Parallelization Strategies for SpMV

Giacomo Mazzucchi: Mat: 248440, `giacomo.mazzucchi@studenti.unitn.it`,
GitRepo: `https://github.com/gmazzucchi/GPU-Computing-2025-248440`

***Abstract*—The sparse matrix-dense vector multiplication (SpMV) is a common linear algebra operation involving a sparse matrix and a dense vector. SpMV is widely used in many real-world applications such as Finite Element Analysis (large sparse stiffness matrices multiply with displacement vectors to simulate structural behavior).**

This deliverable discusses two main parallelization strategies on GPU with CUDA, using a baseline version on CPU and an optimized version using OpenMP. Results show that the thread-per-row strategy is preferable for very sparse matrices, on the other hand, the warp-per-row strategy becomes more effective as the matrix becomes moderately dense.

***Index Terms*—Sparse Matrix, SpMV, CUDA, Parallelization, Storage Format**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a key operation used in many areas like scientific computing, machine learning, and engineering. It involves **multiplying a sparse matrix**—a matrix mostly filled with zeros—**with a dense vector**. Because storing and computing all the zeros would be wasteful, special formats are used to store only the non-zero values. SpMV is at the core of many algorithms, such as solving large systems of equations or analyzing graphs.

The idea behind SpVM is simple, however the parallelization on GPU is not trivial. One major issue is that the **non-zero entries in the matrix are not placed regularly**: this is a problem for parallelization, because it means irregular memory access patterns. Moreover, different rows of the matrix can have very different numbers of non-zero elements, so divide the computational cost evenly among threads is difficult. Some threads end up doing a lot more work than others, which slows down everything. Another challenge is avoiding conflicts when multiple threads try to **write to the same location in memory**. Depending on the chosen algorithm, this can require extra steps like synchronization or the use of atomic operations, which can reduce performance.

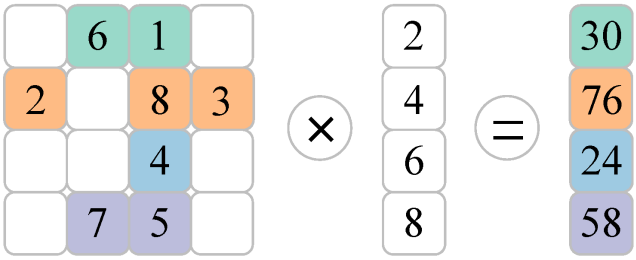

Fig. 1: Sparse Vector Matrix Multiplication example

II. Problem Statement

A. Storage Format

In this work, the assigned compression technique for sparse matrices is the **Compressed Sparse Row (CSR)** format. This format offers an efficient way to store and operate on sparse matrices by saving only the non-zero entries, thereby significantly reducing memory usage and computational overhead.

CSR uses three one-dimensional arrays:

- `values`: stores the non-zero elements of the matrix in row-major order.
- `col_indices`: stores the column indices corresponding to each non-zero value.
- `row_ptr`: an array of length $n + 1$ (where n is the number of rows), indicating the index in `values` where each row starts.

This layout allows constant-time row access and supports efficient implementation of sparse matrix-vector multiplication (SpMV), a fundamental operation in many scientific and engineering applications.

Consider the following sparse matrix A:

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 0 & 0 & 20 \\ 0 & 30 & 0 & 40 \\ 50 & 60 & 70 & 0 \end{bmatrix}$$

Its representation in CSR format is:

$$\begin{aligned} \texttt{values} &= [10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70] \\ \texttt{col\_indices} &= [0,\ 3,\ 1,\ 3,\ 0,\ 1,\ 2] \\ \texttt{row\_ptr} &= [0,\ 1,\ 2,\ 4,\ 7] \end{aligned}$$

Each entry in `row_ptr` points to the starting index in `values` for the corresponding row. For example, the third row ($i = 2$) starts at index 2 and ends at index 4 in the `values` array, meaning it contains two non-zero elements: 30 in column 1 and 40 in column 3.

Comparison with COO format. Had we used the Coordinate (COO) format, we would have needed to store:

$$\begin{aligned} \texttt{row} &= [0,\ 1,\ 2,\ 2,\ 3,\ 3,\ 3] \\ \texttt{col} &= [0,\ 3,\ 1,\ 3,\ 0,\ 1,\ 2] \\ \texttt{values} &= [10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70] \end{aligned}$$

While COO is simpler to construct, it requires storing an additional array of the same size as the non-zero count to keep track of the row indices, increasing memory usage. CSR is thus more space-efficient and enables better performance for row-wise operations such as SpMV.

B. Parallelization

First of all I implemented an approach to parallelize the operation on CPU using **OpenMP**, simply adding the `#pragma omp parallel for` directive. This directive tells the compiler to create threads before the loop and to destroy them after, effectively parallelizing the content of the loop.

Then I parallelized SpMV in **CUDA** using two strategies: **thread-per-row** and **warp-per-row**. In the first method, each thread computes a row's dot product. The warp-per-row strategy is instead an approach that allocates a warp (32 threads) to each row, improving load balancing and memory coalescing through parallel reduction of partial sums.

III. State of the Art

SpVM is a classic problem in parallel computing, and it is challenging due to irregular memory access and work imbalance. To address this, several formats and techniques have been proposed. Chen et al. developed the ESB format, which groups nonzeros into blocks based on matrix structure. This improves memory access and GPU utilization [1]. Kourtis et al. proposed combining different optimization steps like compressing indices and reordering rows to better use memory and threads. Their method works well across different GPUs [2]. Liu and Vinter introduced CSR5, a version of the CSR format that splits the matrix into small tiles and adds metadata to help each GPU thread work more efficiently. CSR5 improves parallelism and reduces waiting time between threads [3]. These formats aim to make SpMV faster by making better use of GPU memory bandwidth and computing power.

IV. Methodology and Contributions

My work was divided into three steps: first I reasoned about a parallelization strategy, then I applied it to a varied dataset, and finally I drew conclusions by capturing similarities among matrices of similar types.

I used the MMIO library [4] to parse matrices in Matrix Market format and the Eigen library to verify the correctness of the results, both for the baseline CPU implementation and for the GPU-parallelized versions of the sparse matrix-vector multiplication algorithm. To measure how long CUDA kernels take to run, I used a function called `profile_kernel`, that uses CUDA events to profile the kernels.

Regarding the actual kernel function, the SpMV using Thread-per-Row parallelization assigns one CUDA thread to each matrix row. Each thread computes the dot product between that row's non-zero elements and the input vector, storing the result in the output vector.

Parallelization using Warp-per-Row, on other hand, maps one warp (32 threads) to each matrix row. Each thread computes partial dot products by iterating over different elements of the row. The partial results are then summed using intra-warp shuffle reduction. Finally, thread 0 in each warp writes the result to the output vector.

Algorithm 1 SpMV using Thread-per-Row parallelization

procedure SPMV_THREAD_PER_ROW($rows$, col_idx, $values$, x, y, num_rows)
  $row \leftarrow blockIdx_x \cdot blockDim_x + threadIdx_x$
  if $row < num\_rows$ **then**
    $dot \leftarrow 0.0$
    for $j \leftarrow rows[row]$ to $rows[row+1]-1$ **do**
      $dot \leftarrow dot + values[j] \cdot x[col\_idx[j]]$
    end for
    $y[row] \leftarrow dot$
  end if
end procedure

Algorithm 2 SpMV using Warp-per-Row parallelization

procedure SPMV_WARP_PER_ROW($rows$, col_idx, $values$, x, y, num_rows)
  $warp\_id \leftarrow (blockIdx_x \cdot blockDim_x + threadIdx_x)\,/32$
  $lane \leftarrow threadIdx_x \bmod 32$
  if $warp\_id < num\_rows$ **then**
    $rs \leftarrow rows[warp\_id]$
    $row\_end \leftarrow rows[warp\_id+1]$
    $sum \leftarrow 0.0$
    for $j \leftarrow rs + lane$ to $row\_end - 1$ step 32 **do**
      $sum \leftarrow sum + values[j] \cdot x[col\_idx[j]]$
    end for ▷ Intra-warp reduction
    for $offset \leftarrow 16$ to 1 step $/2$ **do**
      $sum \leftarrow sum + \texttt{shfl\_down}(sum, offset)$
    end for
    if $lane = 0$ **then**
      $y[warp\_id] \leftarrow sum$
    end if
  end if
end procedure

Dataset	Rows	Columns	Non Zeros	Symmetric	Typology	COO to CSR Conv. Time	Naive CPU Exec. Time	Thread/Row Exec. Time	Warp/Row Exec. Time
pkustk14	151926	151926	7494215	✓	structural	213.45	80.83	76.38	13.21
wiki-Talk	2394385	2394385	5021410	✗	graph	98.03	123.19	47.39	62.10
cage14	1505785	1505785	27130349	✗	graph	414.42	137.78	73.73	62.51
rajat31	4690002	4690002	20316253	✗	circuit	300.35	105.57	29.70	130.73
bcsstk13	2003	2003	42943	✓	fluid	1.22	0.45	0.67	0.10
af23560	23560	23560	484256	✗	fluid	7.59	1.82	1.32	0.78
neos3	512209	518832	2055024	✗	optimization	41.14	15.34	3.19	12.22
ins2	309412	309412	1530448	✓	optimization	48.62	12.74	74.56	11.46
helm2d03	392257	392257	1567096	✓	2D/3D PDE	52.22	16.74	84.53	10.49
great-britain_osm	7733822	7733822	8156517	✓	graph	323.98	263.01	28.53	186.68

TABLE I: Dataset characteristics and SpMV execution times (in milliseconds).

Fig. 2: Execution Time

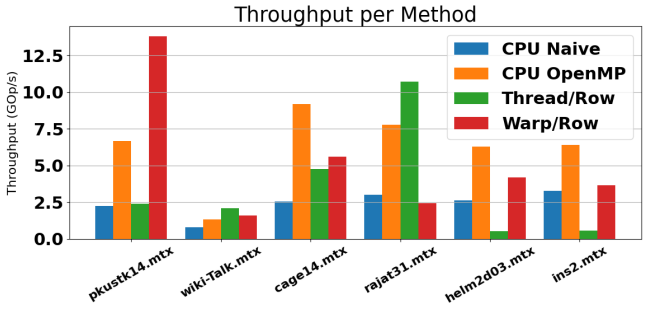

Fig. 3: Memory Throughput

V. System Description and Experimental Set-up

A. System Description

As the University cluster has been inaccessible for a long time, and intermittently at that, I used also the GPU available on my Ubuntu 22.04 Linux system to work regularly on the project.

System	Processor	Cores per Socket	RAM	Accelerator
University cluster	128 x AMD EPYC 9334 32-Core Processor	64 at 3.910 Ghz	16 GB	NVIDIA L40S
Personal Computer	Intel Core i7-4800MQ	8 at 3.70 GHz	16 GB	NVIDIA GeForce GT 730M

TABLE II: System details

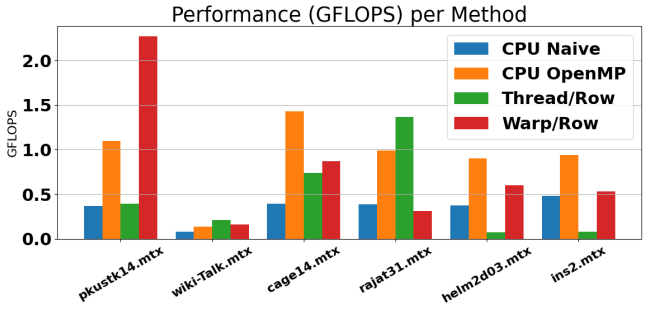

Fig. 4: GFLOPS

B. Dataset description

I used around twenty matrices downloaded from `https://sparse.tamu.edu/` and converted them into CSR format. The matrices were selected to cover a wide range of types and sizes, in order to test the performance of GPU implementations. In particular, the most interesting matrices are pkustk14, cage14, wiki-Talk, and great-britain_osm. The first two are relatively dense, with a high ratio of nonzeros to total elements. On the other hand, the latter two are especially sparse. The results will show significant performance differences between these cases.

VI. Experimental Results

Results show that the **thread-per-row strategy is preferable for very sparse matrices**, where the number of non-zero elements per row is low. In this case, assigning one thread per row ensures that each thread performs a small and roughly balanced amount of work, avoiding minimizing idle threads. The memory access pattern also remains simple and well-distributed across threads.

On the other hand, the **warp-per-row strategy becomes more effective as the matrix becomes moderately dense**, i.e., with a higher average number of non-zero entries per row. In these cases, a single thread in the thread-per-row model would accumulate too much work, causing load imbalance and longer execution times. By assigning an entire warp (32

threads) to process one row collaboratively, the workload is distributed among threads, leading to better use of GPU resources and faster computation.

The situation is **much more complex for OpenMP**, where no clear correlation emerges between matrix types and performance. Results are likely influenced by the specific structure and content of each matrix. Moreover, since OpenMP dynamically manages the number and organization of threads, assuming definitive conclusions becomes even more difficult.

VII. Conclusions

Using a few parallelization strategies (one simple one on the CPU and two more complex ones on the GPU), it is possible to improve performance, with results that follow logical analysis. **Checks with Eigen ensure that the developed algorithms are correct**. Since reading the matrix from input allows one to determine whether it is relatively dense or very sparse, it is possible to choose which GPU algorithm to use at runtime, therefore **improving performance in all cases**.

References

[1] G. Chen, B. Wang, J. Zhan, and L. Zhang, "Esb: A blocked format for evolving-streaming-based sparse matrix-vector multiplication on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1907–1920, 2017.

[2] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–50, 2017.

[3] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 845–854.

[4] R. Boisvert, R. Pozo, and K. Remington, "Matrix market exchange formats: mmio.c i/o library," https://math.nist.gov/MatrixMarket/mmio-c.html, 1996, national Institute of Standards and Technology.