CS546 “Parallel and Distributed Processing”

Homework 3
First name: Guillermo
Last Name: de la Puente
CWID: A20314328
Due date: 3/10/2014

Also included:
matrixNormCuda.cu

The provided file matrixNormCuda.cu contains the code necessary to execute the matrix column normalization algorithm using CUDA parallel capabilities.

Given the input array A and the output array B, the algorithm follows these steps:

1) Copy array A to CUDA memory and prepare an array to hold partial sums of columns.
2) Execute the partial sum algorithm from the class slides on the data from A
3) Transfer the partial sums array to the host, and sequentially reduce the data and obtain a single value sum of all elements in every column. Divide every element by the amount of elements to obtain column mean.
4) Transfer the means array to CUDA.
5) Apply the partial sums algorithm again but applying the transformation (A[i][j] – mean)^2 to the input data
6) Transfer the partial sums to the host
7) Sequentially add the partial results to obtain the total for each column. Divide by amount of elements an calculate square root. These are the standard deviations.
8) Transfer the standard deviation (sigma) array to CUDA.
9) Apply the transformation B[row][col] = (A[row][col] – mean) / standard_deviation on every element of A.
10) Transfer the array B to the host

Every function in CUDA has a different kernel method. The code in matrixNormCuda.cu is extensively commented.

A problem was found when trying to operate with N > 2500 in CUDA because a memory error would be retrieved. Modifying the block size had impact on this, the lower the block size, the lower this limit. However, the block size can't be increased to 64 because the partial sums algorithm requires BLOCK_SIZE*BLOCK_SIZE*2 of floats in shared memory. Because the shared memory is small, a compiler error is received if its size is surpassed.

That's why the following time measures don't show any value for N > 2500 in CUDA.

N	Sequential	CUDA
50	0.04 ms	565 ms
100	0.17 ms	565 ms
200	0.59 ms	565 ms
400	2.2 ms	568 ms
800	9 ms	570 ms
1600	38 ms	585 ms
2400	86 ms	618 ms
4800	382 ms	Memory error
8000	1338 ms	Memory error

It can be seen that the results obtained so far **aren't better** than using the sequential algorithm. The measures have been made in a work node from the Jarvis cluster.

Although it can be seen that the time that the algorithm requires grows very little with CUDA. Most of the time right now is being spent in transferring the arrays from host to device and from device to host.

If a way was found to use CUDA in a way that the memory used was constant and independent from N, the measures show that the algorithm would be very efficient.