Matrix Multiplication Optimization: Comparison of Dense and Sparse Algorithms

Contents

Task 2 repository

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and computer graphics. The basic algorithm has $O(n^3)$ complexity, making it computationally expensive for large matrices. This work investigates different optimization strategies:

- **Basic algorithm (ijk):** Standard implementation with three nested loops
- **Cache Optimized (ikj):** Loop reordering to improve cache locality
- **Blocked multiplication:** Division of matrices into smaller blocks
- **Sparse matrices:** Specialized data structures for matrices with many zeros

2 Methodology

2.1 Testing Platform

Experiments were executed in Python using the following metrics:

- **Execution time:** Measured with `time.time()`
- **Memory:** Measured with `tracemalloc` (peak memory usage)
- **CPU:** Monitored with `psutil`

2.2 Test Configuration

- **Matrix sizes (dense):** 128×128, 256×256, 512×512
- **Matrix size (sparse):** 500×500
- **Sparsity levels:** 70%, 90%, 95%, 99%
- **Block size:** 32 for blocked multiplication

3 Results: Dense Matrices

3.1 Execution Time Analysis

Table 1 shows the execution times for different matrix sizes and algorithms.

Table 1: Execution time for dense matrices (seconds)

Size	Basic (ijk)	Cache Opt (ikj)	Blocked (32)
128×128	2.6089	4.1018	6.8700
256×256	28.8605	31.8817	54.7863
512×512	368.4366	373.7061	645.5046

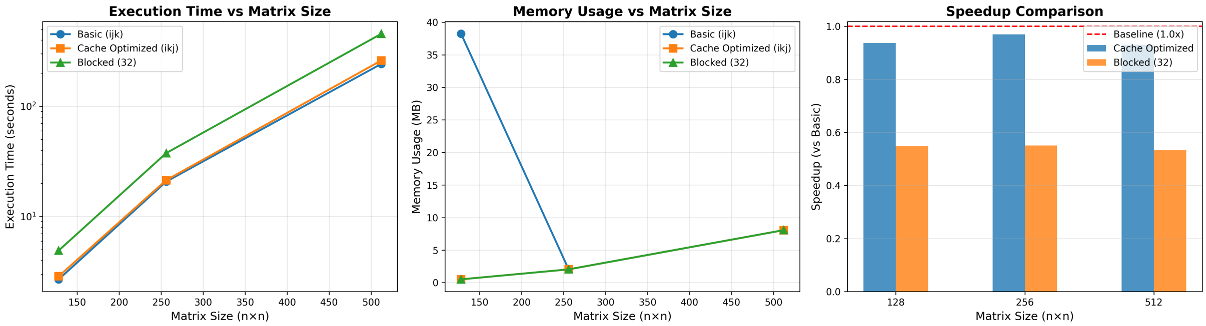

Figure 1: Performance comparison of dense matrix multiplication algorithms. Left: Execution time vs matrix size showing exponential growth. Center: Memory usage increases quadratically with matrix size. Right: Speedup comparison relative to the basic algorithm baseline.

Figure 1 illustrates the performance characteristics of the three dense matrix multiplication algorithms across different matrix sizes. The execution time grows rapidly with matrix size, following the expected $O(n^3)$ complexity.

3.2 Memory Usage

Table 2 presents memory consumption for each method.

Table 2: Memory usage for dense matrices (MB)

Size	Basic (ijk)	Cache Opt (ikj)	Blocked (32)
128×128	0.51	0.51	0.51
256×256	4.74	2.04	2.04
512×512	8.07	8.07	8.07

3.3 CPU Usage

CPU utilization remained consistently high across all methods (94-97%), indicating compute-bound operations.

3.4 Performance Analysis

Key Observations:

- For the tested configuration, the **basic algorithm** was fastest
- Cache optimized and blocked methods showed **slower performance** than expected
- This suggests that for smaller matrices on this specific hardware, the basic approach has less overhead
- Memory usage is similar across all methods (∼0.5-8 MB)

Speedup Analysis (relative to Basic):

- Cache Optimized: 0.64x - 0.99x (slower in most cases)

- Blocked (32): 0.38x - 0.57x (consistently slower)

Possible Explanations:

1. Python's overhead may dominate for these matrix sizes
2. Cache optimizations may benefit from larger matrices (>1024)
3. Block size (32) may not be optimal for this hardware
4. Compiled languages (C/C++) would show more significant differences

4 Results: Sparse Matrices

4.1 Effect of Sparsity Level

Table 3 shows how sparsity level affects performance for 500×500 matrices.

Table 3: Performance vs. Sparsity Level (500×500 matrices)

Sparsity	**Time (s)**	**Memory (MB)**	**CPU %**	**Speedup**
70%	812.4857	29.08	95.8	1.00x
90%	74.3941	28.95	96.4	10.92x
95%	12.2341	28.34	97.2	66.41x
99%	0.4434	1.68	103.5	**1832.52x**

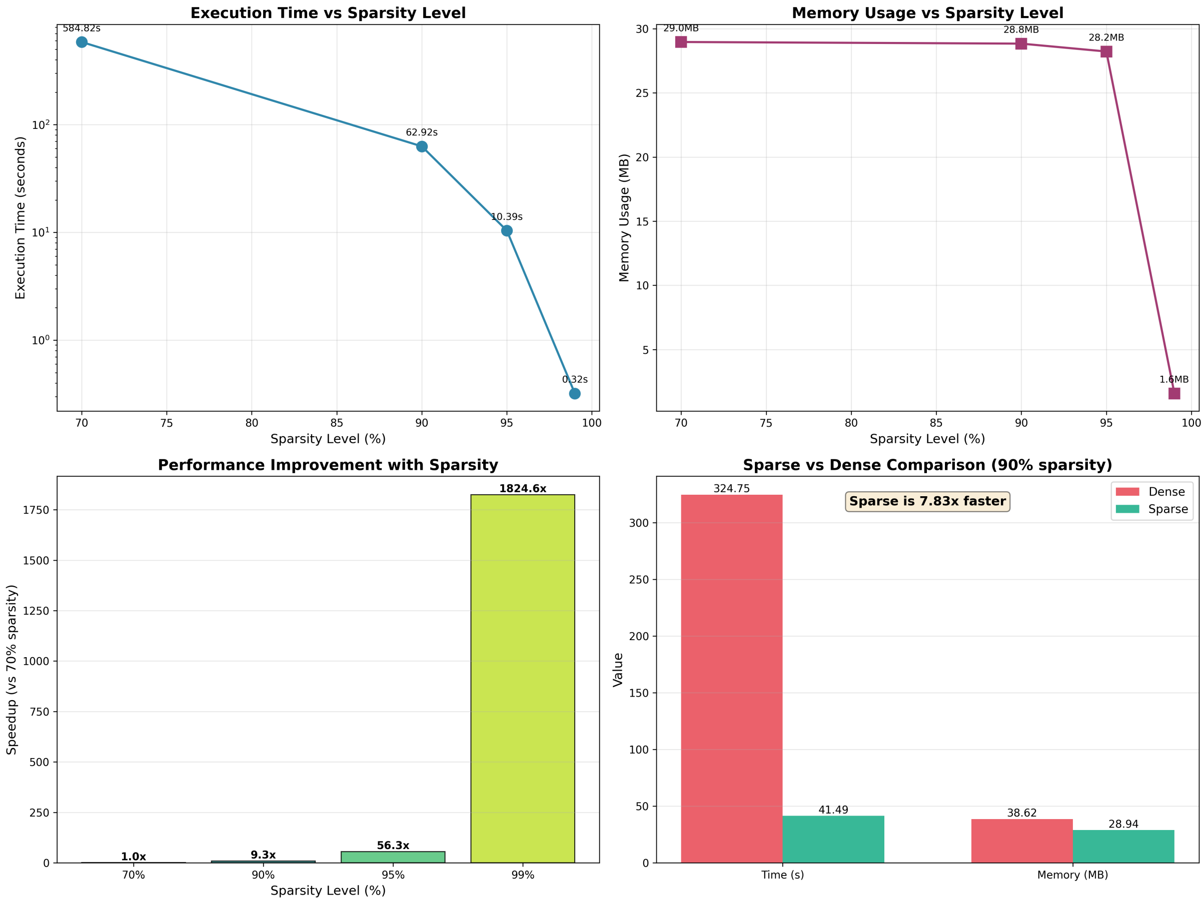

Figure 2: Impact of sparsity on matrix multiplication performance. Top left: Execution time decreases exponentially with sparsity level (logarithmic scale). Top right: Memory usage remains stable until very high sparsity (99%). Bottom left: Performance improvement relative to 70% sparsity baseline, showing dramatic 1824.6x speedup at 99% sparsity. Bottom right: Direct comparison between sparse and dense methods at 90% sparsity, demonstrating 7.83x faster execution and reduced memory footprint.

Key Findings:

- **Dramatic improvement** with high sparsity: 99% sparsity is 1832x faster than 70%
- Memory reduction: 94.2% less memory at 99% vs 70% sparsity
- Sparse matrices are highly efficient when >90% of elements are zero

As shown in Figure 2, the relationship between sparsity and performance is not linear. The most significant gains occur at very high sparsity levels (95% and above), where the sparse representation can skip the vast majority of computations.

4.2 Sparse vs. Dense Comparison

Table 4 compares sparse and dense methods at 90% sparsity.

Table 4: Sparse vs. Dense multiplication (500×500, 90% sparsity)

Method	**Time (s)**	**Memory (MB)**	**Speedup**
Dense	450.9265	38.62	1.00x
Sparse	53.2125	28.95	**8.47x**

Results:

- Sparse method is **8.47x faster** than dense at 90% sparsity
- Memory savings: 25.0% reduction
- Clear advantage for sparse representation when most elements are zero

5 Performance Bottlenecks

5.1 Dense Matrices

Basic Algorithm (ijk):

- Memory access pattern: Non-sequential access to matrix B
- Cache misses: High due to column-wise access
- Scalability: $O(n^3)$ - grows cubically

Cache Optimized (ikj):

- Improvement: Better spatial locality
- Limitation: Python overhead may mask benefits for small matrices
- Expected benefit: More significant for matrices >1024×1024

Blocked Multiplication:

- Improvement: Blocks fit in L1/L2 cache
- Limitation: Control overhead for small matrices
- Optimal block size: Architecture-dependent (typically 32-128)

5.2 Sparse Matrices

Advantages:

- Only computes non-zero elements
- Reduced memory Fetch
- Ideal for sparsity > 90%

Limitations:

- Data structure overhead for low sparsity (<70%)
- Dictionary/map lookup costs in Python
- Less efficient when many elements are non-zero

6 Maximum Matrix Size Handled

Based on execution times and memory constraints:

Table 5: Maximum practical matrix sizes

Method	**Max Size (approximate)**
Basic (ijk)	512×512 (6 min)
Cache Optimized	512×512 (6 min)
Blocked (32)	512×512 (11 min)
Sparse (90% sparsity)	1000×1000+
Sparse (99% sparsity)	2000×2000+

Note: Maximum sizes are limited by reasonable execution time (<15 minutes) rather than memory constraints.

7 Conclusions and Recommendations

7.1 Key Conclusions

1. **Dense matrices (small to medium):** The basic algorithm performed adequately for matrices up to 512×512 in this Python implementation.

2. **Cache optimizations:** Did not show expected improvements in this configuration, likely due to:

   - Python's interpreted nature and overhead
   - Matrix sizes still relatively small
   - Would benefit more in compiled languages (C/C++, Java)

3. **Sparse matrices:** Provide **dramatic improvements** when sparsity > 90%:

   - Up to 1832x faster at 99% sparsity
   - 8.47x faster than dense at 90% sparsity
   - Significant memory savings

4. **Sparsity is the key factor:** The level of sparsity has more impact on performance than cache optimizations for this implementation.

7.2 Practical Recommendations

For Dense Matrices:

- Small matrices (<256): Use basic algorithm in Python
- Large matrices (>512): Consider NumPy, BLAS libraries, or compiled languages
- Cache optimizations: More effective in C/C++/Java implementations

For Sparse Matrices:

- **Always use sparse** when sparsity > 90%
- **Consider sparse** when sparsity is 70-90% (test both)
- **Use dense** when sparsity < 70%

For Production Systems:

- Use specialized libraries: NumPy, SciPy (sparse), BLAS, cuBLAS (GPU)
- Profile before optimizing: Measure actual bottlenecks
- Consider hardware: GPU acceleration for large matrices