CpSc 418
70 points.

Homework 3

Due: Mar. 23, 2016, 11:59pm

Please submit your solution using the handin program. Submit the your solution as
cs418 hw3
Your submission should consist of the following three files:

`hw3.cu` – CUDA source (ASCII text).

`hw3.txt` or `hw3.pdf` – plain, ASCII text or PDF.

The first file, `hw3.cu`, will be your solution to the programming part of the assignment.
You may use any functions that you like from the C, C++, and CUDA libraries.
The second file, `hw3.txt` or `hw3.pdf` should give your solutions to the written parts of the assignment.
No other file formats will be accepted.
Early-bird bonus of 5% for solutions submitted by March 21, 11:59pm.
Some code templates will be posted at
http://www.ugrad.cs.ubc.ca/~cs418/2015-2/hw/3/code.html
This assignment is a preparation for homework 4 where you will speed-up the percolation simulation. I've worked out a solution to that. Along the way, I had to do quite a few experiments to identify the performance bottlenecks in the code. In this assignment, you will quantify the performance limits on the GPUs in the `linXX` boxes. This will form the basis for writing fast percolation simulation code for homework 4. I'm splitting this into two assignments so I can post a solution to homework 3 before homework 4 is due.

1. **GFlops (20 points)**: Measure the floating-point performance of a GPU.
   Consider the logistic map:
   $$x_{i+1} \;=\; \alpha x_i(1 - x_i)$$
   with $0 < x < 1$ and $1 < \alpha < 4$. The fun cases occur with $3.6 < \alpha < 4$; see
   https://en.wikipedia.org/wiki/Logistic_map.
   Each calculation of the recurrence requires three floating point operations: two mulitplications and one subtraction. Let's say that we start with a vector of n values for x_0 and want to compute x_m for each of these values. In other words we compute m steps of the recurrence for each initial value of x. This takes $3 * m * n$ floating point operations.

   (a) (10 points): write a CUDA implementation of the logistic map as described above. Your code should provide a function,

   ```
   void logistic(float *x, int n, int m);
   ```

   where `x` is an array of `n` initial values, and `m` is the number of iterations to compute.
   Your goal is to make a *fast* implementation. Determine how many threads, and blocks you need to use to make the code run quickly. You can pick values of `n` and `m` to get the highest number of floating point operations per second for a kernel that takes at most 0.1 seconds to execute.

   (b) (5 points): How many Gflops (billions of floating point operations per second) does your kernel achieve? Provide data from executing your program that shows how the number of blocks, number of threads per block, and any other critical design decisions affect the performance of your kernel.

   (c) (5 points): Given the observations from part (b), describe how you took these performance trade-offs into account when writing your implementation of the `logistic` function. What observations can you make about writing efficient CUDA programs?
   Your answer doesn't need to be long. Anywhere from five to twenty sentences should be fine. Grading will be based on how your explanations make the design of your code clear, not on length.

2. **Memory bandwidth (20 points)**: Measure the memory-bandwidth of a GPU.
Consider computing the norm (i.e. length) of a vector x

$$\mathrm{norm}(x) \quad = \quad \sqrt{\sum_{i=0}^{n-1} x_i^2}$$

where n is the number of elements of x.

   (a) (10 points): write a CUDA implementation of the norm function as described above. Your code should provide a function,

   ```
   float norm(float *x, int n);
   ```

   where `x` is an array of `n` elements.

   Your goal is to make a *fast* implementation. Determine how many threads, and blocks you need to use to make the code run quickly. You can pick a value value for `n` to get the highest number gigabytes per second for your kernel. 0.1 seconds to execute.

   (b) (5 points): How many gigabytes per second of main memory access does your kernel achieve? Provide data from executing your program that shows how the number of blocks, number of threads per block, memory layout, and any other critical design decisions affect the performance of your kernel.

   (c) (5 points): Given the observations from part (b), describe how you took these performance trade-offs into account when writing your implementation of the `norm` function. What observations can you make about writing efficient CUDA programs?

   As with question 1, your answer doesn't need to be long.

3. **Random number generation(30 points)**
The percolation simulation in perc.cu makes intensive use of the random number generators from the cuRAND package. The host API of `cuRAND` provides functions that return large arrays of random numbers to the CPU host. In this problem; you will write similar functions to produce arrays of random numbers in the *device* memory.

Why do this? Because the percolation simulation needs a large number of random numbers. On the other hand, the CUDA grid that is most efficient from computing random numbers is not the same shape as the CUDA grid that's efficient for simulating percolation. In homework 4, you will combine a kernel for computing random numbers with a kernel for simulating percolation. This problem gets you started on the random number part.

Let's say we want an array of `n` random, `unsigned int` values. You can assume that `n` is at least one million. Our strategy is to determine the number of blocks and threads per block that maximize the speed of random number generation. This means that the code will use lots of random number generators. Each random number generator needs to be initialized. We'll write the initialization in a separate kernel than the random number generation kernel.

Why? Because this lets us launch the random number generation kernel multiple times to get enough random numbers without using huge amounts of global memory. On the other hand, we don't want to incur the overhead of initializing the random number generator each time. In fact, initializing the generator each time would make our code more complicated, because we would need to create new seeds for each launch so that we wouldn't keep getting the same sequence each time.

   (a) (2 points): Use `setup_kernel` from the cuRAND documentation (section 3.6) to initialize the random number generators.

   (b) (18 points): Write a kernel, `rndm`, to fill an array of `nblocks * threads_per_block * m` elements with random numbers generated by the function `curand(curandState)`.

   Your goal is to make a *fast* implementation. Determine how many threads, and blocks you need to use to make the code run quickly. You can pick a value value for `n` to get the highest number of pseudo-random numbers per second for a kernel that takes at most 0.1 seconds to execute.

   (c) (5 points): How many pseudo-random numbers per second does your kernel generate? Provide data from executing your program that shows how the number of blocks, number of threads per block, memory layout, and any other critical design decisions affect the performance of your kernel.

(d) (5 points): Given the observations from part (b), describe how you took these performance trade-offs into account when writing your implementation of the `rndm` kernel. What observations can you make about writing efficient CUDA programs?

As with question 1, your answer doesn't need to be long.

Note: you are welcome to develop your code on any machine that you like. However, please report run-times when running on `linXX.ugrad.cs.ubc.ca` where `XX` is any number from `01` to `25`. Note that these machines run CUDA level 2.1 – that matches anything we're covering in class, but you might find "advanced" CUDA features in other materials that are only available in more recent versions of CUDA.