Lan Nguyen – A20430901
CS546 – Professor Zhiling Lan

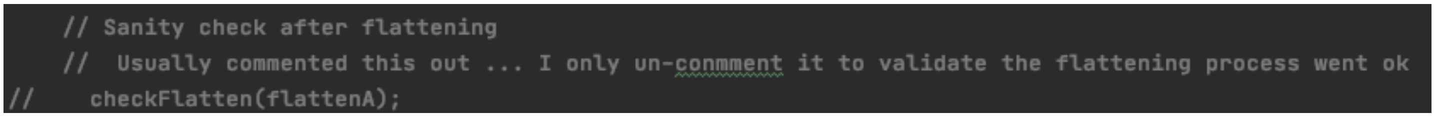
Homework #4

Matrix normalization using CUDA programming

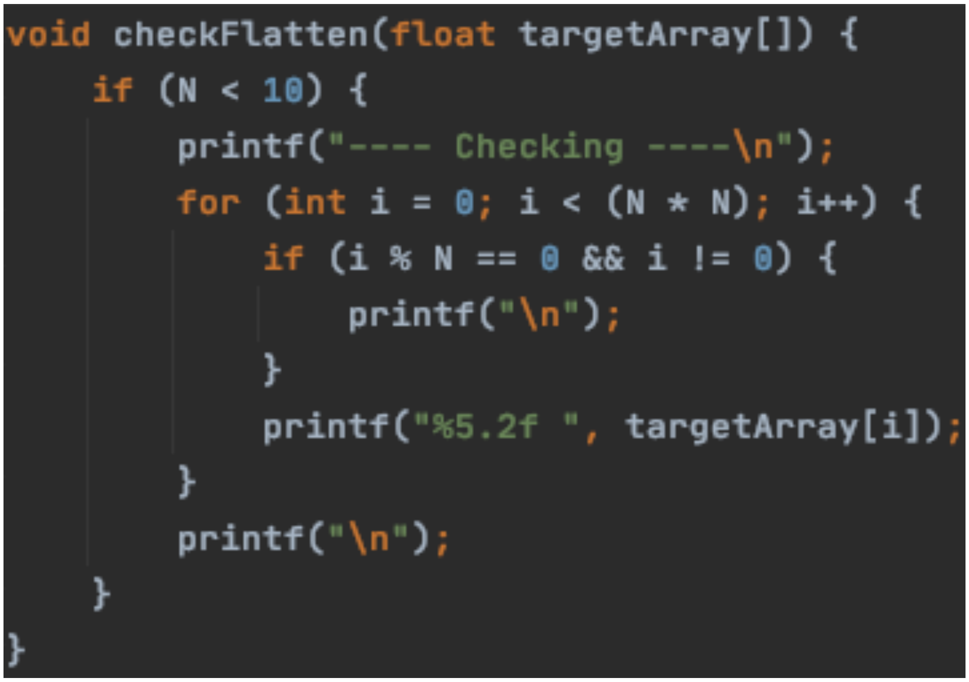
In this part, I will be discussing my process of parallelizing the matrix normalization using CUDA programming with an Nvidia V100 GPU. My hypothesis and assumption prior going into writing the parallelized code for this is that the more processors can be assigned a task to handle a computation task, the better the performance. In addition, I also keep in mind that I should parallelize the most outer loop for an optimal performance. Thus, I designed my code in a way so that the number of matrix dimension is equally divisible by the number of assigned processors. During the process, I encountered several issues. However, these issues were resolved:

- I have had trouble with copying a 2D matrix to GPU device . Understanding how CUDA works, it seems like it's much easier to flatten 2D to 1D then copy from host to device than using CUDA's cudaMemcpy2D function
- Initializing inputs initially was totally random. Thus, I set the random seed to a fixed value of 22
- For verifying the result, I used N=9 so I could have the feasibility of manually verifying it
- During result verification, I also needed to add the print_arrays and checkFlatten functions so that it can display both arrays
- Somehow my serial code requires an additional argument to compile. I believe this is due to math function in the serial code … -lm

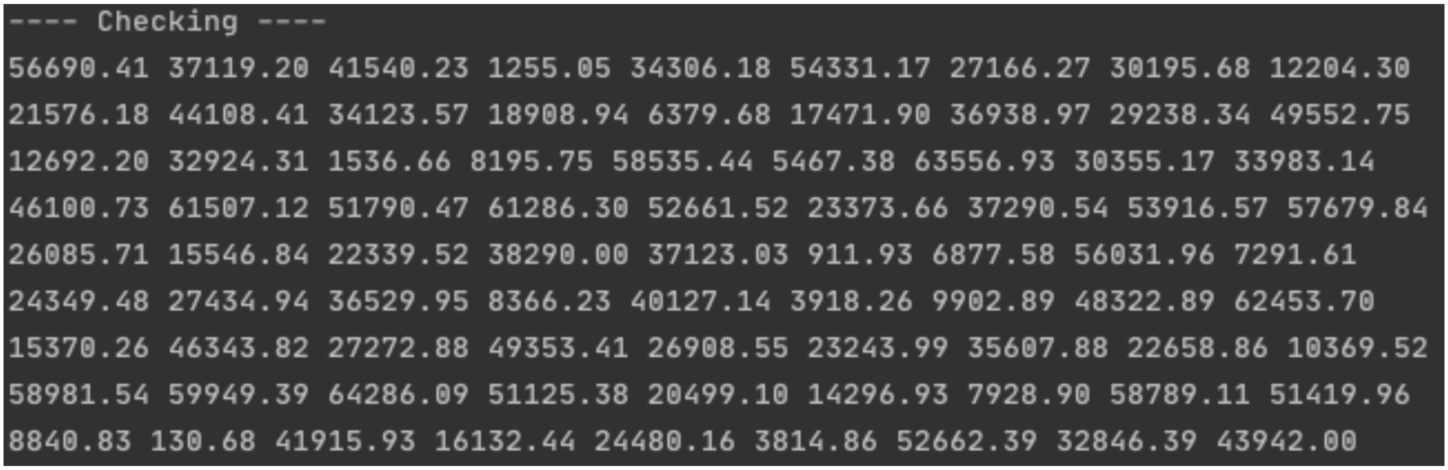
Note that I also leave an un-commented part of the code so that we could easily debug/see the data that were initialized and flattened:



```
    // Sanity check after flattening
    //  Usually commented this out ... I only un-conmment it to validate the flattening process went ok
//    checkFlatten(flattenA);
```



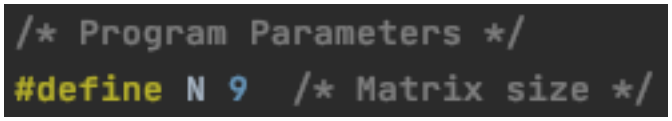
```
void checkFlatten(float targetArray[]) {
    if (N < 10) {
        printf("---- Checking ----\n");
        for (int i = 0; i < (N * N); i++) {
            if (i % N == 0 && i != 0) {
                printf("\n");
            }
            printf("%5.2f ", targetArray[i]);
        }
        printf("\n");
    }
}
```



```
---- Checking ----
56690.41 37119.20 41540.23 1255.05 34306.18 54331.17 27166.27 30195.68 12204.30
21576.18 44108.41 34123.57 18908.94 6379.68 17471.90 36938.97 29238.34 49552.75
12692.20 32924.31 1536.66 8195.75 58535.44 5467.38 63556.93 30355.17 33983.14
46100.73 61507.12 51790.47 61286.30 52661.52 23373.66 37290.54 53916.57 57679.84
26085.71 15546.84 22339.52 38290.00 37123.03 911.93 6877.58 56031.96 7291.61
24349.48 27434.94 36529.95 8366.23 40127.14 3918.26 9902.89 48322.89 62453.70
15370.26 46343.82 27272.88 49353.41 26908.55 23243.99 35607.88 22658.86 10369.52
58981.54 59949.39 64286.09 51125.38 20499.10 14296.93 7928.90 58789.11 51419.96
8840.83 130.68 41915.93 16132.44 24480.16 3814.86 52662.39 32846.39 43942.00
```

Important note: Please kindly adjust size of N on line 11 if you want to change the size of matrix.

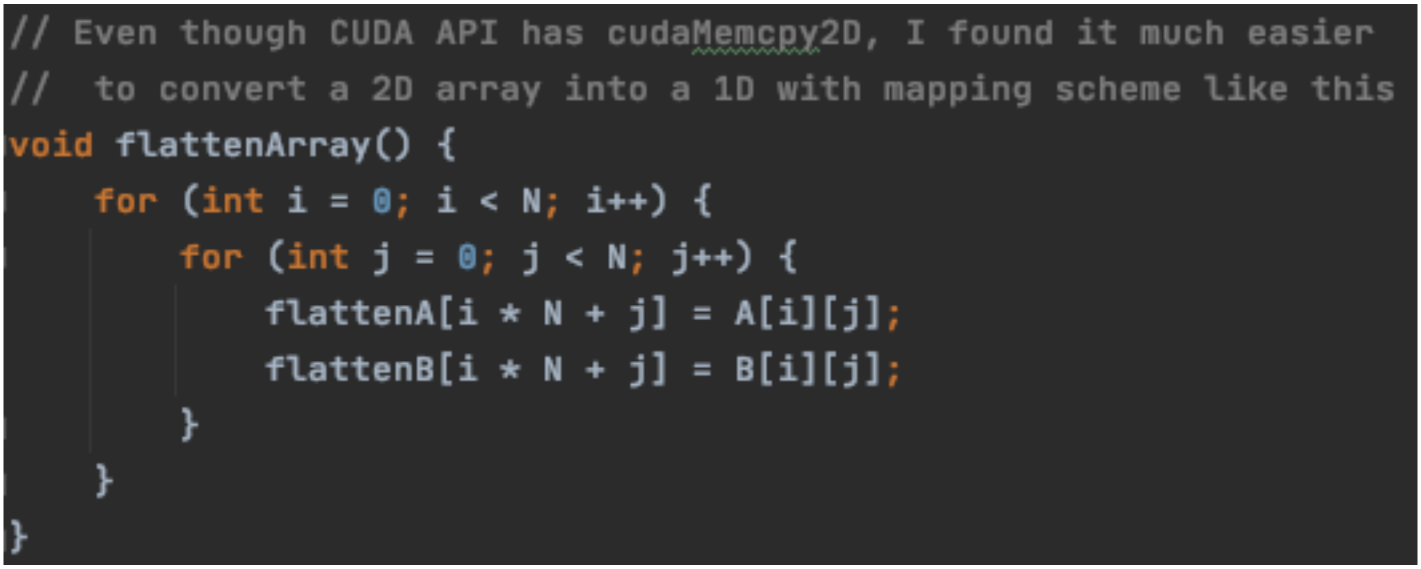
Please also note that if N > 11,000, there might be an error since the data is too large



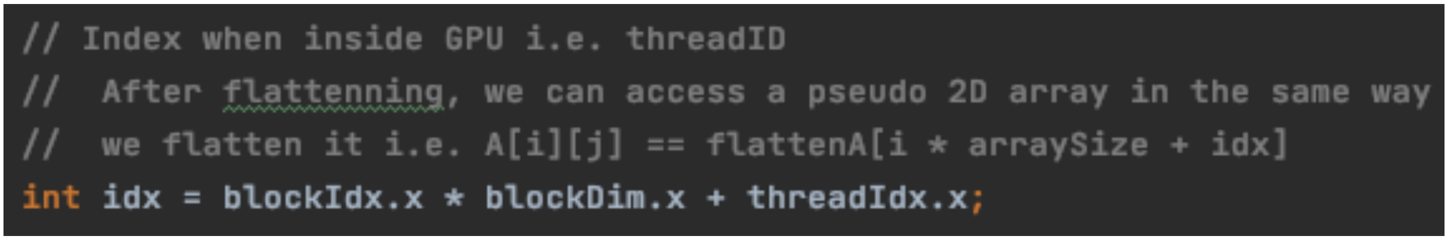
```
/* Program Parameters */
#define N 9  /* Matrix size */
```

Flattening:

This flattening function will allow us to be able to send a pseudo 2D array in a 1D array form from host to device. At the same time, we can easily map its 2D value simply with index when inside a GPU



```
// Even though CUDA API has cudaMemcpy2D, I found it much easier
//  to convert a 2D array into a 1D with mapping scheme like this
void flattenArray() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            flattenA[i * N + j] = A[i][j];
            flattenB[i * N + j] = B[i][j];
        }
    }
}
```



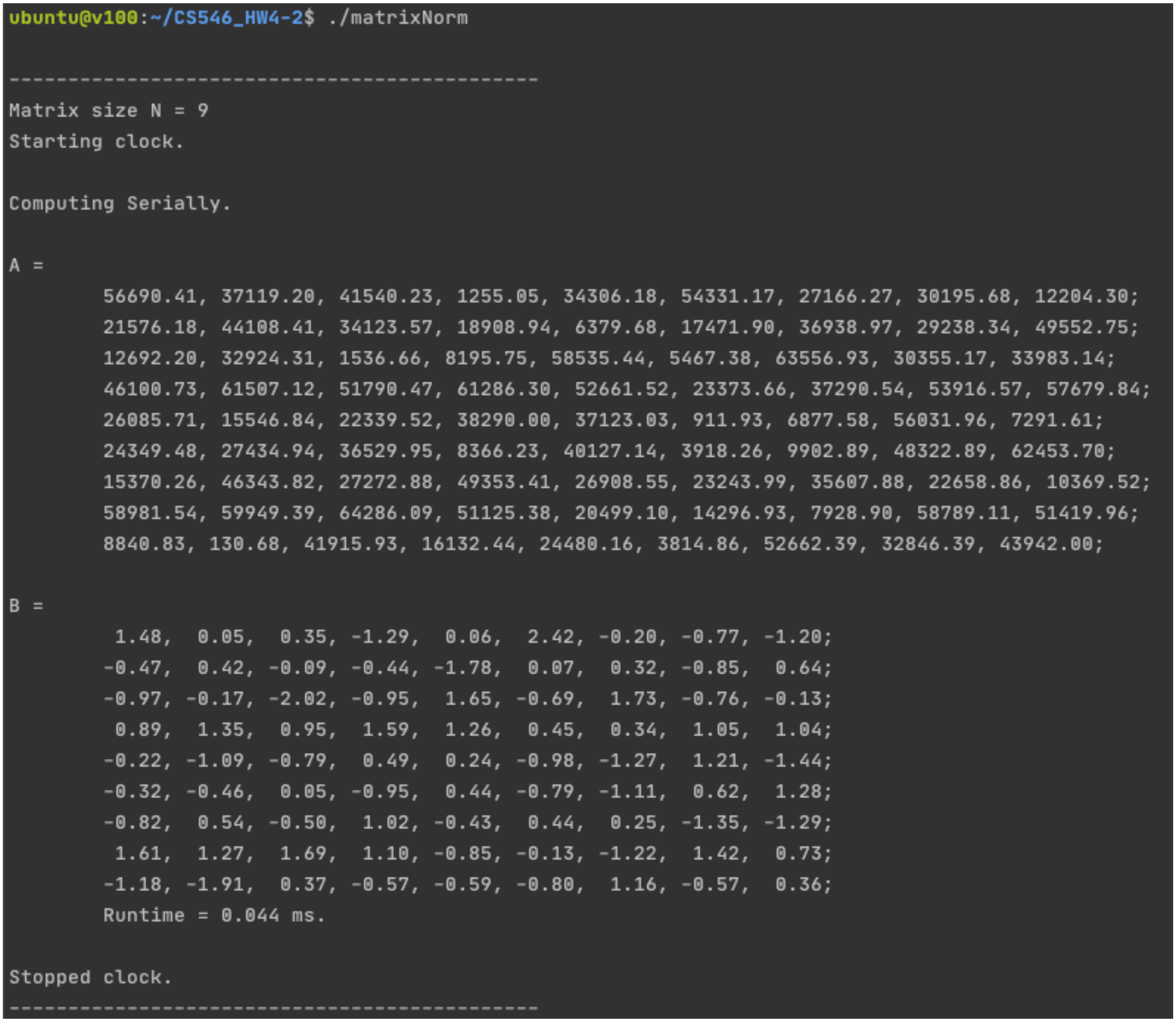
```
// Index when inside GPU i.e. threadID
//  After flattenning, we can access a pseudo 2D array in the same way
//  we flatten it i.e. A[i][j] == flattenA[i * arraySize + idx]
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

Lan Nguyen – A20430901
CS546 – Professor Zhiling Lan

Correctness of a program:

Serial processing

1. Compile: *gcc matrixNorm.c -o matrixNorm -lm*
   Execute: ./matrixNorm
   ****NOTE: Assuming N = 9*



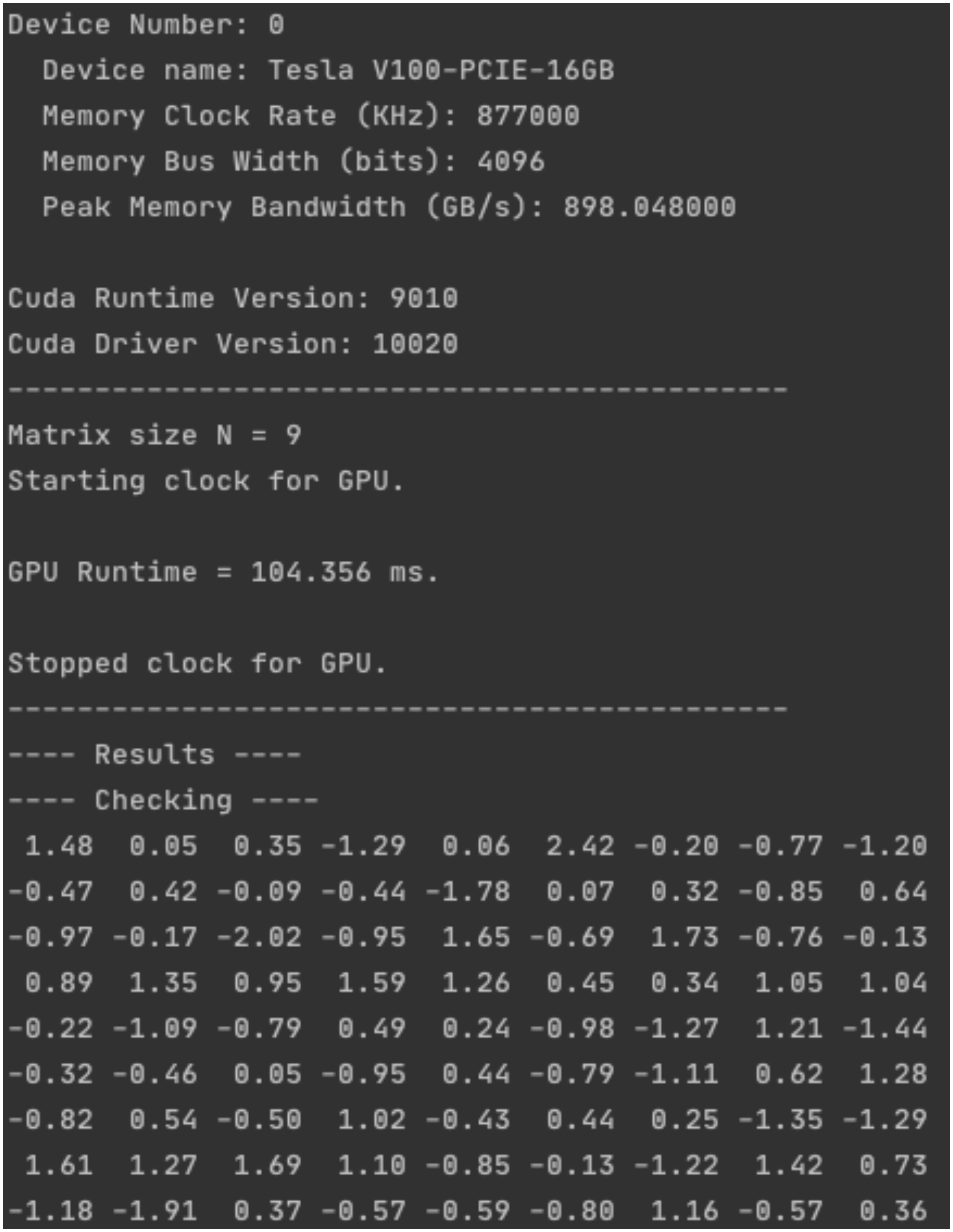
```
ubuntu@v100:~/CS546_HW4-2$ ./matrixNorm

---------------------------------------------
Matrix size N = 9
Starting clock.

Computing Serially.

A =
        56690.41, 37119.20, 41540.23, 1255.05, 34306.18, 54331.17, 27166.27, 30195.68, 12204.30;
        21576.18, 44108.41, 34123.57, 18908.94, 6379.68, 17471.90, 36938.97, 29238.34, 49552.75;
        12692.20, 32924.31, 1536.66, 8195.75, 58535.44, 5467.38, 63556.93, 30355.17, 33983.14;
        46100.73, 61507.12, 51790.47, 61286.30, 52661.52, 23373.66, 37290.54, 53916.57, 57679.84;
        26085.71, 15546.84, 22339.52, 38290.00, 37123.03, 911.93, 6877.58, 56031.96, 7291.61;
        24349.48, 27434.94, 36529.95, 8366.23, 40127.14, 3918.26, 9902.89, 48322.89, 62453.70;
        15370.26, 46343.82, 27272.88, 49353.41, 26908.55, 23243.99, 35607.88, 22658.86, 10369.52;
        58981.54, 59949.39, 64286.09, 51125.38, 20499.10, 14296.93, 7928.90, 58789.11, 51419.96;
        8840.83, 130.68, 41915.93, 16132.44, 24480.16, 3814.86, 52662.39, 32846.39, 43942.00;

B =
         1.48,  0.05,  0.35, -1.29,  0.06,  2.42, -0.20, -0.77, -1.20;
        -0.47,  0.42, -0.09, -0.44, -1.78,  0.07,  0.32, -0.85,  0.64;
        -0.97, -0.17, -2.02, -0.95,  1.65, -0.69,  1.73, -0.76, -0.13;
         0.89,  1.35,  0.95,  1.59,  1.26,  0.45,  0.34,  1.05,  1.04;
        -0.22, -1.09, -0.79,  0.49,  0.24, -0.98, -1.27,  1.21, -1.44;
        -0.32, -0.46,  0.05, -0.95,  0.44, -0.79, -1.11,  0.62,  1.28;
        -0.82,  0.54, -0.50,  1.02, -0.43,  0.44,  0.25, -1.35, -1.29;
         1.61,  1.27,  1.69,  1.10, -0.85, -0.13, -1.22,  1.42,  0.73;
        -1.18, -1.91,  0.37, -0.57, -0.59, -0.80,  1.16, -0.57,  0.36;
        Runtime = 0.044 ms.

Stopped clock.
---------------------------------------------
```

Lan Nguyen – A20430901
CS546 – Professor Zhiling Lan

Parallel processing (CUDA)

2 ways to Run:

1. Compile: *nvcc matrixNormCuda.cu -o matrixNormCuda*
   Execute: ./matrixNormCuda
2. *make*
   ./matrixNormCuda
   ***NOTE: *Assuming N = 9*



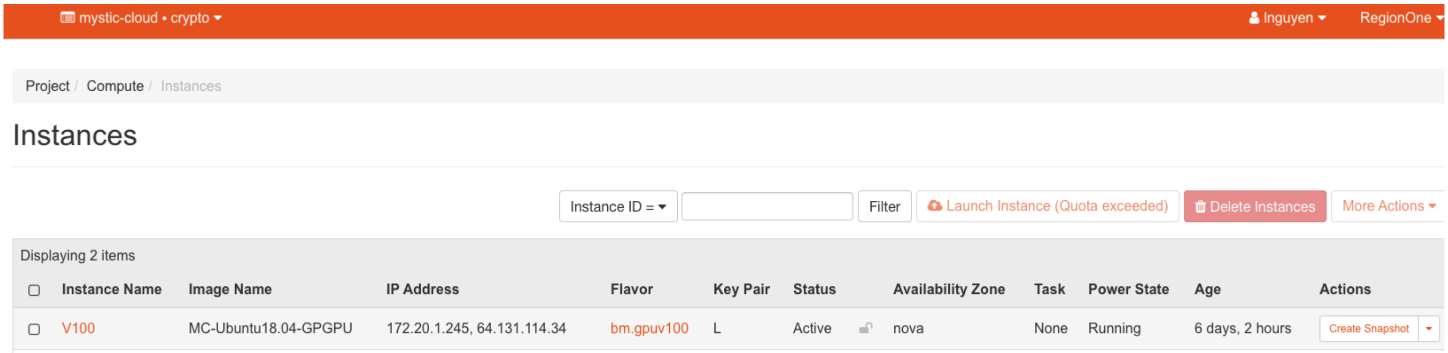
```
Device Number: 0
  Device name: Tesla V100-PCIE-16GB
  Memory Clock Rate (KHz): 877000
  Memory Bus Width (bits): 4096
  Peak Memory Bandwidth (GB/s): 898.048000

Cuda Runtime Version: 9010
Cuda Driver Version: 10020
---------------------------------------------
Matrix size N = 9
Starting clock for GPU.

GPU Runtime = 104.356 ms.

Stopped clock for GPU.
---------------------------------------------
---- Results ----
---- Checking ----
 1.48  0.05  0.35 -1.29  0.06  2.42 -0.20 -0.77 -1.20
-0.47  0.42 -0.09 -0.44 -1.78  0.07  0.32 -0.85  0.64
-0.97 -0.17 -2.02 -0.95  1.65 -0.69  1.73 -0.76 -0.13
 0.89  1.35  0.95  1.59  1.26  0.45  0.34  1.05  1.04
-0.22 -1.09 -0.79  0.49  0.24 -0.98 -1.27  1.21 -1.44
-0.32 -0.46  0.05 -0.95  0.44 -0.79 -1.11  0.62  1.28
-0.82  0.54 -0.50  1.02 -0.43  0.44  0.25 -1.35 -1.29
 1.61  1.27  1.69  1.10 -0.85 -0.13 -1.22  1.42  0.73
-1.18 -1.91  0.37 -0.57 -0.59 -0.80  1.16 -0.57  0.36
```

Potential improvement:

- Since I couldn't make my code run with N size with size a bit larger than 11,000. I'm really curious about it and I would love to make the current code be able to run a larger size N

Machine used:

- Since it was so hard to lease an instance on Chameleon, I develop and ran this code using an instance created from IIT Mystic system. I believe it is almost identical if not the same as Chameleon

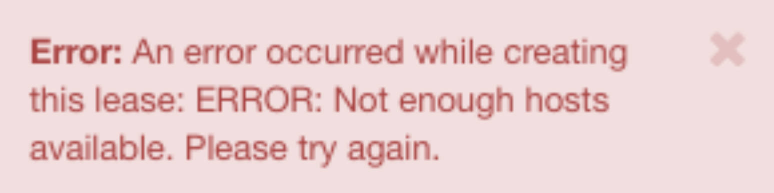


Instances

Displaying 2 items

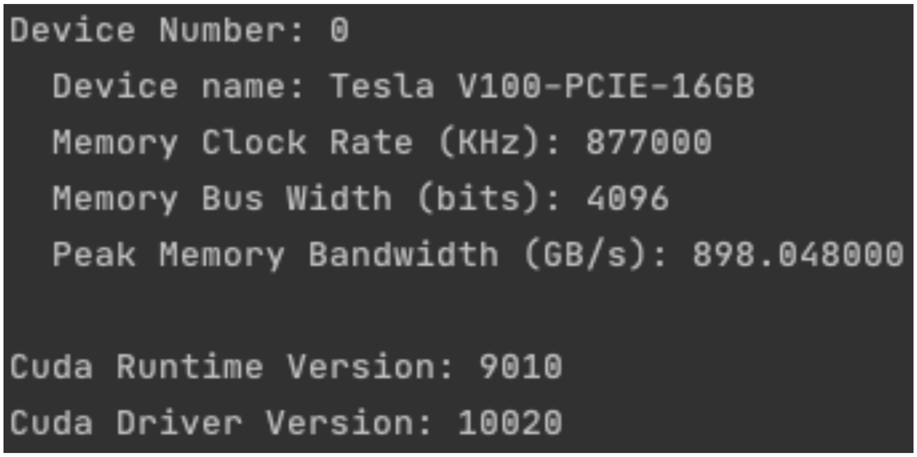
□	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
□	V100	MC-Ubuntu18.04-GPGPU	172.20.1.245, 64.131.114.34	bm.gpuv100	L	Active	nova	None	Running	6 days, 2 hours	Create Snapshot

- Despite multiple attempts to run on Chameleon, error was:



Error: An error occurred while creating this lease: ERROR: Not enough hosts available. Please try again.

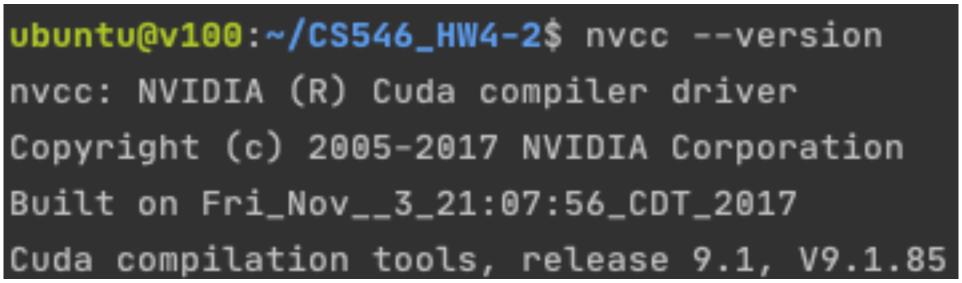
- Specs: Ubuntu 18.04.4 LTS | Release 18.04 | Codename: bionic
  - These specs were retrieved from:
    - cudaGetDeviceProperties();
    - cudaRuntimeGetVersion();
    - cudaDriverGetVersion();



```
Device Number: 0
  Device name: Tesla V100-PCIE-16GB
  Memory Clock Rate (KHz): 877000
  Memory Bus Width (bits): 4096
  Peak Memory Bandwidth (GB/s): 898.048000

Cuda Runtime Version: 9010
Cuda Driver Version: 10020
```

Lan Nguyen – A20430901
CS546 – Professor Zhiling Lan

- nvcc -version



```
ubuntu@v100:~/CS546_HW4-2$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2017 NVIDIA Corporation
Built on Fri_Nov__3_21:07:56_CDT_2017
Cuda compilation tools, release 9.1, V9.1.85
```

Upon receiving outputs that are correct given that I have correctly managed to configure the code so that it utilizes Cuda-capable GPU to parallelize the matrix normalization computation tasks. I have gone ahead and start the performance analysis where I will perform the following:

i. Set various matrix size
***NOTE: Keep in mind that I have hard-coded the random seed = 22
ii. Measuring and discussing runtimes for different size of N
[6,000; 7,000; 8,000; 9,000; 10,000; 11,000]
iii. Plot the measured runtimes

Performance Analysis:

Runtimes INCLUDES distributing data (communication costs)

N Size	Serial runtimes	V100 Runtimes	Difference
6,000	3,235.83 ms	208.694 ms	93.6%
7,000	5,433.06 ms	183.298 ms	96.6%
8,000	6,449.33 ms	241.766 ms	96.3%
9,000	9,679.74 ms	233.11 ms	97.6%
10,000	12,042.2 ms	259.375 ms	97.8%
11,000	14,070.4 ms	289.085 ms	97.9%

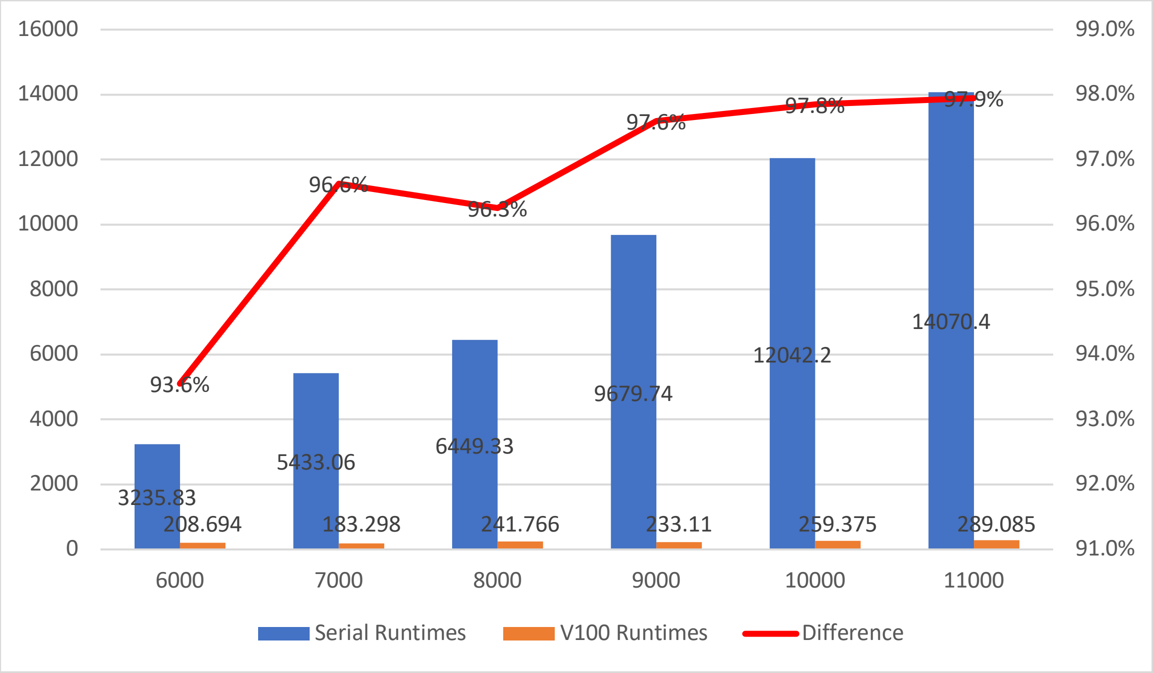


Chart note:

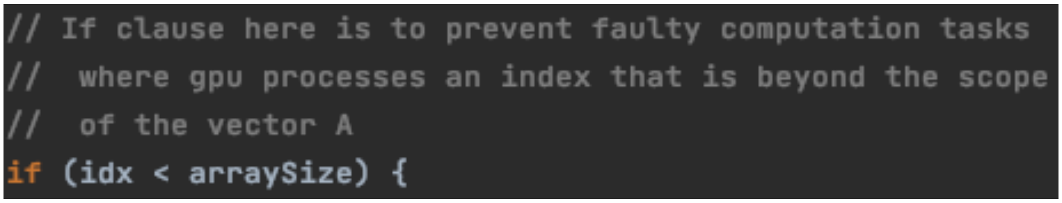
- $Difference = \left|\frac{New - Old}{Old}\right|$

According to the performance data and plot above, I strongly believe the code scale very well. I noticed that sometimes, the speed is not consistent i.e. With same N size, timing can vary a bit. However, the most important thing is that computing tasks carried out by GPU is awesome. We can clearly see that as N size increases, serial runtimes is heavily being affected while V100 runtimes went up a bit or remain the same across all N size. I have always heard the power of GPU in computation tasks, however, with this programming assignment, I'm quite amazed by how GPU is so powerful that it makes serial computation by CPU looks like a machine from several decades ago. In addition, it's also surprisingly simple and straightforward on how I need to adjust the serial CPU code into a parallelize GPU code once I understand how CUDA runs a kernel program.

Even though that matrix normalization using GPU is quite impressive in terms of performance. I believe we as a programmer needs to be very aware of the fact that for a small set of problem (small size array), we should just use CPU and parallelize CPU code. This is due to the fact that there is an overhead cost where it takes time to copy data from host to device and device to host in order for a full computation cycle to be completed.

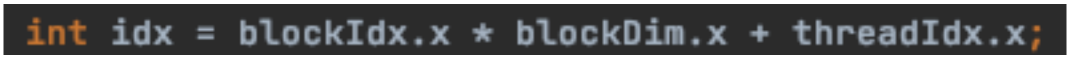
Developing a kernel program for GPU was an extraordinary experience for me. I initially thought it would be really hard. However, by understanding:

1. To use GPU to compute, we need to copy data from CPU memory to GPU memory
2. Prior to copying memory, I need to allocate it on the device
3. Prior to copying memory, I also need to convert 2D array to 1D array by flattening it
4. Launch kernel code with <<<block, thread>>>
5. Before GPU can carry out any compute task, I also added an if clause where its purpose is to prevent faulty computation tasks where GPU processes an index that is beyond the scope of the vector A



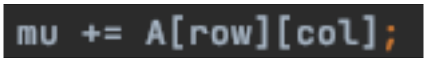
```
// If clause here is to prevent faulty computation tasks
//  where gpu processes an index that is beyond the scope
//  of the vector A
if (idx < arraySize) {
```

6. At this point, I imagine that the most outer loop of normalization function in CPU serial can be parallelize by removing the most outer loop. Then the rest of the code is pretty much the same
7. Since data in GPU now is a flattened data of a 2D array, I can easily map it so that it mimics a 2D array in computation using



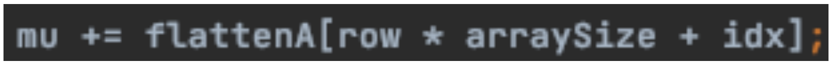
```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

8. To easier visualize my thoughts, I will put a piece of serial code and GPU kernel code side by side for comparison:
   Serial:



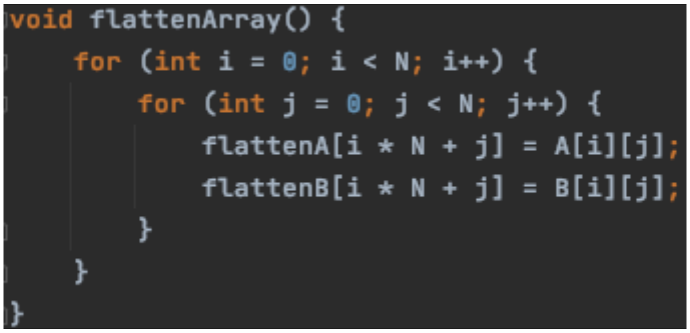
```
mu += A[row][col];
```

GPU:



```
mu += flattenA[row * arraySize + idx];
```

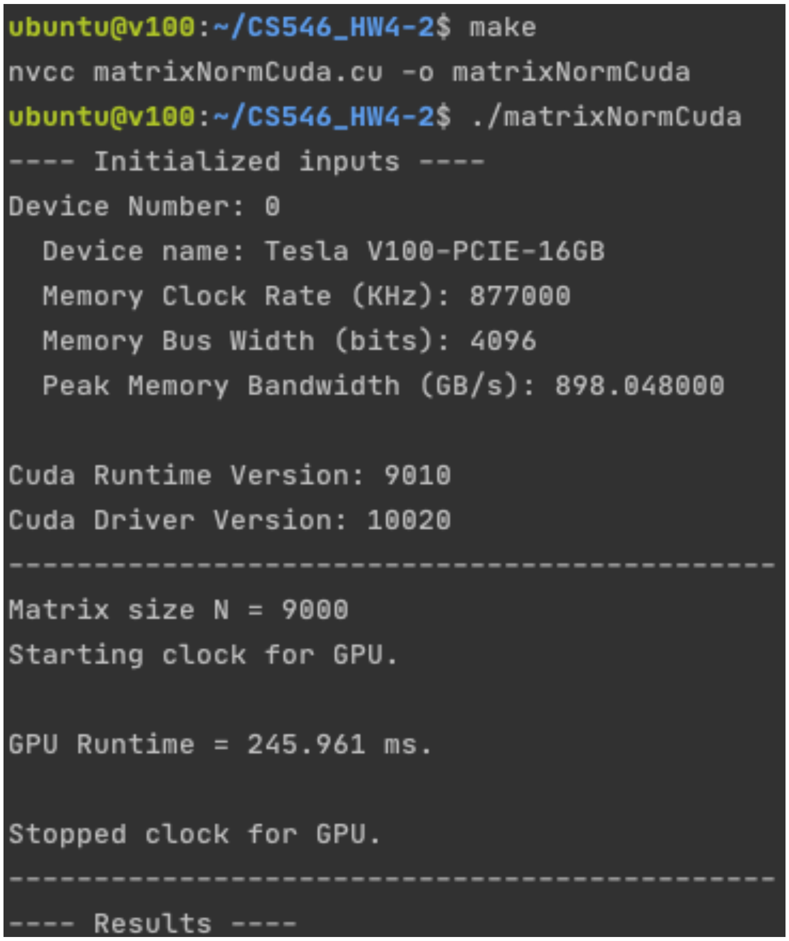
The way I access it is also the same way I convert a 2D to 1D:



```
void flattenArray() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            flattenA[i * N + j] = A[i][j];
            flattenB[i * N + j] = B[i][j];
        }
    }
}
```

9. In addition, since matrix normalization is a 3-part program i.e., calculating mean -> standard deviation -> normalized value. Thus, I also added a "barrier" or __syncthreads() to be exact to prevent data race problem
10. Once the computing tasks are finished, it is being copied from the GPU back to the host machine. Once result is being delivered back to the host machine, the result copied from GPU is in a 1D format (1D but mapped in a way that it mimics 2D array). The way I performed correctness validation is simply by lopping through this returned 1D array (B array) then compare it side by side with the serial code

Most recent run:



```
ubuntu@v100:~/CS546_HW4-2$ make
nvcc matrixNormCuda.cu -o matrixNormCuda
ubuntu@v100:~/CS546_HW4-2$ ./matrixNormCuda
---- Initialized inputs ----
Device Number: 0
  Device name: Tesla V100-PCIE-16GB
  Memory Clock Rate (KHz): 877000
  Memory Bus Width (bits): 4096
  Peak Memory Bandwidth (GB/s): 898.048000

Cuda Runtime Version: 9010
Cuda Driver Version: 10020
---------------------------------------------
Matrix size N = 9000
Starting clock for GPU.

GPU Runtime = 245.961 ms.

Stopped clock for GPU.
---------------------------------------------
---- Results ----
```

Lan Nguyen – A20430901
CS546 – Professor Zhiling Lan
Checklist:

Category	**Full Credit**
Documentation: README	Detailed documentation on both .zip on Blackboard and on GitHub
Design documents	Disccused all 4 requirements: machine used, how I tested for correctness, discussion of any issues I experienced and methods I tried and potential improvement
Performance Analysis	Detailed performance analysis and discussions on page 6-7 with plots of all runtimes with N size [6000, 7000, … 11000]
Code compilation	Code compiles and ran without any errors or warnings
Performance discussion	As discussed on pg. 6-7, code scale well and efficiency is optimal
Makefile	I also included a Makefile for extra credit