GPU Project 2: A Better Version of SDH Computing Program

Instructor: Dr. Yi-Cheng Tu
Course: COP4530/6527 Programming on Massively Parallel Systems
Submission Deadline: October 27, 2021 (Wednesday) at 11:59 PM

1 Overview

In this project you will measure the performance of GPU memory system, perform different experiments, and write a report on your findings. The problem in hand is: computing the spatial distance histogram of a set of points. This is the same problem that you worked on in project 1.

The main objective of this project is to write very efficient CUDA code, unlike in Project 1 where performance was not a grading criterion. You have the freedom to choose any techniques you learned from class to improve your program, these include (but are not limited to): using shared memory to hold input data, manipulating input data layout for coalesced memory access, managing thread workload to reduce code divergence, atomic operations, private copies of output, and shuffle instructions. You can use a combination of different techniques, and the code will be graded based on your rank in a class-wise contest, in which we will use a set of different datasets to test your code. Therefore, your goal is to try all you can think of to optimize your program. That said, we will make your job easier by introducing the following paper published by our group a few years back.

Napath Pitaksirianan, Zhila Nouri, and Yi-Cheng Tu. "Efficient 2-Body Statistics Computation on GPUs: Parallelization Beyond." Proceedings of 45th International Conference on Parallel Processing, pp. 380-385., August 2016.

The paper describes a series of techniques for achieving high performance in dealing with a group of problems that share similar computing patterns as the SDH problem. We will upload a version of the above paper in Canvas.

2 Tasks to Perform

Write a CUDA program to implement the same functionalities as required in Project 1, perform different experiments, and write a short report about your project. The CUDA kernel function results and running time of the kernel(s) should be displayed as output. Thus, your main task it to write an efficient CUDA kernel to compute the SDH. In addition, your program should also include the following features.

Input/Output of Your Program: You have to modify the program to take a different number of command line arguments from what you did in Project 1. Your program should take care of bad or missing inputs and throw appropriate errors in a reasonable way. In particular, here is what we expect in launching your program:

```
./proj2 {#of_samples} {bucket_width} {block_size}
```

where `proj2` is assumed to be the executable after compiling your project. The first two arguments are the same as in Project 1, while the last one is the number of threads within each block your CUDA kernel should be launched.

The output of your program should print out the SDH you computed as in Project 1. Following the SDH, you should add a line to report the performance of your kernel, it should look like the following sample.

```
******** Total Running Time of Kernel = 2.0043 sec *******
```

Please read Section 3 for details of measuring kernel running time.

Project Report Write a report to explain clearly how you implemented the GPU kernels, with a focus on what techniques you used to optimize performance.

3 Measuring Running Time

The running time of the CPU implementation can be measured using different time functions available in the C programming libraries. One such function useful in measuring time is `gettimeofday`. Another is the `rdtsc` instruction supported on Pentium CPUs. You can also use the `clock()` function to record the time. However, these functions cannot be used to measure the running time of GPU kernels. There are special event functions that are used to record the running time of kernels. Following is an example to record running time of a kernel.

```
1:  cudaEvent_t start, stop;
2:  cudaEventCreate(&start);
3:  cudaEventCreate(&stop);
4:  cudaEventRecord( start, 0 );

5:  /* Your Kernel call goes here */

6:  cudaEventRecord( stop, 0 );
7:  cudaEventSynchronize( stop );
8:  float elapsedTime;
9:  cudaEventElapsedTime( &elapsedTime, start, stop );
10: printf( "Time to generate: %0.5f ms\n", elapsedTime );
11: cudaEventDestroy( start );
12: cudaEventDestroy( stop );
```

An *event* in CUDA is essentially a GPU time stamp that is recorded at a user specified point in time. The API is relatively easy to use, since taking a time stamp consists of just two steps: creating an event and subsequently recording an event. For example, at the beginning of some sequence of code, we instruct the CUDA runtime to make a record of the current time. We do so by creating and then recording the event (lines $2-4$). The exact nature of second argument in line 4 is unimportant for our purposes right now (use 0 always).

To time a block of code, we will want to create both a start event and a stop event. We will have the CUDA runtime record when we tell it to do some work on the GPU and then tell it to record when we've stopped.

Unfortunately, there is still a problem with timing GPU code in this way. The fix will require only one line of code but will require some explanation. The trickiest part of using events arises as a consequence of the fact that some of the calls we make in CUDA C are actually asynchronous. For example, when we launch the kernel in line 5, the GPU begins executing our code, but the CPU continues executing the next line of our program before the GPU finishes. This is excellent from a performance standpoint because it means we can be computing something on the GPU and CPU at the same time, but conceptually it makes timing tricky.

You should imagine calls to `cudaEventRecord()` as an instruction to record the current time being placed into the GPU's pending queue of work. As a result, our event won't actually be recorded until the GPU finishes everything prior to the call to `cudaEventRecord()`. In terms of having our stop event measure the correct time, this is precisely what we want. But we cannot safely read the value of the stop event until the GPU has completed its prior work and recorded the stop event. Fortunately, we have a way to instruct the CPU to synchronize on an event, the event API function `cudaEventSynchronize()`.

Now, we have instructed the runtime to block further instruction (line 7) until the GPU has reached the stop event. When the call to `cudaEventSynchronize()` returns, we know that all GPU work before the stop event has completed, so it is safe to read the time stamp recorded in stop. It is worth noting that because CUDA events get implemented directly on the GPU, they are unsuitable for timing mixtures of device and host code. That is, you will get unreliable results if you attempt to use CUDA events to time more than kernel executions and memory copies involving the device.

The function `cudaEventElapsedTime()` is a utility that computes the elapsed time between two previously recorded events. The time in milliseconds elapsed between the two events is returned in the first argument, the address of a floating-point variable.

The call to `cudaEventDestroy()` needs to be made when we're finished using an event created with `cudaEventCreate()`. This is identical to calling `free()` on memory previously allocated with `malloc()`, so we needn't stress how important it is to match every `cudaEventCreate()` with a `cudaEventDestroy()`.

The content about measuring time on GPUs explained in this section is taken from the following reference book:
Jason Sanders and Edward Kandrot. "*CUDA by Example: An Introduction to General-Purpose GPU Programming*". Addison-Wesley, 2011.

4 Environment

All projects will be tested on one machine of the GAIVI cluster. Please follow the instructions posted earlier to connect to the GAIVI machiens. If you prefer to work on your own computer, make sure your project can be executed on the GAIVI computers.

5 Instructions to Submit Project

You should submit one .cu file containing your implementation of the SDH algorithm and a separate .pdf file containing the report. Put both files in a folder, zip the folder and name the zipped file `proj2-xxx.zip` (or `proj2-xxx.tar` if you prefer using TAR for compression), where `xxx` is your USF NetID. Submit the zipped file only via assignment link in Canvas. E-mail or any other form of submission will not be graded. Once you submit your file to Canvas, try to download the submitted file and open it in your machine to make sure no data transmission problems occurred. For that, we suggest you finish the project and submit the file way before the deadline.

6 Rules for Grading

Following are some of the rules that you should keep in mind while submitting the project. Functionality of project will be graded by a set of test cases we run against your code.

In this project, you are supposed to get correct results, i.e., the SDH computed by your CUDA code should be the same as that computed by the CPU code.

- All programs that fail to compile will get zero points. We expect your submission can be compiled by running a simple line of command as in Project 1.
- If, after the submission deadline, any changes are to be made to make the main code work, they should be done within 3 lines of code. This will incur a 30% penalty.
- Program should run for different numbers of CUDA blocks and threads, following the last two arguments to your program. Use of any tricks to make it run on one thread or only on CPU will result in zero points.
- If you submit the same code as you did for Project 1, you will get a (very) low grade in this project.
- We will check the code related to performance measurement carefully. Any tricks played to report shorter running time than the actual one will be treated as cheating and incur a heavy penalty.