Performance Analysis of Sparse Matrix-Vector Multiplication Using CSR Format in Rust

Lukasz Stolzmann

November 22, 2025

Abstract

This paper presents a performance analysis of sparse matrix-vector multiplication (SpMV) implemented in Rust using the Compressed Sparse Row (CSR) format. The study evaluates both serial and parallel implementations using the Rayon library. Performance measurements include execution time, GFLOP/s, and memory bandwidth using the mc2depi sparse matrix. Results show performance improvements with parallelization and compiler optimizations, achieving 5.63 GFLOP/s and 47.9 GB/s on Apple M3 Pro hardware.

Keywords: Sparse matrices, CSR format, SpMV, Rust programming, parallel computing, performance analysis

1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a fundamental kernel in scientific computing, appearing in iterative solvers, eigenvalue computations, graph algorithms, and machine learning applications. Unlike dense matrix operations, SpMV performance is typically memory bandwidth-bounded due to the irregular access patterns and low arithmetic intensity inherent in sparse data structures.

The Compressed Sparse Row (CSR) format is one of the most widely used sparse matrix storage schemes, offering a good balance between memory efficiency and computational performance. CSR stores only the non-zero elements along with their column indices and row pointers, significantly reducing memory requirements for sparse matrices.

This study examines the performance characteristics of SpMV implementations in Rust, a systems programming language that combines memory safety with zero-cost abstractions. The investigation focuses on:

- Serial SpMV implementation using CSR format
- Parallel SpMV using Rayon for multi-threading
- Performance analysis in terms of GFLOP/s and memory bandwidth
- Comparison between debug and release optimization levels
- Memory bandwidth utilization and cache efficiency analysis

2 Methodology

2.1 Sparse Matrix Storage Format

The CSR format represents a sparse matrix using three arrays:

- `values`: Non-zero elements stored row-wise in floating-point format

- `indices`: Column indices of non-zero elements as integers
- `indptr`: Row pointers indicating the start of each row in the values array

This format enables efficient row-wise traversal, which is optimal for SpMV operations where each row is processed independently to compute one element of the result vector. The memory layout provides good cache locality for sequential row access patterns.

2.2 Implementation Framework

The implementations utilize the `sprs` crate (version 0.11), a pure Rust sparse matrix library, and `rayon` (version 1.10) for data parallelism. Matrix Market format files are used for input, ensuring compatibility with standard sparse matrix benchmarks from the SuiteSparse Matrix Collection.

2.3 Algorithms Implemented

2.3.1 Serial SpMV Implementation

The serial SpMV algorithm follows the standard row-wise computation approach:

```
fn spmv_serial(A: &CsMat<f64>, x: &[f64]) -> Vec<f64> {
    let nrows = A.rows();
    let mut y = vec![0.0f64; nrows];

    for i in 0..nrows {
        let mut sum = 0.0;
        if let Some(row) = A.outer_view(i) {
            for (col, val) in row.iter() {
                sum += val * x[col];
            }
        }
        y[i] = sum;
    }
    y
}
```

Listing 1: Serial SpMV Implementation

2.3.2 Parallel SpMV Implementation

The parallel implementation uses Rayon's parallel iterators to distribute row computations across available CPU cores:

```
fn spmv_parallel(A: &CsMat<f64>, x: &[f64]) -> Vec<f64> {
    let nrows = A.rows();

    (0..nrows)
        .into_par_iter()
        .map(|i| {
            let mut sum = 0.0;
            if let Some(row) = A.outer_view(i) {
                for (col, val) in row.iter() {
                    sum += val * x[col];
                }
            }
            sum
        })
        .collect()
}
```

Listing 2: Parallel SpMV Implementation

2.4 Benchmarking Framework

Performance measurements use Rust's `std::time::Instant` for high-precision timing. The benchmarking methodology includes:

- Warm-up iterations to ensure stable CPU frequency and cache state
- Single-run measurements for consistent timing methodology
- GFLOP/s calculation based on 2 floating-point operations per non-zero element
- Memory bandwidth estimation including all data structure access costs

```rust
fn benchmark<F>(name: &str, A: &CsMat<f64>, x: &[f64], f: F)
where F: Fn(&CsMat<f64>, &[f64]) -> Vec<f64>,
{
    println!("\n[INFO] Benchmark: {name}");

    // Warm-up iteration
    let _ = f(A, x);

    let start = Instant::now();
    let _y = f(A, x);
    let dt = start.elapsed().as_secs_f64();

    let nnz = A.nnz() as f64;
    let gflops = (2.0 * nnz) / (dt * 1.0e9);

    // Memory bandwidth calculation
    let total_bytes = estimate_memory_traffic(A);
    let bandwidth_gb_s = total_bytes / (dt * 1.0e9);

    println!("{name}: time = {dt:.6} s, perf = {gflops:.3} GFLOP/s, BW = {
    bandwidth_gb_s:.3} GB/s");
}
```

Listing 3: Benchmark Implementation

2.5 Memory Bandwidth Model

The memory bandwidth calculation estimates the total data movement:

$$\begin{aligned} \text{Total Bytes} &= \text{Values} + \text{Indices} + \text{Pointers} + \text{Vector Access} && (1) \\ &= 8 \cdot \text{nnz} + 4 \cdot \text{nnz} + 4 \cdot (\text{rows} + 1) + 8 \cdot \text{cols} + 8 \cdot \text{rows} && (2) \end{aligned}$$

This model accounts for reading CSR data structures, input vector access, and output vector writes.

2.6 Experimental Setup

- **Hardware**: Apple M3 Pro with 18GB unified memory
- **Software**: Rust 1.83.0 with Cargo build system
- **Test Matrix**: mc2depi.mtx (525,825 $\times$ 525,825, nnz = 2,100,225)
- **Compilation**: Debug mode vs Release mode with `-O3` equivalent optimizations
- **Parallelization**: Rayon with default thread pool (matching CPU core count)

3 Implementation Details

3.1 Code Organization

The project follows Rust best practices for code organization:

- Clean separation between algorithm implementation and benchmarking code
- Use of Rust's type system for compile-time guarantees
- Memory-safe implementations without runtime overhead
- Professional build system using Cargo for dependency management

3.2 Language-Specific Considerations

3.2.1 Rust Advantages

Rust provides several advantages for sparse matrix computations:

- **Zero-cost abstractions**: High-level iterator patterns compile to efficient loops
- **Memory safety**: Bounds checking eliminated in release mode without runtime overhead
- **Parallel safety**: Rayon provides data-race-free parallelism with no performance penalties
- **LLVM backend**: Access to state-of-the-art compiler optimizations

3.2.2 CSR Format Benefits

The CSR format is well-suited for SpMV operations:

- **Sequential memory access**: Row-wise storage matches computation patterns
- **Cache efficiency**: Compact data layout reduces cache misses
- **Parallelization**: Independent row computations enable trivial parallelization
- **Memory efficiency**: Only non-zero elements stored, reducing memory footprint

4 Results and Analysis

4.1 Performance Comparison

Table 1 presents the benchmark results comparing debug and release compilation modes for both serial and parallel implementations using the mc2depi.mtx test matrix.

Table 1: SpMV Performance Results for mc2depi.mtx Matrix (525,825 × 525,825, nnz = 2,100,225)

Implementation	**Time (ms)**	**GFLOP/s**	**Bandwidth (GB/s)**	**Speedup**
Debug Serial	72.348	0.058	0.494	1.0×
Debug Parallel	11.553	0.364	3.092	6.3×
Release Serial	1.405	2.989	25.419	51.5×
Release Parallel	0.746	5.630	47.878	97.0×

4.2 Performance Analysis

4.2.1 Compiler Optimization Impact

Release mode optimizations provide dramatic performance improvements:

- **Serial speedup**: 51.5× faster (72.3ms → 1.4ms)
- **Parallel speedup**: 15.5× faster (11.6ms → 0.75ms)
- **Peak performance**: 5.63 GFLOP/s and 47.9 GB/s achieved in parallel release mode

This shows the effectiveness of compiler optimizations including vectorization and loop unrolling.

4.2.2 Parallelization Effectiveness

Parallel execution shows significant but limited speedups:

- **Debug mode speedup**: 6.3× (memory bandwidth limited)
- **Release mode speedup**: 1.9× (approaching memory bandwidth saturation)
- **Parallel efficiency**: Higher in debug mode due to computational bottlenecks

The limited speedup in release mode reflects the memory-bound nature of SpMV, where performance is constrained by memory bandwidth rather than computational capacity.

4.2.3 Memory Bandwidth Analysis

The memory bandwidth results provide insights into system utilization:

- **Peak bandwidth**: 47.9 GB/s approaches M3 Pro's theoretical memory bandwidth
- **Bandwidth scaling**: Nearly linear improvement from serial to parallel execution
- **Cache effects**: Efficient CSR format utilization with minimal cache misses

5 Discussion

5.1 Performance Observations

The benchmark results demonstrate several important characteristics:

- **Optimization dependency**: Debug vs release performance gap highlights compiler optimization importance
- **Memory bandwidth utilization**: 47.9 GB/s represents efficient use of available hardware bandwidth
- **Parallel efficiency**: Rayon provides effective parallelization with minimal overhead
- **Language performance**: Rust achieves performance competitive with optimized C/C++ implementations

6 Conclusion

This study shows that Rust provides a good platform for sparse matrix computations with memory safety. The CSR format works well for SpMV operations.

Key findings include:

- Release mode optimizations provide significant performance improvements (15-50×)
- Parallel execution achieves good speedups within memory bandwidth limits
- CSR format enables efficient implementation and parallelization
- Peak performance: 5.63 GFLOP/s and 47.9 GB/s memory bandwidth

Rust demonstrates good performance for sparse matrix computations while maintaining memory safety.

7 Future Work

Future extensions could include:

- GPU acceleration using CUDA or ROCm bindings
- Comparison with optimized libraries (Intel MKL, cuSPARSE)
- Testing different sparse matrix patterns and formats
- Implementation of alternative sparse formats (BSR, ELLPACK)

8 Repository and Reproducibility

The complete source code, benchmarking framework, and documentation are available at:

`https://github.com/BigDataULPGC-Lukasz-Stolzmann/TASK-2`

The repository includes:

- Complete Rust implementation with Cargo.toml dependencies and build configuration
- Benchmarking scripts and raw performance data for result verification
- LaTeX source code and compiled PDF for complete documentation
- Instructions for reproducing results across different hardware configurations
- Sample matrices and testing frameworks for extended evaluation