Fast GPU implementation of Spatial Distance Histogram Computation

Abstract:

Computing spatial distance histogram (SDH) in one single thread on CPU involves numerous repeated calculations. The complexity of the algorithm is O (n^2) which is very slow to process large data sets. By using GPU to perform these calculations in parallel, we could get this work done much faster. In this project, we try to modify our first GPU version of SDH. We enable the shared output memory, control the divergence and modify the input array. The final version of GPU program running on NVidia GTX 760 is about 30 times faster than on the Intel® Core™ i7-3770K CPU based on my own machine's data.

Introduction:

One of the most important purposes of this project was to reduce the global memory access of the previous version. There are two ways to perform this: tiling input data and using the private histogram. It is very easy to generate a private histogram. This modification could get a boost of 2-5 times compare to the original version. Another most important boost is to do tiling of the input array, which is difficult to control the last block boundary if add divergence control. I completed this part but did not finish the divergence control. The other boost methods in this version are modifying input array.

Methodology:

The development platform I used was my desktop which runs Ubuntu 12.04LTS, with CUDA 6.5.14. The CPU is a quad-core Intel® Core™ i7-3770K CPU with a 3.50 GHz clock speed. The GPU is NVIDIA GeForce GTX 760 with 2GB global memory and 1152 CUDA Cores. CUDA capability version number is 3.0. GPU max clock rate is 1032Mhz.

Tiling of input array:

The basic idea to reduce global memory access when reading input data (Figure 1) is to place the atom arrays in shared memory. We build two sets of shared input arrays "xyz1 xyz2" that are the equal size with the block size of each CUDA block. So we have n = TotalN/block size number of blocks to proceed. While processing the data, each CUDA block load two pieces of data into xyz1 and xyz2. We define xyz1 as the anchor data, and we do not change it during processing. First we calculate the data within each xyz1; after that we proceed Intra-group pairing in a loop to bring new data to xyz2 to pair with xyz1. After all the data pairing had completed, we got the final data for each thread.

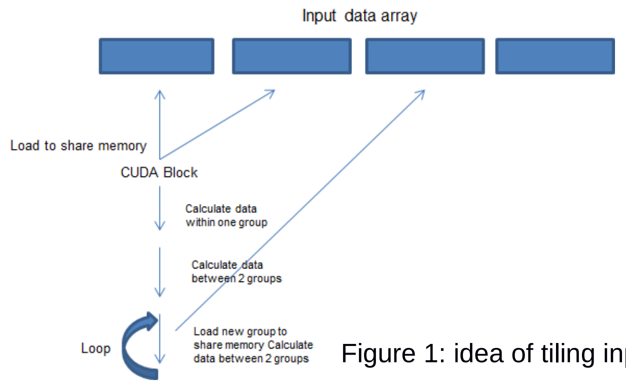
Input data array
Load to share memory
CUDA Block
Calculate data within one group
Calculate data between 2 groups
Loop
Load new group to share memory Calculate data between 2 groups

Figure 1: idea of tiling input

Shared memory of output array:
To reduce the atomicAdd() operation in global memory, we create a private histogram for each block. Each thread will load data into the shared memory first. After all the threads complete their calculation, we combine all the private copy to the global memory.

Divergence:
The original method uses a triangle workload for all the threads, which the first thread has the largest workload, and the last one does not have to do anything. So by using the cycles counting and % increment, we could balance the workload for each thread. In this part, I found there is no performance after I did it. The reason may be % calculation takes a long time. I concluded those codes in my comments.

Results:

For 10000 data points and 500 as the distance difference for each bucket, we choose 256 as the block size. Due to the shared memory size limitation I found too big block size will affect the performance of the program

Here are the data we got:

CPU version time: 818ms
GPU V1 plain version: 94ms
GPU V2with shared output histogram 54ms
GPU V3with divergence control and shared output histogram 54ms
GPU V4with modify input array, divergence control and shared output histogram 43ms
GPU V5with tiling input data, modify the input array, and shared output histogram 30ms.
(Without divergence control due to divergence control in test do not increase performance)

From the figure2, we could see the biggest boosts come from the shared memory of output array and tiling of input data. This data proved that more global memory access will significant reduce the program performance.

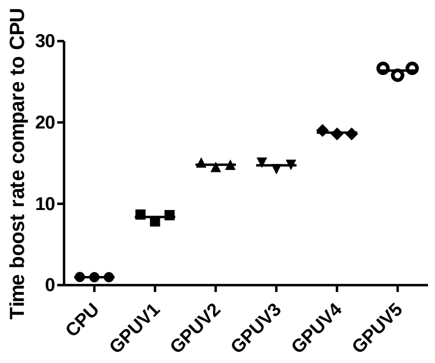

Figure2: different effects of GPU acceleration of SDH

Discussion:
In this project, we learned that the most important idea to boost GPU programming is to reducing the global memory access. By tiling of the input array and create a private copy of output array, I boost the GPU program about three times faster compare to the project1. Although I reported that my GPU program is about 30 times faster than the CPU version, this data is actually based on a good CPU (quad-core Intel® Core™ i7-3770K CPU with a 3.50 GHz clock speed) verse an entry level GeForce GTX 760. Moreover, I included not only the kernel running time but also data shipping process time for input and output of the device. So this number may vary a lot due to CPU and GPU recourses. The remaining problem is divergence does not improve the performance. Further efforts need to be done to demonstrate this problem.