CS406: Parallel Computing
Homework 2 — CUDA Sparse Matrix–Vector Multiply (SpMV)

Sabancı University

December 2, 2025

Problem Description

In this homework you will implement a high-performance Sparse Matrix × Dense Vector multiplication (SpMV) on the GPU using CUDA. As in HW1, you will read real sparse matrices in Matrix Market format, compute $y \leftarrow Ax$, measure performance (GFLOPS), and validate results with a checksum. When dealing with sparse matrices, it is generally better to use compressed storage formats like CSR (Compressed Row Storage) (or known as CSR: compressed sparse row, they are the same). In this homework, just like Homework 1, we have provided a code that reads input matrices into CSR format (into CPU, you will do data transferring to GPU as it is explained later.)

Your goal is to (1) copy the necessary data (i.e. matrices and input vector) to the GPU, (2) complete the provided CUDA kernel (*__global__ void spmv_cuda_kernel()*) that will do the SpMV. You may choose to implement one or more kernel(s) depending on your algorithm. Just make sure to launch your kernel(s) in the *spmv* function.

CSR Arrays Recap. Storing a sparse matrix in CSR uses three arrays: `CSR_ptrs`, `CSR_colids`, and `CSR_values`. For an $n \times m$ sparse matrix:

- `CSR_ptrs`: This array has size $n+1$. The first element is $\texttt{CSR\_ptrs}[0] = 0$. For $i \geq 1$,

$$\texttt{CSR\_ptrs}[i] \;=\; \texttt{CSR\_ptrs}[i-1] \;+\; \text{nnz}(\text{row } i-1),$$

  i.e., it stores the cumulative number of nonzeros up to (but not including) row i. Equivalently, $\texttt{CSR\_ptrs}[i] = \sum_{r=0}^{i-1} \text{nnz}(\text{row } r)$. Thus, if you traverse nonzeros row-wise, $\texttt{CSR\_ptrs}[i]$ is the count of nonzeros until row i. (Note: entries for row i live in indices $k \in [\texttt{CSR\_ptrs}[i], \texttt{CSR\_ptrs}[i+1])$.)

- `CSR_values`: This array has length nnz (the total number of nonzeros). Scanning the matrix row-wise, store each nonzero value in order into `CSR_values`.

- `CSR_colids`: This array also has length nnz. For every stored nonzero value, record its column index in `CSR_colids` at the same position, giving a one-to-one correspondence between `CSR_values` and `CSR_colids`.

Datasets / Matrices

You will use the following matrices from the SuiteSparse Matrix Collection (Matrix Market `.mtx` format) for testing / evaluating your implementation (Same matrices as HW1):

- DIMACS10/netherlands_osm (road network; low degree, planar-like)
- Delaunay_n21 (geometric synthetic graph)
- SNAP/soc-LiveJournal1 (social network; heavy-tailed degree)

What is Provided

We provide a single CUDA skeleton file `spmv_cuda.cu` that:

1. Parses a Matrix Market file on the CPU and builds global CSR arrays (`G_row_ptr`, `G_col_idx`, `G_vals`).
2. Exposes hooks for you to:
   - build device (GPU) memory from the global CSR,
   - implement your CUDA kernel(s),
   - launch/timer logic on the host.
3. Prints GFLOPS and a checksum to help validate correctness.

Your Tasks (What You Must Implement)

1. **Device build.** Allocate/copy CSR (and any persistent vectors you keep) to device memory inside `device_build_from_global_csr()`.
2. **CUDA kernel(s).** Implement `__global__ void spmv_cuda_kernel(...)`. You may use this as a controller that calls other kernels or implement multiple kernels. Design is up to you.
3. **Host wrapper.** In `spmv(...)` perform any H2D/D2H transfers you need, launch your kernel(s), and use the provided CUDA event timing to measure **kernel time**. Fill `y_host` so the checksum is meaningful.

Important Constraints

- **Library usage:** You may use external / online code as long as you cite it properly. However, do *NOT* use direct SpMV implementations from libraries such as cuSPARSE for the graded kernel. You must code the SpMV algorithm yourself.
- **Timing:** Use the provided event timing around your launches. Exclude file I/O and preprocessing from kernel time. This will the most important aspect of your implementation. The final grade will mostly depend on your performance as in Homework 1.
- **GFLOPS:** Report GFLOPS $= \dfrac{\texttt{nnz}}{\text{time (s)} \cdot 10^9}$ for a single SpMV (or the average if you time multiple launches).

- **Correctness:** We will sanity-check via the printed checksum; your result must be consistent across runs.
- **Matrices are NOT necessarily binary:** You may *not* assume the matrices are binary.
- **Single evaluation strategy requirement (SO IMPORTANT).** If you implement more than one algorithm and/or preprocessing method, you must designate *ONE* strategy as the default evaluation path to be used for *ALL* matrices (Choose the best strategy of yours to be default). Provide explicit build/run instructions that select this default. The grader will *not* choose among optional methods. When invoked as `./spmv_cuda.out matrix.mtx` with no extra flags, your program must run this default strategy.

Build & Run

Compiling (example):

```
nvcc -O3 -std=c++17 spmv_cuda.cu -o spmv_cuda
```

Running:

```
./spmv_cuda path/to/matrix.mtx
```

What to Report

For each matrix:

- Kernel time (milliseconds/seconds) measured with CUDA events,
- GFLOPS as defined above,
- The printed checksum value.

Describe your kernel design in detail and any data-layout choices that improved performance.

Deliverables

Submit a single zip (`surname_name.zip`) containing:

1. Your source code(s).
2. A PDF report that complies the following instructions:
    - It should include the pre-processing and implementation tricks you used (at least a few sentences about how you attacked the parallelization problem). Please notice that your code will be graded considering correctness, scalability and speed.
    - It needs to contain build and execution details – e.g., how did you compile the code (which instruction set, optimization parameter etc.) and how I can reproduce your results.
    - 1-2 pages reports are usually not OK.
    - Do not increase the number of pages by leaving more spaces in between lines.

- Perform multiple experiments and report both average execution times and GFLOPS across your experiments.
- Charts and tables must have correct and appropriate captions.

GOOD LUCK!