Sawsane Ouchtal
A20397308
CS 451
Spring 2018

Homework 4
CUDA Programming

1. Program's Design:

The program is divided into the following steps:

- Initialization of the matrix.
- Sequential computation of each column's mean and standard deviation values, each
- Transposition of the matrix to have columns as rows of the new temporary matrix.
- Allocation of memory spaces in the GPU side, including:
    - float d_A[N][N]; // input matrix on the device side
    - float out_A[N][N]; //output matrix retrieved from the device
    - float *temp_d; //column on the device side
    - float *mean_d; //array for the means on the device side
    - float *SD_d; //array for the standart deviations on the device side

- Transfer of each column of the matrix to the GPU in a temp_d array, of the means and the standard deviations array in separate 1d arrays.
- Every array temp_d is passed to the kernel function named **func()** along with the mean, SD and the size of the matrix.
- Transfer of the computed row from the GPU to the CPU and add the rows up to the output matrix.

The SLURM execution file following lines:

```
#SBATCH -p gpu-shared
#SBATCH --gres=gpu:k80:1
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=6
```

According to XSEDE's documentation, users should always set --ntasks-per-node equal to 6 x [number of GPUs requested] when using K80 "gpu-shared" jobs.[1]

2. Performance Evaluation:

This section serves to compare the running time of the parallel code versus the sequential code. The sequential and the parallel code make use of the **gettimeofday** function.

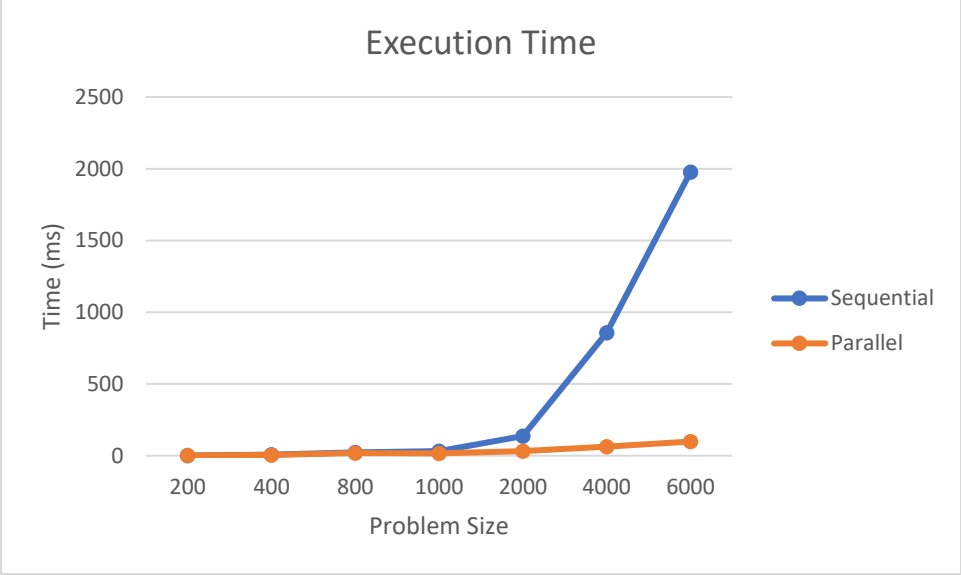
```
cudaDeviceSynchronize();
gettimeofday(&start, &tzdummy);

/******Kernel Function*********/

cudaDeviceSynchronize();
gettimeofday(&stop, &tzdummy);
```

According to CUDA's developer's guide[2], measuring the elapsed time of the code's execution, it is needed to synchronize the CPU thread with the GPU "by calling cudaDeviceSynchronize() immediately before starting and stopping the CPU timer".
The table below summarizes the execution times for both the sequential and parallel codes while varying the problem size, which is the matrix size:

Problem size	200	400	800	1000	2000	4000	6000
Sequential	1.161	6.866	23.179	32.085	137.443	856.973	1977.04
Parallel	3.189	6.43	18.22	15.667	31.614	63.736	98.62



Graph of the data from the previous table

The graph shows that the GPU performance is not any better from that of the CPU with small problem size, in this case smaller 1000*1000 matrix. But as the problem's complication increases, the execution time of the CPU is much longer than that of the GPU. Another observation is that the execution time of the GPU is not exponential as it is using the CPU, that is the GPU's performance does not seem to drastically slow down when the computations get more complex.

Note: The correctness of the program was tested against the provided sequential code, the matrix size was fixed to 20 and the matrix values were set static instead of randomly generated ones. Finally, both outputs were compared to be found similar.

[1] https://portal.xsede.org/sdsc-comet
[2]http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#optimizing-cuda-applications