Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

I have uploaded all my code file in Jarvis under home/rchhatrala/HW4 directory.

- **File Details:**
  - **GPUMatrixNorm.cu**
    - Cuda parallel program file.
  - **gpu.c**
    - Serial code for matrix normalization give.

- **GPU cuda code explanation:**
  - In this assignment we are implementing Matrix normalization using GPU with CUDA. In this normalization process we are first calculating the mean and standard deviation for every column as we are performing column matrix normalization. Then we are using these calculated mean and standard daviation to perform column matrix normalization for each element of matrix.
  - In my implementation I also perform sum reduction in **__global__ void meanCalculation** and **__global__ void calculate_SD** function in GPUMatrixNorm.cu file.

- **Code Explanation of GPUMatrixNorm.cu**

In cuda programming we have host function and device functions. Host functions are the functions of C programing which runs on CPU and Device function are the CUDA functions which run on the GPU parallely by many threads.

Here in my cuda program for matrix normalization I used the dim3 datatype for block and threads.
I define block size of 1*N(X*Y) so I assume that each column is one block and define number of thread as nt*1(X*Y) where nt is passed as argument while running the code.

Host Function:

- **Main() function :**
  - Main() is the host function. Execution of program start from here.
  - In this function we are allocating memory for host variables as well as device variables. Then we will initialize inputs which input matrix, Matrix Size and number of threads.
  - Memory allocation to device variables.

```
cudaMalloc(&d_in, sizeof2d);
cudaMalloc(&d_out, sizeof2d);
cudaMalloc(&d_mean, sizeof1d);
cudaMalloc(&d_sd, sizeof1d);
```

- After we initialize matrix in host variable then we copy that matrix to device variable by using the following function.
  - **cudaMemcpy(d_in, A, sizeof2d, cudaMemcpyHostToDevice);**

- After we reveive inputs at device GPU, it will compute the result and copy device variable bake to CPU by below command.
  - **cudaMemcpy(B, d_out, N * N * sizeof(float), cudaMemcpyDeviceToHost);**

❖ **void parameters() functions:**
- This function will deal with the arguments passed during the run time of program.
- This function will set the Matrix dimension variable N, it also set the number of thread parameter for the program.
- I define the 8000 as the max matrix dimension and 1024 as max number of thread per block, So this function will take care of both of these parameters. If will exceed the valued of these both parameters then max limit then program will exit and provide the appropriate error message.

Device Functions:

❖ **__global__ void meanCalculation:**
- This function will calculate the mean for the every column and store it in d_mean array.
- I also perform sum reduction in this function. First every thread will do the sum of their element if it works for more than one element and store that intermediate sum in col_data array. Then thread 0 of each block will perform total sum for column and then mean for that column will be calculated from this total sum.
- Here __global__ keyword will show the scope of the function. It shows that this is the device function. This device function is call from Host main() function.
- While calling this function we have to pass the number of block as first argument of the three triangular bracket and second argument is number of thread per block.
  - **meanCalculation<<<dimGrid, dimBlock, sizeof1d>>>(d_in, d_mean, N,nt);**
- dimGrid shows the number of block. In dim3 datatype I set dimGrid as Nx1.
- dimBlock shows the number of thread per block which we will pass at run time as nt.

❖ **__global__ void calculate_SD:**
- This function will perform the standard deviation calculation for each column and store the standard deviation form each column in d_sd.
- Here in this function every thread will do work for their part of elements and then store the intermediate result in col_sd_data. Then thread 0 will perform sum reduction on all thread work that is store in col_sd_data and then store final sum result in col_sd_total.

After dividing col_sd_total with N(Matrix Dimension) we will get the final standard deviation for each column and store it in d_sd.

- **__global__ void matrixColumnNorm:**
  - This is the main normalization function which will perform normalization on each element of matrix.
  - This function will use the mean and standard deviation for each column stored in d_mean and d_sd respectively and store the final result into the d_out array.
  - After this function finishes its execution we get our final column normalized matrix. Then main() function will copy back the final output from device(GPU) to host(CPU) by below function.
    - **cudaMemcpy(B, d_out, N * N * sizeof(float), cudaMemcpyDeviceToHost);**

- **Cuda function that are used in the program.**

  - **cudaMalloc():** This function work similar as free() function of C. This function will release the memory that are allocated to device variables.

  - **cudaMemcpy():** Copy host data to device and also used to copy device data back to host.
    - **cudaMemcpy(destination, source, size of data , cudaMemcpyHostToDevice);**
    - Last argument is tag which shows data is coping from where to where like **cudaMemcpyHostToDevice** tells us that data is coping from host to device.

  - **cudaDeviceSynchronize():** A device kernel function launch is asynchronous. So it returns control to the CPU thread immediately after starting up the GPU process, before the kernel has finished executing. So we can perform wait for Kernel function to finish by using this function.
    - We are this function because we want the result of mean function and standard deviation

  - **cudaEventSynchronize():** by using this function we can wait fro cuda event to complete.

  - **cudaFree():** By calling this function we can frees the pointer allocated during cudaMelloc() function.

Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

- **Running instruction for CUDA program:-**
  GPUMatrixNorm.cu this cuda code file is present in my Jarvis account in rchhatrala/HW4 directory.

  1. **vi GPUMatrixNorm.cu**
     - copy code from **GPUMatrixNorm.cu** provided in attach submittion.
     - save it by :wq.
  2. **qlogin –q interactive.q**
  3. **nvcc GPUMatrixNorm.cu -o GPUMatrixNorm**
     - Above command will compile the CUDA code and generate the **GPUMatrixNorm** object file.
  4. **./ GPUMatrixNorm 4000 128**
     - Here in first argument we are passing Matrix Dimension(N) and in second argument we are passing number of thread per block(nt).

- **Analysis of CUDA program:**
  To analysis the CUDA program we are using the three parameters.
  - Cuda program execution time
  - Bandwidth used by threads.
    - Size of matrix / time
  - Through put of execution(Gflop/sec)
    - Total work / time

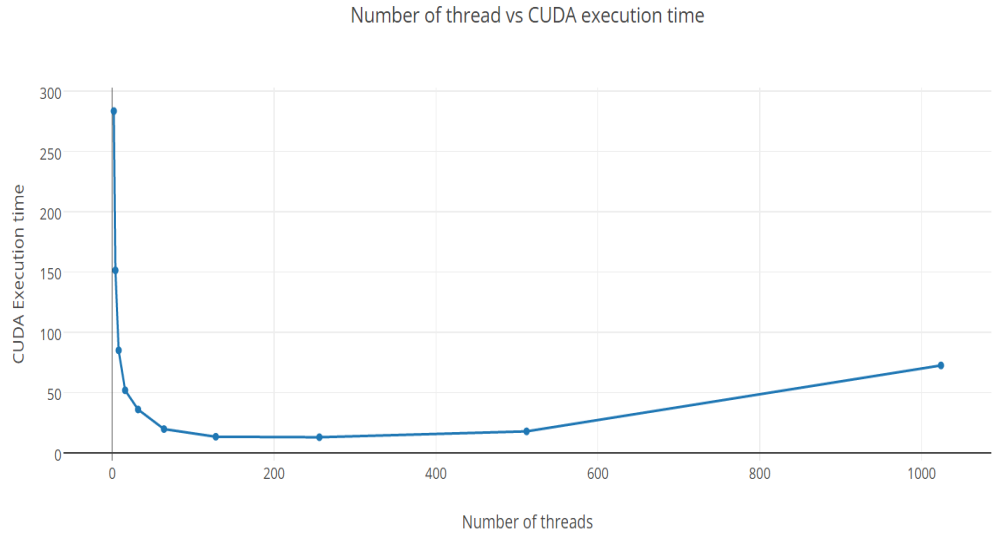
- I am performing the test for matrix size N=2000 and N=4000 and we are also changing the number of thread and recording the above parameters.

- For N=2000

Number of Thread	CUDA Execution time	Band Width	Throughput(Gflop/sec)
2	283.526	0.1128	0.0566
4	151.354	0.2164	0.106
8	84.977	0.3765	0.1888
16	51.8268	0.6174	0.3096
32	35.911	0.891	0.4469
64	19.6502	1.6284	0.81677
128	13.288	2.408	1.2077
256	12.879	2.4845	1.2461
512	17.7456	1.8032	0.9044
1024	72.453	0.441	0.221

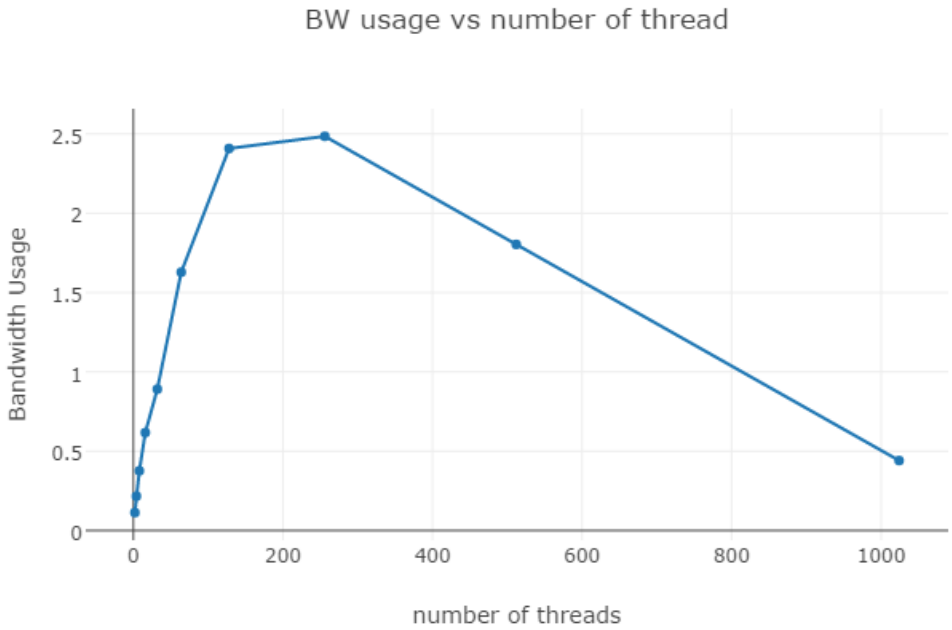
Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

- Plot for number of thread vs CUDA execution time.



  - From above graph we can see that upto 256 number of thread CUDA Execution time is decreasing and after that number of thread(nt=512,1024) time is againg start increases.

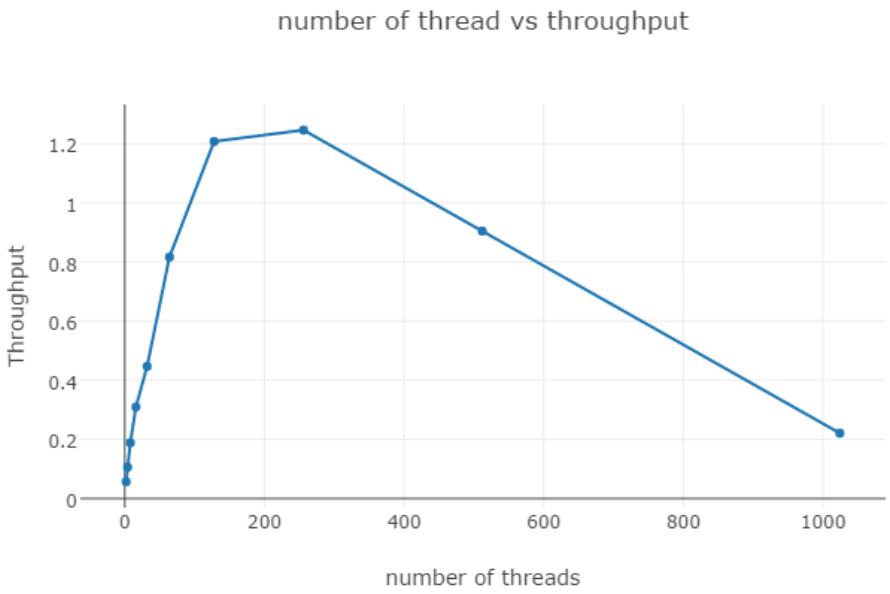
- Plot for number of thread vs Bandwidth usage.



  - From above graph we can see tha upto 256 thread bandwidth usage is increases as upto that many thread cuda execution time decreases.
  - After 256 thread Bandwidth usage decreases as cuda time increases as we divide the matrix size with time.

Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

- Polt for number of thread vs throughput.



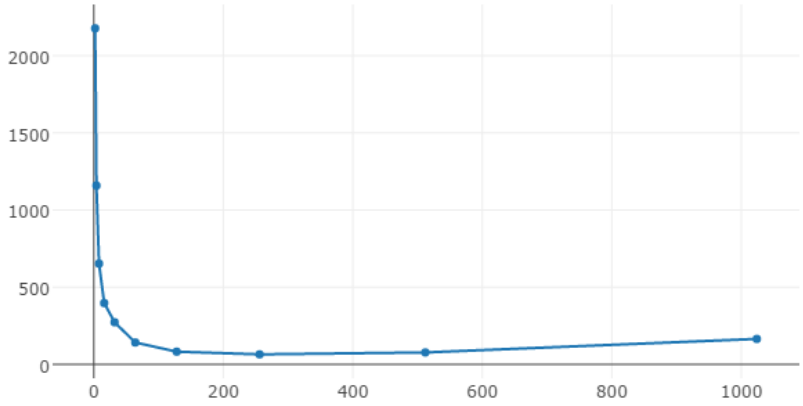
  - From above we are seeing the similar result as bandwidth usage graph. Here as well through put increases upto 256 thread and then decreases. This happens because throughput is the total_work / total_time, so after 256 thread time increases so throughput decreases as both have inverse relation with each other.

➢ For N(Matrix size)=4000

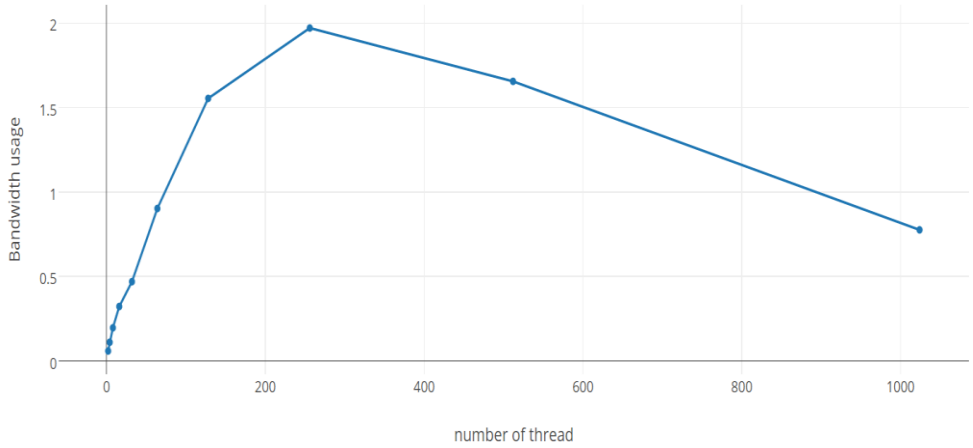
Number of Thread	CUDA Execution time	Band Width	Throughput(Gflop/sec)
2	2176.83	0.05877	0.0284
4	1158.22	0.1105	0.0553
8	652.437	0.1961	0.0982
16	397.178	0.3222	0.1614
32	272.751	0.4692	0.235
64	141.834	0.9024	0.4519
128	82.3659	1.554	0.7783
256	64.948	1.9707	0.9807
512	77.354	1.6549	0.8288
1024	164.954	0.7759	0.3886

Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

- **Plot for Number of thead vs CUDA time**

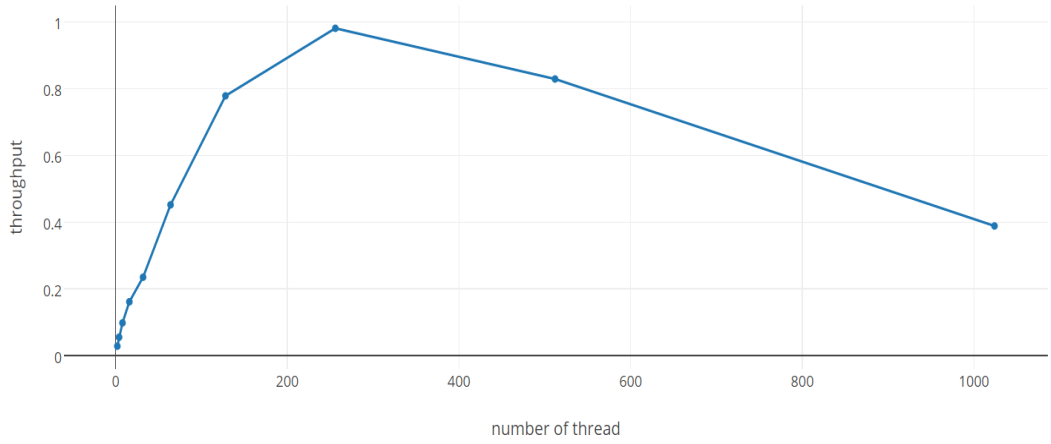


- **Plot for number of thread vs Bandwidth usage.**



Riten Chhatrala
rchhatrala@hawk.iit.edu
CWID:-A20385102

- **Plot for number of thread vs Throughput**



From matrix dimension N=4000 we are getting similar result as N=2000.

Conclusion:

- So from above analysis we can see that the up to 256 number of thread, CUDA program execution time decreases, Bandwidth usage increase and Throughput is increases. So for 256 thread all three analysis parameter showning their best values.
- After 256 number of thread CUDA execution time increases, bandwidth usage decreases and throughput decreases. Here throughput and bandwidth usage is inversely promotional to the execution time.
- Here we can say that this thing happens because GPU environment has some number of ALUs and threads will run on this ALUs in parallel. So if we increases threads more than number of ALUs then threads need to perform context switching. Context switching of threads cost more in execution of program which leads to increase in execution time, decrease in Bandwidth usage and also decreases the throughput.