Sparse Matrix Vector Multiplication (spmv)

a.) Discussion of parallel algorithms implemented in your program

The parallel algorithms used in our program are 1 Dimensional Block Diagonal Partitioning and Equal Workload Load Balancing.

Block diagonal partitioning is one technique to improve parallelization. This is breaking down the matrix into smaller blocks and assigning them to different processors or threads so that concurrency is achieved. Each block or submatrix would contain the diagonal elements of the original matrix and possibly some additional overlapping elements to avoid dependency issues. In our algorithm we used a simple one dimensional block diagonal partitioning where the matrix would be divided into blocks of rows. We chose one dimensional block diagonal partitioning as SpMV has a possibility to be skewed along one dimension, when the non zero elements are clustered in the same row or column. One dimensional block diagonal partitioning will help on the load balancing when the matrix is skewed. Moreover the algorithm is simple to and memory efficient which is beneficial to memory constrained environments in our case using Google Colab the default memory limit is only around 12GB RAM.

Load balancing is a method of equally distributing the work to the pool of resources. There are many benefits of load balancing but in this paper we would focus on its effect on the overall performance of an application. As load balancing has a concept of equally distributing the work to the available resources it can improve overall performance of an application by reducing latency. In our paper the load balancing we used is a simple work distribution where we equally distribute the sub matrices which have been partitioned to threads for parallel processing.We used the equal workload for load balancing as we leverage on its simplicity to understand. Furthermore, this algorithm has a low overhead on computational resources which would be beneficial for us since we would be using Google Colab's free version.

b.) Execution time comparison between sequential and parallel

A. Performance Analysis of Sequential SpMV C implementation

For the baseline reference of performance of SpMV, we used the sequential C version as the starting reference of the execution time. It is executed 30 times and sequentially. It could be observed on Table 1 that as the size of the matrix increases, the execution time also increases significantly. To learn the execution time of the kernel for C, the time.h library is used and the function call is the only that was timed.

Table 1: Average execution time (in ms) of the sequential SpMV C implementation.

Matrix Size	Kernel Execution Time
1,024x1,024	3322.70ms
2,048x2,048	13438.70ms
4,096x4,096	55283.26ms
8,192x8,192	242148.90ms

B. Performance Analysis of Parallel SpMV CUDA implementation

The plain Parallel SpMV CUDA implementation performed a lot better than the Sequential C version. This is also executed 30 times and the number of threads that was set is 1,024. The nvprof is utilized to know the performance of the kernel. The time it takes for the memcpy for both HtoD and DtoH was also recorded and can be seen in Table 2. It could be observed that as the matrix size increases, besides the execution time of the kernel, the CUDAmemcpy HtoD increases while the DtoH remains small. But despite the overhead of the memcpy, it is still significantly faster than the Sequential C version due to its parallel processing power.

Table 2: Average execution time (in ms) of the parallel SpMV CUDA implementation.

Matrix Size	Kernel Execution Time	CUDA memcpy HtoD	CUDA memcpy DtoH
1,024x1,024	1.3660ms	124.85us	2.5040us
2,048x2,048	2.9792ms	756.93us	2.4070us
4,096x4,096	4.9423ms	3.4534ms	3.0870us
8,192x8,192	7.6325ms	13.648ms	4.1900us

C. Performance Analysis of Sequential SpMV C implementation with Load Balancing and Block Diagonal Partitioning

Similar to the setup of the Sequential C version, the Sequential SpMV C with Load Balancing and Block Diagonal Partitioning is also executed 30 times. And the performance was measured by timing the function that performs the SpMV using the time.h library of C. It can be observed between Table 1 and Table 3 that for smaller matrices, the Sequential C version performs slightly better but as the matrix size increases, the Sequential C with Load Balancing and Block Diagonal Partitioning is performing slightly better.

In contrast with the plain CUDA implementation, the CUDA implementation still performs better than the C version and it is as expected as there was no parallelism implemented for the C implementations.

Table 3: Average execution time (in ms) of the sequential SpMV C implementation with Load Balancing and Block Diagonal Partitioning.

Matrix Size	Kernel Execution Time
1,024x1,024	3375.73ms
2,048x2,048	13932.56ms
4,096x4,096	54418.30ms
8,192x8,192	217693.30ms

D. Performance Analysis of Parallel SpMV CUDA implementation with Load Balancing and Block Diagonal Partitioning

The Parallel SpMV CUDA with Load Balancing and Block Diagonal Partitioning is also executed 30 times and with 1,024 threads. Similar to the plain Parallel SpMV CUDA. The performance of the Parallel SpMV CUDA with Load

Balancing and Block Diagonal Partitioning as seen in Table 4 is similar to the performance of the plain SpMV seen in Table 2 where it only performed slightly better when the matrix size is of 4,096x4,096 which could be attributed to the generated sparse matrix, where it could have lesser nonzeroes compared to the one used in the plain parallel SpMV.

It could also be observed that similar to the C versions, the CUDA versions performances with Load Balancing and Block Diagonal Partitioning and without Load Balancing and Block Diagonal Partitioning are quite similar to each other. Further proving that applying the two techniques does not improve the performance that much but it might be a different case when the matrix becomes larger as we could observe that the performance improves as the matrix size increases.

Table 4: Average execution time (in ms) of the parallel SpMV CUDA implementation with Load Balancing and Block Diagonal Partitioning.

Matrix Size	Kernel Execution Time	CUDA memcpy HtoD	CUDA memcpy DtoH
1,024x1,024	1.3802ms	137.69us	2.6800us
2,048x2,048	3.0406ms	710.37us	2.7550us
4,096x4,096	4.4541ms	3.2295ms	3.1980us
8,192x8,192	7.7081ms	13.430ms	4.2540us

c.) Detailed analysis and discussion of results

Overall, CUDA runs faster than its C counterpart. However the difference of CUDA parallel vs CUDA with Load Balancing and Block Diagonal Partitioning resulted in the plain SpMV execution running faster in most cases, and the implementation with load balancing and block diagonal partitioning only ran faster on the input size 4096x4096 matrix size. Additionally for the C sequential implementation and C with Load Balancing and Block Diagonal Partitioning it resulted to the sequential implementation to run faster on smaller inputs, 1024x1024 and 2048x2048, and the implementation with Load Balancing and Block Diagonal Partitioning to run faster on larger inputs, 4096x4096 and 8192x8192. But this could be influenced by the sparseness of the matrix that was generated at the time of the test execution.

The CUDA implementation as expected is a lot faster than the C version due to its parallel processing power. And the increase in the execution time is related to the size of the matrix. It was also observed that since the sparse matrix is being generated randomly with at least more than 50% of the values to be 0, there are executions where the speed is faster than the rest due to its sparseness.

The proposed implementation of the SpMV using Load Balancing and Block Diagonal Partitioning for CUDA has a similar performance as with the plain SpMV as observed in the Results and Discussion section. Results have shown that the combination of the two algorithms did not have that much impact on the performance of the SpMV for CUDA. However for C, at smaller inputs the sequential implementation ran faster and at larger inputs the implementation with Block Diagonal Partitioning and Load Balancing ran faster.