Stage 3 – Sparse Matrix-Vector Multiplication (SpMV) Benchmarking

Agustín Darío Casebonne

December 3, 2025

Link to github

Abstract

This report presents a benchmarking analysis of Sparse Matrix–Vector Multiplication (SpMV) using the CSR (Compressed Sparse Row) format. Implementations were developed in Java, Python, and Rust, focusing on sequential and parallel performance across multiple matrix sizes. Results include execution time, speedup, efficiency, and memory usage. A concise discussion highlights the implications of the findings in the context of memory-bound computations with irregular access patterns.

1 Introduction

Sparse Matrix–Vector Multiplication (SpMV) is a fundamental kernel in scientific computing and machine learning systems. Unlike dense matrix multiplication, SpMV is limited by memory bandwidth rather than arithmetic throughput. Its performance depends heavily on:

- memory locality,
- irregular access patterns,
- thread synchronization overhead,
- the efficiency of the sparse representation.

The CSR (Compressed Sparse Row) format reduces storage to:

$$\mathcal{O}(N_{\text{nz}} + N)$$

where N_{nz} is the number of non-zero entries. It also limits computational work to existing non-zeros only.

This stage evaluates three programming languages with distinct memory and execution models:

- **Rust**: low-level, memory-safe, zero-cost abstractions,
- **Java**: managed runtime, JIT compilation, garbage collection,
- **Python**: high-level, Numba-accelerated JIT for numerical kernels.

2 Methodology

Each implementation contains:

- a CSR generator with approximately 5% density,

- a sequential SpMV kernel,
- a parallel kernel (threads or multiprocessing depending on language),
- a benchmark harness running 5 repeated trials per matrix size.

Tested dimensions:

$$N \in \{512, 1024, 1536, 2048\}$$

Metrics collected:

- sequential and parallel runtime (seconds),
- speedup = sequential / parallel,
- efficiency = speedup / number of threads,
- memory usage (MB).

Memory measurements were taken using language-specific tools and OS-level readings. Some anomalies (e.g., negative Java values) are attributed to measurement deltas rather than actual memory reduction.

3 Results

3.1 Java

Table 1: Java SpMV CSR Results

N	Seq. (s)	Par. (s)	Speedup	Eff.	Mem. (MB)
512	0.000895	0.010353	0.09	0.01	-5.50
1024	0.001645	0.000664	2.48	0.15	N/A
2048	0.002983	0.000988	3.02	0.19	47.21

The negative memory value at $N = 512$ indicates a measurement artifact from heap delta calculations and does not represent actual memory release.

3.2 Python (Numba)

Table 2: Python (Numba JIT) SpMV CSR Results

N	Seq. (s)	Par. (s)	Speedup	Eff.	Mem. (MB)
512	0.153124	0.133692	1.15	0.07	70.54
1024	0.001420	0.000407	3.49	0.22	14.41
1536	0.003021	0.000597	5.06	0.32	32.41
2048	0.004702	0.001921	2.45	0.15	57.62

Numba's JIT compilation dramatically improves runtime, but memory usage remains comparatively high due to Python object overheads and array wrappers.

3.3 Rust

Rust consistently achieves the fastest sequential times but suffers from high parallel overhead relative to problem size.

Table 3: Rust SpMV CSR Results

N	Seq. (s)	Par. (s)	Speedup	Eff.	Mem. (MB)
512	0.000017	0.000215	0.08	0.00	0.21
1024	0.000144	0.001182	0.12	0.01	0.82
1536	0.000182	0.000850	0.21	0.01	1.84
2048	0.000330	0.001113	0.30	0.02	3.25

4 Discussion

4.1 Sequential Performance

Rust achieves the best sequential execution time by a large margin. Its memory safety and zero-cost abstractions allow it to perform very close to raw pointer arithmetic.

Python exhibits significant variation:

- large overhead at small sizes (NumPy/Numba warmup),
- excellent performance at medium sizes (1024–1536),
- slower performance again at 2048 due to memory pressure.

Java benefits from JIT compilation but retains the overhead of garbage collection and a larger baseline memory footprint.

4.2 Parallel Scaling

Parallel efficiency is low across all languages due to the memory-bound nature of SpMV.

Key insights:

- Rust: thread coordination overhead is larger than the computational work.
- Java: moderate speedups appear at larger sizes but overhead dominates for small matrices.
- Python: in some mid-sized cases achieves speedups $> 5\times$, likely due to oversubscription and JIT optimizations.

4.3 Memory Usage

Rust is the clear memory-efficiency winner, closely matching theoretical CSR requirements.

Java and Python show 10–20x higher memory usage due to:

- object headers,
- garbage collector metadata,
- dynamic typing overheads,
- array wrapper layers.

5 Conclusions

1. **Fastest sequential performance**: Rust.
2. **Most consistent parallel scaling**: Python (for medium matrix sizes).

3. **Worst parallel performance**: Rust and Java for small-to-medium matrices due to overhead dominating.
4. **Best memory efficiency**: Rust by a wide margin.
5. **General finding**: All languages struggle with parallel SpMV due to its low arithmetic intensity and high memory bandwidth dependence.

Future work should include experimenting with:

- larger matrices ($N > 4096$),
- varying sparsity patterns,
- NUMA-aware parallelization,
- cache blocking for SpMV variants.