# Optimized CUDA Kernels for CSR sparse matrix multiplication

## CIS6010, Fall 2025

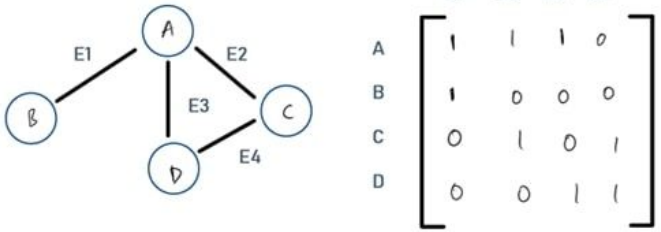
Anh Tran, Aditya Agarwal, Rajath Ramana
December 08, 2025

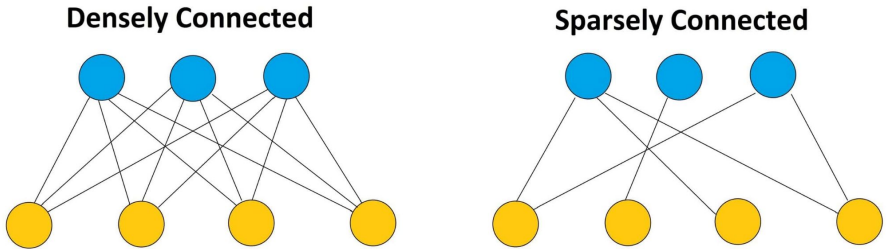
Penn Engineering

# Contents:

# Motivation

- CSR SPMM is important in many ML workloads [8] (Graph Neural Networks, Sparse Linear Layer).
- Existing Libraries (Eg. cuSPARSE [2, 3], torch.sparse) don't handle it very well.
- Need custom SPMM kernels optimised for irregular patterns.



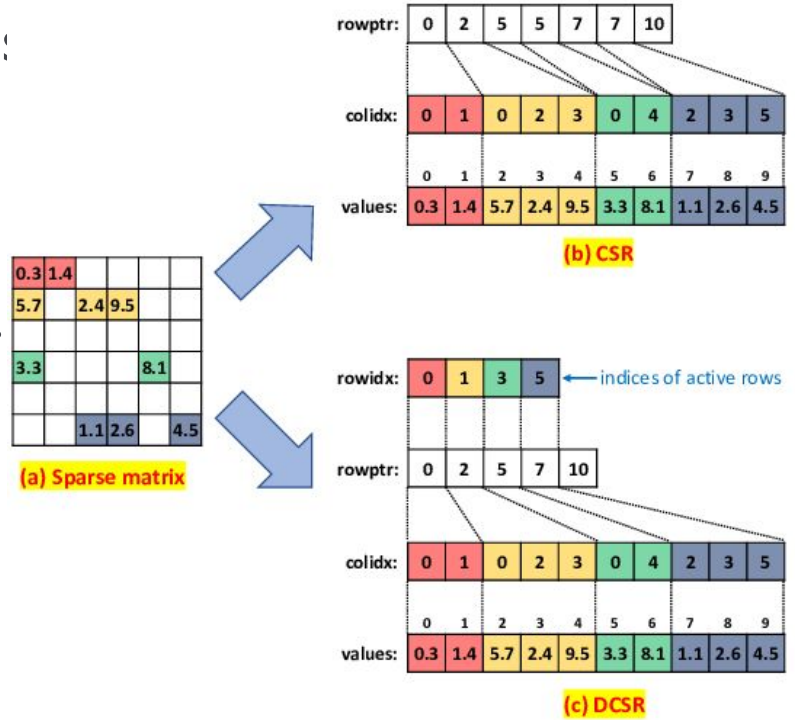



# Background

- CSR Format - Stores only non-zeroes using row_ptr, col_idx and values, this is good for unstructured sparsity.
- cuSPARSE - NVIDIA's sparse lib, slow for irregular sparse patterns.
- torch.sparse - Pytorch supports CSR format, relies on cuSPARSE, lacks custom kernel and speed.
- Research shows that irregular sparsities are common.

# CSR Illustration

- CSR [1] stores non-zero values, column position in col_idx
- row_ptr marks where each non-zero value starts in a row.
- DCSR - also has row_idx which stores number of all non-zero rows.
- Use Case - Matrices where a lot of the rows are empty.



# Contributions of this work

- Highly optimized CUDA kernel, outperform cuSPARSE.
- Implemented naive, warp-per-row, shared-memory, and FP4 vectorized variants for irregular sparsity.
- Python API, outperform torch.sparse.
- Empirical Exp, analysis on performance.
- Unified benchmarks to test and compare various implementations.

# Methods

## Naive Approach

- Each thread is responsible for computing one output element C[row,col] by taking a sparse row of A (CSR) and a full column of B.
- The thread iterates over the nonzeros in its assigned row of A and accumulates products with the corresponding elements from B.

## WarpPerRow Approach

- One warp handles one entire sparse row of A, improving locality and reducing redundant loads of that row's nonzero elements.
- Each lane in the warp computes different output columns (lane, lane+32, lane+64, …), enabling parallel computation of the full row of C.

# Methods

Analyze workload of threads via **main loop**

A[M, N] x B[N, K] => C[M, K]

## Naive Approach

```
sum = 0
for each nonzero entry j in row:
    A_col  = column_index[j]
    A_val  = value[j]
    sum += A_val * B[A_col, col]
```

- Workload for 1 thread is typically smaller
- But loop length is arbitrary, hard to optimized

## WarpPerRow Approach

```
for each assigned output column col = lane, lane+32,
lane+64, ... < K:
    sum = 0

    for each nonzero j in this sparse row:
        a = value[j]
        c = column_index[j]

        sum += a * B[c, col]

    old_value = (beta != 0) ? C[row, col] : 0
    C[row, col] = alpha * sum + beta * old_value
```

- Workload for 1 thread is heavier (nested loops), but the (outer) loop length are predictable

=> facilitate various optimization techniques: shared memory, packed load/store

# Methods

A[M, N] x B[N, K] => C[M, K]

**Optimization strategies:**

- Threads can cooperatively load A into sharedmem

=> reduce global mem traffic

- Packed load/store: CUDA cores allow load/store a pack of **4x float32**

=> can be applied for B and C
=> vectorize memory accesses
=> reduce complexity of outer loop

**WarpPerRow Approach**

```
for each assigned output column col = lane, lane+32,
lane+64, ... < K:
    sum = 0

    for each nonzero j in this sparse row:
        a = value[j]
        c = column_index[j]

        sum += a * B[c, col]

    old_value = (beta != 0) ? C[row, col] : 0
    C[row, col] = alpha * sum + beta * old_value
```

# Experiment & Analysis

## Experiment Setup:

Matrix sizes tested:

- small = (512, 1024, 64)
- medium = (1024, 2048, 128)
- large = (4096, 4096, 128)
- XL = (8192, 4096, 256)

Densities: 0.01 ~ 0.30 (sprasity 0.99 ~ 0.70)

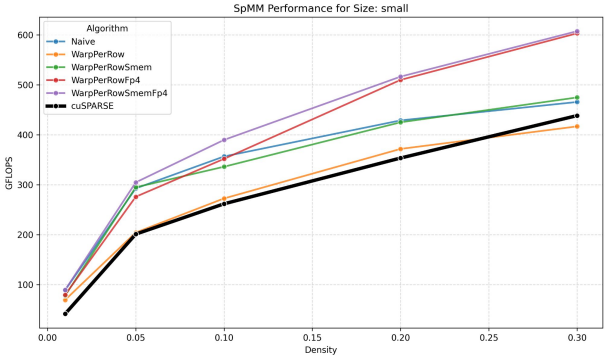
These are **typical sizes and sparsities** in Linear Layers or GNNs [5, 6, 7].

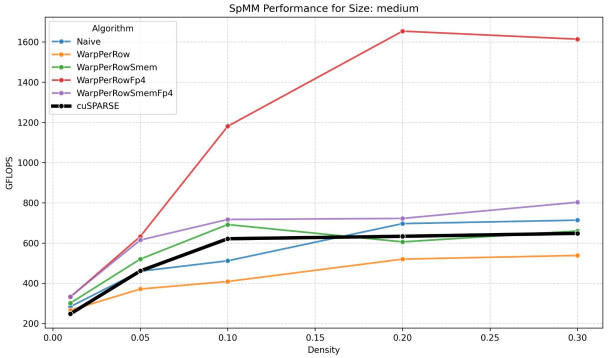
## Kernels - 5 variants

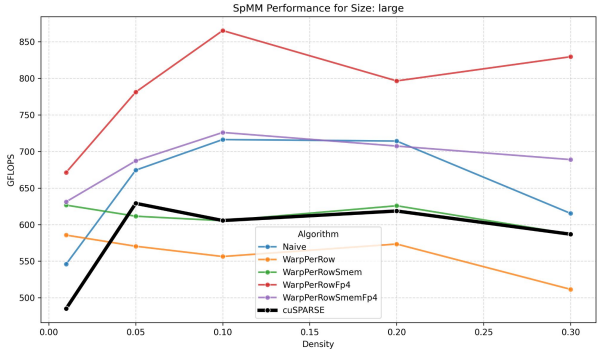
- naive
- warp per row (wpr)
- wrp + sharedmem
- wrp + fp4 (vectorized load/store 4x float32)
- wrp + sharedmem + fp4

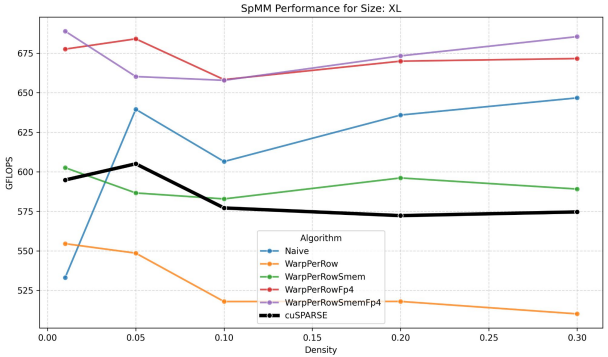
**Hardware**: Nvidia GeForce RTX4070

# Performance vs cuSPARSE




- **WarpPerRowFp4** is consistently the top performer, with **WarpPerRowSmemFp4** is comparable on very small or very large matrices multiplication.
- Performance improvement widen with matrix size -> showing the kernels scale well.
- Our kernels dominate even at high densities -> the kernels can handle a wide sparsity range efficiently.

*Note*: baseline denoted by **bold, dark lines**.

# Performance vs torch.sparse

| Algorithm | mean GFLOPS | mean speedup |
|---|---|---|
| WarpPerRowFp4 | 727 | 1.483 |
| WarpPerRowSmemFp4 | 595 | 1.294 |
| Naive | 531 | 1.161 |
| WarpPerRowSmem | 520 | 1.134 |
| *cuSPARSE* | *488* | *1.000* |
| WarpPerRow | 444 | 0.964 |

*Our kernels vs cuSPARSE*

| Algorithm | mean GFLOPS | mean speedup |
|---|---|---|
| WarpPerRowSmemFp4 | 479 | 2.297 |
| WarpPerRowFp4 | 466 | 2.267 |
| *torch sparse addmm* | *347* | *1.547* |
| WarpPerRowSmem | 324 | 1.702 |
| Naive | 310 | 1.633 |
| WarpPerRow | 275 | 1.553 |
| *torch sparse mm* | *240* | *1.000* |

*Our Python APIs vs torch.sparse*

Our kernels outperform existing libraries:

- **1.48× faster** than cuSPARSE
- **1.38× faster** than torch.sparse.addmm
- **2.30× faster** than torch.sparse.mm

# Breakdown of Performance Factors

| Metric | naive | wpr | wpr_smem | wpr_fp4 | wpr_smem_fp4 |
|---|---|---|---|---|---|
| GFLOPS | 273.39 | 255.77 | 278.31 | 320.44 | 303.36 |
| Memory Throughput (% peak) | 94.72 | 86.41 | 76.94 | 73.43 | 73.13 |
| Memory Bandwidth (GB/s) | 12.55 | 11.57 | 12.10 | 13.10 | 11.93 |
| L1/TEX Hit Rate (%) | 73.78 | 33.04 | 2.95 | 18.68 | 3.77 |
| L2 Hit Rate (%) | 97.12 | 98.93 | 98.82 | 98.76 | 98.76 |
| Mem Pipes Busy (%) | 94.72 | 86.41 | 76.94 | 24.63 | 19.34 |
| Coalescing Efficiency ( / 32) | 22.7 | 22.7 | 31.9 | 28.9 | 31.9 |
| Warp Cycles / Instruction | 29.02 | 29.98 | 16.12 | 68.66 | 52.07 |
| Achieved Occupancy (%) | 63.54 | 91.68 | 46.84 | 77.57 | 46.94 |
| Active Warps per SM | 30.50 | 44.0 | 22.48 | 37.23 | 22.53 |
| IPC (inst / cycle) | 1.05 | 1.47 | 1.39 | 0.54 | 0.43 |
| Total Instructions Executed | 178,205,120 | 267,000,000 | 243,099,537 | 88,314,836 | 76,548,614 |

*Profiling data. Size [8192 x 4096 x 256]. Density 0.1*

Better coalescing

- **Shared memory** allows us to restructure the matrix layout and access pattern, to fit the warps better
- Coalescing efficiency improved significantly with shared mem

# Breakdown of Performance Factors

| Metric | naive | wpr | wpr_smem | wpr_fp4 | wpr_smem_fp4 |
|---|---|---|---|---|---|
| GFLOPS | 273.39 | 255.77 | 278.31 | 320.44 | 303.36 |
| Memory Throughput (% peak) | 94.72 | 86.41 | 76.94 | 73.43 | 73.13 |
| Memory Bandwidth (GB/s) | 12.55 | 11.57 | 12.10 | 13.10 | 11.93 |
| L1/TEX Hit Rate (%) | 73.78 | 33.04 | 2.95 | 18.68 | 3.77 |
| L2 Hit Rate (%) | 97.12 | 98.93 | 98.82 | 98.76 | 98.76 |
| Mem Pipes Busy (%) | 94.72 | 86.41 | 76.94 | 24.63 | 19.34 |
| Coalescing Efficiency ( / 32) | 22.7 | 22.7 | 31.9 | 28.9 | 31.9 |
| Warp Cycles / Instruction | 29.02 | 29.98 | 16.12 | 68.66 | 52.07 |
| Achieved Occupancy (%) | 63.54 | 91.68 | 46.84 | 77.57 | 46.94 |
| Active Warps per SM | 30.50 | 44.0 | 22.48 | 37.23 | 22.53 |
| IPC (inst / cycle) | 1.05 | 1.47 | 1.39 | 0.54 | 0.43 |
| Total Instructions Executed | 178,205,120 | 267,000,000 | 243,099,537 | 88,314,836 | 76,548,614 |

*Profiling data. Size [8192 x 4096 x 256]. Density 0.1*

Reduce pressure on memory

- Naive/WPR are ~94% (heavily memory-bound). Shared-memory cuts this to ~77%, and FP4 variants drop to ~20–25%, a 4-5x reduction in memory load.
- FP4 packing + shared-memory reuse move the kernel from memory-bound to compute-bound.

# Breakdown of Performance Factors

| Metric | naive | wpr | wpr_smem | wpr_fp4 | wpr_smem_fp4 |
|---|---|---|---|---|---|
| GFLOPS | 273.39 | 255.77 | 278.31 | 320.44 | 303.36 |
| Memory Throughput (% peak) | 94.72 | 86.41 | 76.94 | 73.43 | 73.13 |
| Memory Bandwidth (GB/s) | 12.55 | 11.57 | 12.10 | 13.10 | 11.93 |
| L1/TEX Hit Rate (%) | 73.78 | 33.04 | 2.95 | 18.68 | 3.77 |
| L2 Hit Rate (%) | 97.12 | 98.93 | 98.82 | 98.76 | 98.76 |
| Mem Pipes Busy (%) | 94.72 | 86.41 | 76.94 | 24.63 | 19.34 |
| Coalescing Efficiency ( / 32) | 22.7 | 22.7 | 31.9 | 28.9 | 31.9 |
| Warp Cycles / Instruction | 29.02 | 29.98 | 16.12 | 68.66 | 52.07 |
| Achieved Occupancy (%) | 63.54 | 91.68 | 46.84 | 77.57 | 46.94 |
| Active Warps per SM | 30.50 | 44.0 | 22.48 | 37.23 | 22.53 |
| IPC (inst / cycle) | 1.05 | 1.47 | 1.39 | 0.54 | 0.43 |
| Total Instructions Executed | 178,205,120 | 267,000,000 | 243,099,537 | 88,314,836 | 76,548,614 |

*Profiling data. Size [8192 x 4096 x 256]. Density 0.1*

Fewer instructions

- SMEM slightly reduce # of instructions to 243M and 76M (fewer global-access insns)
- FP4 packing lowers total instructions from 178M (naive) to 88M and 76M (FP4 variants) (**more aggressive; basically, we shorten the loop**)

# Breakdown of Performance Factors

| Metric | naive | wpr | wpr_smem | wpr_fp4 | wpr_smem_fp4 |
|---|---|---|---|---|---|
| GFLOPS | 273.39 | 255.77 | 278.31 | 320.44 | 303.36 |
| Memory Throughput (% peak) | 94.72 | 86.41 | 76.94 | 73.43 | 73.13 |
| Memory Bandwidth (GB/s) | 12.55 | 11.57 | 12.10 | 13.10 | 11.93 |
| L1/TEX Hit Rate (%) | 73.78 | 33.04 | 2.95 | 18.68 | 3.77 |
| L2 Hit Rate (%) | 97.12 | 98.93 | 98.82 | 98.76 | 98.76 |
| Mem Pipes Busy (%) | 94.72 | 86.41 | 76.94 | 24.63 | 19.34 |
| Coalescing Efficiency ( / 32) | 22.7 | 22.7 | 31.9 | 28.9 | 31.9 |
| Warp Cycles / Instruction | 29.02 | 29.98 | 16.12 | 68.66 | 52.07 |
| Achieved Occupancy (%) | 63.54 | 91.68 | 46.84 | 77.57 | 46.94 |
| Active Warps per SM | 30.50 | 44.0 | 22.48 | 37.23 | 22.53 |
| IPC (inst / cycle) | 1.05 | 1.47 | 1.39 | 0.54 | 0.43 |
| Total Instructions Executed | 178,205,120 | 267,000,000 | 243,099,537 | 88,314,836 | 76,548,614 |

*Profiling data. Size [8192 x 4096 x 256]. Density 0.1*

One things that we are not (yet) able to fully explain is cache hit rates.

Share mem + vectorized load/store (Fp4) has low hit rates but, somehow, higher throughput.

**Our hypothesis**:

- many threads in naive kernel loads the same cols/rows

=> more frequent accesses
=> explain higher rates
=> naive kernels might not benefit a lot from this

- significantly lower Memory pipe busy and total instructions of smem and fp4 compensate the poor cache rates.

# Conclusions

- Developed a clean, modular CSR-SpMM library with multiple CUDA kernel variants.
- Warp-optimized kernels **outperform cuSPARSE and PyTorch sparse** on many sparsity regimes.
- Provided a simple PyTorch API and benchmarking tools for easy evaluation and extension.
- Demonstrated clear performance trade-offs and a solid foundation for future kernel research.

# Shortcomings and Future Scope

- Limitations:
  - Handles general CSR matrices but does not yet optimize for extreme irregular sparsity patterns.
  - Dynamic warp scheduling was explored but not fully integrated or validated.
- Future scope:
  - Add advanced performance features (streaming, better tiling, load balancing) and broaden Python/CUDA interfaces for more sparsity formats and end-to-end use.

# Work Distribution

Anh: Developed 5 kernels, debugged and integrated Python APIs, designed benchmarks/experiments, analyzed results.

Aditya & Rajath: Implemented the PyTorch integration stack/interface and contributed to developing the naïve shared-memory kernel.

All team members: Collaborated on benchmarking, verification, and end-to-end system integration, comparing the custom CSR-SpMM kernels with PyTorch's sparse operations.

# Reference

[1] Matt Eding, *"Sparse Matrices"* — a comprehensive overview of sparse-matrix data structures and formats (COO, CSR/CSC, DOK, LIL, BSR, DIA), with pros/cons for each. matteding.github.io

[2] NVIDIA Developer Blog: *"Accelerating Inference with Sparsity Using the NVIDIA Ampere Architecture and NVIDIA TensorRT"* — describes how TensorRT 8.0 + Ampere-GPU Sparse Tensor Cores leverage 2:4 structured sparsity for ~50% zeros and deliver large performance and efficiency gains for neural-network inference. NVIDIA Developer

[3] NVPL SPARSE documentation: "Sparse Matrix Formats" — outlines storage formats (COO, CSR, CSC, etc.) supported by NVPL, with detailed definitions of how sparse matrices are represented in memory. NVIDIA Docs

[4] cuSPARSE Generic API documentation — reference guide for generic sparse-matrix operations under NVIDIA's CUDA ecosystem. NVIDIA Docs

# Reference

[5] Jonathan Frankle and Michael Carbin, The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, published at ICLR 2019.

[6] Victor Sanh, Thomas Wolf, and Alexander M. Rush, Movement Pruning: Adaptive Sparsity by Fine-Tuning, published at NeurIPS 2020.

[7] Tianlong Chen, Yongduo Sui, Xuxi Chen, Aston Zhang, and Zhangyang Wang, A Unified Lottery Ticket Hypothesis for Graph Neural Networks, published at ICML 2021.

[8] Tasou, I., Mpakos, P., Vichos, A., et al. Sparse Computations in Deep Learning Inference. arXiv:2512.02550, 2025.