Accelerating Sparse Matrix-Vector Multiplication (SpMV) Using CUDA: Performance Analysis

Amin Mohamed Amin ElSayed
Department of Computer and Systems Engineering, Alexandria University
Alexandria, Egypt
Student ID: 21010310
Email: es-amin.mohamedamin2026@alexu.edu.eg

I. Problem Statement

A. Importance

Sparse matrix-vector multiplication (SpMV), $\mathbf{y} = A\mathbf{x}$, is a fundamental kernel in scientific computing. It plays a central role in **iterative solvers** such as Conjugate Gradient and GMRES, **eigenvalue problems**, **graph analytics** like PageRank and network analysis, and **machine learning tasks** involving sparse features or recommendation systems.

SpMV is inherently memory-bound: each non-zero element contributes a single multiply-add, but accessing values and indices consumes significant memory bandwidth. Irregular sparsity patterns cause non-contiguous memory access, poor cache utilization, and high latency. As a result, SpMV typically achieves only a small fraction of the processor's peak floating-point performance. Efficient implementations must therefore optimize memory access, exploit parallelism, and tune for hardware to accelerate real-world applications across science, engineering, and data analytics.

B. Formulation

Given a sparse matrix $A \in \mathbb{R}^{m\times n}$ with $\text{nnz} \ll mn$ non-zero elements and a dense vector $\mathbf{x} \in \mathbb{R}^n$, the sparse matrix-vector multiplication computes $\mathbf{y} \in \mathbb{R}^m$ as:

$$y_i = \sum_{j\in\mathcal{N}(i)} a_{ij}x_j, \quad i = 1,\dots,m, \tag{1}$$

where $\mathcal{N}(i) = \{j : a_{ij} \neq 0\}$ is the set of column indices with non-zero entries in row i. The complexity is $\mathcal{O}(\text{nnz})$, requiring $2 \cdot \text{nnz}$ floating-point operations. Common storage formats include Compressed Sparse Row (CSR), which stores values, column indices, and row pointers, requiring $2 \cdot \text{nnz} + m + 1$ storage locations.

II. Algorithm Design

The SpMV algorithm is designed following **Foster's methodology** for parallel computation, which relies on four principles: *Partitioning, Communication, Agglomeration, and Mapping*.

A. Foster's Principles Applied to SpMV

1) **Partitioning:** The matrix is divided by rows, with each row treated as an independent task, enabling parallel execution.
2) **Communication:** Threads access shared elements of the input vector; minimizing memory fetches is crucial for performance.
3) **Agglomeration:** Several rows are grouped into a block to reduce communication overhead; shared memory can store frequently used vector elements.
4) **Mapping:** Each row is assigned to a thread, and threads are organized into blocks to balance workload and maximize GPU utilization.

B. Serial Algorithm

The serial implementation processes the matrix row by row, computing a dot product between non-zero elements and the corresponding entries in the input vector. It has linear complexity, $O(\text{nnz})$, and serves as a reference for verifying the parallel implementation.

C. Parallel Algorithm (CUDA)

In the parallel version, each row is handled by a GPU thread. Threads are grouped into blocks to improve occupancy and reduce memory latency, allowing the GPU to outperform the serial version, especially for large matrices.

III. Pseudocode

A. Serial SpMV

```
1) Input: $A$, $x$, size $N$
2) For $i = 0 \dots N-1$:
    a) $\text{sum} = 0$
    b) For each non-zero $A_{ij}$: $\text{sum} \mathrel{+}= A_{ij} \cdot x_j$
    c) $y[i] = \text{sum}$
3) Output: $y$
```

B. Parallel SpMV (CUDA)

```
1) Input: $A$, $x$, size $N$
2) $i = \textbf{blockIdx.x} \cdot \textbf{blockDim.x} + \textbf{threadIdx.x}$
3) If $i < N$:
    a) $\text{sum} = 0$
    b) For each non-zero $A_{ij}$: $\text{sum} \mathrel{+}= A_{ij} \cdot x_j$
```

c) $y[i] = \text{sum}$

4) **Output:** y

IV. Validation

To ensure correctness, the CUDA kernel was validated against the serial CPU implementation across all matrix sizes and sparsity levels. Results were compared element-wise using a tolerance of 1×10^{-6} to account for floating-point differences. All configurations returned `valid = 1`, confirming that the parallel implementation is numerically reliable.

V. Performance Evaluation

The Sparse Matrix-Vector Multiplication (SpMV) performance was evaluated for several sparsity levels: 0.5, 0.7, 0.85, 0.9, 0.95, 0.975, and 0.99. For each case, both serial CPU and CUDA GPU execution times were measured. Speedup is defined as the ratio of serial execution time to the best CUDA execution time. In this report, I focused on the results for sparsity levels 0.975 and 0.7, which illustrate representative performance trends for high and medium sparsity matrices.

A. Performance Tables

TABLE I
Sparsity = 0.975

Matrix Size	Serial (ms)	CUDA (ms)	Block Size	Speedup
512 x 512	0.0052	0.0067	32	0.77
1024 x 1024	0.0237	0.0089	32	2.66
2048 x 2048	0.1029	0.0109	64	9.47
4096 x 4096	0.5264	0.0429	32	12.27
8192 x 8192	2.2700	0.1566	32	14.50
16384 x 16384	10.350	0.7696	64	13.45

TABLE II
Sparsity = 0.7

Matrix Size	Serial (ms)	CUDA (ms)	Block Size	Speedup
512 x 512	0.0996	0.0320	32	3.11
1024 x 1024	0.4146	0.0593	32	6.99
2048 x 2048	1.7507	0.1320	32	13.27
4096 x 4096	7.7829	0.7684	32	10.13
8192 x 8192	32.331	2.1710	256	14.89
16384 x 16384	125.361	10.738	64	11.67

B. Analysis

The performance results show that the execution time of the serial SpMV implementation increases rapidly with matrix size, while the CUDA implementation scales much better due to parallel execution. Higher sparsity reduces the computation, resulting in lower execution times for both serial and GPU versions, whereas denser matrices increase the workload. Speedup generally improves with matrix size, with small matrices seeing little benefit from GPU acceleration due to overheads, while larger matrices achieve significant speedups. Block size also affects performance, as smaller block sizes provide good occupancy for most cases, but larger blocks are sometimes used for low-sparsity large matrices to maximize GPU throughput. Overall, CUDA-based SpMV consistently outperforms the serial version for medium to large matrices, and the speedup is influenced by matrix size, sparsity, and block configuration.

VI. Execution Time vs. Problem Size

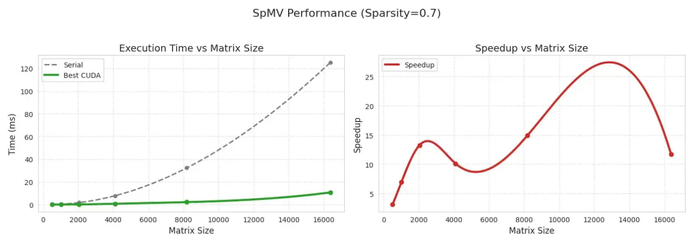

Fig. 1. SpMV performance for sparsity = 0.7.

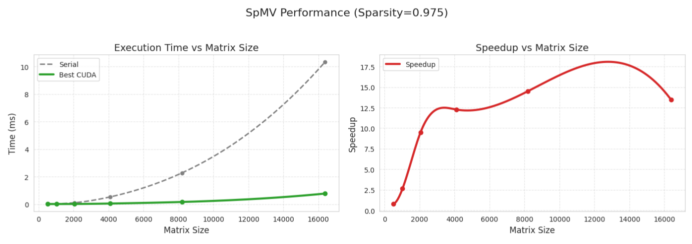

Fig. 2. SpMV performance for sparsity = 0.975.

For a sparsity of 0.7, serial execution reaches 125 ms for a 16,384 × 16,384 matrix, while CUDA completes it in under 11 ms, with peak speedup for medium-sized matrices. At a sparsity of 0.975, execution times are generally lower, and CUDA remains the fastest for medium-sized matrices.

VII. Analysis and Tuning Insights

The performance of sparse matrix multiplication depends strongly on the nature of the problem, particularly matrix size and sparsity. **First**, the choice of block size is critical: very sparse matrices benefit from small blocks, medium sparsity and matrix sizes favor intermediate blocks, and denser matrices achieve the best performance with large blocks. **Second**, speedup behavior is nonlinear. Smaller matrices show limited gains, while larger matrices exhibit disproportionate acceleration in most plots, highlighting thresholds where GPU execution becomes highly effective. Overall, GPU acceleration is most beneficial for large matrices with sufficient non-zero elements, whereas small and highly sparse matrices are more efficiently executed on the CPU.

VIII. Challenges Encountered

When attempted very large problem sizes, such as one billion rows/cols, the Colab session crashed before completion.

References

[1] B. Schmidt, J. Á. Gonzalez-Martinez, C. Hundt, M. Schlarb, *Parallel Programming: Concepts and Practice*, Morgan Kaufmann, 2017. ISBN 978-0128498903.