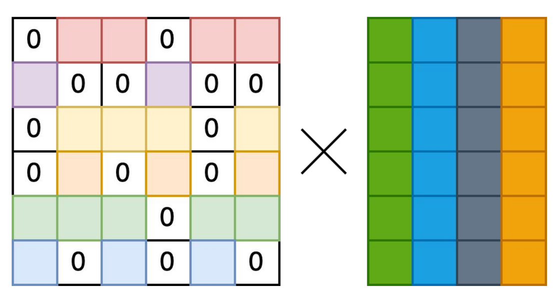
Cuda Implementations of Sparse Matrix Multiplication (SpMM)

Team members: Mingxin Li, Zephyr Zhao

Introduction



$$DenseC = op(SparseA)@op(DenseB)$$

- **Memory Efficiency**: They reduce memory and computation requirements by focusing only on non-zero elements.
- **Diverse Storage Formats**: Many different storage formats exists, tailored for different use cases and performance needs.
- **Challenge**: a) Sparse matrices only focus on non-zero elements, thus the data exhibits access randomness; b) The variety of storage formats and the inherent randomness of sparse matrices makes it difficult to parallelize common operations on GPUs such as matrix multiplication
- **Project Focus**: Our project implements GPU kernels for sparse matrix multiplication (with dense matrix) of various storage formats, and evaluate their performance against sequential SpMM implementation and existing SpMM toolkits

Research Questions

For this project, we hope that our results can help answer the following research questions:

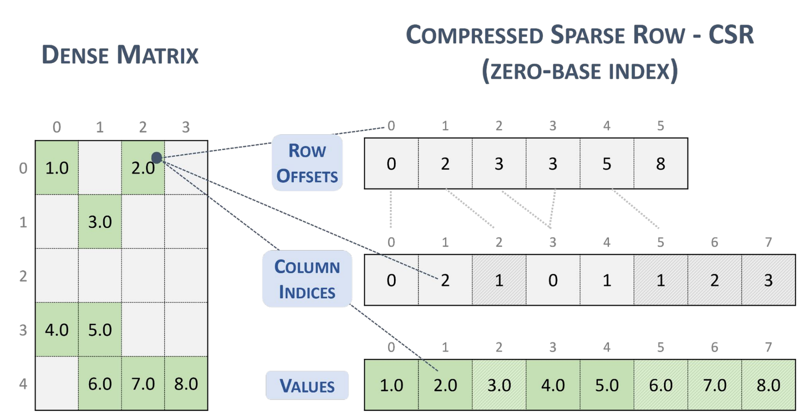
- Which SpMM storage format is most efficient in the context of a sparse matrix multiplied by a dense matrix?
- How does speedup respond to matrix size?
- How does speedup respond to sparsity/density?

Requirements

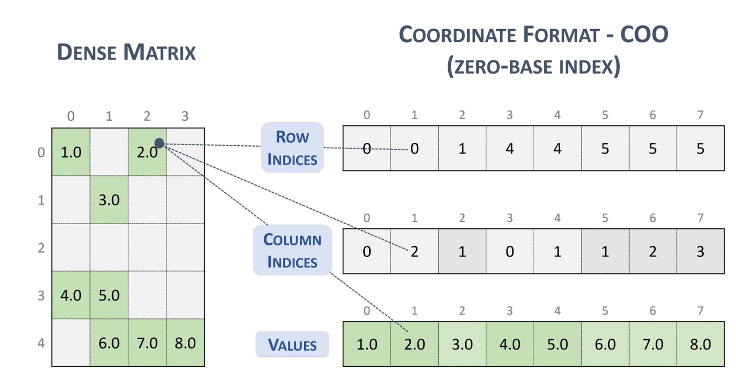
- **Correctness**:
  - Converting input matrices to different storage formats correctly
  - Sequential and parallel implementations of all formats produce correct results
  - Correctly handle matrices of different sizes
- **Performance**:
  - Calculate large matrices in reasonable runtime

Design

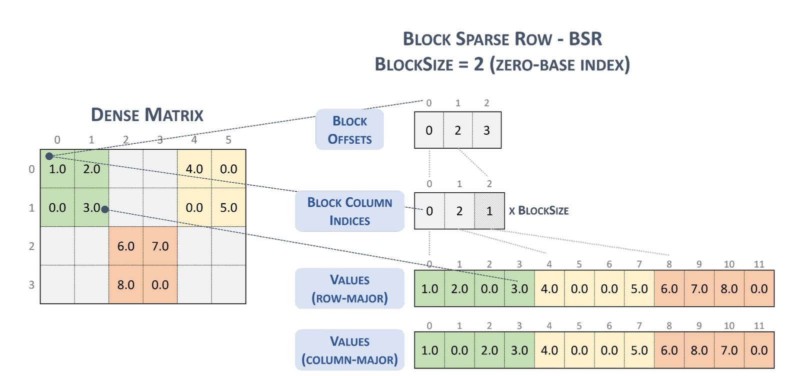
- **Data Source**: We chose varies sized sparse matrices from SuiteSparse Matrix Collection [1] to be used as test cases for both correctness and performance.
- **Storage Formats**: We choose to implement GPU kernels for the following storage formats: Compressed Sparse Row (CSR), Coordinate (COO), Block Sparse Row (BSR) and ELLPack-C
- **Platform**:
  - CPU: AMD EPYC 9534, 256 Cores, 1.5TB Mem
  - GPU: H100 SXM, 80 GB Mem
  - CUDA: CUDA Toolkit 12.4



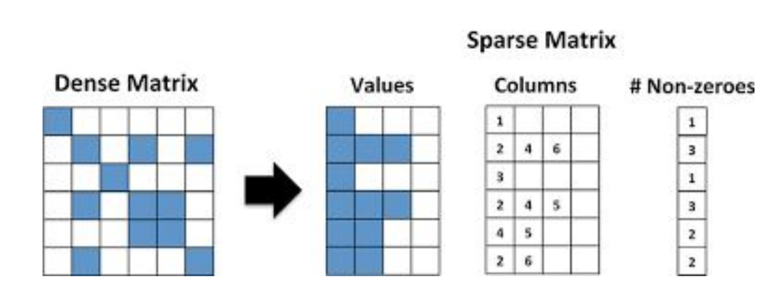
CSR



COO



BSR

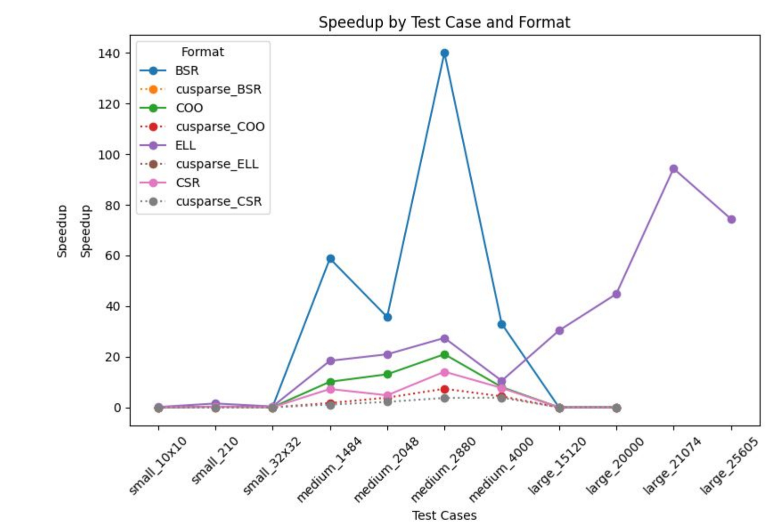


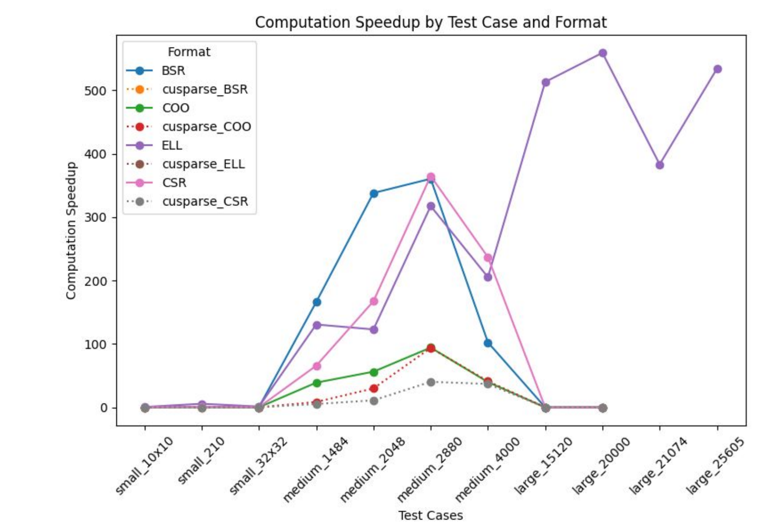
ELL

Methodology

- **Timing:** We timed the prologue, execution, and epilogue for comparison, involving the whole execution process.
- **CPU vs GPU:** We implemented sequential and parallel kernels for each storage format.
- **Kernel Performance Comparison:** Nvidia cusparse Library (supporting only COO and CSR format in version 12.4)
- **Validation:** Using torch C++ API allclose, with Absolute Tolerance 1e-3, Relative Tolerance 1e-2.
- **Profiling:** Nsight Compute.
- **Fine-Tune:** a) Use shared memory to reduce global memory access; b) Transform B as column-major to reduce random memory access and exploit locality; c) Use read-only cache; d) Use intra-warp communication; e) SpTC (ongoing);

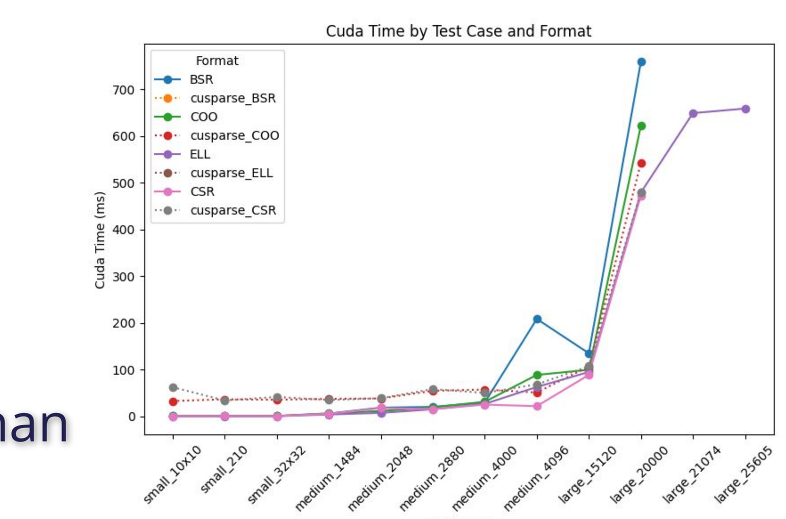
Results



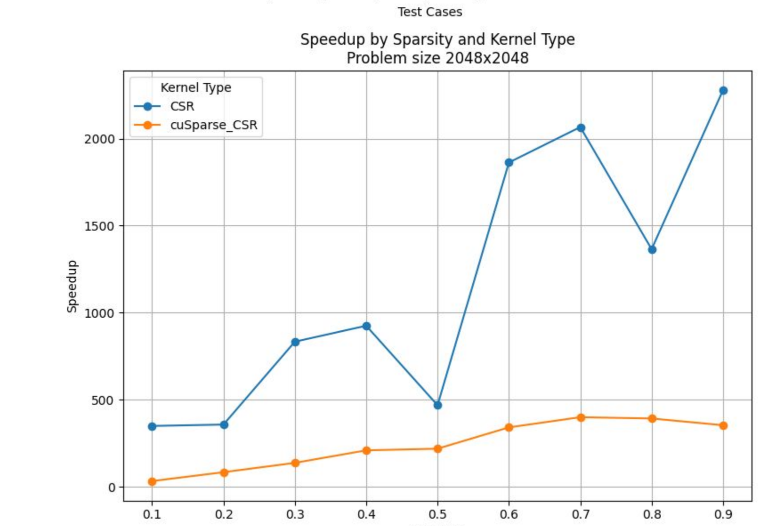


- BSR gains the highest speedup; COO the lowest speedup
- Total speedup << computation speedup
- Speedup increases then generally decreases after reaching a certain size

- CSR is the fastest
- Runtime increases as problem size increases
- Our implementation performs comparably well if not better than cuSparse



- Better performance as density increases
- Our CSR implementation perform better than cuSparse CSR implementation



Conclusion

- For our specific context, CSR storage format provides the most flexibility and performance; our solution achieved a 1.5 to 6 times speedup compared to cuSPARSE;
- As data density increases, the acceleration effect of the GPU continues to improve, but exhibiting diminishing returns.

References

1. SuiteSparse Matrix Collection: https://sparse.tamu.edu/about
2. cuSparse: https://docs.nvidia.com/cuda/cusparse/index.html
3. Matrix Collection Format: https://math.nist.gov/MatrixMarket/formats.html#coord