CS406: Parallel Computing – Homework 2
High-Performance CUDA SpMV: Adaptive Binning vs. Heuristic Tuning

Kerem Tufan
ID: 32554

December 17, 2025

Abstract

This project explores the optimization of Sparse Matrix-Vector Multiplication (SpMV) on NVIDIA GPUs. The primary challenge in SpMV is the irregular memory access pattern caused by varying row lengths. We implemented and compared two strategies: (1) **Adaptive Row Binning**, which aims for perfect load balance via indirect addressing, and (2) **Heuristic-Based Tuning**, which prioritizes low memory latency using a "Threads-Per-Row" (TPR) heuristic. Experimental results on an NVIDIA Titan X (Pascal) show that the memory overhead of binning is prohibitive for sparse matrices. The Heuristic strategy, combined with RCM reordering and Texture Cache optimizations, proved superior for road networks, achieving **31.25 GFLOPS** and outperforming the NVIDIA **cuSPARSE** library by approximately **34%**.

1 Introduction

Sparse Matrix-Vector Multiplication ($y = Ax$) is a memory-bound operation. On SIMT architectures like GPUs, performance is dictated by two factors:

1. **Memory Coalescing:** Ensuring adjacent threads access adjacent memory addresses.
2. **Warp Divergence:** Handling rows with vastly different non-zero counts without stalling the GPU.

In this assignment, I developed a "Vector CSR" solver that addresses these issues. I applied **Reverse Cuthill-McKee (RCM)** reordering to improve spatial locality and evaluated two distinct scheduling algorithms to handle row irregularity.

2 Optimization Methodology

2.1 Preprocessing: RCM Reordering

Before transferring data to the GPU, I applied the RCM algorithm to the input matrix.

- **Problem:** In standard CSR, column indices are scattered. When a warp reads the input vector x, it triggers multiple cache lines, wasting bandwidth.

- **Solution:** RCM renumbers nodes to cluster non-zeros around the diagonal. This ensures that threads in a warp access the vector x in a localized range, significantly increasing L1/L2 cache hit rates.

2.2 Strategy A: Adaptive Row Binning (Experimental)

This strategy attempts to solve the load imbalance problem explicitly.

- **Concept:** We preprocess rows into bins (Sparse, Medium, Dense) and launch separate kernels. This ensures a warp never processes a short row alongside a long row.
- **Implementation:** The kernel uses an indirect lookup array: `row = row_ids[tid]`.
- **Bottleneck:** My analysis revealed that for ultra-sparse matrices (like road networks), the cost of reading `row_ids` from global memory is higher than the actual floating-point computation. The latency of indirect addressing significantly degrades performance.

2.3 Strategy B: Heuristic-Based Tuning (Selected)

Instead of reordering rows, this strategy tunes the kernel configuration to the matrix density. The host code calculates the average non-zeros (avg_nnz) and dispatches a template-specialized kernel.

- **TPR=2 (Road Networks):** Assigns only 2 threads per row. This reduces the number of idle threads in a warp from 31 (in a naive kernel) to nearly zero.
- **TPR=4 (Meshes):** A middle ground for regular grids.
- **TPR=32 (Social Networks):** Uses the full warp for heavy rows, relying on hardware threading to hide latency.

2.4 Kernel-Level Optimizations

Both strategies utilize the following low-level CUDA optimizations:

1. Read-Only Texture Cache (`__ldg`)

Standard global loads go through the L1 cache, which is volatile. I used the `__ldg()` intrinsic for reading `row_ptr`, `col_idx`, and `x`. This forces the GPU to use the Read-Only (Texture) cache pipeline, which is optimized for spatial locality and relieves pressure on the L1 cache.

2. Warp Shuffle Reduction

Traditional reductions use Shared Memory and `__syncthreads()`. I replaced this with `__shfl_down_sync()`. This allows threads to exchange data directly via registers, eliminating synchronization barriers and shared memory bank conflicts.

3 Experimental Setup

All benchmarks were conducted on the following high-performance computing environment:

- **GPU:** NVIDIA TITAN X (Pascal Architecture)

– CUDA Cores: 3584

– Memory: 12 GB GDDR5X (Bandwidth $\approx$ 480 GB/s)

– **Max Clock:** 1911 MHz (Graphics), 5005 MHz (Memory)

- **Software Stack:**

  – Driver Version: 545.23.08

  – CUDA Version: 12.3

  – Compiler Flags: `-O3 -std=c++17`

- **Clock Policy:** A warm-up loop of 200 iterations was used to stabilize the GPU clocks at their maximum application frequency ($\approx 1417 - 1911$ MHz) and mitigate cold-start latency.

- **Baseline:** Results are compared against NVIDIA's optimized **cuSPARSE** library.

4 Results and Analysis

4.1 Case 1: Netherlands OSM (Ultra-Sparse)

Topology: Road network (Avg degree $\approx$ 2). Irregular structure but very low density.

Analysis: This was the decisive test. Strategy A failed because the overhead of "Binning" (indirect memory access) was too high for such short rows. Strategy B, configured with **TPR=2**, minimized thread waste and maximized bandwidth. Notably, my implementation achieved **31.25 GFLOPS**, significantly outperforming the optimized cuSPARSE library.

Table 1: Netherlands OSM Results

Method	**Time (ms)**	**GFLOPS**
Strategy A (Binning)	32.10	15.19
cuSPARSE (Ref)	20.92	23.34
Strategy B (TPR=2)	**15.62**	**31.25**

Checksum (Strategy B): 9.136285×10^{30}

4.2 Case 2: Delaunay n21 (Regular Mesh)

Topology: Geometric mesh (Avg degree $\approx$ 6). Very regular row lengths.

Analysis: Load balancing was not a major issue here due to the regularity of the mesh. While NVIDIA's **cuSPARSE** proved slightly faster (40.33 GFLOPS) due to closed-source micro-optimizations, our Strategy B (TPR=4) remained highly competitive at **38.31 GFLOPS**, proving that the heuristic approach is valid for structured grids.

Table 2: Delaunay n21 Results

Method	Time (ms)	GFLOPS
Strategy A (Binning)	43.80	28.74
cuSPARSE (Ref)	**31.20**	**40.33**
Strategy B (TPR=4)	32.84	38.31

Checksum (Strategy B): 2.888635×10^{47}

4.3 Case 3: soc-LiveJournal1 (Scale-Free)

Topology: Social network. Power-law distribution (some rows have > 1000 non-zeros).

Analysis: This case represents the toughest challenge due to heavy tails. Interestingly, our Heuristic Strategy (Strategy B) achieved **10.54 GFLOPS**, effectively matching the more complex Binning Strategy. This indicates that the TPR=32 kernel is robust enough to handle irregular workloads without the need for explicit load balancing. Although cuSPARSE (11.10 GFLOPS) holds a slight lead, our solver delivers competitive performance with a much simpler codebase.

Table 3: soc-LiveJournal1 Results

Method	Time (ms)	GFLOPS
Strategy A (Binning)	655.20	10.54
cuSPARSE (Ref)	**622.00**	**11.10**
Strategy B (TPR=32)	654.68	10.54

Checksum (Strategy B): 1.973638×10^{153}

5 Numerical Stability

The checksums for Strategy B (shown below the tables) matched the CPU baseline verification. Minor deviations in the least significant digits are expected due to the **non-associative nature of floating-point arithmetic** ($(a + b) + c \neq a + (b + c)$). The parallel reduction tree in CUDA sums elements in a different order than the serial CPU loop, but the result is algorithmically correct.

6 Conclusion

This study demonstrates that for GPU SpMV, **simplicity and memory latency often outweigh perfect load balancing**.

- **Indirect Addressing is Costly:** The "Adaptive Binning" strategy failed on sparse matrices because the cost of fetching row indices exceeded the computation time.
- **Heuristics Win:** The "Heuristic Tuning" strategy (Strategy B) provided the best balance. By aggressively tuning for road networks (TPR=2) and using RCM for locality, we achieved a **1.34x speedup over cuSPARSE**.

7 Reproduction: Build and Execution

To reproduce the results, compile with `-O3` to enable vectorization and run the solver. The code automatically detects the matrix type and selects the optimal heuristic.

```
# 1. Compile (Requires CUDA 12.3 and C++17)
nvcc -O3 -std=c++17 spmv_cuda.cu rcm.cpp -o spmv_cuda

# 2. Run (Example for LiveJournal)
./spmv_cuda ./soc-LiveJournal1.mtx

# 3. Output will display Time, GFLOPS, and Checksum.
```