Assignment 4 Report - COL380

Name - Ujjwal Mehta

Entry No. - 2020CS10401

Overview of the Assignment:

In this assignment we were supposed to do sparse matrix multiplication in Cuda on GPU of HPC IIT Delhi using Offload programming. The input and output files were in byte format with specifications given in the assignment pdf.

Approach to Multiply Matrix:

1. Input Organisation of Matrices on Host and Device Memory
The code to multiply the sparse matrices as well as output the binary byte file is present in the main.cu file, in this assignment we first read the binary byte file into a dynamic array of k blocks on the host code (where I have defined the struct of Block as Block4 [for m = 4] and Block8[for m = 8]) then after reading the first sparse matrix, we for that array according to first i index of block followed by j index of block using sort function present in Thrust library, and after doing this we calculate the offset for accessing each row of sparse matrix in a separate array so that we can directly get the row of that sparse matrix by the use of this offset. Similarly we read the second matrix and transfer both of this matrices as well as offset arrays to the device memory.

2. Grid and Block Organisation of the Kernel for Performing Matrix Multiplication
In order to multiply the matrices, I have a grid of 2 dimension as well inside that grid each block is also 2 dimensional and the exact dimension of grid and block for the kernel matrix_multiplication launch is as follows:

```
int block_x = ((n/m + 15)/16); // n and m are as given in assignment
dim3 grid(block_x,block_x,1);
dim3 block(16,16,1);
```

Here inside each block thread, it computes a specific row and column element of the result matrix i.e. each thread will calculate a different element of the resultant matrix where the row and column is calculated as follows:

```
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;
```

3. Finally Output writing of Byte File
Finally after the resultant matrix computation in the device memory, we transfer the resultant matrix on the host memory using **cudaMemcpy** and finally we write all the non zero blocks into the output binary file.

Optimizations for Matrix Multiplication:

After trying several approaches, these are the final optimizations that made the matrix multiplication faster.

1. After reading the first matrix, I send it immediately to the device using **cudaMemcpyAsync** which makes reading the second matrix parallel with the data transfer of first matrix, which reduces data transfer latency. We do the similar transfer for second matrix.
2. Second matrix transfer is done in parallel with the first matrix using **cudaStream** by sending them from different streams.
3. In order to reduce the access time of elements of matrix, rather then using **cudaMallocManaged**, I used **cudaMalloc** to store matrices on device.
4. In order to reduce the data transfer further, I used **unsigned int** for computation rather then **long long** which reduced the transfer size by half.
5. In order to further reduce accessing time of elements of matrix, I store them in offset form and for that I transfer them in a sorted manner.
6. Further I store each block of a matrix in a linear manner rather then in 2D form, which helps in better access time.

Observations:

After testing my code on various test cases I got the following runtime:

n value	m value	k value	Runtime
20000	4	25000	13 Seconds
8192	4	100000	50 Seconds
32128	8	2500000	213 Seconds
32124	4	10000000	520 Seconds
10000	8	690	1.5 Seconds

Acknowledgement:

I thank HPC (High performance computer) IIT Delhi for letting me use its resources.