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

Agustín Darío Casebonne

December 3, 2025

Link to GitHub Repository

Abstract

This report presents a comprehensive benchmarking analysis of Sparse Matrix–Vector Multiplication (SpMV) using the Compressed Sparse Row (CSR) representation. Implementations were developed in Java, Python (with Numba), and Rust, covering both sequential and parallel versions. The study evaluates execution time, speedup, efficiency, and memory consumption for multiple matrix dimensions. A technical discussion interprets the results within the context of memory-bound numerical kernels and programming language runtime characteristics. The analysis highlights both algorithmic limitations and language-specific behaviors, providing a thorough evaluation aligned with high-performance computing criteria.

1 Introduction

Sparse Matrix–Vector Multiplication (SpMV) is a fundamental kernel in scientific computing, graph processing, and machine learning pipelines. Unlike dense matrix multiplication—which is compute-bound and dominated by floating-point operations—SpMV is inherently *memory-bound.* Its performance is constrained by:

- irregular memory access patterns,
- low arithmetic intensity (typically below 1 FLOP/byte),
- cache-line underutilization,
- overheads from thread scheduling and synchronization.

The CSR format significantly reduces storage by maintaining only non-zero entries:

$$\mathcal{O}(N_{\text{nz}}) \quad \text{vs.} \quad \mathcal{O}(N^2)$$

and improves spatial locality relative to other sparse formats.

Three languages were selected due to their contrasting approaches to memory and execution:

- **Rust**: zero-cost abstractions, deterministic memory access, minimal runtime overhead,
- **Java**: JIT compilation, automatic memory management, predictable multi-threading,
- **Python**: high-level semantics augmented with Numba JIT compilation for numerical kernels.

This stage focuses on evaluating how the CSR structure interacts with each runtime model under both sequential and parallel workloads.

2 Methodology

Each implementation includes:

- a CSR generator with approximately 5% non-zero density,
- an optimized sequential SpMV kernel,
- a parallel SpMV kernel (threads or multiprocessing),
- a reproducible benchmark harness running five trials per configuration.

The evaluated dimensions were:

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

The following metrics were collected:

- execution time (sequential and parallel),
- speedup: Seq/Par,
- efficiency: Speedup/Threads,
- estimated memory usage (MB), based on runtime-level tools.

It is important to note that SpMV performance is highly sensitive to memory subsystem behavior; therefore, perfect scaling is neither expected nor achievable on commodity hardware.

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	10.81
1536	0.002284	0.0003290	0.69	0.04	28.18
2048	0.002983	0.000988	3.02	0.19	47.21

The anomalous negative memory value at $N = 512$ arises from Java's incremental heap resizing and GC activity. It does not reflect 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

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

3.3 Rust

4 Discussion

4.1 Sequential Performance

Rust achieves the fastest sequential execution by a significant margin, benefiting from:

- tight control of memory access,
- absence of garbage collection,
- predictable data layout.

Python with Numba shows excellent scaling beyond warm-up costs, while Java's sequential results exhibit moderate noise due to JVM warm-up and GC events.

4.2 Parallel Scaling

As expected for a memory-bound kernel, speedup remains modest:

- Rust suffers from thread scheduling and synchronization overhead relative to its extremely small per-row workload,
- Java exhibits moderate speedup at large sizes but overhead dominates for small matrices,
- Python achieves the best scaling due to Numba's efficient loop fusion, vectorization opportunities, and simplified threading model.

None of the implementations achieve linear scaling, which is consistent with theoretical expectations for SpMV on shared-memory systems.

4.3 Memory Usage

Rust is closest to theoretical CSR memory requirements. Java and Python incur overhead from:

- runtime metadata,
- dynamic array structures,
- garbage collector or reference counting infrastructure.

These observations align with expectations for managed runtimes.

5 Conclusions

1. **Rust** provides the fastest and most memory-efficient sequential implementation.
2. **Python (Numba)** achieves the highest parallel speedup at mid-range matrix sizes.

3. **Java** displays stable performance for large matrices but incurs significant runtime overhead.
4. The inherently memory-bound nature of SpMV limits achievable parallel efficiency across all tested languages.

Future work could explore:

- larger matrices ($N > 4096$),
- variable sparsity patterns and real-world matrices,
- NUMA-aware thread placement,
- alternative sparse formats such as ELL, HYB, and blocked CSR.