Sparse Matrix-Vector Multiplication Benchmark

https://github.com/klapa0/big_data_studies

Kacper Klasen, FB390707

January 30, 2026

1 Introduction

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in scientific computing, numerical simulations, and data analysis. Its performance depends heavily on the storage format of the sparse matrix and memory access patterns due to irregular accesses to the input vector. The goal of this experiment is to benchmark two approaches:

1. **Naive CSR** — standard row-wise traversal using the Compressed Sparse Row format.
2. **Cache-blocked CSR (Bucketed/Compact)** — reorganizing non-zero elements into contiguous blocks per column to improve cache locality.

2 Methodology

The `mc2depi.mtx` matrix from the Matrix Market repository was used as the test case. It has 525,825 rows and columns with 2,100,225 non-zero elements. The input vector $\mathbf{x}$ was filled with ones. The SpMV operation was repeated 50 times to obtain stable timings.

The benchmarks were compiled using:

```
g++ main.cpp -O3 -fopenmp -std=c++17 -o spmv
```

The OpenMP parallelization was applied to:

- **Naive CSR**: parallel over rows.
- **Blocked CSR**: parallel over column blocks, with atomic operations on the output vector to avoid race conditions.

Different numbers of threads were tested by setting `OMP_NUM_THREADS` or using `omp_set_num_threads(` in the code.

3 Implementation Details

3.1 Naive CSR

Each row is computed independently:

$$y_i = \sum_{k=A.ptr[i]}^{A.ptr[i+1]-1} A.val[k] \cdot x[A.col[k]], \quad i = 0, \ldots, n-1$$

This allows direct parallelization over rows with no synchronization.

3.2 Blocked CSR (with atomic updates)

The matrix is divided into column blocks of size $CB = 4096$. Non-zero elements are reorganized into a compact structure:

- $B.b\_ptr$ — start index of each block in the compact arrays.
- $B.row\_indices$, $B.col\_indices$, $B.val\_data$ — flattened arrays of non-zero entries per block.

Parallelization is applied over blocks:

```
#pragma omp parallel for
for (int b = 0; b < B.nblocks; b++)
    for (int j = B.b_ptr[b]; j < B.b_ptr[b+1]; j++)
        #pragma omp atomic
        y[B.row_indices[j]] += B.val_data[j] * x[B.col_indices[j]];
```

Atomic operations are required because multiple blocks may update the same row.

4 Results

Threads	Naive CSR [s]	Blocked CSR (atomic) [s]
1	0.00406	0.02268
2	0.00199	0.01158
4	0.00115	0.00604
8	0.00102	0.00395
16	0.00096	0.00356

Table 1: Average execution time per iteration for different numbers of threads.

Both implementations produced identical results (maximum element-wise difference $< 10^{-9}$), confirming correctness.

5 Analysis

The naive CSR implementation scales well with the number of threads because:

- each row is independent,
- no synchronization is required,

- memory access is cache-friendly.

The blocked CSR version is slower despite parallelization. This is due to:

- **atomic operations** on the output vector, which serialize updates and cause cache line bouncing,
- multiple threads potentially updating the same row concurrently,
- overhead of reorganizing the matrix into blocks, which does not benefit the cache for this moderately sparse matrix.

As a result, increasing the number of threads improves performance only partially. The overhead of atomic operations dominates at higher thread counts.

6 Conclusion

Memory optimizations such as cache blocking are not universally beneficial. For the `mc2depi.mtx` matrix:

- Naive CSR with OpenMP parallelization over rows outperforms blocked CSR with atomic updates.
- Atomic operations limit scalability and introduce variability in execution times.
- Cache blocking would only be beneficial for extremely large matrices or on architectures where row-level parallelism is insufficient.

7 References

- Matrix Market: `https://math.nist.gov/MatrixMarket/`
- OpenAI ChatGPT, assistance with benchmark analysis and LaTeX formatting.