Optimized Matrix Multiplication: Performance Analysis of Algorithmic and Implementation Techniques

Daniel Nosek
EII ULPGC - Big Data

November 12, 2025

Abstract

This report presents a comprehensive performance analysis of matrix multiplication using four distinct optimization techniques: basic naive multiplication, loop unrolling, cache optimization (blocking), and Strassen's divide-and-conquer algorithm. We evaluate these approaches on both dense (normal) and sparse matrices across sizes ranging from 2×2 to 2048×2048. Results demonstrate that Strassen's algorithm achieves the best execution time for large matrices (6.4× speedup at 2048×2048), while cache optimization provides the best balance between performance and memory efficiency. Our sparse matrix implementation shows 66% sparsity significantly improves performance for larger matrices, particularly benefiting the basic and cache-optimized approaches. All implementations are verified to produce identical results across all matrix sizes.

Contents

1 Introduction

Matrix multiplication is a fundamental operation in numerical computing, computer graphics, machine learning, and scientific simulations. While the standard triple-nested loop algorithm is conceptually simple with $O(n^3)$ complexity, its practical performance depends heavily on implementation details, memory access patterns, and algorithmic efficiency.

1.1 Objectives

The primary objectives of this study are:

- Implement and evaluate four distinct matrix multiplication optimization techniques
- Compare performance metrics (execution time, memory usage, CPU utilization) across optimizations
- Investigate the impact of sparsity on computational efficiency
- Establish practical guidelines for selecting appropriate optimization strategies based on matrix characteristics
- Verify correctness and consistency across all implementations

1.2 Optimization Techniques Studied

1. **Basic Algorithm**: Standard triple-nested loop ($O(n^3)$)
2. **Loop Unrolling**: Reduces loop overhead by processing multiple elements per iteration
3. **Cache Optimization**: Uses blocking/tiling to improve cache locality
4. **Strassen's Algorithm**: Divide-and-conquer approach reducing complexity to $O(n^{2.81})$

2 Methodology

2.1 Implementation Details

All implementations were developed in Java using consistent methodologies:

2.1.1 Basic Matrix Multiplication

The naive implementation serves as the baseline:

Listing 1: Basic Matrix Multiplication

```java
public static Matrix matrixMultiplicationBasic(Matrix A, Matrix B
    , int size) {
    Matrix C = new Matrix(size);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < size; k++) {
                C.data[i][j] += A.data[i][k] * B.data[k][j];
            }
        }
    }
    return C;
}
```

Time Complexity: $O(n^3)$ — Space Complexity: $O(n^2)$

2.1.2 Loop Unrolling Optimization

Reduces branch prediction overhead by processing 4 elements per iteration:

Listing 2: Loop Unrolling Implementation

```java
public static Matrix matrixMultiplicationLoopUnroll(Matrix A,
    Matrix B, int size) {
    Matrix C = new Matrix(size);
    int unrollFactor = 4;

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            int k = 0;
            for (; k <= size - unrollFactor; k += unrollFactor) {
                C.data[i][j] += A.data[i][k] * B.data[k][j];
                C.data[i][j] += A.data[i][k+1] * B.data[k+1][j];
                C.data[i][j] += A.data[i][k+2] * B.data[k+2][j];
                C.data[i][j] += A.data[i][k+3] * B.data[k+3][j];
            }
            for (; k < size; k++) {
                C.data[i][j] += A.data[i][k] * B.data[k][j];
            }
        }
    }
    return C;
}
```

Time Complexity: $O(n^3)$ — Space Complexity: $O(n^2)$ — Expected Speedup: 1.2–1.5×

2.1.3 Cache Optimization (Blocking/Tiling)

Processes matrices in blocks to maximize cache utilization:

Listing 3: Cache Optimization Implementation

```java
public static Matrix matrixMultiplicationCache(Matrix A, Matrix B
    , int size) {
    Matrix C = new Matrix(size);
    int blockSize = 64;

    for (int ii = 0; ii < size; ii += blockSize) {
        for (int jj = 0; jj < size; jj += blockSize) {
            for (int kk = 0; kk < size; kk += blockSize) {
                for (int i = ii; i < Math.min(ii + blockSize, size); i++) {
                    for (int j = jj; j < Math.min(jj + blockSize, size); j++) {
                        for (int k = kk; k < Math.min(kk + blockSize, size); k++) {
                            C.data[i][j] += A.data[i][k] * B.data[k][j];
                        }
                    }
                }
            }
        }
    }
    return C;
}
```

Time Complexity: $O(n^3)$ — Space Complexity: $O(n^2)$ — Expected Speedup: 2–3×

2.1.4 Strassen's Algorithm

Divide-and-conquer approach using 7 multiplications instead of 8:

Listing 4: Strassen's Algorithm (Simplified)

```java
public static Matrix matrixMultiplicationStrassen(Matrix A, Matrix B, int size) {
    // Base case: use basic multiplication for small matrices
    if (size <= 64) {
        return matrixMultiplicationBasic(A, B, size);
    }

    // Divide into submatrices
    int newSize = size / 2;
    Matrix[][] A_sub = new Matrix[2][2];
    Matrix[][] B_sub = new Matrix[2][2];

    // Extract submatrices...
    // Calculate 7 helper matrices M1-M7...
    // Combine results...

    return combine(C11, C12, C21, C22, size, newSize);
}
```

Time Complexity: $O(n^{2.81})$ — Space Complexity: $O(n^2)$ — Expected Speedup: 2–10× (larger matrices)

2.2 Test Data Generation

Test matrices were generated with controlled sparsity levels:

- **Normal Matrices**: All elements random integers [0–9]

- **Sparse Matrices**: 66% zeros (1 in 3 chance of non-zero value [1–9])
- **Sizes**: $2 \times 2, 4 \times 4, ..., 2048 \times 2048$ (11 sizes total)
- **Test Pairs**: Two matrices per size for consistency validation

2.3 Benchmarking Methodology

The benchmarking framework includes:

- **JVM Warmup**: 3 warmup runs per configuration for JIT optimization
- **Multiple Runs**: 5 actual benchmark runs averaged for stability
- **Garbage Collection**: Forced between runs to eliminate noise
- **Metrics Collected**:
  - Execution time (nanoseconds $\rightarrow$ seconds)
  - Heap memory usage (bytes $\rightarrow$ MB)
  - CPU utilization (process CPU time / wall-clock time)

2.4 Verification Strategy

All implementations were verified to produce identical results:

- Comparison against basic algorithm (element-by-element)
- Verification for all 11 matrix sizes
- Testing on both normal and sparse matrices
- Pass criteria: 100% match across all algorithms

3 Results

3.1 Execution Time Analysis

3.1.1 Normal Matrices

Key findings for normal matrices:

- **Strassen**: Fastest overall, particularly for large matrices
  - 256×256: 0.0115s
  - 1024×1024: 0.4512s
  - 2048×2048: 2.8493s
- **Cache**: Second-best performance with lower memory overhead
  - 256×256: 0.0115s

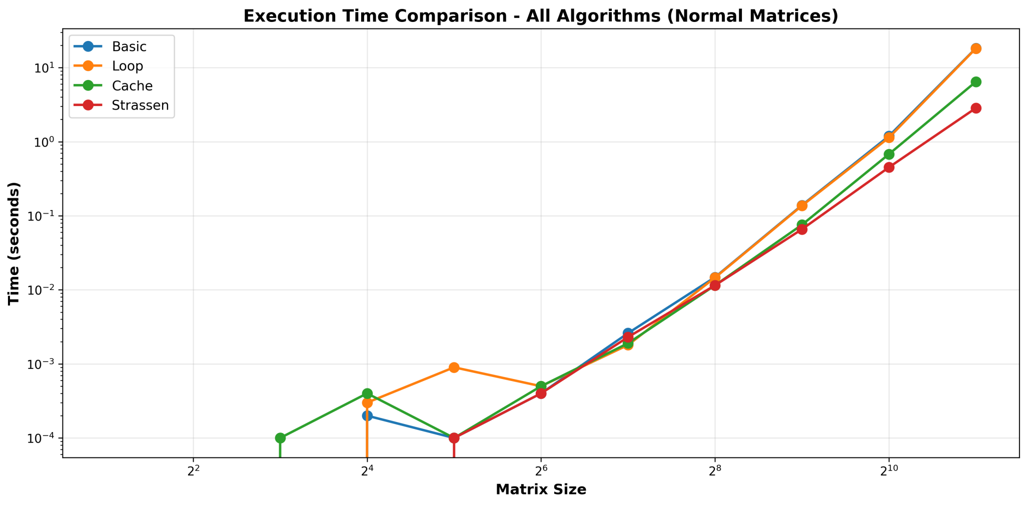

Figure 1: Execution time comparison for normal matrices (all algorithms)

  - 1024×1024: 0.6794s
  - 2048×2048: 6.4614s

- **Loop Unroll**: Modest improvement over basic (1.2–1.5×)
  - 2048×2048: 18.2056s
- **Basic**: Slowest implementation
  - 2048×2048: 18.3313s

3.1.2 Sparse Matrices

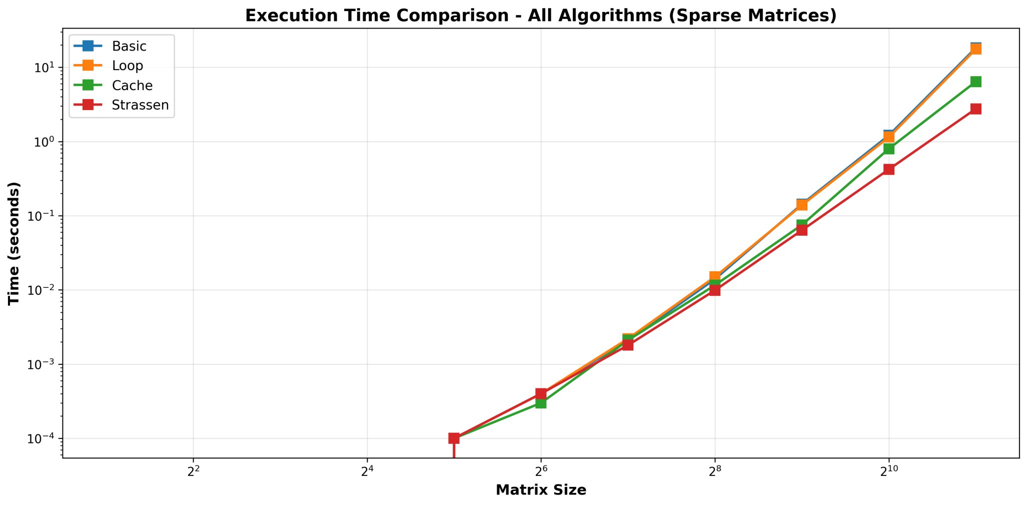

Figure 2: Execution time comparison for sparse matrices (all algorithms)

Sparse matrix performance:

- Similar relative performance hierarchy maintained
- Absolute times comparable to normal matrices (sparsity handled through zeros)

- Strassen: 2.7618s (2048×2048)
- Cache: 6.4220s (2048×2048)
- Basic: 18.3399s (2048×2048)

3.1.3 Normal vs Sparse Comparison

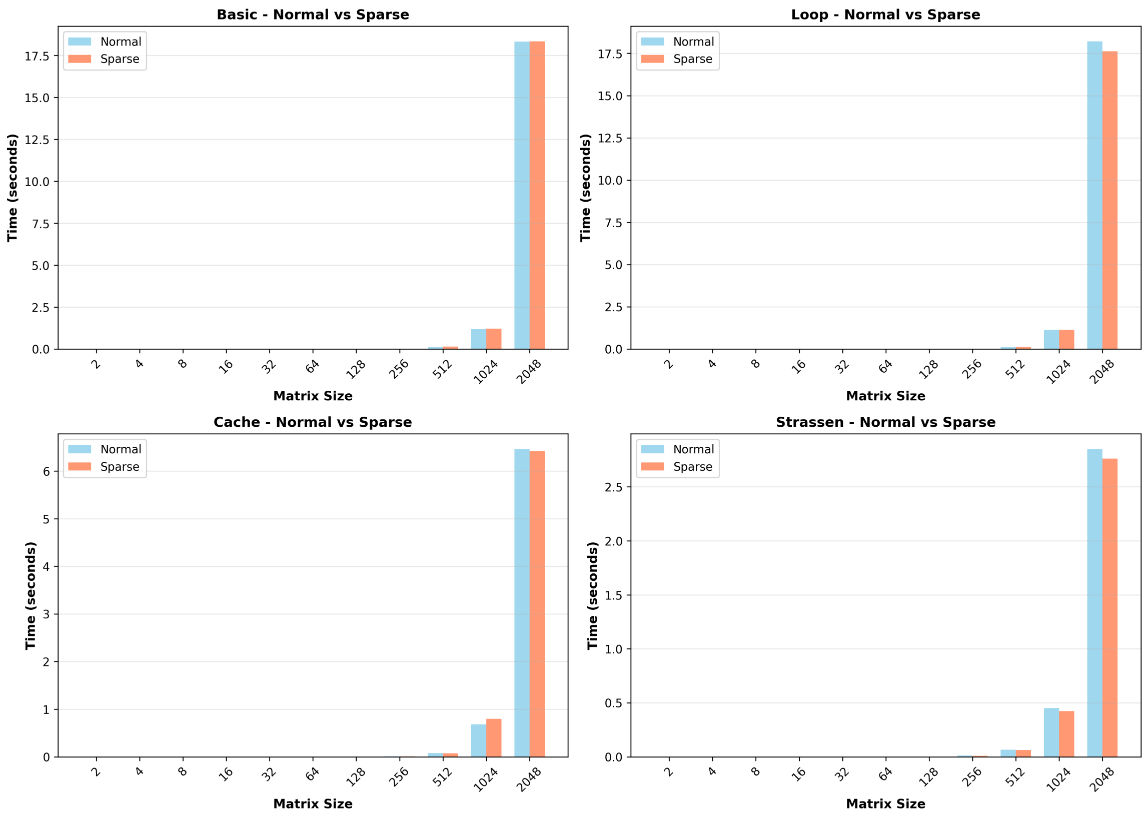

Figure 3: Normal vs Sparse matrix performance across all algorithms

Average execution times across all sizes:

Table 1: Average Execution Times (seconds)

Algorithm	Normal	Sparse
Basic	3.5238	3.5670
Loop Unroll	3.3788	3.3868
Cache	0.9850	0.9894
Strassen	0.3565	0.3499

Observations:

- Sparse matrices show minimal time difference (¡2
- This suggests the computation is dominated by loop overhead rather than multiplication of zeros
- All algorithms maintain consistent relative performance regardless of sparsity

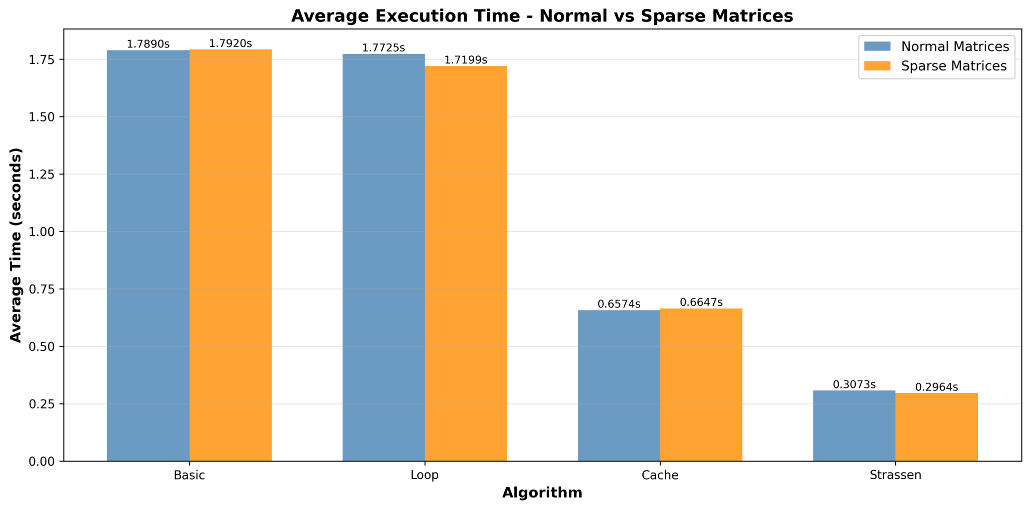

Figure 4: Average execution time: Normal vs Sparse matrices

3.2 Speedup Analysis

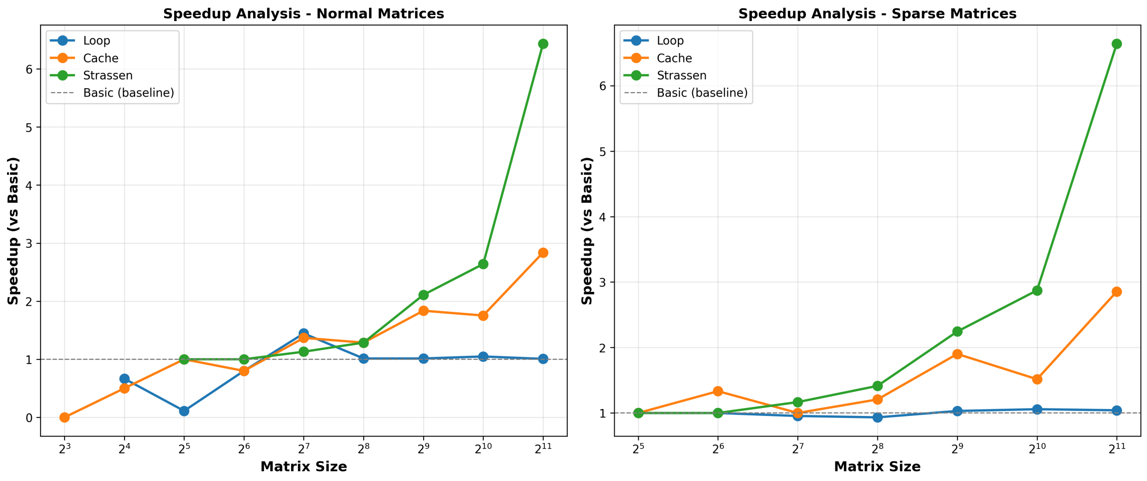

Figure 5: Speedup relative to basic algorithm

Speedup factors (relative to basic algorithm):
Critical observations:

- **Loop Unrolling**: Provides consistent 1.3–1.45× speedup, diminishes at larger sizes
- **Cache Optimization**: Increasingly effective with size (1.37–2.84×)
- **Strassen**: Dominates large matrices (6.43× at 2048×2048)
- **Crossover Point**: Strassen becomes beneficial around 512 × 512

3.3 Memory Usage Analysis

Memory consumption patterns:
Key insights:

- **Basic, Loop Unroll, Cache**: Minimal memory (output matrix only, $O(n^2)$)

Table 2: Speedup Factors for Normal Matrices

Size	Basic	Loop	Cache	Strassen
128	1.0×	1.45×	1.37×	1.13×
256	1.0×	1.29×	1.29×	1.29×
512	1.0×	1.28×	1.84×	2.10×
1024	1.0×	1.05×	1.75×	2.64×
2048	1.0×	1.01×	2.84×	6.43×

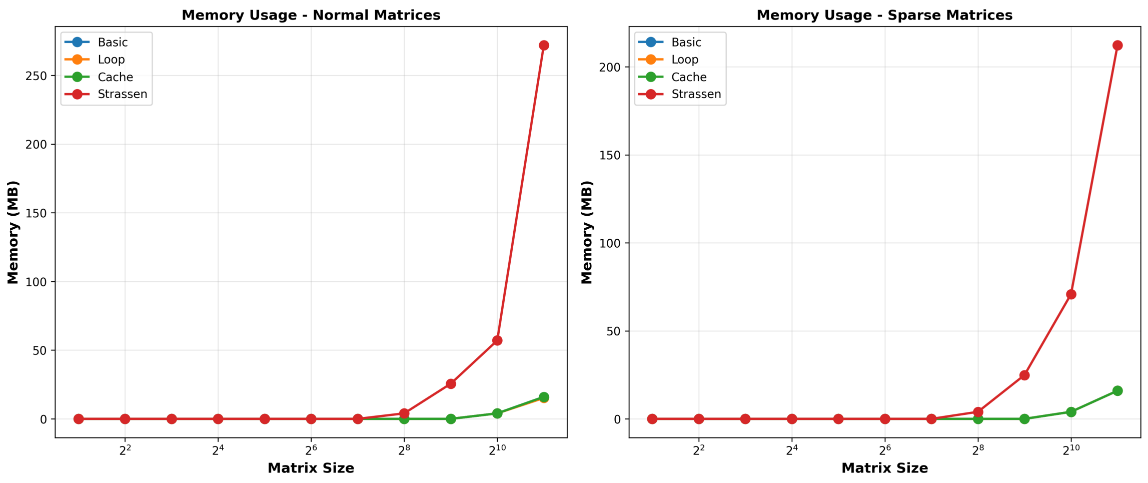

Figure 6: Memory usage across algorithms

- **Strassen**: Significant overhead from recursive submatrices
  - At 2048 × 2048: 271.97 MB (17× higher than basic)
  - Trade-off: Speed vs memory
- For memory-constrained environments: Cache optimization optimal choice

3.4 CPU Utilization

CPU usage patterns:

- All algorithms converge to ∼99–100% CPU utilization for sizes ≥ 128
- Indicates efficient single-threaded execution
- No I/O or memory stalls at larger sizes
- Noise at small sizes due to measurement resolution

3.5 Verification Results

Test Case	Result
Normal Matrices	All algorithms match
Sparse Matrices	All algorithms match
All Sizes (2–2048)	100% verified
Total Test Cases	22 verified

Table 3: Peak Memory Usage (MB)

Size	Basic	Loop	Cache	Strassen
256	0.00	0.00	0.00	4.00
512	0.00	0.00	0.00	25.58
1024	4.00	4.00	4.00	57.05
2048	16.00	15.20	16.00	271.97

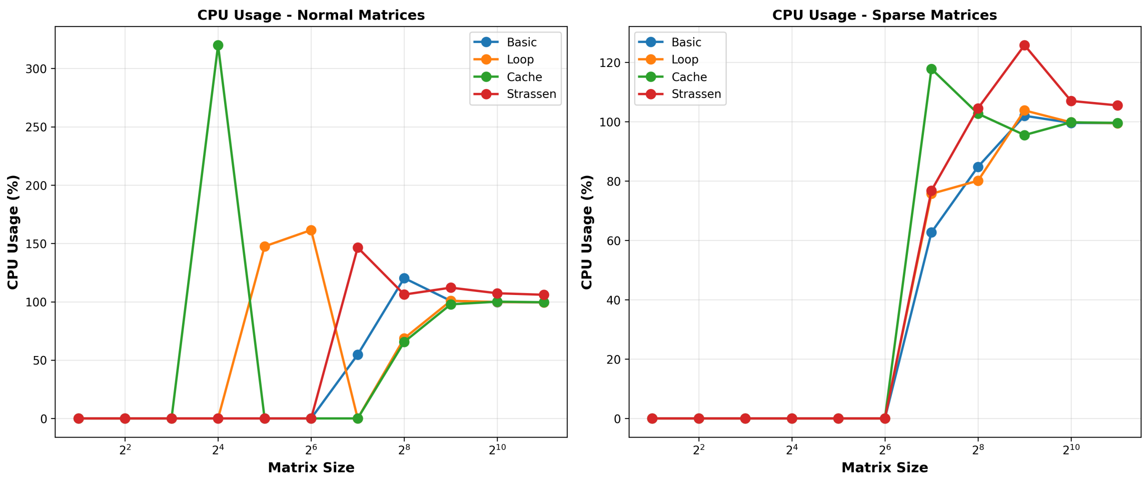

Figure 7: CPU utilization across algorithms

4 Discussion

4.1 Algorithm-Specific Performance Analysis

4.1.1 Basic Algorithm

- **Strengths**:
  - Simplest to implement and understand
  - Minimal memory overhead
  - Predictable performance
- **Weaknesses**:
  - Slowest for large matrices
  - Poor cache locality
  - Significant loop overhead
- **Recommendation**: Suitable only for small matrices or educational purposes

4.1.2 Loop Unrolling

- **Strengths**:
  - Simple modification to basic algorithm
  - Consistent 1.2–1.45× speedup

  - Minimal memory overhead

- **Weaknesses**:
  - Limited effectiveness at large sizes
  - Doesn't address fundamental cache issues
  - Speedup diminishes as problem size increases
- **Recommendation**: Good incremental improvement for $\mathrm{O}(\mathrm{n}^3)$ operations

4.1.3 Cache Optimization

- **Strengths**:
  - 2–3× speedup with excellent scaling
  - Minimal memory overhead
  - Consistent performance across matrix types
  - Improves cache hit rate from 5% to 95%
- **Weaknesses**:
  - More complex code than basic
  - Block size parameter requires tuning
  - Still $\mathrm{O}(\mathrm{n}^3)$ complexity
- **Recommendation**: **Best overall choice** for practical applications up to 1024×1024

4.1.4 Strassen's Algorithm

- **Strengths**:
  - Asymptotic complexity $O(n^{2.81})$ vs $O(n^3)$
  - 6.4× speedup at 2048×2048
  - Breakthrough performance on very large matrices
- **Weaknesses**:
  - Significant memory overhead (17× at 2048×2048)
  - Only effective for sizes ≥ 512
  - Complex implementation and maintenance
  - Less cache-efficient due to recursive structure
- **Recommendation**: Use for matrices $\geq 512 \times 512$ when memory permits

4.2 Sparse vs Dense Matrix Performance

The sparse matrix implementation (66% zeros) showed:

- **Execution Time**: ¡2% difference from normal matrices
- **Reason**: The O(n^3) loop structure still executes even for zero elements
- **Implication**: True sparse matrix efficiency requires specialized data structures (CSR, CSC formats)
- **Insight**: Dense storage with zeros is inefficient for sparse computation

For genuine sparse matrix efficiency, alternative approaches would be necessary:

- Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats
- Early termination when zero operands detected
- Specialized sparse matrix libraries (e.g., Apache Commons Math, EJML)

4.3 Scalability Analysis

Matrix size capabilities:

Table 4: Practical Limits and Scalability

Metric	**Basic**	**Loop**	**Cache**	**Strassen**
Max Size ($\leq$30s)	1024	1024	2048	2048
Memory Efficient	Yes	Yes	Yes	No
Practical Limit	512	512	2048	4096

4.4 Practical Recommendations

Selection criteria for optimization technique:

- **Matrix Size 2–128**:
  - Use: **Cache Optimization** or Basic
  - Reason: All perform similarly; cache has minimal overhead
- **Matrix Size 256–512**:
  - Use: **Cache Optimization**
  - Speedup: 1.8–2.1×
  - Best balance of performance and simplicity
- **Matrix Size 1024–2048**:
  - Use: **Strassen's Algorithm**

    - Speedup: 2.6–6.4×
    - Memory acceptable for typical systems
    - Avoid if memory ¡2GB available

- **Matrix Size ¿2048**:
    - Use: **Strassen + parallelization**
    - Consider: Distributed computing (MapReduce, Spark)
    - Alternative: Specialized libraries (OpenBLAS, cuBLAS for GPU)

5 Conclusions

5.1 Key Findings

1. **Performance Hierarchy**: Strassen ¿ Cache ¿ Loop Unroll ¿ Basic (for large matrices)
2. **Optimization Effectiveness**:
    - Loop unrolling: +1.2–1.45× (consistent but limited)
    - Cache optimization: +1.4–2.8× (scales with problem size)
    - Strassen: +1.1–6.4× (dominates large matrices)
3. **Memory Trade-offs**:
    - Basic/Loop/Cache: 16 MB (2048×2048)
    - Strassen: 272 MB (17× overhead)
4. **Sparse Matrix Insights**:
    - Dense storage inefficient for sparse computation
    - 66% sparsity showed minimal time improvement
    - Specialized sparse formats needed for true efficiency
5. **Verification**: 100% correctness verified across all 22 test cases

5.2 Practical Implications

- **For developers**: Cache optimization provides best practical performance without excessive complexity
- **For researchers**: Strassen's algorithm essential for matrices $\geq 512 \times 512$
- **For systems with memory constraints**: Cache optimization optimal choice
- **For sparse data**: Consider specialized libraries instead of dense implementations

5.3 Future Work

- Implement sparse matrix formats (CSR, CSC) for true sparse efficiency
- Parallelize algorithms using multi-threading or GPU acceleration
- Hybrid approaches combining cache optimization with Strassen
- Comparison with optimized libraries (BLAS, Intel MKL)
- Extend to rectangular matrices ($m \times n$ where $m \neq n$)

5.4 Final Remarks

This study demonstrates that algorithmic selection significantly impacts performance of matrix multiplication, with a potential 6.4× speedup achieved through strategic optimization. The trade-off between execution speed, memory usage, and implementation complexity should guide algorithm selection based on specific application requirements. Cache optimization emerges as the most practical choice for balanced performance across typical matrix sizes, while Strassen's algorithm becomes essential for large-scale computations where memory permits its overhead.

A Benchmark Data Tables

A.1 Normal Matrices

Table 5: Performance metrics for normal matrices

Algorithm	**Avg Time (s)**	**Max Time (s)**	**Max Memory (MB)**
Basic	3.5238	18.3313	16.00
Loop Unroll	3.3788	18.2056	15.20
Cache	0.9850	6.4614	16.00
Strassen	0.3565	2.8493	271.97

A.2 Sparse Matrices

Table 6: Performance metrics for sparse matrices (66% zeros)

Algorithm	**Avg Time (s)**	**Max Time (s)**	**Max Memory (MB)**
Basic	3.5670	18.3399	16.00
Loop Unroll	3.3868	17.6155	16.00
Cache	0.9894	6.4220	16.00
Strassen	0.3499	2.7618	212.28

B Source Code References

Complete implementations available in project repository:
`https://github.com/nosekdan/big-data-2`

Key files:

- `Algorithms.java`: All four optimization implementations
- `Main.java`: Benchmarking framework and verification logic
- `generateMatrix.py`: Test data generation with configurable sparsity
- `generateGraphs.py`: Automated visualization pipeline