

Project: Sparse Matrix Sparse Matrix Multiplication (100 points)

Due Date: December 9, 2025 (Tuesday), no extensions are allowed

Due Time: 11:59 pm on Gradescope

Collaborators: You can do this project in groups of 3.

Note: **GENERATIVE AI IS ALLOWED IN THIS ASSIGNMENT. HOWEVER, YOU MAY USE ONLY AUTHORIZED AI TOOLS PROVIDED BY UVA (<https://copilot.microsoft.com> (choose UVA Work option)). See the AI policy on Canvas for more details.**

1. Project Overview

This project implements sparse matrix–sparse matrix multiplication (SpMSpM) of a square sparse matrix **A** with a square matrix **B** itself, i.e., it computes

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}.$$

The baseline implementation runs on the CPU, and a GPU variant is to be implemented for performance comparison. Input matrices are stored in compressed sparse row (CSR) format, and the output matrix is accumulated in coordinate (COO) format. The codebase also includes utilities for reading matrices from files, managing host and device memory, and timing different phases of the computation.

2. Code Structure

2.1. `matrix.cu`

The corresponding implementation in `matrix.cu` provides:

- Host-side creation and destruction routines for COO and CSR matrices (e.g., `createEmptyCOOMatrix`, `createCSRMatrix`).
- Device-side creation and destruction routines (e.g., `createEmptyCOOMatrixOnGPU`, `createEmptyCSRMatrixOnGPU` and their `free*OnGPU` counterparts).
- Utility functions to copy data between host and device (`copyCSRMatrixToGPU`, `copyCOOMatrixFromGPU`) and to clear device-side matrices (`clearCOOMatrixOnGPU`).

These abstractions allow the main program and the GPU kernels to work with sparse matrices without manually managing raw device pointers.

2.2. `kerneli.cu`

The file `kerneli.cu` contains the skeleton for each implementation. Use them as follows:

- `kernel0.cu`: GPU parallel implementation without optimizations.
- `kernel1.cu`–`kernel4.cu`: GPU parallel implementation with incremental optimizations. Each file accumulates all optimizations from the previous kernel and adds one additional optimization.

You need to implement at least two optimizations on top of the GPU parallel kernel. Make sure `kernel4.cu` has the fastest version, even if you just implemented 2 versions, copy the fastest one to `kernel4.cu`.

No points will be given if the program fails to compile or run; in that case, we will not check your code to give partial points.

2.3. `main.cu`

The main driver provides:

- A CPU baseline implementation `spmspm_cpu` that performs SpMSpM using CSR input and COO output. It typically uses a temporary accumulator per row, scans the relevant rows of the right-hand-side matrix, and then compresses the nonzeros into the output COO structure.

- A verification routine `verify` that compares GPU and CPU results. By default, a quick sanity check is used; passing `-v` enables exact element-wise comparison after sorting both output matrices.
- Argument parsing and control logic to select which GPU versions to run and which input matrix file to read.
- Setup of host- and device-side data structures, as well as timing of CPU execution, memory transfers, and each GPU kernel.

2.4. README.md and Makefile

The `README.md` file provides a short textual description of the project and summarizes compilation and execution options. The `Makefile` builds a single executable `spmspm` by compiling all `.cu` files with `nvcc` and linking them together.

3. Grading

Final submissions will be assessed for correctness and execution time. If all four optimizations produce correct results on the GPU, they are eligible to receive full credit for the project, pending report evaluation (see next section). The two teams with the fastest execution times (on a range of GPUs) will receive an additional 10 points on the final exam (however, the maximum score on the final exam remains at 100). Execution times will be measured by the course staff using a range of GPUs available in the UVA CS department. To prepare, test your code on as many different GPUs as possible and consider applying multiple optimizations to improve performance.

Grade distribution:

- Code: 60%
- Report: 40%

4. Submission

Write a report explaining your parallel GPU implementation as well as each optimization implemented. Describe in detail and include how each optimization benefits performance in this case, and how did you implement them. Include a table of the execution time observed in your report too for each GPU used. **Submit your code and the report to GradeScope as ZIP folder.**