Matrix Multiplication Performance Analysis

Kacper Janiszewski

GitHub repo: `github.com/Janisz11/Individual_assignment_2`

1 Report (Section 2.2)

1.1 Experimental Setup

All experiments were implemented in C++17 and compiled with:

```
g++ -O3 -march=native -std=c++17
```

Execution time was measured using `std::chrono::high_resolution_clock`. We define an algorithm as *efficient* if its runtime is less than or equal to

$$120\,000 \text{ ms} = 120 \text{ s} = 2 \text{ minutes}.$$

We consider the following dense algorithms:

- **Baseline**: classical dense $O(n^3)$ matrix multiplication using three nested loops over rows, columns and the shared dimension.
- **Optimized #1 (Transposed-B)**: the same $O(n^3)$ algorithm, but with matrix B pre-transposed to improve cache locality when accessing columns of B.
- **Optimized #2 (Blocked)**: dense $O(n^3)$ multiplication with cache blocking (tiling), operating on sub-blocks that fit better into CPU caches.

For sparse matrices, we use the CSR (Compressed Sparse Row) format and implement sparse–dense matrix multiplication (SpMM).

1.2 Execution Time and Maximum Matrix Size (Dense)

We benchmarked the three dense implementations for square matrices of sizes

$$N \in \{128, 256, 512, 1024, 2048, 4096\}.$$

The following table shows the measured execution times (in milliseconds). The speedup in parentheses is computed relative to the baseline at the same matrix size. For $N = 4096$ the baseline time is only estimated from cubic scaling; it was not executed to avoid an excessively long run.

Figure 1 illustrates these results graphically.

Table 1: Execution time of dense algorithms (in ms) and speedup versus baseline.

N	Baseline	Transposed-B	Blocked
128	1.28	0.84 (1.52×)	0.34 (3.73×)
256	10.66	8.33 (1.28×)	3.21 (3.33×)
512	366.48	79.72 (4.60×)	36.01 (10.18×)
1024	5432.77	693.78 (7.83×)	268.10 (20.26×)
2048	60882.36	6535.71 (9.32×)	2297.38 (26.50×)
4096	≈ 487058.87 (est.)	52285.37 (9.32× est.)	16180.11 (30.10× est.)

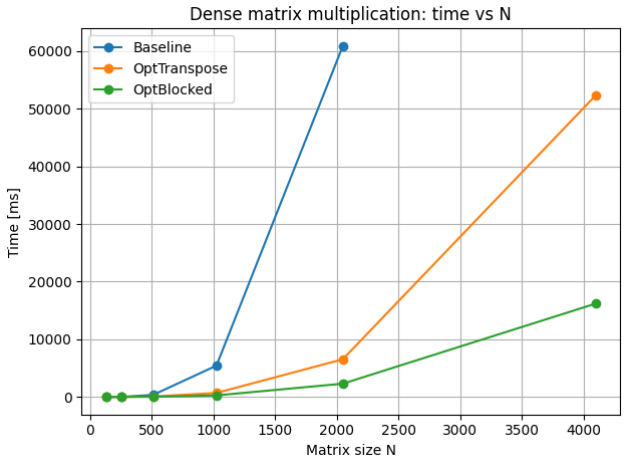

Figure 1: Dense matrix multiplication: runtime vs. matrix size N.

Observations For small matrices, all three methods are fast, though the blocked algorithm is already approximately 3–4× faster than the baseline. As N grows, the differences increase significantly. For $N = 2048$:

- The transposed-B version is approximately 9.3× faster than the baseline.
- The blocked algorithm is approximately 26.5× faster than the baseline.

Both optimized methods clearly improve cache behavior and reduce the effective cost of the $O(n^3)$ computation.

For $N = 4096$, the baseline time was only estimated to be around 487 seconds (≈ 8.1 minutes), so it was not executed in practice. The optimized algorithms were executed and remained well below the 2-minute threshold.

1.2.1 Maximum Matrix Size Handled Efficiently (Dense)

Using the 2-minute threshold (120 000 ms) as the efficiency criterion:

- **Baseline:** For $N = 2048$, the baseline takes $\approx 60\,882$ ms, which is still below the 2-minute threshold and is considered efficient at this size. For $N = 4096$, the estimated time ($\approx 487\,058$ ms) clearly exceeds the time budget, so the baseline algorithm is no longer efficient. The maximum tested efficient size is therefore

$$N_{\text{max, baseline}} = 2048.$$

- **Transposed-B:** At $N = 2048$, the transposed-B method requires about $6\,536$ ms, and at $N = 4096$ it still remains efficient with $\approx 52\,285$ ms, which is below 2 minutes. Within the tested range, the maximum efficient size is

$$N_{\text{max, transposed}} = 4096.$$

- **Blocked:** At $N = 2048$, the blocked algorithm runs in about $2\,297$ ms, and at $N = 4096$ in about $16\,180$ ms, with a very large margin with respect to the 2-minute threshold. Within the tested range, the maximum efficient size is

$$N_{\text{max, blocked}} = 4096.$$

Under the chosen 2-minute threshold, all three dense algorithms can handle matrices up to 2048×2048, but only the optimized implementations remain efficient even at 4096×4096.

1.3 Sparse Matrices: Synthetic CSR versus Dense (Different Sparsity Levels)

To study the impact of sparsity, we fixed the matrix size at $N = 1000$ and varied the density of non-zero entries in matrix A across

$$\{1\%, 5\%, 10\%, 20\%, 50\%\}.$$

For each density, we compared:

- the dense baseline ($A_{\text{dense}} \cdot B_{\text{dense}}$),
- the CSR-based sparse–dense multiplication ($A_{\text{CSR}} \cdot B_{\text{dense}}$).

The following table summarizes the execution times and speedups.
Figure 2 shows the corresponding runtime curves as a function of density.

Performance Comparison and Sparsity Effect The dense baseline performs approximately N^3 operations independent of the number of zeros, so its runtime is nearly constant across densities. In contrast, the CSR-based method performs work proportional to the number of non-zero entries (nnz), so its runtime increases as the matrix becomes denser.

As a consequence:

Table 2: Synthetic sparse matrix ($N = 1000$): dense versus CSR for different densities.

Density	nnz	Dense [ms]	CSR [ms]	Speedup
1%	9 926	871.86	2.85	306.07×
5%	49 960	821.94	10.87	75.59×
10%	99 822	794.83	18.83	42.22×
20%	199 458	796.65	47.85	16.65×
50%	499 387	781.25	109.08	7.16×

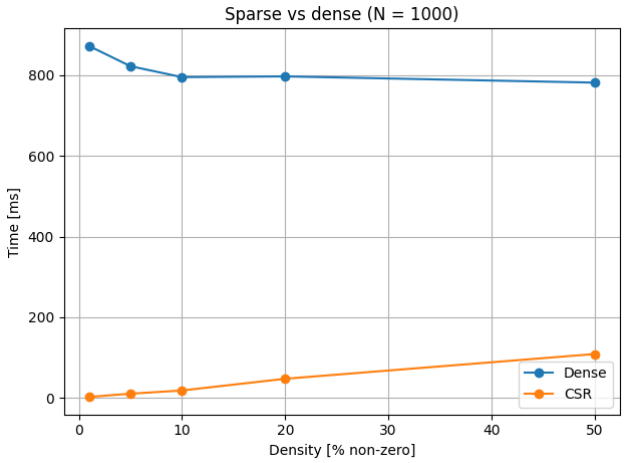

Figure 2: Sparse vs. dense runtime for $N = 1000$ as a function of density (percentage of non-zeros).

- For highly sparse matrices (1% density, approximately 99% zeros), CSR is about 306× faster than the dense baseline.
- For moderately sparse matrices (50% density), CSR is still about 7× faster, although the advantage is smaller.

This confirms that the sparsity level has a strong impact on performance: the sparser the matrix, the more beneficial CSR-based computation becomes. Under the 2-minute threshold, both dense and CSR methods are efficient for $N = 1000$, but CSR is clearly preferable for highly sparse matrices.

1.4 Real Sparse Matrix: `mc2depi` (SuiteSparse)

We also evaluated a real-world sparse matrix from the SuiteSparse collection, `mc2depi`, with the following properties:

- Size: 525 825 × 525 825,

- Number of non-zero entries: $\text{nnz} = 2\,100\,225$,
- Density: approximately $\dfrac{2\,100\,225}{525\,825^2} \approx 0.00076\%$, or approximately 99.99924% zeros.

We loaded `mc2depi.mtx` in Matrix Market format into a CSR structure and performed sparse matrix–dense matrix multiplication:

$$C = A \cdot B,$$

where:

- A is the `mc2depi` matrix in CSR format ($525\,825 \times 525\,825$),
- B is a dense matrix of size $525\,825 \times 16$,
- C is the dense result of size $525\,825 \times 16$.

The measured results were:

- Load time (parsing the Matrix Market file into CSR): $\approx 5\,785.27$ ms,
- Best SpMM runtime (over 3 runs): ≈ 13.33 ms,
- Approximate throughput: ≈ 5.04 GFLOP/s (using $2 \cdot \text{nnz} \cdot 16$ floating-point operations).

Feasibility of Dense Representation A dense $525\,825 \times 525\,825$ matrix of double-precision values would require

$$525\,825^2 \cdot 8 \text{ bytes} \approx 2.0 \text{ TB}$$

of memory for a single matrix A. Storing A, B, and C densely would require approximately 6 TB of RAM, which is far beyond the capacity of a typical workstation. In contrast, the CSR representation of A requires only $O(\text{nnz})$ storage and fits easily into main memory.

For `mc2depi`, dense matrix multiplication is practically infeasible, while sparse CSR-based multiplication completes in milliseconds. This experiment illustrates that for extremely sparse, very large matrices, sparse representations are not just faster but the only viable choice.

1.5 Memory Usage

1.5.1 Dense Matrices

For an $N \times N$ dense matrix of double-precision values, the memory footprint is $8N^2$ bytes. The dense algorithms simultaneously store at least three matrices (A, B, C), so the total size is approximately $24N^2$ bytes. Approximate memory usage for $N \in \{128, 256, 512, 1024, 2048\}$ is shown in Table 3.

The transposed-B algorithm stores an additional N^2 doubles for the transposed matrix B^T, so at $N = 2048$, the total memory for dense matrices reaches approximately 128 MB, which remains acceptable for a modern machine.

Table 3: Approximate memory usage for three dense $N \times N$ matrices (A, B, C).

N	Memory (A + B + C)
128	≈ 0.38 MB
256	≈ 1.50 MB
512	≈ 6.00 MB
1024	≈ 24.00 MB
2048	≈ 96.00 MB

1.5.2 Synthetic Sparse Matrices ($N = 1000$)

For $N = 1000$, the dense baseline uses three dense matrices, requiring

$$3 \cdot 1000^2 \cdot 8 \text{ bytes} = 24 \text{ MB}.$$

The CSR storage for A consists of:

- `values`: a double for each non-zero entry,
- `col_index`: an integer for each non-zero entry,
- `row_ptr`: an integer per row plus one.

For $N = 1000$, the row pointer overhead is negligible. The approximate storage for A in CSR is dominated by $\text{nnz} \cdot (8+4)$ bytes. For the tested densities:

- 1% (nnz $\approx 9\,926$): ≈ 0.12 MB,
- 5% (nnz $\approx 49\,960$): ≈ 0.58 MB,
- 10% (nnz $\approx 99\,822$): ≈ 1.15 MB,
- 20% (nnz $\approx 199\,458$): ≈ 2.29 MB,
- 50% (nnz $\approx 499\,387$): ≈ 5.72 MB.

CSR therefore reduces the memory footprint of A by roughly an order of magnitude (or more) compared to the dense representation, especially at low densities.

1.5.3 Real Sparse Matrix `mc2depi`

For `mc2depi`:

- Dense A alone would require ≈ 2.0 TB of memory,
- CSR A requires only on the order of tens of MB,
- The dense matrices B and C of size $525\,825 \times 16$ require $525\,825 \cdot 16 \cdot 8 \approx 67.2$ MB each.

The total memory for sparse SpMM (CSR A, dense B and C) is around 160 MB, which is easily manageable on a typical machine. This highlights the dramatic memory savings of sparse representations for large, sparse problems.

1.6 Observed Bottlenecks and Performance Issues

1.6.1 Dense Algorithms

The main observations for dense algorithms are:

- The baseline implementation suffers from poor cache locality when accessing columns of B, leading to rapid runtime growth as N increases. For $N = 2048$, the cost is already above 60 seconds and grows quickly beyond that.
- The transposed-B algorithm improves spatial locality by storing B^T, so both A and B^T are accessed row-wise. This reduces cache misses and yields a consistent 7–10$\times$ speedup for large matrices.
- The blocked algorithm further increases cache reuse by operating on sub-blocks that fit into cache. Its runtime grows much more slowly with N, but for very large sizes it becomes limited by memory bandwidth and the cost of streaming data through the memory hierarchy.

1.6.2 Sparse Algorithms

For sparse (CSR) algorithms:

- For synthetic sparse matrices, the dense baseline pays the full $O(N^3)$ cost regardless of the number of zeros, so it becomes inefficient as soon as the matrix is significantly sparse.
- The CSR algorithm performs work proportional to nnz, and its runtime is mostly bounded by memory bandwidth: reading the `values` and `col_index` arrays and updating the output.
- For `mc2depi`, the dominant cost is actually I/O and parsing of the Matrix Market file (several seconds), not the SpMM computation itself (about 13 ms). Dense multiplication is infeasible due to memory capacity constraints, so the limiting factor for dense methods is RAM, while the sparse method remains inexpensive.

1.7 Summary

The experiments demonstrate that:

- For moderately large dense matrices, cache-optimized algorithms (transposed-B and blocked) are essential to achieve acceptable performance and to scale up to $N \approx 4000$ within the 2-minute time budget.

- For large and highly sparse matrices, sparse formats such as CSR are mandatory, dramatically reducing both runtime and memory usage and enabling problems that are completely infeasible with dense representations.