Advanced CUDA Strategies to Parallelize 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 the optimization of a basic strategy used as a baseline (thread per row and warp per row), using more advanced CUDA techniques, for example shared memory, warp intrinsics, loop unrolling and read only cache. I implemented two versions: thread per row using loop unrolling and read only cache, and warp per row using shared memory. The first one does not generally improve results, meanwhile the second one improves significantly the performance, especially in relatively dense matrices.

***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.

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:

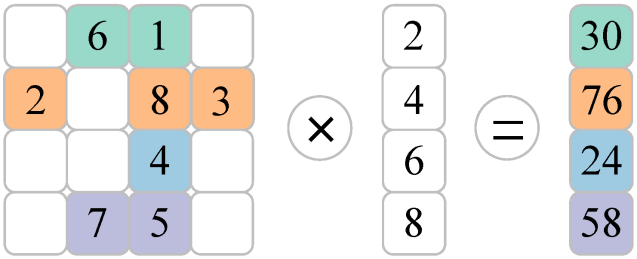


Fig. 1: Sparse Vector Matrix Multiplication example

- `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:

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

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:

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

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

The baseline strategies for SpMV in **CUDA** are **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.

C. Optimizations

The main principle is that global memory accesses are very slow and can severely limit performance. To address this, two key optimization techniques are applied.

Shared memory is used to cache portions of the input vector x that are frequently accessed by threads in the same block. Since shared memory has much lower latency than global memory, this reduces access time and improves throughput, especially when threads reuse the same data. **Read-only cache** allows the GPU to route loads through a cache optimized for read-only data, reducing pressure on global memory bandwidth.

In addition to memory optimizations, **loop unrolling** can be applied. This technique manually expands the loop body, reducing the number of iterations and loop-control instructions. It improves instruction-level parallelism and enables better pipelining.

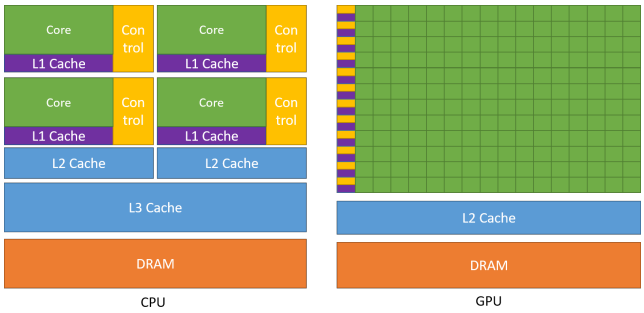


Fig. 2: CUDA Memory Diagram

III. State of the Art

SpMV is a classic problem in parallel computing, and it is challenging due to **irregular memory access**. To address this, solution have been attempted. Chen and other researchers developed the ESB format, which groups nonzeros into blocks based on matrix structure. This improves memory access and **GPU utilization** [1]. Kourtis and other researchers proposed combining different optimization steps like compressing indices and reordering rows to better use memory and threads. Their method works well across different GPUs as paper [2] reports. 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.

Another class of optimizations focuses on staging parts of the input vector in **shared memory** to reduce redundant memory accesses. This is useful in particolar when the nonzero column indices are localized or clustered. Bell and Garland proposed warp-centric SpMV, showing that assigning one warp per row can better balance work and enable coalesced memory access [4]. More recent work by Anzt et al. explored performance bounds for sparse matrix kernels and highlighted the benefits of caching input vector values in shared memory for certain matrix patterns [5]. The idea is that when several threads access neighboring elements of the input vector, caching those elements can improve speed. This technique is also supported by NVIDIA's cuSPARSE and CUTLASS libraries, which internally implement shared memory buffering. Using shared memory doesn't always help, but it can speed things up a lot for matrices where most nonzero values are close to the diagonal [6].

IV. Methodology and Contributions

For Deliverable 1, I used thread per row for sparse arrays and warp per row for relatively dense arrays. Of these strategies, for complete clarity, I attach the pseudocode in the algorithm 1 and algorithm 2. After that I tried to apply improvements by reasoning about possible inefficiencies particularly related to the memory accesses described in section I.

As the first deliverable, I used the MMIO library [7] to parse matrices in Matrix Market format and **the Eigen library to verify the correctness of the results**. To measure how long CUDA kernels take to run, I used a function called `profile_kernel`, that internally calls CUDA events to profile the kernels. I do a **startup warmup** when I start the program and compute **geometric mean** over more kernel runs to prove robustness of the results. Unfortunately, I could not successfully run the `ncu` program on the University Cluster due to a permission error:

```
==ERROR== ERR_NVGPUCTRPERM - The user
does not have permission to access NVIDIA
GPU Performance Counters on the target
device 0. For instructions on enabling
permissions and to get more information
see
```

For this reason, I profiled my implementation with `nsys`, knowing well however the limitations of the latter program when compared to the former.

Regarding the actual kernel function, as the first deliverable, the SpMV using Thread-per-Row parallelization (algorithm 1) 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.

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

Parallelization using Warp-per-Row (algorithm 2), 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 2 SpMV using Warp-per-Row parallelization

procedure SPMV_WARP_PER_ROW($rows$, col_idx, $values$, x, y, num_rows)
  $warp\_id \leftarrow \frac{\texttt{blockIdx\_x}\cdot\texttt{blockDim\_x}+\texttt{threadIdx\_x}}{32}$
  $lane \leftarrow threadIdx_x \bmod 32$
  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

A. Thread-per-Row with Loop Unrolling and `__ldg()` Read-Only Cache

The kernel (algorithm 3) initializes an accumulator `sum = 0.0` and iterates over the non-zero elements of the row. To improve performance, the loop is **unrolled by a factor of 4**. This reduces loop control overhead and helps the compiler generate more efficient instructions. The implementation is very similar to the baseline and its purpose is to see how much impact these strategies alone actually have.

B. Warp-per-Row with Shared Memory Caching

As any warp per row strategies, every row is processed by a group of **32 threads**. To speed up the calculation, the kernel (algorithm 4) tries to load part of the input vector into fast shared memory. It first finds the smallest and largest column indices used in that row. This tells us what part (or tile) of the vector we need. **If this tile is small enough (1024 or fewer values), the warp loads it into shared memory**. For the tile size I chose 8KB to be conservative enough and support a variety of GPUs, but it can be easily tuned accordingly to the hardware specification. Then, instead of reading from global memory (which is slower), the warp reads from shared memory. This makes the multiplication faster. Each thread in the warp computes part of the sum, and then they combine their results using warp shuffle instructions. In the end, one thread writes the final result to the output vector.

C. Profiling

As I mentioned earlier, since `ncu` was not available, I also profiled the implemented versions with `nsys`, checking the execution time, call stack, GPU utilization and other parameters. With the available information, I confirmed only the profiling results obtained with cuda events in the code, without any additional insights.

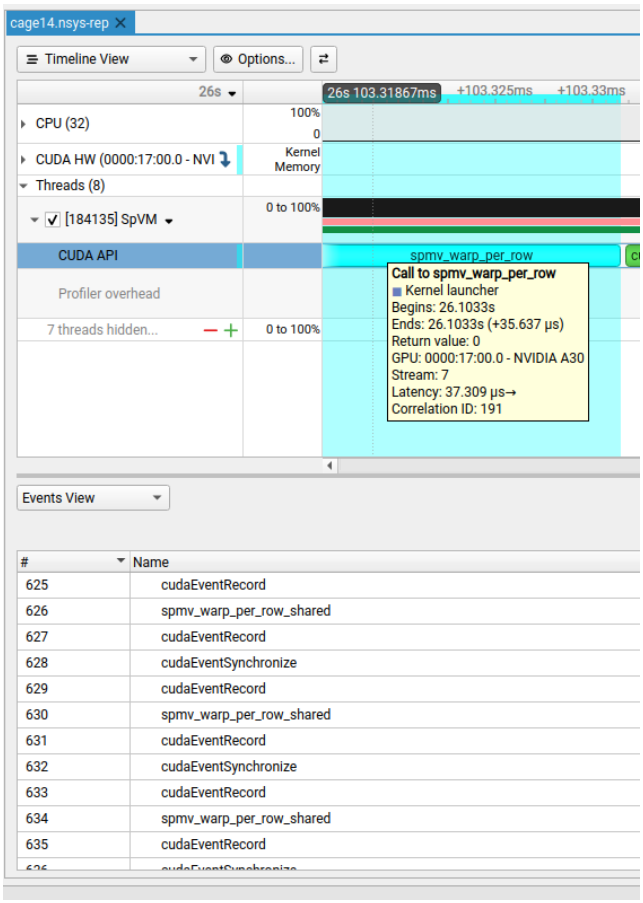


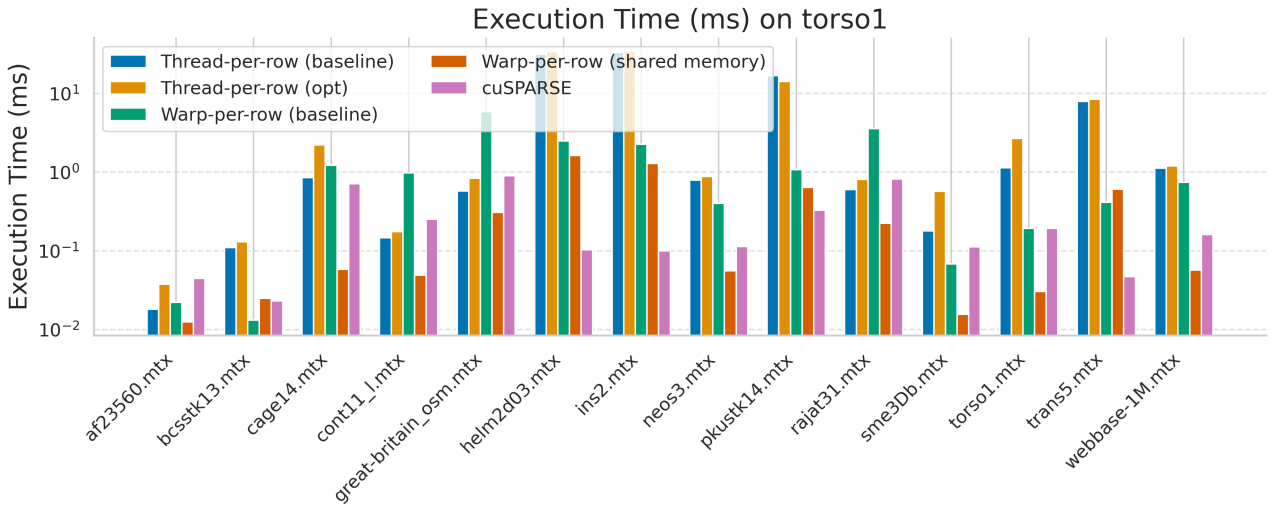
Fig. 3: Screenshot from the nsys analysis

Algorithm 3 Thread-per-Row SpMV with Loop Unrolling and Read-Only Cache

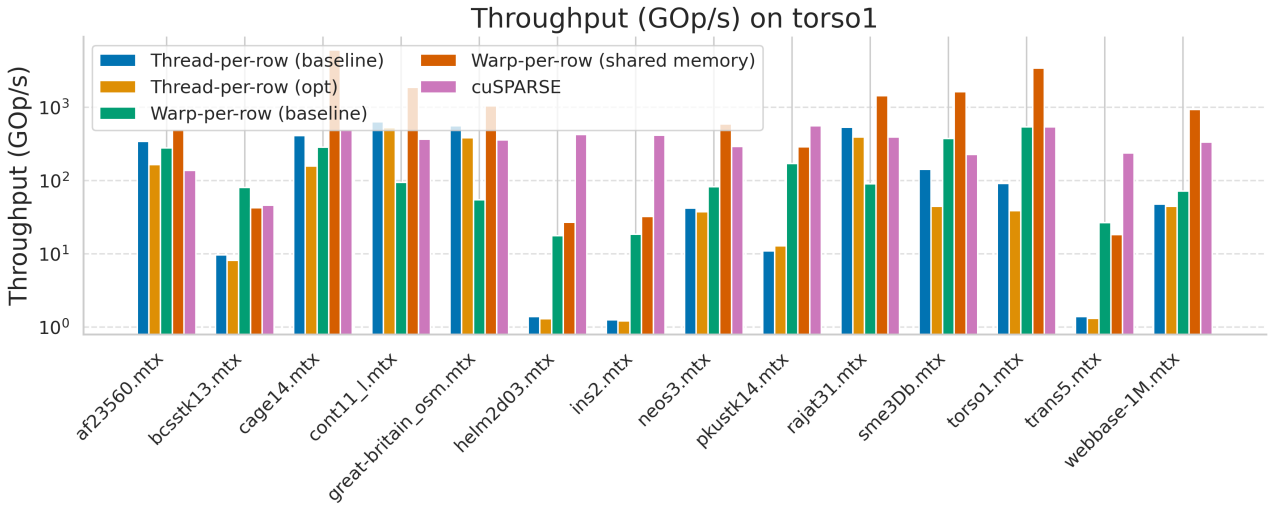
procedure THREADPERROWSPMV(CSR matrix $(row\_ptr, col\_idx, values)$, input vector x, output vector y)
  for each thread assigned to a row i **do**
    Identify the range of nonzeros $(start, end)$ for row i
    Initialize local accumulator $sum \leftarrow 0$
    for nonzeros in chunks of 4 **do**
      Load 4 column indices and 4 values
      Multiply each value with corresponding x using read-only cache
      Accumulate the results
    end for
    for remaining nonzeros **do**
      Multiply value with corresponding x and accumulate
    end for
    Write sum to output $y[i]$
  end for
end procedure

Algorithm 4 Warp-per-Row SpMV with Shared Memory

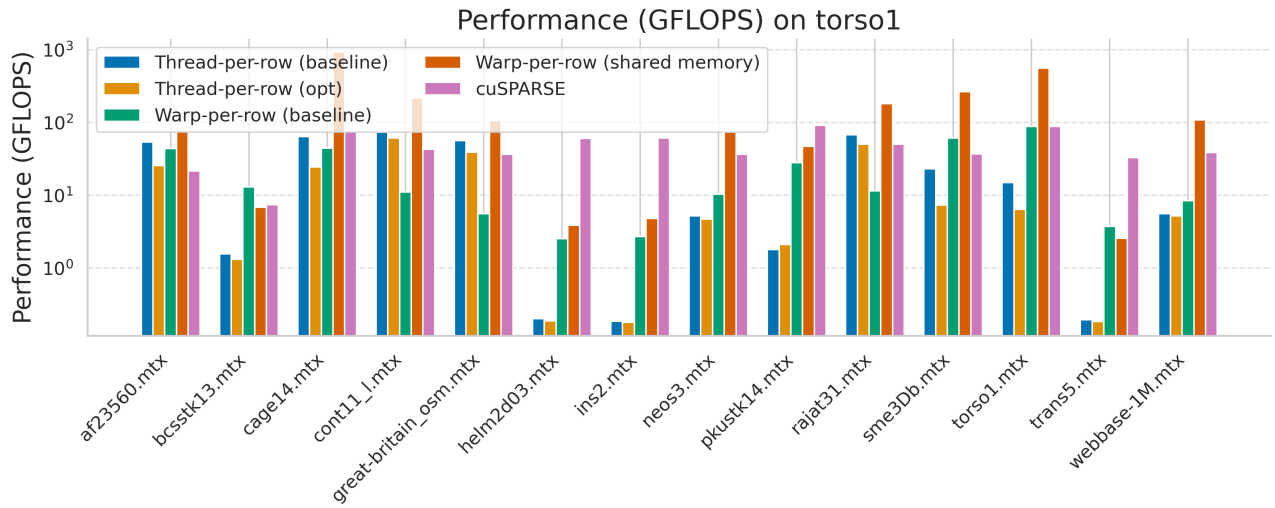
procedure WARPPERROWSPMVSHARED(CSR matrix $(row\_ptr, col\_idx, values)$, input vector x, output vector y)
  for each warp assigned to a row i **do**
    Identify nonzero range $(start, end)$ for row i
    Compute the min and max column indices accessed in this row
    Attempt to stage the required portion of x in shared memory
    if tile fits in shared memory **then**
      Load x tile cooperatively into shared memory
      for nonzeros in row i **do**
        If $x[col]$ is cached, access shared memory
        Else, fall back to global memory
        Multiply and accumulate
      end for
    else
      Fallback: access x directly from global memory
      for nonzeros in row i **do**
        Multiply and accumulate
      end for
    end if
    Perform warp-level reduction of partial sums
    Write final result to output $y[i]$
  end for
end procedure



(a) Execution Time



(b) Memory Throughput



(c) GFLOPS

Fig. 4: Comparison of execution time, memory throughput, and performance (GFLOPS) across different SpMV implementations.

Dataset	Rows	Cols	NNZ	Sym	Type	CSR (ms)	cuSPARSE (ms)	TPR Base (ms)	TPR Opt. (ms)	WPR Base (ms)	WPR Shared (ms)
pkustk14	151926	151926	7494215	✓	structural	116.39	0.33	16.72	14.21	1.07	0.64
cage14	1505785	1505785	27130349	✗	graph	214.72	0.71	0.85	2.22	1.23	0.06
rajat31	4690002	4690002	20316253	✗	circuit	190.42	0.81	0.60	0.81	3.56	0.22
bcsstk13	2003	2003	42943	✓	fluid	0.82	0.02	0.11	0.13	0.01	0.03
trans5	116835	116835	798312	✗	simulation	7.52	0.05	7.92	8.38	0.41	0.60
af23560	23560	23560	484256	✗	fluid	4.44	0.04	0.02	0.04	0.02	0.01
cont11_l	41600	41600	1965616	✓	optimization	53.85	0.25	0.15	0.18	0.98	0.05
sme3Db	472850	472850	9373548	✗	structural	17.14	0.11	0.18	0.57	0.07	0.02
TSOPF_RS_b2383	2383	2383	60119	✗	power	129.37	0.32	0.48	2.23	0.25	0.01
helm2d03	392257	392257	1567096	✓	2D PDE	27.88	0.10	31.40	33.66	2.48	1.63
great-britain_osm	7733822	7733822	8156517	✓	graph	186.63	0.90	0.58	0.83	5.87	0.31
webbase-1M	1000005	1000005	3105536	✗	web	32.73	0.16	1.12	1.20	0.74	0.06
neos3	512209	518832	2055024	✗	optimization	20.74	0.11	0.79	0.88	0.40	0.06
ins2	309412	309412	1530448	✓	optimization	27.03	0.10	33.06	34.38	2.26	1.29
torso1	116158	116158	8516501	✗	biomedical	67.24	0.19	1.14	2.68	0.19	0.03

TABLE I: Dataset characteristics and SpMV execution times (in milliseconds) for various CUDA implementations.

V. System Description and Experimental Set-up

A. System Description

As the University cluster has been inaccessible for some periods of time (and I had problems with VPN), I used also the GPU available on my Ubuntu 22.04 Linux system to work regularly on the project. The data collected, however, has been gathered exclusively from the University cluster (edu01 or edu02 does not depend on me).

System	Processor	Frequency	Accelerator
Personal Computer	Intel Core i7-4800MQ	3.70 GHz	NVIDIA GeForce GT 730M
edu01	Intel Xeon Silver 4309Y	2.800 GHz	NVIDIA A30
edu02	AMD EPYC 9334 32-Core Processor	3.910 GHz	NVIDIA L40S

TABLE II: System details

B. Dataset description

Exactly as the first deliverable, 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, ins2 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.

VI. Experimental Results

In the first deliverable, I observed that for very sparse matrices, the Thread-Per-Row (TPR) strategy was more effective, while for relatively dense matrices, the Warp-Per-Row (WPR) approach worked better. This is because assigning one thread to an entire row becomes too expensive when the row is long or dense.

The **Warp-Per-Row** strategy with shared memory performs very well, especially on large matrices with a structured pattern or narrow bandwidth, such as `cage14` and `helm2d03`. The key advantage is that part of the input vector is loaded into shared memory, reducing the need for repeated slow global memory accesses. Warp-level processing also helps balance the workload better across threads, especially for rows with many elements.

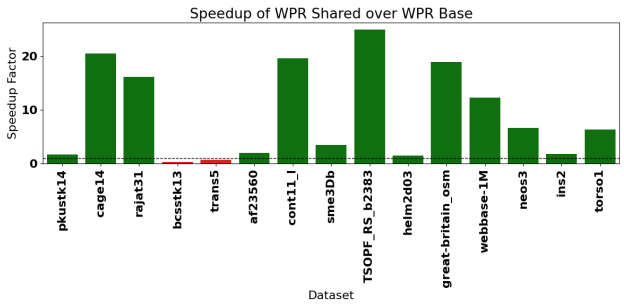


Fig. 5: Shared memory speedup

Shared memory is most effective when the nonzero elements in a row are close together. In these cases, we can take advantage of data locality to make memory access much faster. However, if the rows are very irregular or too much data is needed (more than 48–96 KB per streaming multiprocessor), shared memory may not help and can even slow things down.

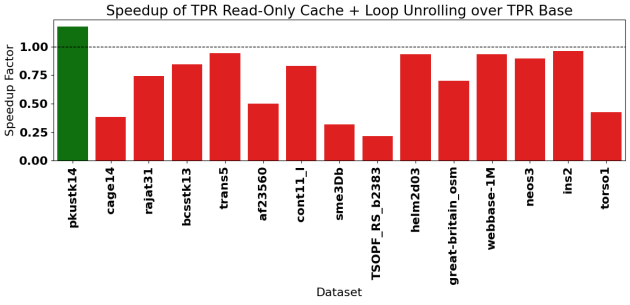


Fig. 6: TPR optimized speedup

In contrast, the **Thread-Per-Row (TPR)** approach does not scale well. It has problems with memory access patterns and does not fully use the parallel power of the GPU. Even with optimizations, TPR performs poorly on matrices with rows of very different lengths or extremely large sizes, like `great-britain_osm`. Some techniques like loop unrolling and using the read-only cache also did not improve performance. One possible reason is that modern GPUs, like the NVIDIA L40S and A30, may already apply similar optimizations automatically. These GPUs have deep pipelines, which should benefit from loop unrolling, but in practice, it did not help much in this case.

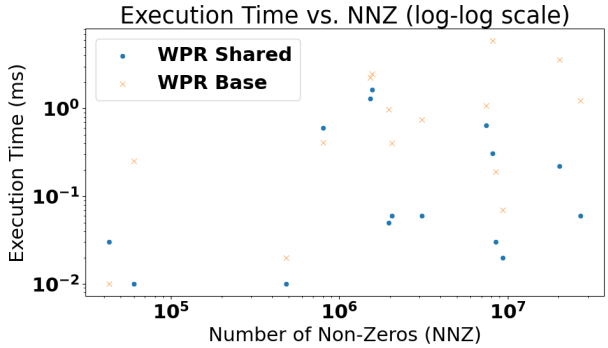


Fig. 7: Execution Time vs Matrix Sparsity

VII. Conclusions

In contrast to the first deliverable, with the use of shared memory, **the warp-per-row strategy clearly becomes the best**. This holds true both for relatively dense matrices (like cage14) and for extremely sparse ones (like great-britain_osm), where the version using only global memory performs much worse. The optimized thread-per-row version gave poor results, while the cuSPARSE version was still the best for almost all the tested matrices. Final note, because every matrix multiplication in every run is verified using Eigen, the correctness of the output is always verified.

References

[1] Y. Chen *et al.*, "Esb: An efficient sparse block format for sparse matrix-vector multiplication on gpus," in *Proceedings of the ACM Conference on Supercomputing*, 2009.

[2] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th Conference on Computing Frontiers*, 2008.

[3] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, 2015.

[4] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," NVIDIA Technical Report NVR-2008-004, Tech. Rep., 2008.

[5] H. Anzt, J. Dongarra *et al.*, "Designing efficient sparse matrix kernels on gpus," *ACM Transactions on Mathematical Software (TOMS)*, 2017.

[6] K. Madduri *et al.*, "An optimized sparse matrix-vector multiplication algorithm for large-scale graph mining on gpus," *Proceedings of IEEE IPDPS*, 2019.

[7] 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.