TileSpGEMM: The Tiled Algorithm for Parallel Sparse General Matrix-Matrix Multiplication on GPUs

Anjali Patil(12140210), Chirag Sharma(12140520), Garima Tata(12140690)

August 9, 2024

1 Abstract

Our problem statement is to design and implement an efficient tiled sparse matrix-matrix multiplication (SPGEMM) algorithm for large-scale sparse matrices on modern computing architectures. Sparse General Matrix-Matrix Multiplication (SpGEMM) is a key operation in sparse linear algebra that plays a vital role in tasks like sparse linear solvers, graph processing systems, and machine learning algorithms. Essentially, it involves multiplying two matrices with a sparse structure, resulting in another sparse matrix. Despite its importance, parallelizing SpGEMM is quite challenging, especially on contemporary parallel platforms such as GPUs, Xeon Phi, and distributed clusters, due to the sparsity of its input and output matrices.

2 Introduction

We have used Tile Data Structure and Tiled approach to SpGEMM for sparse matrix matrix multiplication.
In the step 1, we performed a symbolic SpGEMM for the sparse matrix multiplication and identifies potential non-zero tile regions in the output sparse matrix.**C**.
In step 2 we worked on each tile of C and implemented a function to identify matching non-empty tiles in A and B for corresponding tile in C. Next, we constructed bitmask for each tile in C using matching pairs generated before and lower level structure of tile. Finally, when all the row masks in C tiles are generated.
At last, in step 3, we need to select **accumulator**, If the number of non-zeros in a tile is greater than t_nnz (threshold- 75% of tile size), use dense_accumulator. Otherwise, use sparse_accumulator for better efficiency.
We tried implementing parallelism for processing multiple tiles concurrently and used atomic operations (AtomicAdd) for safe updates in a multi-threaded environment.

3 Approach/Algorithm

Due to dependency issue of CUDA, we were not able to run the existing code. So, we implemented complete algorithm from stratch, first we wrote sequential code than we tried to parallelize each tile computation though leveraging the use of Threads

3.1 First Step (Count Number of Sparse tiles in resultant Matrix)

We used two sparse matrices, **A** and **B**, as input and performed symbolic SpGEMM multiplication between their high level tile structure(this reduces the size of both matrix which indirectly reduces computation cost) to identify sparse tiles in the output matrix **C**, This step calculates potential number of the non-zero elements in the resulting matrix $\mathbf{C}'$ which is the high level structure of resultant matrix C.

Tile Information, that we produced in this step are:
1. tileRowidx_C: Stores the row index of the non-zero tile in **C**.

2. tileColidx_C: Stores the column index of the non-zero tile in **C**.
3. numtileC: Stores the number of non-zero tiles in resultant matrix.

3.2 Second Step (Generating Tile Structure)

1. First to find the pairs of matching tiles for each non-zero tiles in C, we used Linear search to find the intersecting non-empty tiles between A's row and B's column that returns a list of this intersecting non-empty tiles.

2. Then for each non-zero tile in C we calculated its bitmask using list of matching generated previously. Essentially in this process, for each pair of matched tiles Aik and Bkj, traverse all the non-zeros of Aik, and consider the column index of each nonzero as the row index of the bit mask of Bkj. Then, the AtomicOr operation is used for all the extracted row masks to obtain the resultant row mask of Ci, and finally, we obtain the mask of each tile. Finally we generated row pointer array and number of non-zero value for each tile.

3.3 Third step (Accumulator)

1. The bit mask of each tile that was generated in the step 2 is used to generate the column index of all nonzeros of a tile and then the atomic add operation is operated to compute the value. In this way, the intermediate products can be directly accumulated at a certain position according to the column index.

2. In Dense Accumulator, we directly calculate values of the tile by directly multiplying matching tiles(finded in step 2) in matrix A and matrix B.

4 Evaluation Methodology

- We wrote sequential code in C++ and then parallelized the code on each tile using CUDA threads.
- **Hardware Platform**: We conducted experiments on Google Colab, which has an NVIDIA Tesla T4 GPU.
- **Input Dataset**: We created our own dataset consisting of matrices ranging in size from 128 to 3072, with varying sparsity levels (0.1, 0.2, 0.25, 0.3). These matrices were stored in files (.txt format), accessible for anyone to use through file reading.
- **Third-party Frameworks**: We implemented SPGEMM (Sequential and Parallel) and Tile-SpGEMM (Sequential) as third-party frameworks. We then compared:
  - TileSpGEMM (Parallel) with SPGEMM (Sequential)
  - TileSpGEMM (Parallel) with SPGEMM (Parallel)

5 Experimental Results

- Optimal tile size increases with the size of matrices, for us optimal tile size came out to be 64.
- For very small matrices, SpGEMM works better than Tile SpGEMM (for all tile values).
- As the matrix size increases, Tile SpGEMM dominates over SpGEMM provided the adequate tile size is set a priori.
- For very large matrices, small tile sizes perform extremely poorly since the abstraction outweighs the processing time.
- For relatively dense matrices (density 25%), the execution time of SpGEMM increases, but for Tile SpGEMM, this increment is not significant, and it still outperforms SpGEMM in the case of larger matrices.

TileSpGEMM(Sequential) with SpGEMM (Sequential)

Figure 1: TileSpGEMM(Sequential) v/s SpGEMM (Sequential)

Figure 2: TileSpGEMM(Sequential) with optimal tile 32 v/s SpGEMM (Sequential)

TileSpGEMM(Parallel) with SpGEMM Parallel

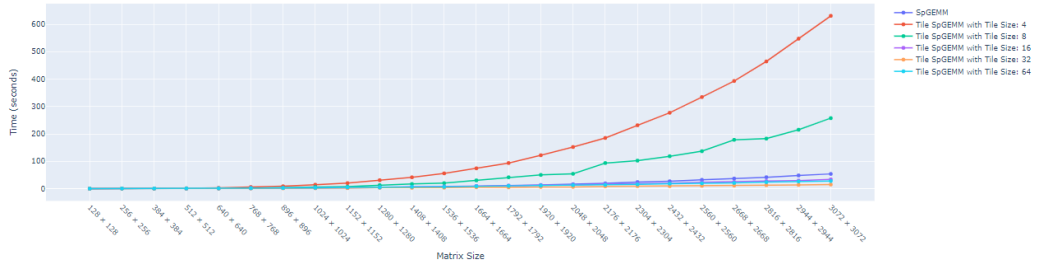

Figure 3: TileSpGEMM(Parallel) v/s SpGEMM(Parallel)

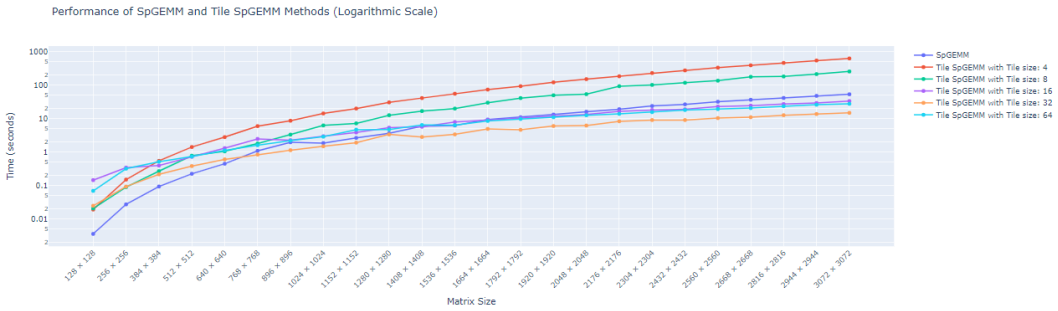

Figure 4: TileSpGEMM(Parallel) v/s SpGEMM(Parallel) on logarithmic Scale

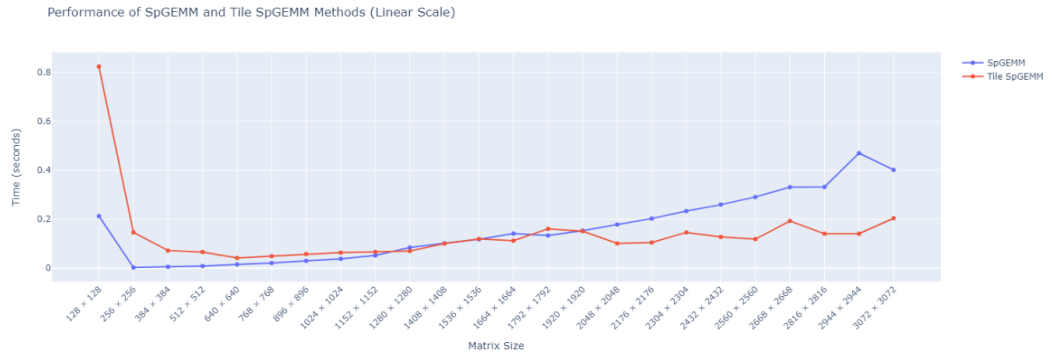

Figure 5: TileSpGEMM(Parallel) With tile size 64 v/s SpGEMM(Parallel)

6 Related Work

Some related work in the field of Sparse Matrix-Matrix Multiplication has been done. For example, using Cusparse and Nsparse libraries. However, Nsparse and Cusparse libraries, due to their proprietary nature and dependency issues (e.g., requiring CUDA 11.4 and Google colab is CUDA 12.2), are not easily feasible for general use. SPGEMM is also a promising technique, but it has certain shortcomings such as load imbalance, high global space complexity, and unsatisfactory data locality and sparse accumulator selection.

7 Conclusion

Writing such a complicated algorithm was very challenging for us, but we managed to successfully implement both sequential and parallel code.

Through these experiments, we acquired insights into a new algorithm for matrix-matrix multiplication, which proved to be significantly more optimal. We discovered that up to a certain matrix size and tile size, Tile Spgemm outperforms SPGemm. For us, the optimal tile size was found to be 64.

In our work, we tried to make both tilewise and bitwise conversions parallel. However, we acknowledge that further optimizations are possible. Considering the practical applications and possibility to optimize future prospects are visible.

8 References

1. TileSpgemm Paper(https://dl.acm.org/doi/pdf/10.1145/3503221.3508431)
2. Tile Spgemm Github (https://github.com/SuperScientificSoftwareLaboratory/TileSpGEMM/tree/main/src)