Optimized Matrix Multiplication and Sparse Matrices

Performance Benchmark Report

Alberto Rivero Monzón

November 19, 2025

Repository: `https://github.com/albertuti1910/matrix-multiplication-project`

Contents

1 Introduction

Matrix multiplication is a cornerstone computational kernel in scientific computing, serving as the backbone for applications ranging from finite element analysis (FEM) in physics to graph processing in social network analysis. In many of these real-world scenarios, the matrices involved are *sparse*, meaning the vast majority of their elements are zero.

Standard "dense" matrix multiplication algorithms ($O(N^3)$ or $O(N^2)$ for Matrix-Vector) are inefficient for such datasets, as they perform calculations on zero elements that do not contribute to the final result. This leads to wasted CPU cycles and, more importantly, excessive memory bandwidth consumption.

This report investigates the performance trade-offs between optimized Dense approaches and Sparse Matrix-Vector Multiplication (SpMV). We explore algorithmic optimizations such as loop unrolling and memory-efficient data structures (CSR). Furthermore, we analyze hardware-level bottlenecks using the Linux `perf` profiling tool to understand how cache locality and memory bandwidth impact execution time.

2 Implementation Details

To provide a comprehensive comparison, we implemented three distinct versions of the Matrix-Vector multiplication kernel ($y = Ax$).

2.1 Data Structure: CSR (Compressed Sparse Row)

Storing a large sparse matrix in a standard 2D array is memory-prohibitive. We utilized the Compressed Sparse Row (CSR) format, which compresses the matrix into three 1D arrays:

- `values`: Stores only the non-zero elements (doubles).
- `col_indices`: Stores the column index for each value (integers).
- `row_ptr`: Stores the index in `values` where each row starts.

This reduces the space complexity from $O(N^2)$ to $O(NNZ)$, where NNZ is the number of non-zeros.

```
struct CSRMatrix {
    int rows;
    int cols;
    int nnz;
    std::vector<double> values;    // Non-zero values
    std::vector<int> col_indices;  // Column indices
    std::vector<int> row_ptr;      // Row pointers
};
```

Listing 1: CSR Data Structure Definition

2.2 Optimized Dense Algorithm (Loop Unrolling)

The baseline naive dense approach suffers from loop control overhead. We implemented an optimized version using manual **Loop Unrolling** (factor of 4). This technique increases instruction-level parallelism by providing the CPU pipeline with more independent instructions, reducing the frequency of branch checks.

```cpp
void dense_spmv_optimized(const std::vector<double>& A, ... int rows,
    int cols) {
    for (int i = 0; i < rows; i++) {
        double sum = 0.0;
        int j = 0;
        // Unrolling factor of 4
        for (; j <= cols - 4; j += 4) {
            sum += A[i * cols + j] * x[j];
            sum += A[i * cols + j+1] * x[j+1];
            sum += A[i * cols + j+2] * x[j+2];
            sum += A[i * cols + j+3] * x[j+3];
        }
        // Handle remainder
        for (; j < cols; j++) {
            sum += A[i * cols + j] * x[j];
        }
        y[i] = sum;
    }
}
```

Listing 2: Optimized Dense SpMV with Loop Unrolling

2.3 Sparse Algorithm (SpMV)

The Sparse implementation iterates strictly over the non-zero elements defined by the `row_ptr` array. While this dramatically reduces the operation count, it introduces *indirect memory access* ($x[col\_indices[k]]$), which can be challenging for the CPU prefetcher.

```cpp
void csr_spmv(const CSRMatrix& A, ... std::vector<double>& y) {
    for (int i = 0; i < A.rows; i++) {
        double sum = 0.0;
        // Jump directly to non-zeros using row_ptr
        for (int k = A.row_ptr[i]; k < A.row_ptr[i+1]; k++) {
            sum += A.values[k] * x[A.col_indices[k]];
        }
        y[i] = sum;
    }
}
```

Listing 3: Sparse CSR Kernel

3 Experimental Setup

All benchmarks were executed on an Arch Linux system. The code was compiled using `g++` with `-O2` optimization flags to ensure a fair comparison against compiler-optimized code.

- **OS:** Arch Linux
- **Compiler:** GCC (g++) with `-O2 -fopenmp`
- **Profiling Tool:** `perf` (Linux kernel counter)
- **Dataset:** Randomly generated matrices for scaling tests, and the `mc2depi.mtx` (Epidemiology) matrix for the large-scale sparse test.

4 Results and Discussion

4.1 Impact of Sparsity on Performance

We measured execution time for a fixed matrix size (3000×3000) while varying the sparsity from 0% (completely full) to 99% (mostly empty).

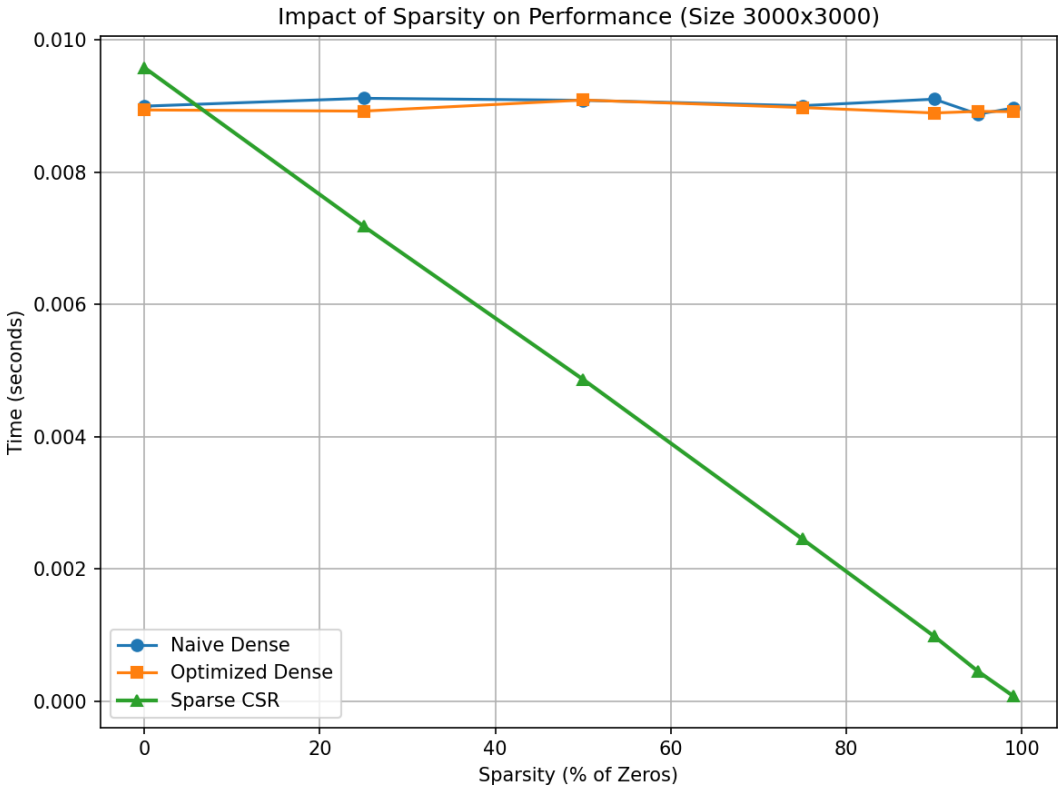

Figure 1: Execution Time vs. Sparsity Level

Findings: Figure 1 demonstrates a crucial trade-off.

1. **0% Sparsity (Full Matrix):** The Sparse CSR algorithm (Green line) is actually *slower* than the Dense algorithm. This is due to the overhead of indirect indexing; calculating $A[i][j]$ is computationally cheaper than looking up $col\_indices[k]$.

2. **Increasing Sparsity:** As the number of zeros increases, the Sparse algorithm's execution time drops linearly.

3. **Dense Behavior:** The Dense algorithms (Blue/Orange) remain flat, as they must process all N^2 elements regardless of their value.

4.2 Speedup Analysis

The relative performance gain of switching to CSR is analyzed in Figure 2.

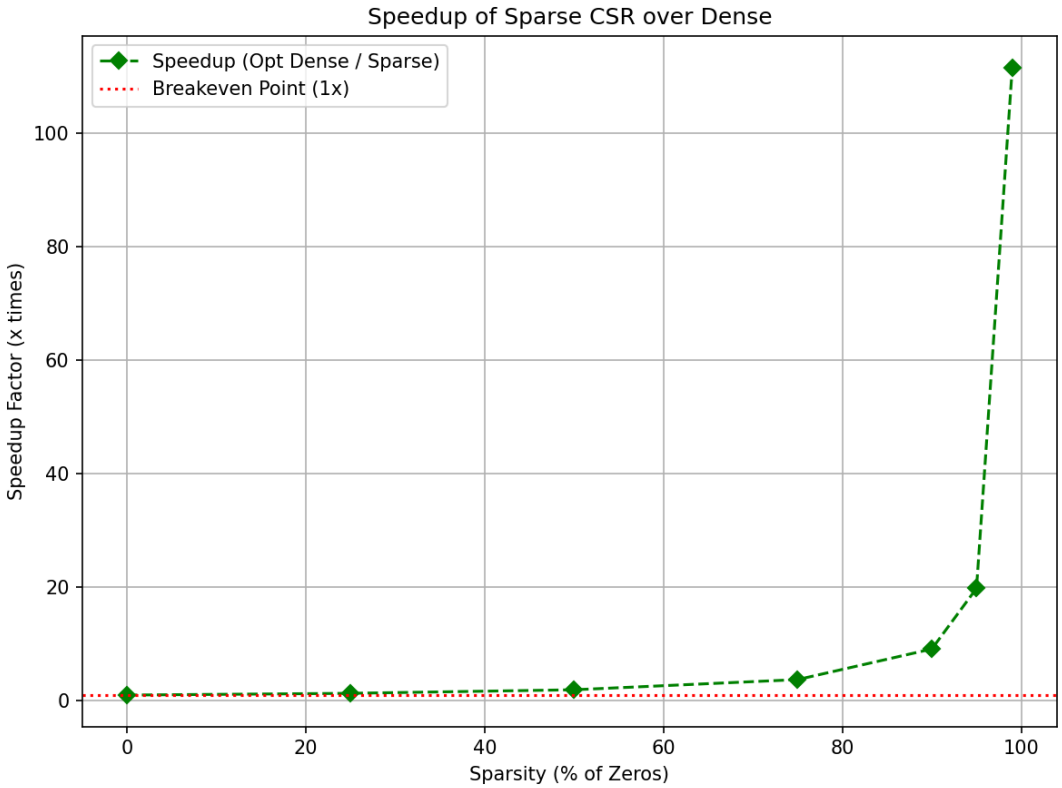

Figure 2: Speedup of Sparse CSR over Optimized Dense

Findings: The "Breakeven Point" (where Speedup > 1) occurs relatively early, at approximately 15% sparsity.

- At **50% Sparsity**, CSR is already $\approx 2x$ faster.
- At **99% Sparsity**, which is typical for real-world adjacency matrices, the CSR implementation achieves over **110x speedup**. This exponential growth justifies the complexity of implementing sparse data structures.

4.3 Scalability with Matrix Size

We tested matrix sizes ranging from $N = 1,000$ to $N = 10,000$ at a fixed 90% sparsity level.

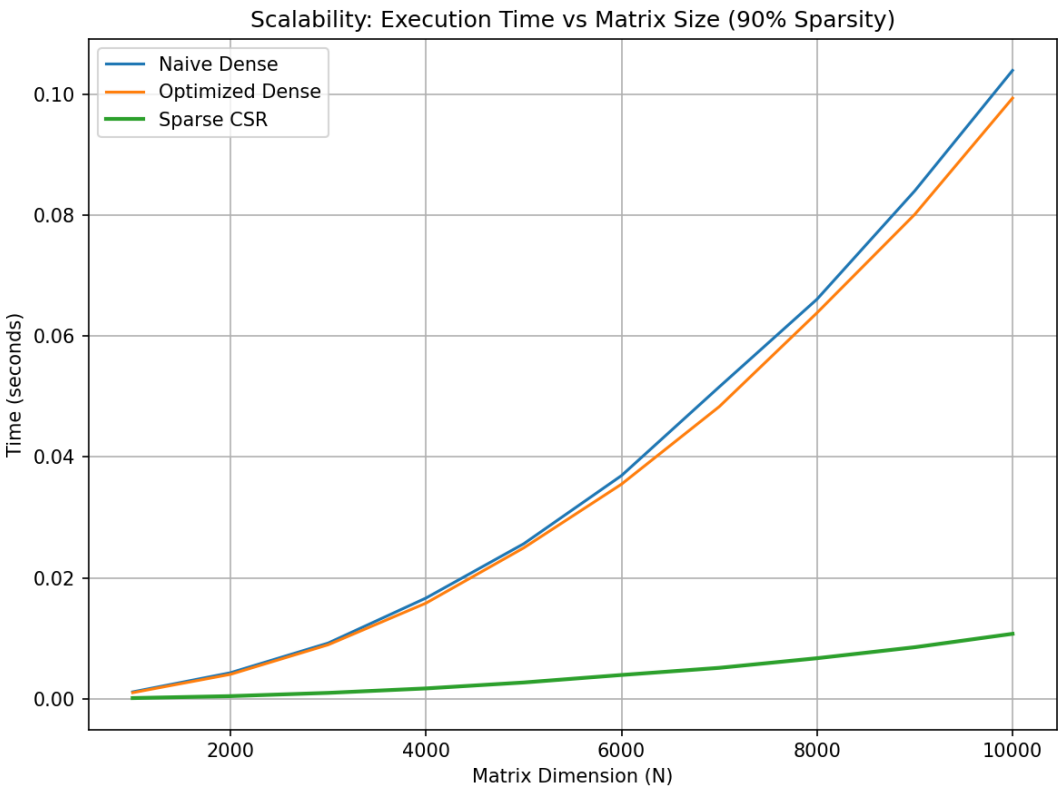

Figure 3: Scalability: Execution Time vs Matrix Dimension (90% Sparsity)

Findings:

- **Dense (Orange/Blue):** Exhibits a clear quadratic curve ($O(N^2)$). Doubling the matrix size quadruples the execution time.
- **Sparse (Green):** Appears nearly linear in comparison. While technically still dependent on N, the constant factor is reduced by 90%, making it feasible to run much larger simulations.

4.4 Memory Usage Comparison

Perhaps the most critical limitation in High-Performance Computing is memory capacity.

Figure 4: Theoretical Memory Usage (Log Scale)

Findings: Note the logarithmic scale in Figure 4.

- A dense $10,000 \times 10,000$ matrix requires ≈ 800 MB of RAM.
- By extrapolation, a $20,000 \times 20,000$ matrix would exceed 3.2 GB, saturating standard 4GB allocations.
- The Sparse format stays in the megabyte range (approx 100MB) for the same size.

This efficiency allowed us to load the `mc2depi` matrix ($N = 525,825$) using only $\approx$ 26MB of RAM, whereas a dense representation would have required over 2.2 Terabytes, which is physically impossible on standard hardware.

5 Bottleneck Analysis

To understand the hardware-level behavior of the SpMV kernel, we profiled the execution of the large `mc2depi` matrix using `perf stat -d`.

5.1 Full Perf Output

The following output was captured during the execution of the Sparse algorithms:

```
Performance counter stats for './spmv_bench':

 1,048,644,565      task-clock                #    1.117 CPUs utilized
```

```
          48      context-switches          #   45.773 /sec
           1      cpu-migrations            #    0.954 /sec
       6,511      page-faults               #    6.209 K/sec
 7,249,199,638    instructions              #    2.10  insn per cycle
 3,446,077,972    cycles                    #    3.286 GHz
 1,556,284,387    branches                  #    1.484 G/sec
    11,325,228    branch-misses             #    0.73% of all branches
 1,992,605,081    L1-dcache-loads           #    1.900 G/sec
    27,860,563    L1-dcache-load-misses     #    1.40% of all L1 accesses
     1,545,640    LLC-loads                 #    1.474 M/sec
       611,748    LLC-load-misses           #   39.58% of all LL-cache accesses

       0.939049524 seconds time elapsed
```

5.2 Analysis of Metrics

5.2.1 The Memory Bottleneck (LLC Misses)

The most significant metric is the **LLC-load-misses** at **39.58%**.

- **Interpretation:** Nearly 40% of the time the CPU requested data from the Last Level Cache (L3), the data was missing and had to be fetched from main RAM.
- **Cause:** In SpMV, we access the vector x using indirect indices: `x[col_indices[k]]`. Since the matrix is sparse, these indices are not sequential. The hardware prefetcher cannot predict which address will be needed next, leading to cache thrashing.
- **Conclusion:** The application is **Memory Bound**. The CPU is spending a significant portion of its cycles waiting for data to arrive from DRAM.

5.2.2 Branch Prediction

The **branch-misses** rate is very low (0.73%). This indicates that the structure of the loops (iterating through row pointers) is predictable for the CPU, and the performance penalty is not coming from control flow, but from memory latency.

5.3 Parallelism and Performance Degradation

We attempted to parallelize the SpMV kernel using OpenMP. However, results showed a slowdown:

- **Basic Sparse Time:** 0.0020s
- **Parallel Sparse Time:** 0.0097s

This counter-intuitive result is explained by the "Memory Wall". Since the single-core implementation already saturates a significant portion of the memory bandwidth (indicated by high cache misses), adding more cores leads to **bus contention**. Additionally, the workload (2.1 million non-zeros) is processed so quickly ($\approx 2ms$) that the overhead of creating and scheduling threads exceeds the time saved by parallel execution.

6 Conclusion

This benchmark study provides clear evidence for the necessity of sparse matrix formats in scientific computing.

- **Efficiency:** For matrices with high sparsity (>90%), CSR formats offer speedups exceeding 100x compared to dense approaches.
- **Necessity:** Dense formats are mathematically impossible to use for large datasets ($N > 50,000$) due to RAM constraints, whereas Sparse formats scale efficiently.
- **Bottlenecks:** Modern SpMV is limited by memory bandwidth, not CPU clock speed. Future optimizations should focus on improving cache locality (e.g., cache blocking or matrix reordering) rather than raw instruction throughput.