Assignment 4:

Syed Alle Mustafa

CS-451

A20519664

Description Of Environment:

The code was compiled and ran on CUDA friendly environment on ubuntu 20.04, Image named "CC-Ubuntu20.04-CUDA" on chameleon was used, it had Quadro rtx 6000 gpu, following are the details:

```
cc@mustafa-hw4:~/cuda-samples/Samples/0_Introduction/vectorAdd$ lspci -nnk | grep VGA -A1
03:00.0 VGA compatible controller [0300]: Matrox Electronics Systems Ltd. Integrated Matrox
G200eW3 Graphics Controller [102b:0536] (rev 04)
        DeviceName: Embedded Video
--
3b:00.0 VGA compatible controller [0300]: NVIDIA Corporation TU102GL [Quadro RTX 6000/8000]
[10de:1e30] (rev a1)
        Subsystem: Dell TU102GL [Quadro RTX 6000/8000] [1028:12ba]
```

And the version of nvcc used is 11.8, can be seen as follow:

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:33:58_PDT_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0
```

Algorithm:

The algorithm, takes the input of matrix dimension from the user, and then calculates the number of blocks on grid, where the number of threads is fixed to be 256, and the blocks per grid is determined in such a that each thread is responsible for a separate column, for this it is set to thread size i.e 256 added by number of columns and divided by 256, this way each column has its own column to process.

```
CUDA kernel launch with 40 blocks of 256 threads
Total columns: 10000, total threads: 10240
```

```
CUDA kernel launch with 47 blocks of 256 threads
Total columns: 12000, total threads: 12032
```

```
CUDA kernel launch with 55 blocks of 256 threads
Total columns: 14000, total threads: 14080
```

```
CUDA kernel launch with 63 blocks of 256 threads
Total columns: 16000, total threads: 16128
```

Then two matrices are allocated and one of them (input matrix) is initialized based on random numbers. A vector for storing column average and standard deviation is allocated space. All three of these data structures are shared to device. On kernel every thread calculates the sum, average and then standard deviation of a unique column and stores them on the vector (between the operations) on the index devoted for that column. Once the average and standard deviation is found out the matrix is normalized column wise and returned to the host. Then the allocated memory for each object is cleared.

Efficiency:

The code is algorithm is efficient since each thread is independent and if the column size is not large the blocks per grid is also very less. If we compare it to a serial program, following is the time taken by serial program:

```
--------------------------------
Matrix size N = 6000
Starting clock.

Computing Serially.
Runtime = 1574.83 ms.

Stopped clock.
```

If we do the same work size in CUDA, we get the following result:

```
[Matrix Normalization of 6000 rows and columns]

---------------
Matrix size N = 6000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 24 blocks of 256 threads
Total columns: 6000, total threads: 6144
Copy output data from the CUDA device to the host memory
Runtime: 0.021 ms.
Test PASSED
Done
```

Which is exponentially greater when done serially, even if we take a matrix of size 45000, The results will still be very great:

```
[Matrix Normalization of 45000 rows and columns]

---------------
Matrix size N = 45000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 176 blocks of 256 threads
Total columns: 45000, total threads: 45056
Copy output data from the CUDA device to the host memory
Runtime: 0.022 ms.
Test PASSED
Done
```

Correctness:

For correctness, matrix input was initialized with same numbers across the column, but not for the rows, to check whether the mean is the number chosen, which is the index of the column, and the standard deviation calculated should be zero. And this was proven true and can be seen using following results:

```
Hello from thread 2853, avg =  2853.000000
Hello from thread 2854, avg =  2854.000000
Hello from thread 2855, avg =  2855.000000
Hello from thread 2856, avg =  2856.000000
Hello from thread 2857, avg =  2857.000000
Hello from thread 2858, avg =  2858.000000
Hello from thread 2859, avg =  2859.000000
Hello from thread 2860, avg =  2860.000000
Hello from thread 2861, avg =  2861.000000
Hello from thread 2862, avg =  2862.000000
Hello from thread 2863, avg =  2863.000000
Hello from thread 2864, avg =  2864.000000
Hello from thread 2865, avg =  2865.000000
Hello from thread 2866, avg =  2866.000000
Hello from thread 2867, avg =  2867.000000
Hello from thread 2868, avg =  2868.000000
Hello from thread 2869, avg =  2869.000000
Hello from thread 2870, avg =  2870.000000
Hello from thread 2871, avg =  2871.000000
Hello from thread 2872, avg =  2872.000000
Hello from thread 2873, avg =  2873.000000
Hello from thread 2874, avg =  2874.000000
Hello from thread 2875, avg =  2875.000000
Hello from thread 2876, avg =  2876.000000
Hello from thread 2877, avg =  2877.000000
Hello from thread 2878, avg =  2878.000000
Hello from thread 2879, avg =  2879.000000
Test PASSED
Done
cc@mustafa-hw4:~/cuda-samples/Samples/0 Introc
```

```
thread 1453, calculated std as  0.000000
thread 1454, calculated std as  0.000000
thread 1455, calculated std as  0.000000
thread 1456, calculated std as  0.000000
thread 1457, calculated std as  0.000000
thread 1458, calculated std as  0.000000
thread 1459, calculated std as  0.000000
thread 1460, calculated std as  0.000000
thread 1461, calculated std as  0.000000
thread 1462, calculated std as  0.000000
thread 1463, calculated std as  0.000000
thread 1464, calculated std as  0.000000
thread 1465, calculated std as  0.000000
thread 1466, calculated std as  0.000000
thread 1467, calculated std as  0.000000
thread 1468, calculated std as  0.000000
thread 1469, calculated std as  0.000000
thread 1470, calculated std as  0.000000
thread 1471, calculated std as  0.000000
thread 2880, calculated std as  0.000000
thread 2881, calculated std as  0.000000
thread 2882, calculated std as  0.000000
thread 2883, calculated std as  0.000000
```

This experiment was done for large number of dimensions. To visualize this on small dimension of matrix the input and output can be observed:

```
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 1 blocks of 256 threads
Total columns: 10, total threads: 256
Copy output data from the CUDA device to the host memory
Runtime: 1.815 ms.
Test PASSED

----Input Vector----
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
0.000000      1.000000      2.000000      3.000000      4.000000      5.000000      6.000000      7.000000      8.000000      9.000000
----Output Vector----

0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     Done
```

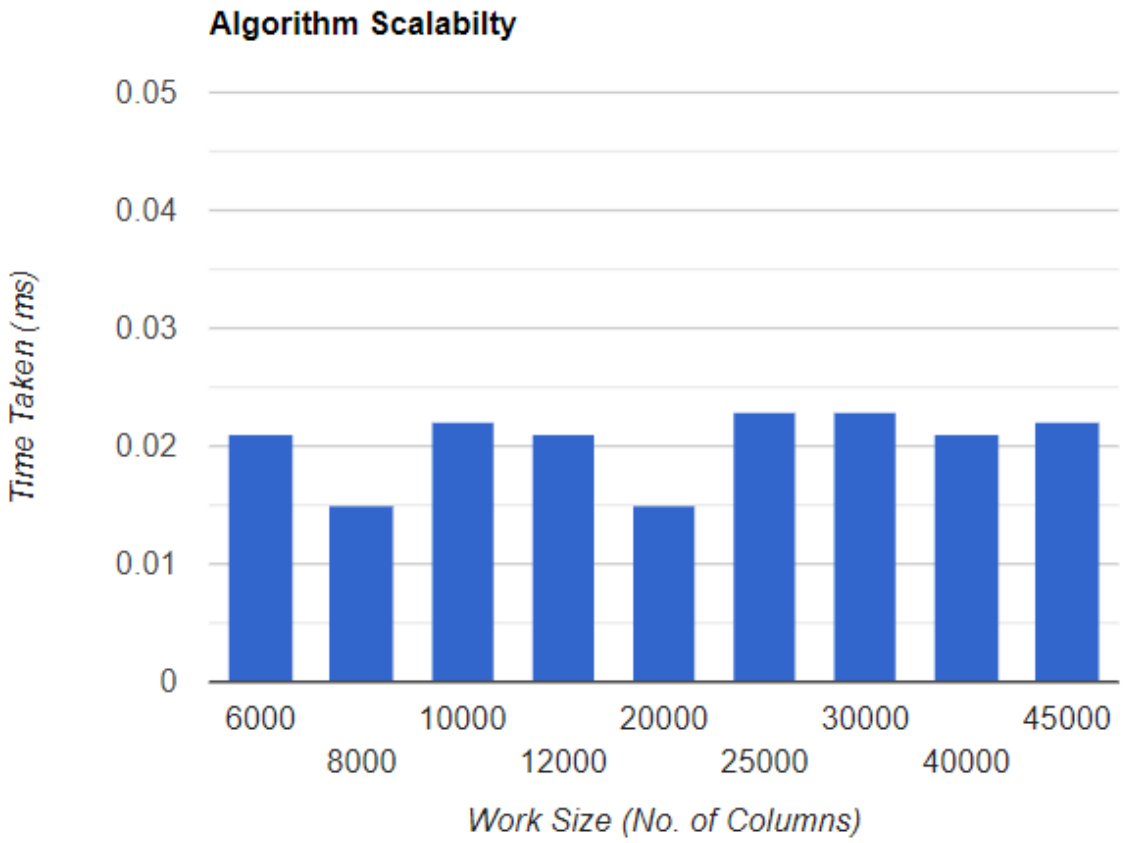
The output for random number is given in the present code, but it was much time consuming to prove it using random number, and since each thread is independent, and have separate memory index for calculations of mean and standard deviation, it makes much sense the algorithm runs perfectly. Following is the output for random values,

```
----Input Vector----
56696.250000      18959.404297      9684.888672      8976.958008      33318.402344      53615.531250      46545.597656      59207.566406      2638.521973      30905.832031
27518.597656      26926.482422      32005.269531      14833.659180      17386.101562      54394.312500      37803.039062      16462.990234      40356.160156      29998.578125
52149.386719      41234.238281      29302.548828      27643.515625      55828.203125      23223.724609      60311.410156      29563.619141      59684.339844      55503.816406
50206.539062      50844.589844      8927.221680      59891.429688      59821.546875      42245.625000      47970.960938      40831.144531      35917.191406      50609.480469
6200.978027      63435.789062      11999.963867      38206.246094      12733.449219      29386.066406      27064.562500      50536.488281      45849.054688      1884.721191
14999.067383      32462.443359      43118.960938      44301.617188      60105.960938      33411.164062      1989.341309      54881.371094      62974.781250      61673.683594
44849.187500      47645.324219      46982.273438      53776.410156      42000.750000      41267.820312      30486.033203      24435.710938      16562.968750      867.225769
9509.194336      22763.945312      64303.015625      21509.158203      60970.195312      11500.464844      50895.222656      22498.755859      62036.953125      31208.279297
24383.478516      11500.020508      63670.722656      1966.437988      55801.636719      58240.683594      35377.601562      57790.976562      47586.054688      32816.382812
53928.660156      26899.240234      14925.707031      35374.933594      15139.650391      56926.457031      11106.756836      45625.683594      15826.170898      27669.724609
----Output Vector----

1.450669     -0.868354     -1.074520     -1.143718     -0.240579     1.161132     0.913362     1.543480     -1.795640     0.086601
-0.043841     -0.352522     0.023701     -0.821266     -1.063047     1.209864     0.421822     -1.101366     0.083542     0.039907
1.217774     0.573842     -0.109280     -0.115995     0.921436     -0.740630     1.687328     -0.290757     1.046517     1.352584
1.118260     1.196069     -1.111799     1.659475     1.127583     0.449662     0.993501     0.406428     -0.137618     1.100687
-1.135752     2.011293     -0.960612     0.465556     -1.303229     -0.355022     -0.181936     1.006952     0.357211     -1.407027
-0.685104     0.005907     0.570523     0.801148     1.142265     -0.103153     -1.591761     1.275794     1.210455     1.670128
0.843850     0.988931     0.760608     1.322801     0.207627     0.388476     0.010432     -0.608049     -1.101891     -1.459394
-0.966301     -0.622027     1.612834     -0.453734     1.186880     -1.474211     1.157914     -0.727899     1.163730     0.102167
-0.204425     -1.351316     1.581723     -1.529695     0.920065     1.450551     0.285455     1.455828     0.443752     0.184931
1.308910     -0.354285     -0.816658     0.309673     -1.179015     1.368313     -1.079145     0.703093     -1.138600     -0.079952     Done
cc@mustafa-hw4:~/cuda-samples/Samples/0_Introduction/vectorAdd$
```

Scalability:

The algorithm is scalable perfectly, and as the work size increases the time taken either remains the same or minimizes very slightly, it can be seen by the following bar chart:



The following bar chart is made based on following results (not the time doesn't contain the copying data from and to the device, but the screenshots in the end contains them too):

Without Data Handling:

```
[Matrix Normalization of 6000 rows and columns]

---------------
Matrix size N = 6000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 24 blocks of 256 threads
Total columns: 6000, total threads: 6144
Copy output data from the CUDA device to the host memory
Runtime: 0.021 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 8000 rows and columns]

----------------
Matrix size N = 8000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 32 blocks of 256 threads
Total columns: 8000, total threads: 8192
Copy output data from the CUDA device to the host memory
Runtime: 0.015 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 10000 rows and columns]

----------------
Matrix size N = 10000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 40 blocks of 256 threads
Total columns: 10000, total threads: 10240
Copy output data from the CUDA device to the host memory
Runtime: 0.022 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 12000 rows and columns]

----------------
Matrix size N = 12000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 47 blocks of 256 threads
Total columns: 12000, total threads: 12032
Copy output data from the CUDA device to the host memory
Runtime: 0.021 ms.
Test PASSED
Done
```

```
cc@mustafa-hw4:~/cuda-samples/Samples/0_Introduction/vectorAdd$ ./practise 20000
[Matrix Normalization of 20000 rows and columns]

---------------
Matrix size N = 20000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 79 blocks of 256 threads
Total columns: 20000, total threads: 20224
Copy output data from the CUDA device to the host memory
Runtime: 0.015 ms.
Test PASSED
Done
```



```
[Matrix Normalization of 25000 rows and columns]

----------------
Matrix size N = 25000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 98 blocks of 256 threads
Total columns: 25000, total threads: 25088
Copy output data from the CUDA device to the host memory
Runtime: 0.023 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 30000 rows and columns]

----------------
Matrix size N = 30000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 118 blocks of 256 threads
Total columns: 30000, total threads: 30208
Copy output data from the CUDA device to the host memory
Runtime: 0.023 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 40000 rows and columns]

---------------
Matrix size N = 40000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 157 blocks of 256 threads
Total columns: 40000, total threads: 40192
Copy output data from the CUDA device to the host memory
Runtime: 0.021 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 45000 rows and columns]

---------------
Matrix size N = 45000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 176 blocks of 256 threads
Total columns: 45000, total threads: 45056
Copy output data from the CUDA device to the host memory
Runtime: 0.022 ms.
Test PASSED
Done
```

Time Taken with Data Handling:

```
cc@mustafa-hw4:~/cuda-samples/Samples/0_Introduction/vectorAdd$ ./practise 8000
[Matrix Normalization of 8000 rows and columns]

---------------
Matrix size N = 8000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 32 blocks of 256 threads
Total columns: 8000, total threads: 8192
Copy output data from the CUDA device to the host memory
Runtime: 230.897 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 10000 rows and columns]

---------------
Matrix size N = 10000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 40 blocks of 256 threads
Total columns: 10000, total threads: 10240
Copy output data from the CUDA device to the host memory
Runtime: 358.89 ms.
Test PASSED
Done
```



```
[Matrix Normalization of 12000 rows and columns]

---------------
Matrix size N = 12000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 47 blocks of 256 threads
Total columns: 12000, total threads: 12032
Copy output data from the CUDA device to the host memory
Runtime: 512.391 ms.
Test PASSED
Done
```

```
cc@mustafa-hw4:~/cuda-samples/Samples/0_Introduction/vectorAdd$ ./practise 14000
[Matrix Normalization of 14000 rows and columns]

---------------
Matrix size N = 14000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 55 blocks of 256 threads
Total columns: 14000, total threads: 14080
Copy output data from the CUDA device to the host memory
Runtime: 696.205 ms.
Test PASSED
Done
```

```
[Matrix Normalization of 16000 rows and columns]

----------------
Matrix size N = 16000
Starting Clock

Copy input data from the host memory to the CUDA device
CUDA kernel launch with 63 blocks of 256 threads
Total columns: 16000, total threads: 16128
Copy output data from the CUDA device to the host memory
Runtime: 906.276 ms.
Test PASSED
Done
```