CS267 Project - SpMM, SDDMM and SDDMM_SpMM Kernels in CUDA C/C++

Tzu-Chuan(Jim) Lin
University of California, Berkeley
tzu-chuan_lin@berkeley.edu

Pin-Lun(Byron) Hsu
University of California, Berkeley
byron_hsu@berkeley.edu

Chin-An(Daniel) Chen
University of California, Berkeley
chinanchen@berkeley.edu

Abstract

Sparse Matrix-Matrix multiplication(SpMM) and Sampled Dense-Dense Matrix Multiplication(SDDMM) are computational kernels that commonly appear in a back-to-back fashion in graph neural networks(GNN). More precisely, when the sparse matrix represents an adjacency matrix of a graph and the dense matrix represents a stack of nodes' feature vectors, calling SpMM and SDDMM back-to-back will aggregate the information from their neighbors according to their dot-product interactions' weights. In this project, we explored different ways to perform blocking on SpMM, SDDMM separately and provided a fused kernel - SDDMM_SpMM. We also compared the performances of these kernels with cuSPARSE [1] and DGEMM [2] on randomly generated with different zero ratios, sizes of the matrices, and the different blocking sizes (i.e. thread block size) of the GPU kernels. Finally, our SpMM implementation outperforms cusparsespmm(), SDDMM implementation outperforms cusparsesddmm() and cublasDgemm() when the dimensions of the sparse and dense matrices are smaller than 2048 and 1024 respectively. However, surprisingly, the fused SDDMM_SpMM runs slower than calling our SDDMM and SpMM kernels back-to-back. We release all the experimental code into Github [1] if the readers are interested in extending this work.

1. Introduction

Graphs are a general representation of data, such as images, text, and social networks. Here, we take social media networks as an example. In the graph, each vertex could stand for a user, and the edges between each pair of vertices represent the connections; oftentimes, the graph of n users would be stored in an adjacency Matrix S with size (n, n) . On top of that, if the vertices have a set of features, such as gender, age, and height, then the vertex matrix A would then have a dimension of (n, f). Below is a short description of the GNN computation process (Fig. 1). Based on the usage of SDDMM and SpMM mentioned, we aim to optimize SDDMM_SpMM using CUDA kernels to achieve a better performance than calling SDDMM and SpMM separately, in order to speed up the inference of graph neural networks.

Here are the summary of related operations:

- AA^T is used to calculate the interactions between each pair of the nodes in the graph. In other words, $(AA^T)_{i,j}$ is the interaction posed on node j from node i.
- S the adjacency matrix records where the edges are in the graph.
- $(S \odot AA^T)$ used to mask the weights matrix.
- $(S \odot AA^T)A$ would then be used to calculate the next hidden features.

Compressed Sparse Row (CSR) is a way to store the matrix where most elements are zero (i.e. sparse matrix). Instead of one two-dimensional matrix, CSR represents the matrix S by three one-dimensional arrays, which contain non-zero values,

[1] https://github.com/jimlinntu/cs267-final/

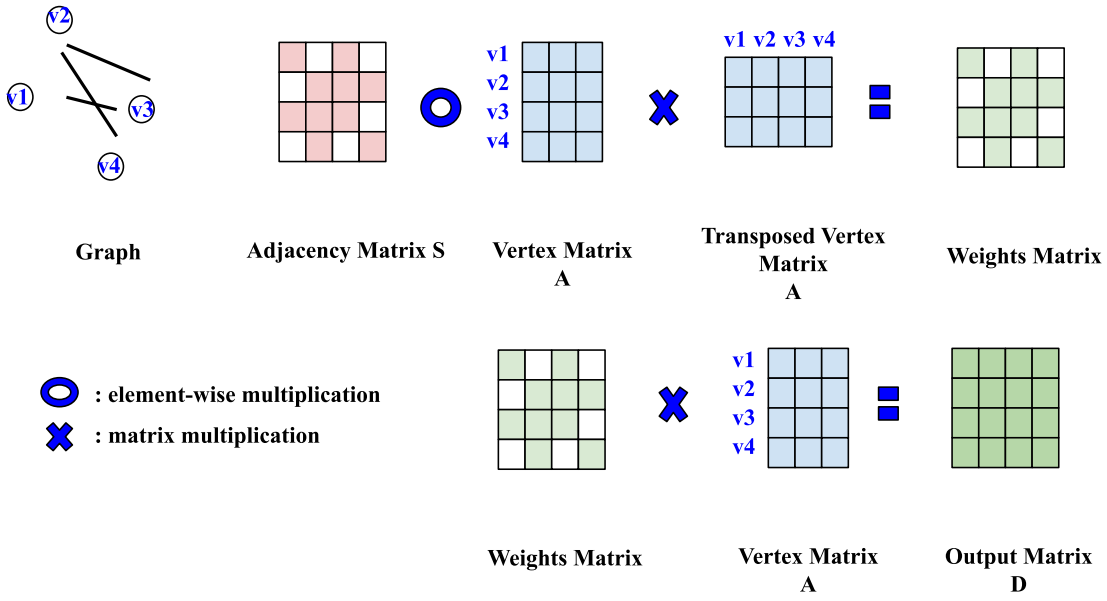


Figure 1. Dataflow for GNN using SDDMM and SpMM

the offset for the rows, and the column indices for each non-zero value (Fig. 2). Using CSR format when performing matrix multiplication can save lots of redundant work potentially because we can avoid multiplication that will be eventually masked.

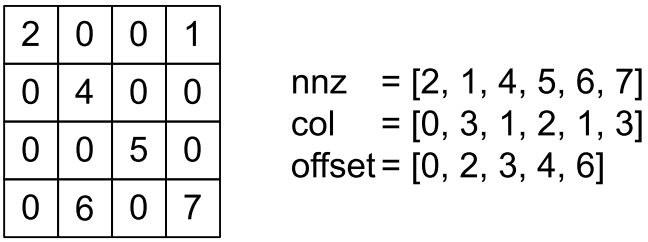


Figure 2. An example of CSR format

In our project, we define three operations:

- $SpMM(S, A) = SA$
- $SDDMM(S, A) = S \odot AA^T$
- $SDDMM\_SpMM(S, A) = (S \odot AA^T)A$

where S is a sparse matrix and A is a dense matrix.

Note that the common definition of $SDDMM$ accepts three arguments S, A, B (i.e. $SDDMM(S, A, B) = S \odot AB^T$). In this project we simplify the definition of $SDDMM$ to only accept two arguments because we focus more on the implementation ideas. The ideas we shared in this project can easily apply when $SDDMM$ accepts one more argument - B.

Our contributions to this project are listed below:

- We provide different ways of implementing $SpMM$ (Section 4), $SDDMM$ (Section 5), and $SDDMM\_SpMM$ (Section 6), share the excruciating details of the implementations and benchmark their performances in Section 7.
- We compare our methods to the vendor's implementations (i.e. cuSPARSE and cuBLAS). We found that our SpMM, SDDMM kernels outperformed the vendor's kernels when $S.num\_rows \leq 2048$ and $A.num\_cols \leq 1024$, while the fused kernel SpMM_SDDMM (Section 6.2) does not outperform the implementation in Section 6.1 as we originally expected.
- We organize our codebase in a highly-structured format where readers can easily plug in their own implementations and compare the performances with implementations provided in our codebase.

2. Implementation Overview

Considering that our work is about experimenting with different CUDA computational kernels, we organize our codebase extremely rigorously, hoping that readers interested in our codebase can understand our implementation details without a huge effort.

First of all, because our implementation would involve a lot of data movement between the host (i.e. CPU) and the device (i.e GPU), we created several classes with proper member functions to handle this tedious conversion and prettify our code. For the host matrix, we use `HostDenseMat` and `HostSparseMat` to represent a dense matrix and a sparse matrix on the host respectively. While for the device matrix, we use `DeviceDenseMat` and `DeviceSparseMat` to represent a dense matrix and a sparse matrix on the device respectively. For all `Host*Mat`, `.to_device(Device*Mat &d)` are associated with these classes so that CUDA runtime API such as `cudaMemcpy` is hidden from the main algorithms and called implicitly. And for all `Device*Mat`, we add `.copy_to_host(Host*Mat &h)` function so that the readers can understand the semantics of the implementations in the main algorithms easier and avoid calling `cudaMemcpy` explicitly as well.

Secondly, for {`Host|Device`}`DenseMat`, there will be three main member variables associated with it: 1) `num_rows`, 2) `num_cols` and 3) `double *vals`. Notice that we store all of our matrix in a row-major order (i.e. `vals[i * num_cols + j]` will access an element at row i and column j of the matrix).

Thirdly, for {`Host|Device`}`SparseMat`, we follow the compressed row format (CSR). Therefore, we have 6 main member variables associated with this class: 1) `num_rows`, 2) `num_cols`, 3) `nnz` (the number of non-zeros of this matrix), 4) `int *offsets`, 5) `int *cols`, 6) `double *vals`. In this format, to loop over all the non zero elements in row i in this matrix, one will need to first access `offsets[i]` and `offsets[i+1]`. Based on these two values, one can use $j \in [offsets[i], offsets[i+1])$ to access an element in `double *vals` (i.e. `vals[j]`) and use it to get the column index (0-index) by `cols[j]`.

In addition, we create `MatrixGenerator` class for the experimental purpose. There are three kinds of generating functions: 1) `generate_sparse_csr`, 2) `generate_binary_sparse_csr` and 3) `generate_dense`. The difference between 1) and 2) is: the latter only generates a matrix with 0 or 1 because we realized `cusparseSDDMM` only supports a binary sparse matrix as input. Therefore, by having 2), it is fairer to compare our implementations with `cusparseSDDMM`.

In terms of the algorithms, in `include/algo.cuh`, we defined three classes: 1) `CusparseAlgo`, 2) `CublasAlgo` and 3) `Algo`. As the naming suggest, the methods (`spmm`, `sddmm` and `sddmm_spmm`) in 1) are implemented using `cusparseSpMM` and `cusparseSDDMM` from [1]. While the methods - {`spmm|sddmm|sddmm_spmm`}`_by_dgemm` in 2) are implemented using `cublasDgemm` from [2]. Lastly, the methods in `Algo` class are implemented by the authors. We will detail the implementation later in Section 4, 5 and 6.

Regarding to the correctness of our implemented method, we provided `Checker` class with `check_correctness_spmm`, `check_correctness_sddmm` and `check_correctness_sddmm_spmm`, where we generate a hundred small random matrices and then compare our results to vendor's implementations (i.e. cuSPARSE, cuBLAS) to make sure our implementations are correct.

3. Related Works

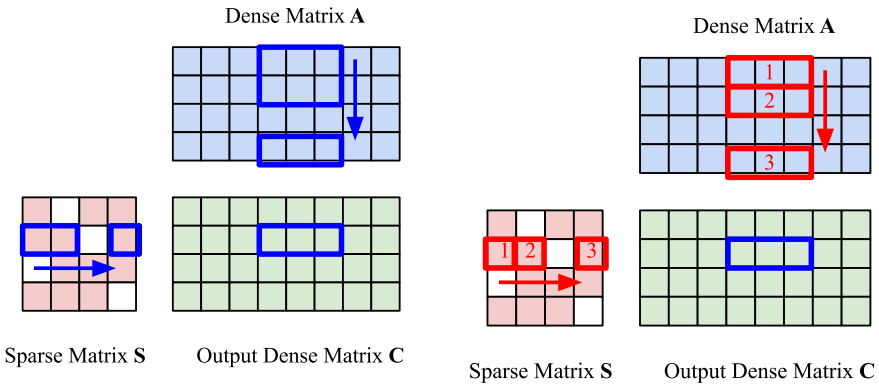
In [3], the authors fuse SDDMM and SpMM into one kernel by proposing a new kernel called FusedMM. They split the kernel into five user-defined functions to make it flexible. Also, they maximize resource utilization by SIMD. FusedMM even without any optimization, runs up to 9.8× faster than DGL [4] library. However, FusedMM is only implemented on CPU in this paper. We aim to explore the performances of these computational kernels on GPU in our paper.

More optimization tricks on SDDMM and SpMM are proposed in [5], including Hierarchical 1-Dimensional Tiling, Vector Memory Operations, Row Swizzle Load Balancing. It can achieve 1.2–2.1× speedups and up to 12.8× memory savings on sparse Transformer and MobileNet models.

In [6], they developed a row-reordering technique to improve the performance of SpMM and SDDMM. The existing techniques can only deliver high performance when consecutive rows of the sparse matrix have non-zeroes at identical columns. They propose a cluster-based row-reordered procedure to put similar rows together to enhance data reuse. The evaluation shows this technique can achieve up average 1.19x speedup over the state-of-the-art alternative for SpMM.

4. $C := SpMM(S, A)$ Implementation

4.1. `spmm_no_shm`



(a) `spmm_no_shm` (b) `spmm_with_shm`

Figure 3

We divided the output matrix (C) into several blocks with a height of 1, and for each block, each thread is in charge of an element in C. Referring to Figure 3a, when we are calculating the blue block in the C matrix, we will walk through the blue blocks in A and S matrix and ignore the zero value in S.

4.2. `spmm_with_shm`

Referring to Figure 3b, this is an improved version of Section 4.1. When we walk through S and A, we will divide them into sub-blocks (numbers 1, 2, and 3 on the red block) and calculate them one by one. Furthermore, to accelerate memory access, we stored the elements of the S sub-block in shared memory so we do not need to request an element in A multiple times from the GPU global memory, given the fact that all the threads in a block are all accessing and using the elements from the same row in A.

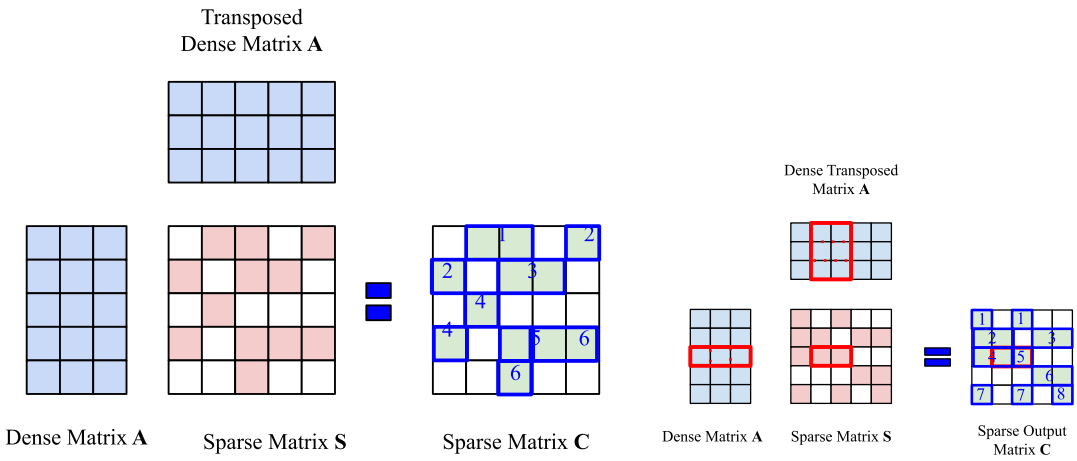
4.3. `spmm_with_shm_improved`

This implementation is exactly the same as Section 4.2. The only difference is: when we launch the kernel, instead of using `kernel<<<dim3(A.num_cols/TILE_WIDTH, A.num_rows), dim3(TILE_WIDTH, 1)>>>`, we use `kernel<<<dim3(A.num_rows, A.num_cols/TILE_WIDTH), dim3(1, TILE_WIDTH)>>>`. In other words, x and y are swapped. Surprisingly, we found out that by using `x` to index rows of C and `y` to index columns of C, the runtime is reduced significantly compared to Section 4.2. We will share more experimental results in Section 7.

5. $C := SDDMM(S, A)$ Implementation

5.1. `sddmm_block_over_nnz_wo_shm`

In this method (Figure 4a), we flatten the output sparse matrix C and divide it into multiple blocks by `blocksize`. The elements in the same block might scatter across different rows because there is no guarantee that the number of non-zero elements in S is a multiple of `blocksize`. The calculating process of an output element is quite naive, we just need to



(a) `sddmm_block_over_nnz_wo_shm` (b) `sddmm_with_tid_mapping`

Figure 4

walk through a row in A and a column in A^T and multiply the dot product result by the corresponding element in S (i.e. $C_{i,j} = (A_{i,:} \cdot A_{j,:}) * S_{i,j}$). One thing to note that each thread will need to perform a **binary search** over `offsets` in order to know the row index of this element. However, because the elements in a block might not sit on the same row, we cannot use shared memory to accelerate.

5.2. `sddmm_with_tid_mapping`

In order to leverage shared memory, we want the elements in a block to sit on the same row (Figure 4b). We divide the elements on each row into blocks, and if the number is not a multiple of `blocksize`, we will create padded elements to fill the gap. The padded element would not produce output but just help align the blocks. We create a mapping to map threads to elements in C, while some threads are mapped to padded elements. We can utilize shared memory on A when we walk through A and A^T to speed up the process.

5.3. `sddmm_block_over_nnz_if_same_row_use_shm`

This method is the extension of Section 5.1. The main idea is the same as Figure 4a. One observation of Section 5.1 is: when a thread block happens to operate on elements that sit on the same row in the sparse matrix, we will notice that all threads in that block will access the **exact same row** in A matrix. According this observation, we let a thread block to load the elements on row i of A together and put these elements in the shared memory. After that, each thread will be able to use the elements they have loaded together to compute the dot product between row i of A and the column j of A^T (which is row j of A, no need to explicitly transpose A).

5.4. `sddmm_block_over_nnz_but_in_same_row`

The main concept of this method is similar to Section 5.2. However, one critical flaw of the method in Section 5.2 is: it will first inspect the whole sparse matrix S by CPU, populate some data structures on CPU and then copy this populated data structure to GPU. This does not make sense because we want the algorithm to run on GPU as much as possible to exploit more parallelism. In order to offload the same concept from Section 5.2, we maintain an array called `block_offsets` on GPU. We first launch a kernel to count how many blocks do we need for each row (i.e. `block_offsets[i]` equals the number of thread blocks for row i). In this way, we will gaurantee that each thread block will operate elements in S in the same row so that we can take advantage of the shared memory on the row of A just as Figure 4b shows.

After knowing how many blocks for each row of S we need to launch, we will further compute prefix sum on `block_offsets` in parallel using `thrust` (the illustration of how to get `block_offsets` is shown in Figure 5). Then, we will launch the CUDA kernel. In the kernel, each thread block will first need to perform a **binary search** over `block_offsets` to find which row am I(this thread) operating on? Then, a thread block will first load elements from row i of A together in parallel just as Figure 4b and then each thread will compute the dot product in parallel independently.

5.5. `sddmm_launch_kernel_as_dense_matrix`

We came up with this idea because we noticed that in the methods mentioned above, there are always several preprocessing steps to make sure that all the elements a thread block process stays on the same row in S. Therefore, to avoid the preprocessing steps, we choose launch the kernel as if S is a dense matrix. In Figure 6a, we use the red color to denote the nonzero elements and the blue rectangle to denote the thread block. As you can see, when we launch the kernel in this way, each thread block will be guaranteed to operate on elements that are in the same row in S. One weakness of this method is: when S's number of rows or number of columns are extremely large, we will launch lots of thread blocks that have no work to do (for example, in Figure 6a, 2 rightmost thread blocks on the second row of S will **immediately return** due to no work).

5.6. `sddmm_dynamic_parallelism`

One pitfall mentioned in Section 5.5 inspired this implementation. According to [7], a CUDA kernel can launch child kernels on the fly. By taking this advantage, we will not need to launch too many redundant thread blocks because we will first launch `S.num_rows` threads and then each thread will launch `S.number of cols on that row` threads. However, even though this implementation seems to save lots of redundant thread blocks, in reality, the runtime of this implementation is really poor. We will mention the experimental runtime in Section 7.

The illustration is shown in Figure 6b. By comparing Figure 6a and 6b, you can easily tell the difference between two implementations.

6. $C := SDDMM\_SpMM(S, A)$ Implementation

6.1. `sddmm_spmm_naive_back2back_calls`

The most naive way of fusing two operations together is to launch two separate CUDA kernels. The advantage of this implementation is that two kernels can block over different matrices. In our case, the first kernel (i.e. $SpMM$) will block over the sparse matrix S while the second kernel (i.e. $SDDMM$) will block over the output matrix C. In this way, there will be **no race condition across thread blocks** when writing to the output matrix C and thus no need of any synchronization primitives.

6.2. `sddmm_spmm_block_over_sparse_launch_as_dense_matrix`

By observing the implementation in Section 6.1, you will notice that the result of $S \odot AA^T$ must be written to the global memory on GPU so that the second kernel can execute the $(S \odot AA^T)A$ operation. However, you can quickly notice that we do not actually need the result of $S \odot AA^T$ **on the global memory**.

Therefore, in this implementation, we choose to only launch a kernel that blocks over the sparse matrix S. In this way, each thread block will first compute $S' = (S \odot AA^T)$ **without** writing the result (i.e. S') to the global memory. S' will only

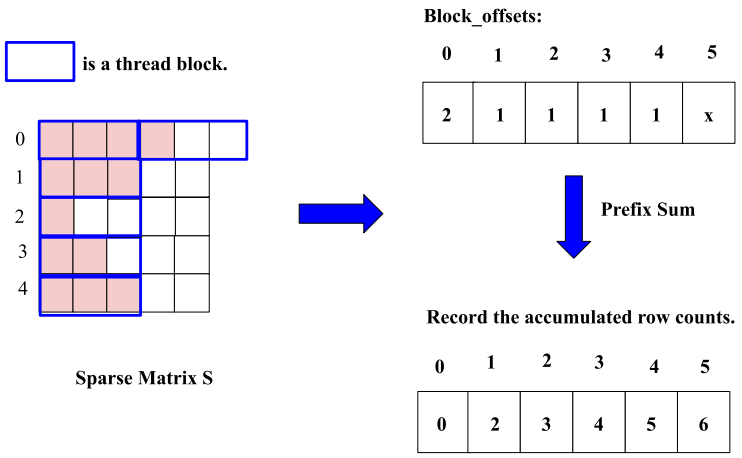
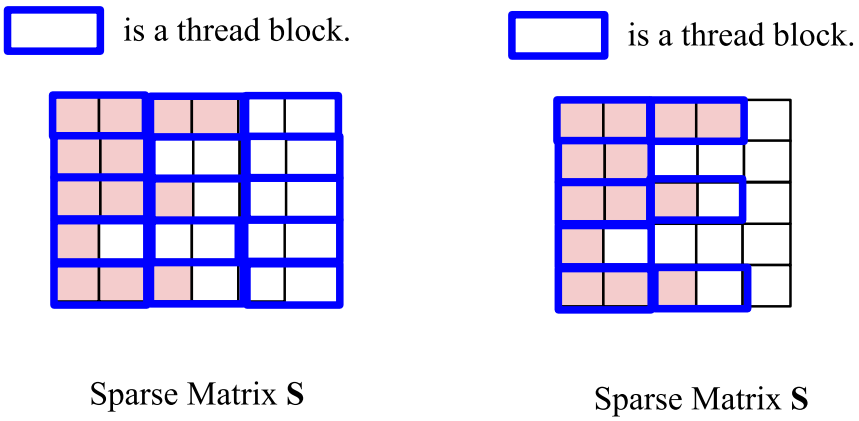


Figure 5. `sddmm_block_over_nnz_but_in_same_row`



(a) `sddmm_launch_kernel_as_dense_matrix` (b) `sddmm_dynamic_parallelism`

Figure 6

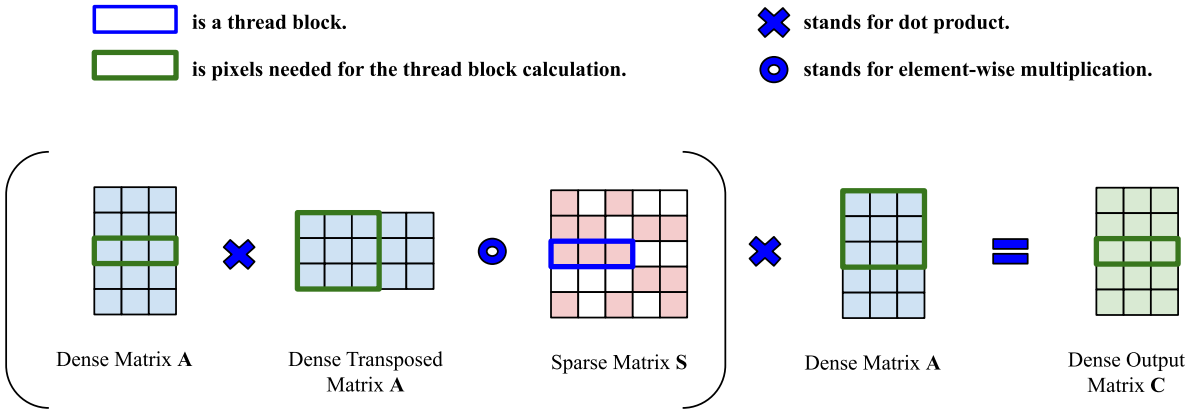


Figure 7. `sddmm_spmm_block_over_sparse_launch_as_dense_matrix`

be stored into the shared memory for each thread block. After finishing computing S', each thread block will increment the its $S'A$ result into the output matrix C. One weakness of this method is: we will need `atomicAdd` (due to race condition on writing) when incrementing the $S'A$ to the output matrix C. For example, let $S'_{I,K}$ denote a thread block in S'. Every thread block with index (I, K) will need to touch $C_{I,J}$ for all $J \in [0, A.num\_cols)$. In other words, thread block (I, K_1) and thread block (I, K_2) will potentially increment $C_{I,0}$ at the same time, forcing us to use `atomicAdd` to avoid race condition on the output matrix C.

The illustration of this implementation is shown in Figure 7.

7. Experiment

7.1. Environment Setup

All of the experiements are done on Cori GPU [8], using the environment set up script - `set_cori_env.sh` in our Github repo [2]. At the time we did the experiment, the CUDA version on Cori is 11.4.0.

We conducted several experiments regarding 1) the matrix sizes and 2) sparsity(zero ratio) of the matrices and 3) the thread block sizes of the launching kernels versus the execution time.

Note that we use (m, n) as a shorthand for $(S.num\_rows, A.num\_cols) = (m, n)$ in this section.

[2] https://github.com/jimlinntu/cs267-final/blob/master/set_cori_env.sh

7.2. SpMM Experiments

7.2.1 Matrix Size

We ran the experiments on four different size configurations for SpMM (fig. 8). Along the x-axis, the sizes for both sparse matrix S and dense matrix A were doubled, and the execution times of each kernel were recorded on the y-axis accordingly. Unsurprisingly, we found that the log execution time is relatively linear w.r.t the size of the two input matrices, and the kernel using the shared memory gave us the best performance. On top of that, in the configuration $(1024, 512)$, cusparsespmm cost the longest time might attribute to the memory preallocation overhead because it is required to call `cusparseSpMM_bufferSize` and then `cudaMalloc` before running `cusparseSpMM`.

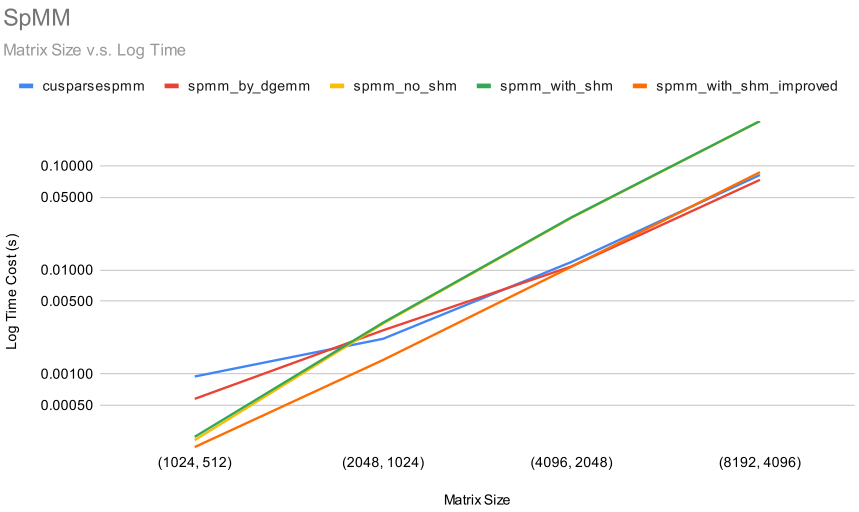


Figure 8. SpMM: Execution Time vs Matrix Sizes when the sparsity is 0.9

7.2.2 Sparsity

To find out the relationship between the sparsity and the execution time, this experiment performed with four different sparsity settings (fig. 9). For DGEMM, because it treats every matrix as dense, it is reasonable that the execution time will not vary by the sparsity. On the other hand, compared to the cuSPARSE, our proposed methods is more sensitive (i.e. the slope is steeper) w.r.t the sparsity of the matrix.

7.2.3 Block Size

The experiment tested on the fixed configuration $(8192, 4096)$ and recorded the execution time with the various block sizes for the launching kernel. Shown in fig. 10, the block size will not affect the execution time significantly.

7.3. SDDMM Experiments

7.3.1 Matrix Size

Similar to the settings from SpMM, the experiment tested on four different configurations of sparse and dense matrices for SDDMM with the sparsity set to 0.9 and block size set to 16. In (fig. 11), both cusparsesddmm and DGEMM showed almost the same trend through the different configurations, which might be caused by in cusparse, before the certain sparsity, the kernel might treat the sparse matrix as a dense matrix, which is the same as DGEMM. Besides, there is a larger slope occurred between $(2048, 1024)$ and $(4096, 2048)$ using dynamic parallelism. It might be caused by the imbalance of zeros in each row of the sparse matrix, which made the kernels launched by each row differ from each other and increased the overall execution time.

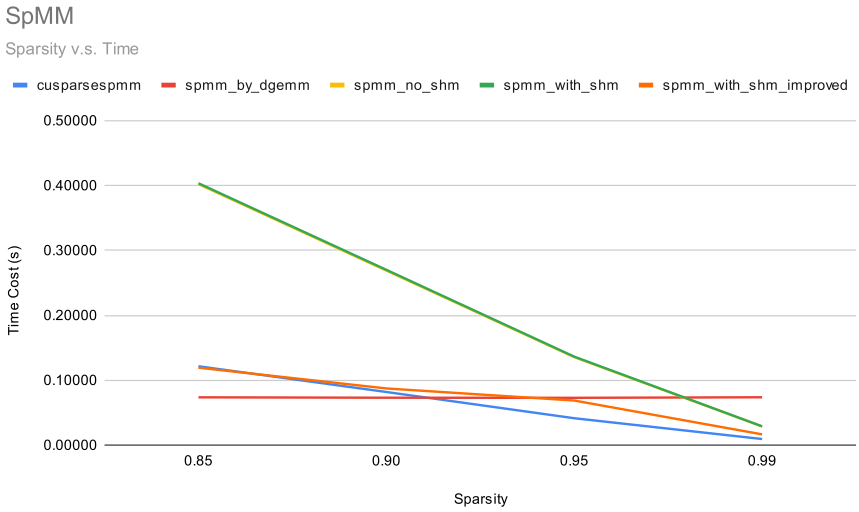


Figure 9. SpMM: Execution Time vs Sparsity of Matrix S with $(8192, 4096)$

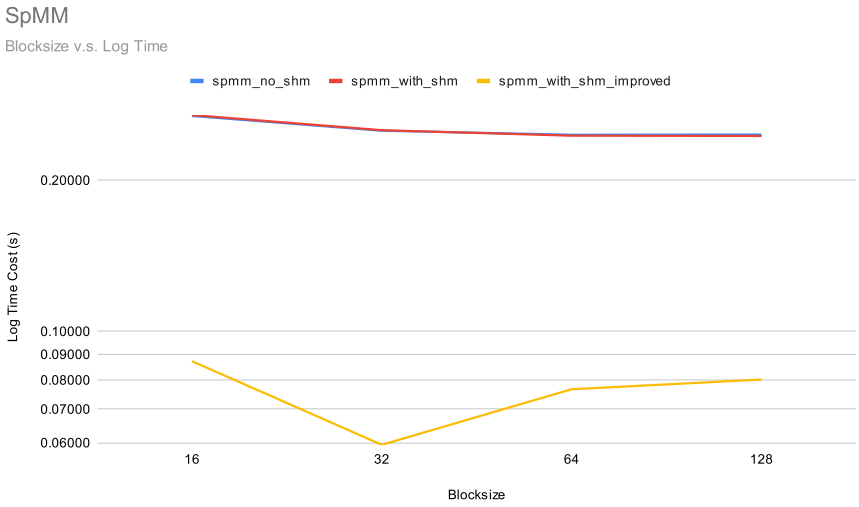


Figure 10. SpMM: Execution Time vs Kernel Launching Block Sizes with $(8192, 4096)$

7.3.2 Sparsity

For the sparsity, here the experiment tested on the four settings: 0.85, 0.9, 0.9, 0.99 (fig. 12). As we expected, the execution time of DGEMM would not be affected by the sparsity. Besides, the kernel using dynamic parallelism cost the longest time might be due to the overhead of the nested kernels. In addition, when the sparsity becomes larger (large than 0.9), we hypothesize that the cuSPARSE starts to treat the sparse matrix as sparse instead of a dense matrix.

7.3.3 Block Size

Unlike the effect of block size on SpMM, for SDDMM, because the operation consists of the dot product with the dense matrix A and the transposed dense matrix A^T, the larger the block size, the longer the sequential time it would take to perform the dot product. Hence, in the figure. 13, the execution time increases as the block size increases.

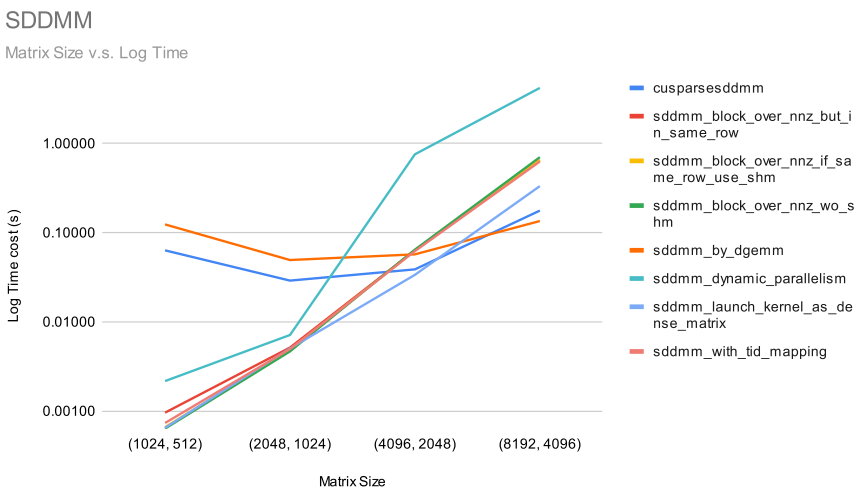


Figure 11. SDDMM: Execution Time vs Matrix Sizes when the sparsity is 0.9

7.4. SpMM_SDDMM Experiments

7.4.1 Matrix Size

As shown in Figure. 14, the fused SpMM_SDDMM kernel (i.e. `sddmm_spmm_block_over_sparse_launch_as_dense_matrix`) did not outperform the method that calls SDDMM and SpMM back to back (i.e. `sddmm_spmm_naive_back2back_calls`). In SpMM_SDDMM, we needs to block over the sparse matrix S to avoid duplicate computations of AA^T; however, the trade-off occurred when writing the results to the output dense matrix C due to the need for atomic write (mentioned in Section 6.2), undermining the gain from fusing two kernels.

7.4.2 Sparsity

In Figure. 15, the experiment showed the result of the execution time of SpMM_SDDMM under different sparsity settings. It is reasonable that when bigger the sparsity is, the more advantage we could get from representing the sparse matrices in CSR format instead of in dense format.

7.4.3 Block Size

For the block size (Fig. 16), because in SpMM_SDDMM, we need to perform dot products on AA^T and $(S \odot AA^T)A$, the block size would directly affect the sequential time running in SDDMM. Hence, the execution time increases as the block size increases.

8. Conclusions

In this project, we implemented a bunch of methods using shared memory, dynamic parallelism, and different blocking techniques for the kernels frequently used in graph neural networks - SDDMM, SpMM, and the fused SDDMM_SpMM - in CUDA C/C++. In addition, we tested different sizes of matrices, the sparsity of the sparse matrix, and the thread block size of the launched kernels with with vendor's CUDA libraries: cuSPARSE and cublasdgemm. Finally, our SpMM outperforms cusparseSpMM when the configuration is $\leq (2048, 1024)$. And our SDDMM kernel runs faster when the matrix size is not too large ($\leq (4096, 2048)$, but the fused kernel does **not** run faster than calling SDDMM and SpMM back to back with our separate kernels as we previous expected potentially due to `atomicAdd`.

9. Team Work

Tzu-Chuan Lin first prototyped the codebase.

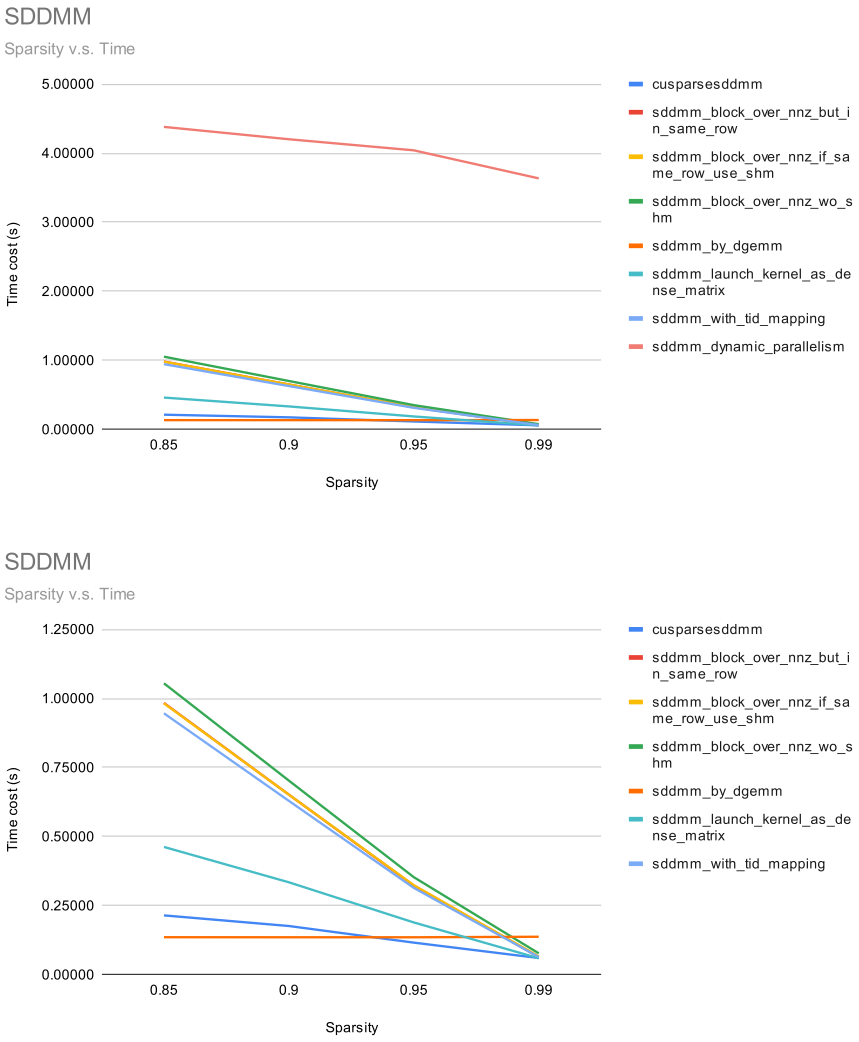


Figure 12. SDDMM: Execution Time vs Sparsity of Matrix S with $(8192, 4096)$

Then, Tzu-Chuan Lin and Pin-Lun Hsu implemented the codebase, while Chin-An Chen did the analyses, plottings and write-up.

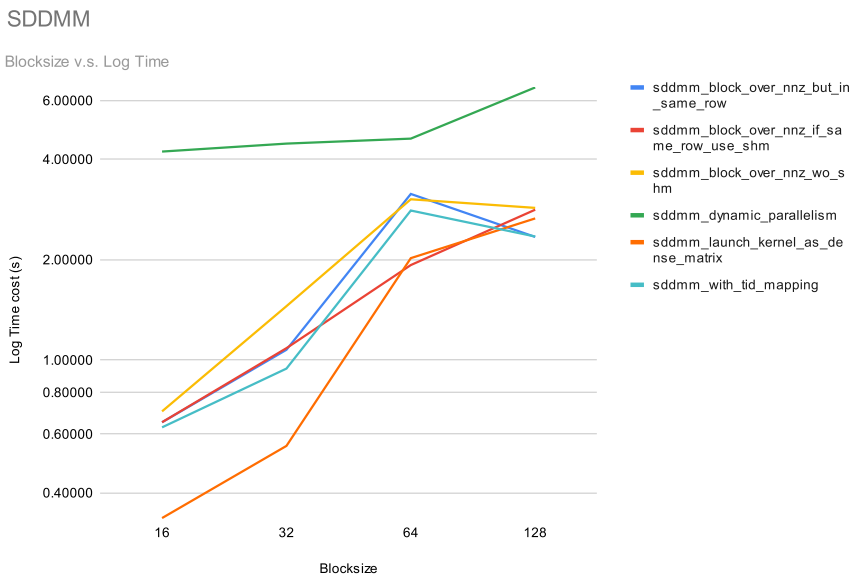


Figure 13. SDDMM: Execution Time vs Kernel Launching Block Sizes with $(8192, 4096)$

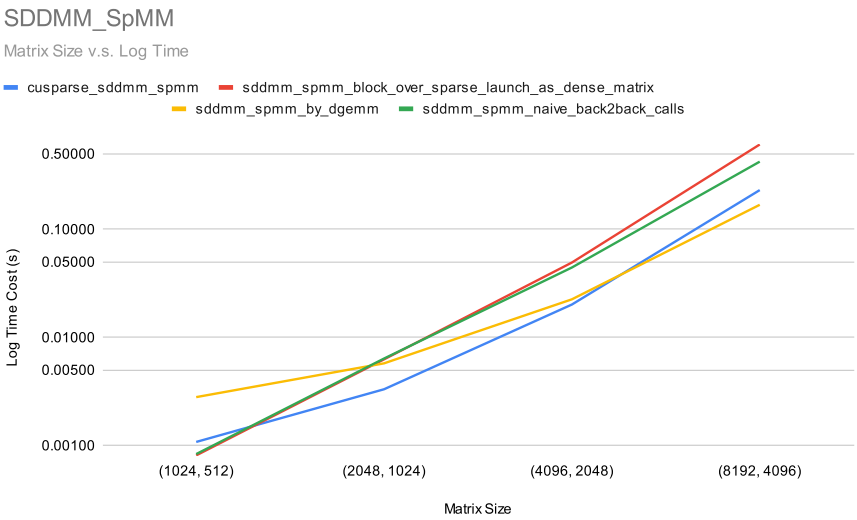


Figure 14. SDDMM_SpMM: Execution Time vs Matrix Sizes when the sparsity is 0.9

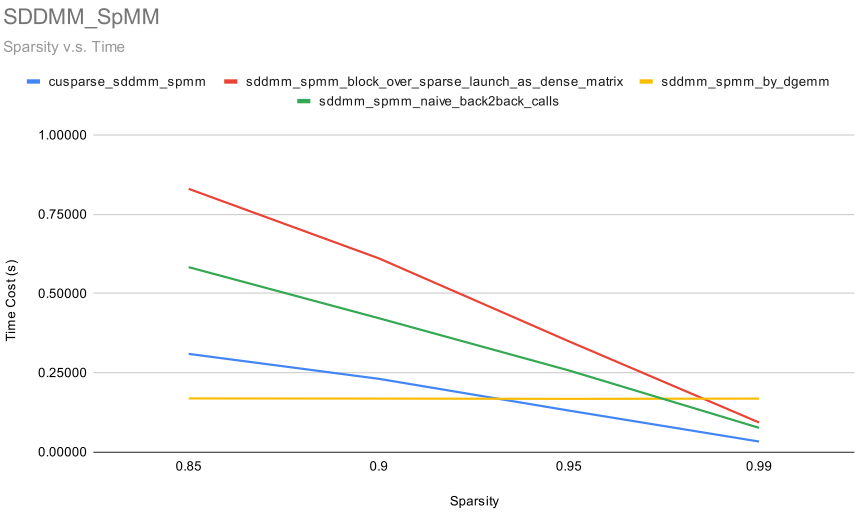


Figure 15. SDDMM_SpMM: Execution Time vs Sparsity of Matrix S with $(8192, 4096)$

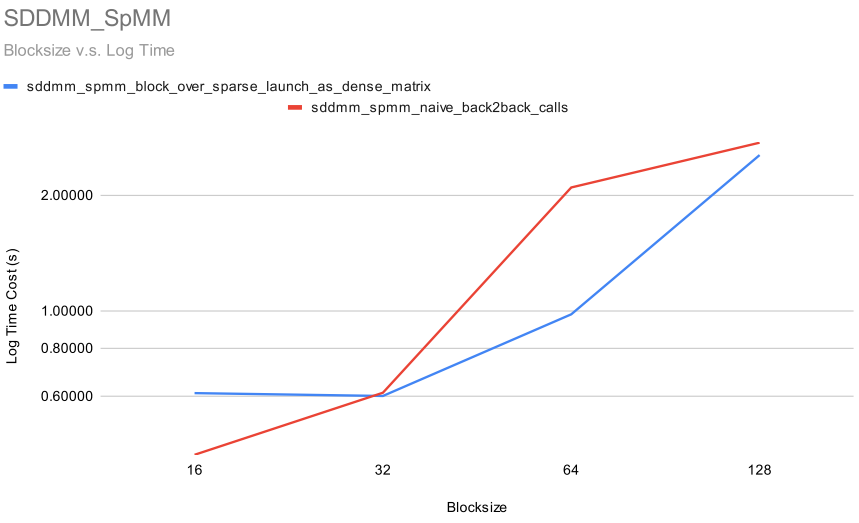


Figure 16. SDDMM_SpMM: Execution Time vs Kernel Launching Block Sizes with $(8192, 4096)$

References

[1] "cuSPARSE." https://docs.nvidia.com/cuda/archive/11.4.0/cusparse/index.html. [Online; accessed 2-May-2022]. 1, 3

[2] "cuBLAS." https://docs.nvidia.com/cuda/archive/11.4.0/cublas/index.html. [Online; accessed 2-May-2022]. 1, 3

[3] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," *CoRR*, vol. abs/2011.06391, 2020. 3

[4] "Dgl." (https://www.dgl.ai/. 3

[5] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," 2020. 4

[6] P. Jiang, C. Hong, and G. Agrawal, *A Novel Data Transformation and Execution Strategy for Accelerating Sparse Matrix Multiplication on GPUs*, p. 376–388. New York, NY, USA: Association for Computing Machinery, 2020. 4

[7] "CUDA Dynamic Parallelism." https://docs.nvidia.com/cuda/archive/11.4.0/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism. [Online; accessed 2-May-2022]. 6

[8] "Cori GPU." https://docs-dev.nersc.gov/cgpu/. [Online; accessed 2-May-2022]. 7