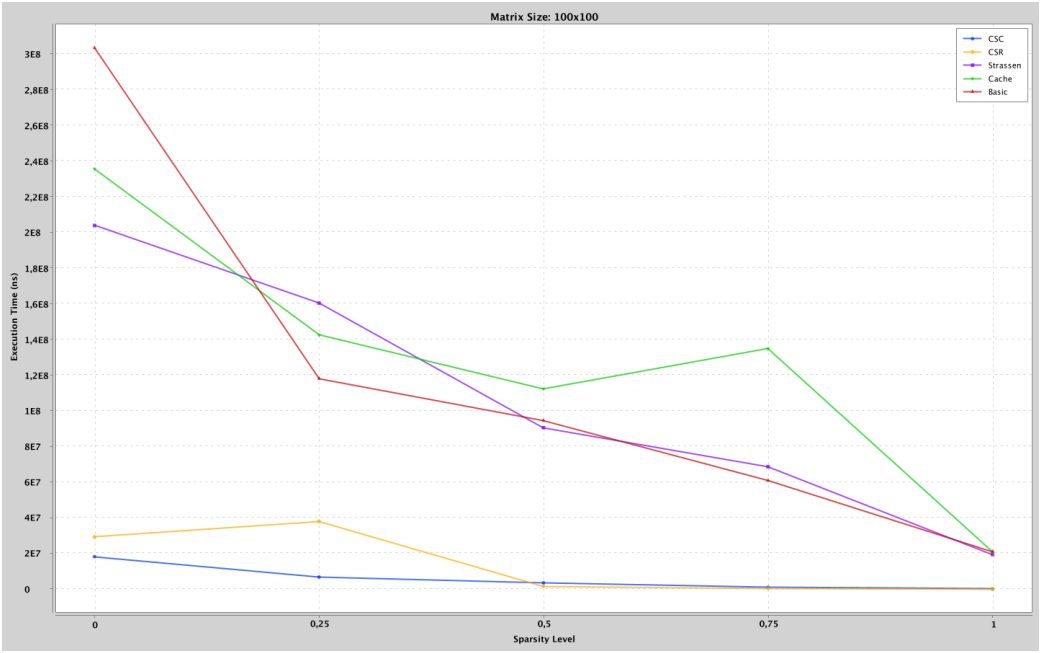

Figure 3: Execution Time 100*100

- **Matrix 500x500:**CSC and CSR remain efficient while Cache and Basic algorithms improve with higher sparsity levels. Finally we can see the advantages of the Strassen with low sparsity levels.

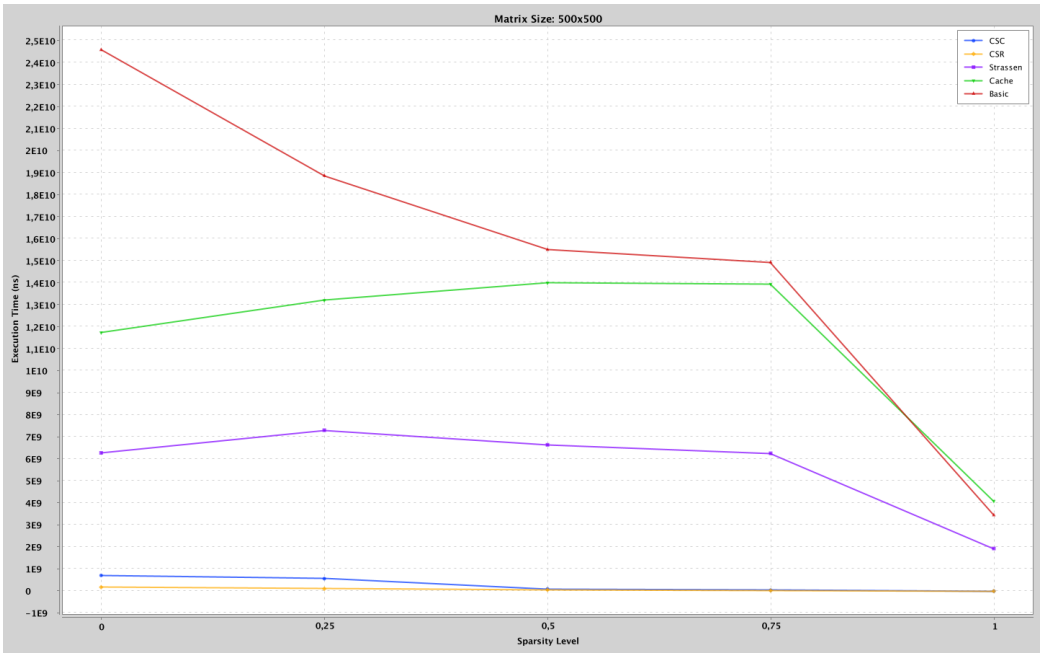

Figure 4: Execution Time 500*500

- **Matrix 1000x1000:** Similar results to the 500*500 matrix case but with higher execution time.

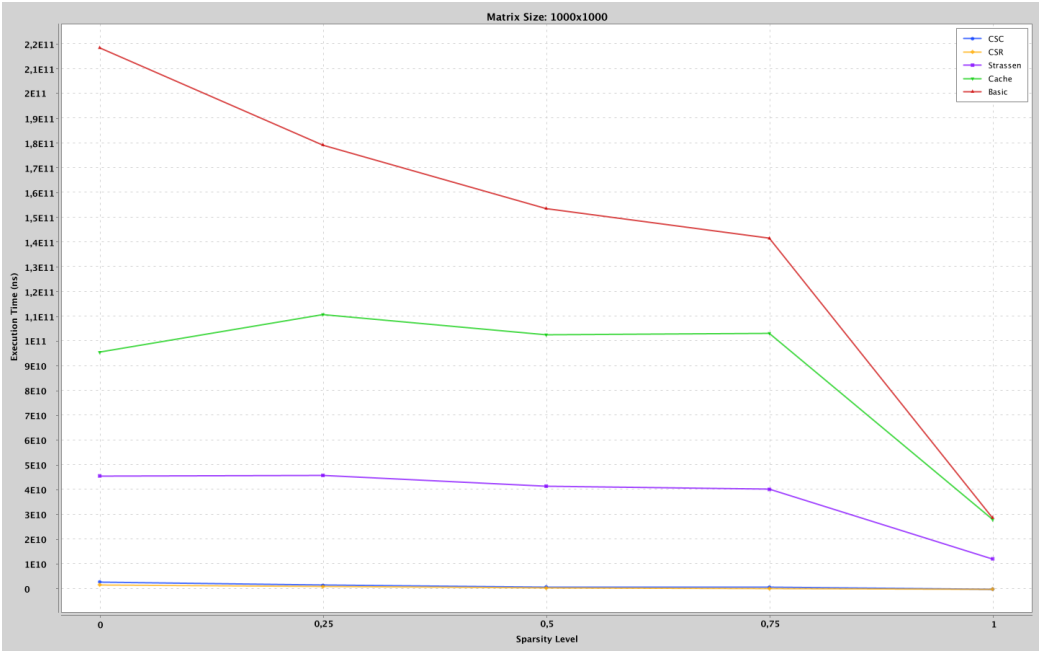

Figure 5: Execution Time 1000*1000

4.2 First Experiment: Memory Usage Comparison for CSC, CSR, Strassen, CacheOptimized, and Basic

Memory usage patterns were assessed across matrix sizes:

- **10x10:** the memory usage remains high for all algorithms at a low sparsity level (dense matrix). CSC and CSR use slightly less memory as they are optimized to handle sparse structures by focusing only on non-zero elements. In contrast, Strassen, CacheOptimized, and Basic show almost identical memory usage, which remains constant across sparsity levels. This indicates that for small matrices, memory optimization from sparsity is minimal as the overhead of storing matrix structure dominates.

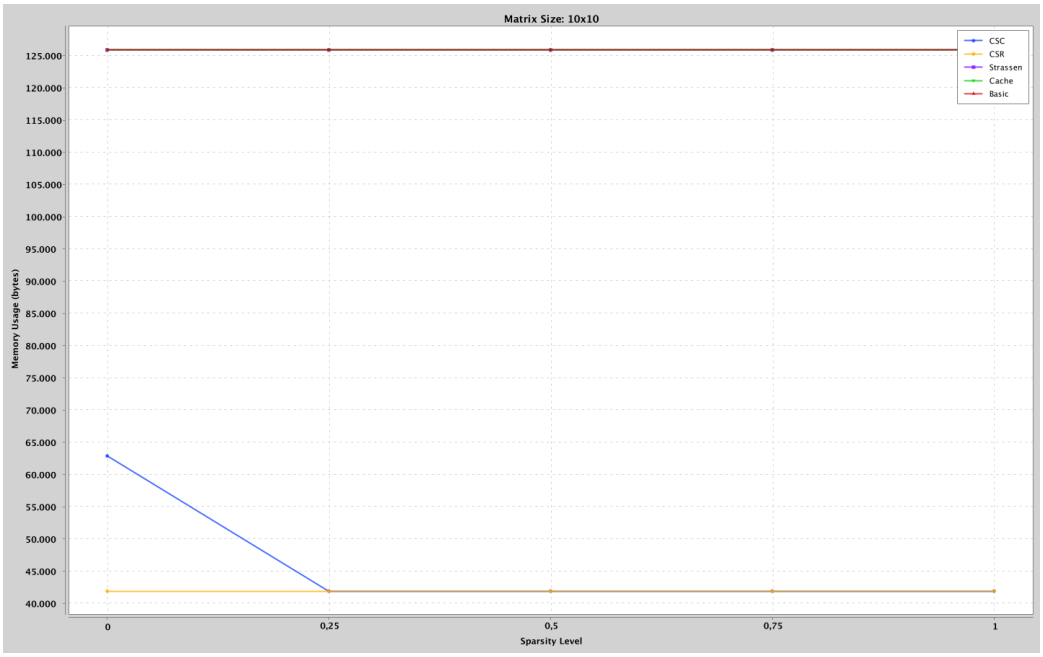

Figure 6: Memory Usage 10*10

- **50x50:** CSC and CSR demonstrate more efficient memory usage at higher sparsity levels, reflecting their capability to optimize storage by ignoring zero elements. Strassen, however, show significantly lower memory usage with dense matrices, and their consumption increases slightly as sparsity increases. Basic remains relatively consistent but does not benefit much from sparsity, highlighting its lack of optimization. CacheOptimized begins to show improvements at higher sparsity but does not outperform CSC and CSR, which remain the most memory-efficient for sparse matrices.

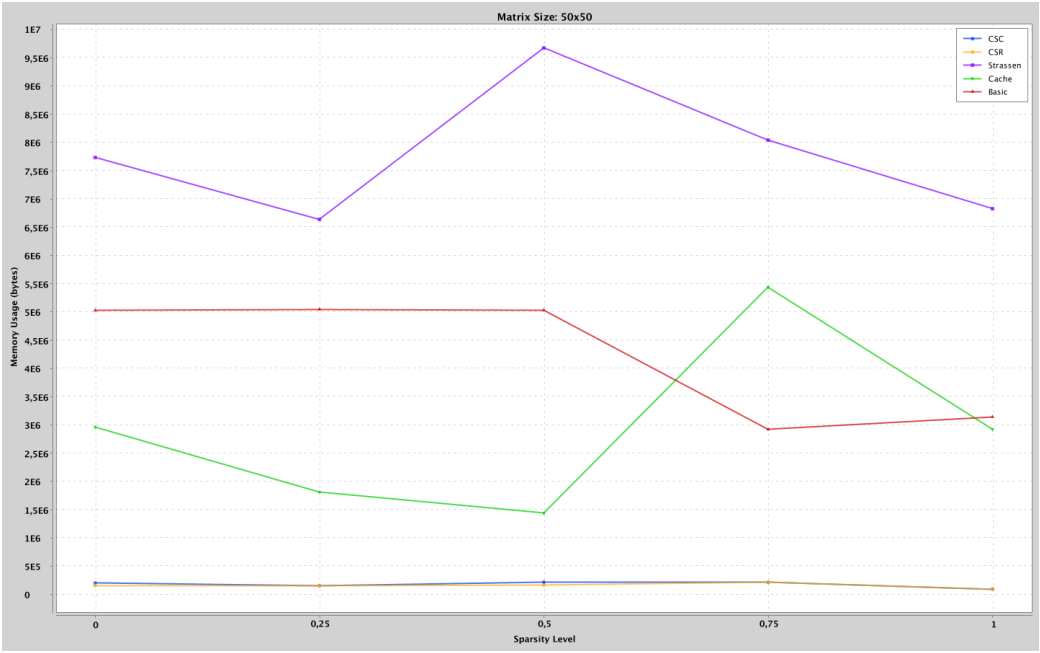

Figure 7: Memory Usage 50*50

- **Matrix 100x100:** The memory usage patterns become more distinct. CSC and CSR maintain efficient memory usage, particularly at high sparsity, where they focus solely on non-zero elements. Strassen's memory usage, although still high, shows gradual improvement with increased sparsity. CacheOptimized also starts to show benefits, reducing its memory footprint as sparsity increases but still remaining less efficient than CSC and CSR for sparse matrices. Basic continues to use significant memory regardless of sparsity level, underscoring its limitations in handling sparse structures effectively.

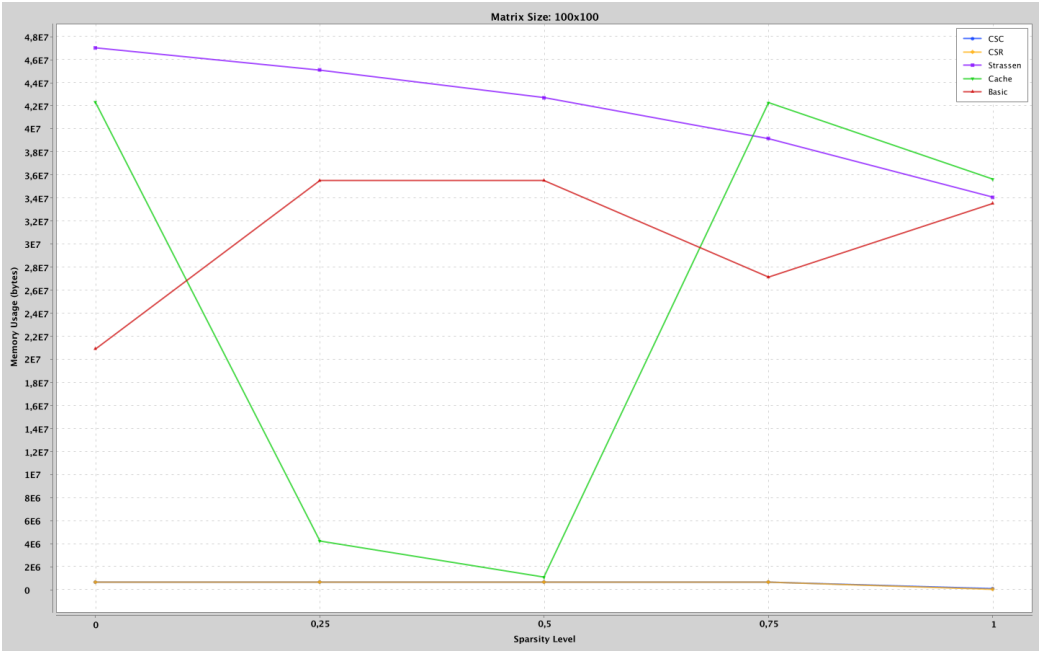

Figure 8: Memory Usage 100*100

- **500x500:** CSC and CSR demonstrate their advantages in memory efficiency, with both algorithms showing substantial reductions in memory usage at higher sparsity levels. Strassen and CacheOptimized exhibit a noticeable decrease in memory usage as sparsity increases. Basic, however, continues to have high memory consumption, showing little benefit from sparsity and reflecting its inefficiency for large matrices where optimization is crucial.

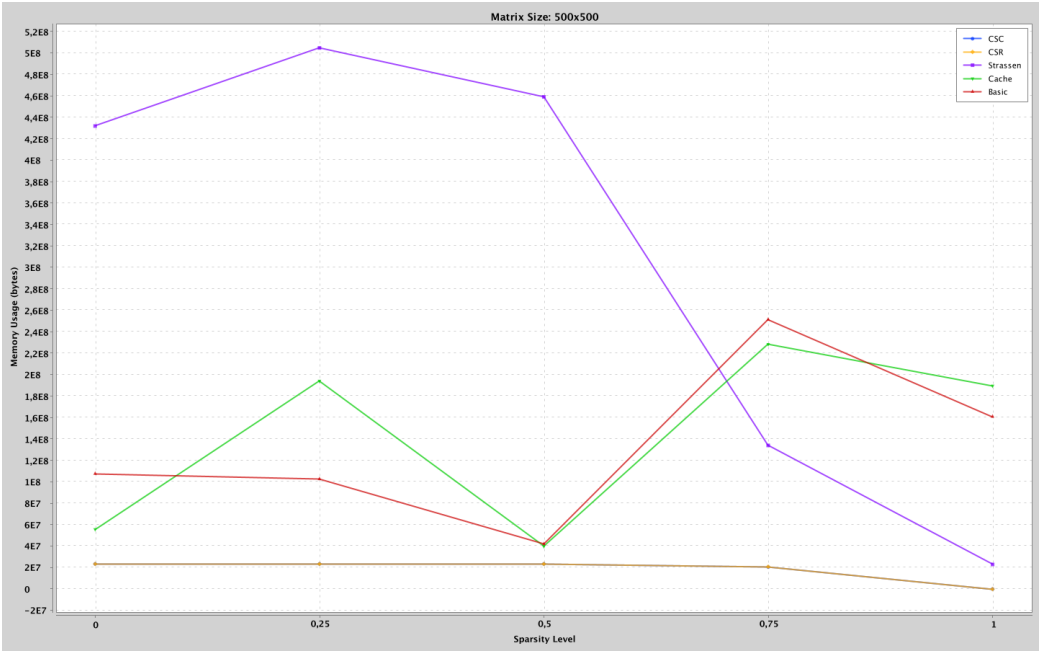

Figure 9: Memory Usage 500*500

- **1000x1000:** In the largest matrix size tested, 1000x1000, CSC and CSR continue to be the most memory-efficient for sparse matrices, demonstrating their ability to handle large and sparse datasets effectively. Strassen and CacheOptimized show significant reductions in memory usage as sparsity increases. Basic remains the least efficient in memory usage, with minimal improvement from sparsity, highlighting its limitations for large-scale, sparse matrix applications.

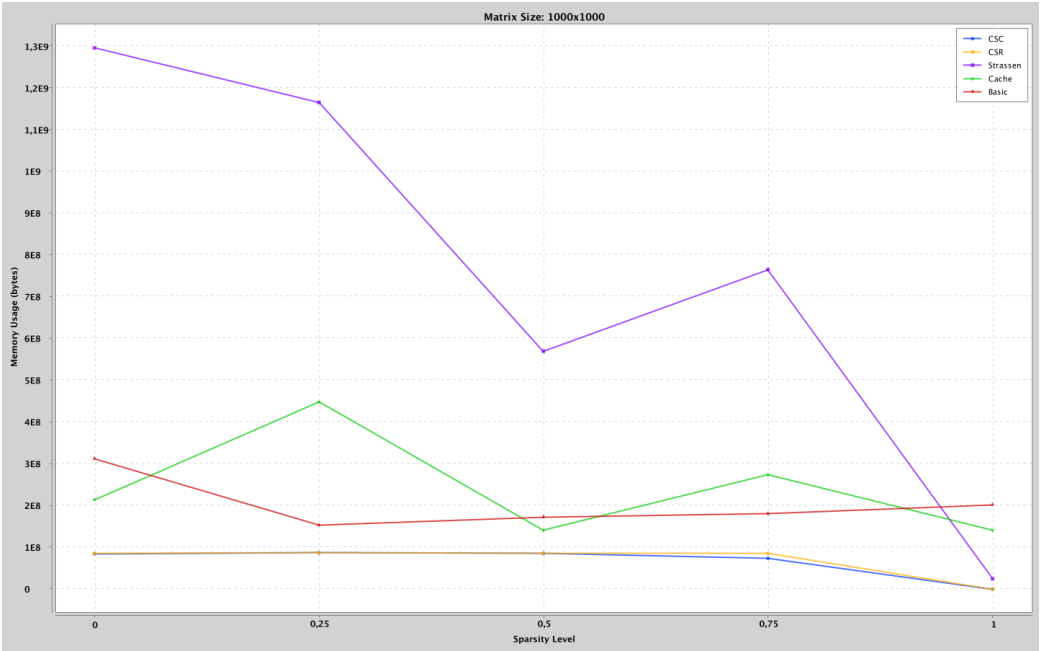

Figure 10: Memory Usage 1000*1000

5 Maximum Matrix Size

Testing across various matrix sizes indicated that 1000x1000 was the maximum matrix size that could be efficiently handled by the optimized algorithms. For matrices of size 2000x2000, certain algorithms required excessively long execution times, rendering them inefficient for large-scale applications. This increase in computational demands and memory usage highlighted the limitations of some approaches in managing very large matrices within a practical timeframe. Therefore, analysis was focused on matrix sizes up to 1000x1000, where performance remained efficient and processing requirements were manageable across the tested algorithms. This threshold provided a balanced assessment of algorithm efficiency without surpassing feasible computational limits.

6 General Conclusions

In summary, **CSC and CSR** provide robust performance across different matrix sizes and sparsity levels, particularly excelling in large, sparse matrices where other algorithms face performance and memory challenges. **CacheOptimized** is well-suited for larger, highly sparse matrices but less efficient for dense matrices due to limited block optimization opportunities. **Strassen** shows advantages in reducing multiplications for smaller, dense matrices but suffers from high overhead in sparse cases. Finally, the **Basic algorithm** proves inefficient in both time and memory, especially for large-scale matrices, highlighting the importance of optimized algorithms for practical big data and scientific computing applications.

7 Conclusions Based on Matrix Size

The performance of each algorithm varied with matrix size, revealing distinct patterns:

- **CSC and CSR:** These algorithms maintain excellent scalability across all matrix sizes, particularly in sparse matrices. Their memory usage remains efficient as the matrix size grows, due to their focus on storing only non-zero elements.
- **CacheOptimized:** This algorithm demonstrates manageable growth in memory usage as matrix size increases, particularly excelling in large, sparse matrices. Its cache optimization approach becomes more beneficial with increasing matrix size and sparsity, as it minimizes memory usage by focusing on blocks with non-zero elements. However, in dense matrices, CacheOptimized does not achieve the same level of efficiency as CSC and CSR, since there are fewer opportunities to avoid redundant memory storage.
- **Strassen:** With larger matrices, Strassen's memory usage grows significantly, particularly in dense matrices where it struggles to optimize memory effectively. Although it shows some improvement in memory usage with increased sparsity, the overhead of handling sub-blocks limits its efficiency for large, sparse matrices. Strassen may be suitable

for medium-sized dense matrices where its focus on reducing multiplications is beneficial, but it becomes less effective in large matrices, especially when sparsity is high.

- **Basic:** The Basic algorithm experiences a rapid increase in memory usage with matrix size, regardless of sparsity level. Its lack of optimization for sparse elements means that it continues to allocate memory for all elements, resulting in excessive memory consumption, especially for large matrices. As matrix size grows, the inefficiency of Basic becomes more apparent, making it unsuitable for applications where large-scale matrices and memory efficiency are required.

7.1 Scalability Test with Williams/mc2depi Matrix

In this second experiment, we aimed to evaluate the scalability and performance of the Compressed Sparse Column (CSC) algorithm by performing a matrix multiplication of a large sparse matrix by itself.

The CSC algorithm completed the multiplication in **222.4789 seconds**. The matrix's sparse structure, with only 2,100,225 non-zero entries out of a possible 276,446,025 (approximately 0.76% density), made it a suitable candidate for sparse matrix optimizations.

7.1.1 Results and Discussion

The CSC algorithm proved efficient for this operation due to its ability to skip zero entries, reducing unnecessary computations and memory usage significantly. Given the matrix's large size (over 525,000 rows and columns), using dense matrix multiplication algorithms would have resulted in extremely high memory consumption and prolonged computation times, rendering the operation infeasible on standard hardware. In contrast, CSC's sparse format allowed the algorithm to focus only on non-zero elements, leveraging the matrix's sparsity to minimize resource use.

8 Future Work

Future work could explore optimizing Strassen for sparse matrices, enhancing CacheOptimized for dense matrices. Additionally, exploring hybrid and

parallelized versions of CSC, CSR, and CacheOptimized could yield further performance gains.