CpSc 418 **Homework 4** Due: Apr. 15, 2016, 11:59pm
50 points + 10 points extra credit

Please submit your solution using the handin program. Submit the your solution as
cs418 hw4
Your submission should consist of the following two (or three) files:

`hw4.cu` – CUDA source (ASCII text) for Q1.

`hw4.txt` or `hw4.pdf` – plain, ASCII text or PDF.

`hw4x.cu` – CUDA source (ASCII text) for Q2 if you decide to attempt the extra credit problem.

The first file, `hw4.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, `hw4.txt` or `hw4.pdf` should give your solutions to the written parts of the assignment.
No other file formats will be accepted.
Some code templates are posted at
http://www.ugrad.cs.ubc.ca/~cs418/2015-2/hw/4/code.html
This assignment takes the results from homework 3 to produce a fast percolation simulator. There are two questions. Q1 is the main question – you combine the random-number generation kernel from HW3 with an optimized percolation simulation to write a faster percolation simulator. I'll provide lots of suggestions for that code.

Grading: I'll take your score on this (as a percentage), and use it to replace the lowest score that you got on any other assignment (again as a percentage) if that raises your grade. With the extra credit, it's possible to get 120% on this homework. If you don't do this assignment, your homework grade will be calculated from the other three homework assignments.

1. **Faster Percolation (50 points)**: The code that I used for percolation demos in class is available at:
   http://www.ugrad.cs.ubc.ca/ cs418/2015-2/hw/4/src/perc0.cu
   It achieves roughly 244 million vertex updates per second (using the GPU). The limit was the performance of the random number generator. The basic problem is that the percolation threads need to communicate, which means they all need to be in the same block. On the other hand, the random-number generators can't reach their peak performance running in a kernel that only has one block of threads.

   My solution puts the random number generation in one kernel, and the percolation computation in a separate kernel. The random number generator writes its results to global memory. The percolation kernel uses these precomputed random numbers to perform its computation. By separating the computation into two kernels, the number of blocks and threads can be separately optimized for each task. This comes at a cost of launching an extra kernel and communicating between the two kernels using shared memory.

   I have posted most of my optmized code here:
   http://www.ugrad.cs.ubc.ca/ cs418/2015-2/hw/4/src/perc.cu
   It achieves about 4.6 billion vertex updates per second. To make this into an interesting homework problem, I've removed about seven lines of code. These are marked with `TODO:` comments in the source code. This means that I've written all the stuff for launching kernels and reporting results. Your task in this assignment is to write the GPU code that makes the percolation simulation run fast.

   The version that I posted, perc.cu, compiles with a few warnings about unused variables (you can think of those warnings as "hints"). It runs *really* fast – because the code isn't doing anything. In the remainder of this problem statement, I'll describe the optimizations that I did. I won't give the details. Feel free to use these and/or improve on them.

   First, I separated random number generation into a separate kernel from the percolation simulation as described above. Note that the largest simulation my code can run is limited by the amount of memory on the GPU for stroring these random numbers.

Second, I observed that the percolation computation can be optimized using bitwise-logical operations. The basic idea is that instead of updating a single vertex with the statement:

```
next = (v[myId] || v[id1]) && (curand_uniform(myRandState) <= p);
```

my code updates 32 vertices at a time using bitwise logical operations with a statement like:

```
x = (x | (x >> 1)) & *r;
```

where `x` is a bit-vector of vertex states, and `r` is a pointer into the array of pre-computed random bits. To do this, each thread needs to know the state of its left-neighbour. One approach is to have each thread store its value in shared-memory, and read its neighbour's value from shared memory.

Having written and tested the code, I replaced it with `TODO:` comments in the `perc` kernel. You need to write your own implementation. In `hw4.txt` (or `hww.pdf`) describe any trade-offs that you encountered and how they affected the design of your code.

My code uses 64-bit values (i.e. `uint64_t`) in the kernel for the percolation simulation. With this, each thread only accesses shared memory (one read and one write) every 32 updates. I haven't done the speed measurements to see if this matters much – I would expect that the global memory accesses to read `*r` takes more time.

Third, I wrote my code to use exactly one warp. One warp of 32 threads with 32 vertices (i.e. bits) per thread simulates a percolation network with a width of 1024. This seems big enough to get interesting results. It also makes my code simpler, and (presumably faster), because it doesn't need to perform syncthreads operations. Of course, if the simulation kernel could use more threads, it would achieve a higher throughput.

Fourth, I wrote the random-number generation kernel to generate random bits. In particular, it takes a parameter, `p`, that is the probability that a vertex is marked if at least one of its parent vertices is marked. The random-number generation kernel produces an array of `uint` values. Each `uint` is a 32 bit quantity, where each bit of the `uint` is 1 with probability `p` and 0 with probability $1 - p$. These bits are independent random variables (to within the limits of the quality of the `curand` library, which appears to be pretty good).

By generating the bits this way, my code calls `curand(randstate_t)` once for each bit, in other words, 32 times for each `uint` that it writes to the global memory. This lets my code run longer simulations before running out of space. It also reduces the number of global memory accesses performed by the kernels.

Having written and tested the code for random-bit generation, I replace it with `TODO:` comments in the `rndb` kernel. You need to write your own implementation. In `hw4.txt` (or `hww.pdf`) describe any trade-offs that you encountered and how they affected the design of your code.

Fifth, my timing measurements indicate that the new version is about 20 times faster than the original.

2. **Fastest Percolation (10 points, extra credit)**: For Q1, I described a particular set of design choices for optimizing the percolation simulation. For this problem, see how fast you can make a percolation simulation run. The figure of merit is vertex-updates-per-second. You can change the width of the graph, but it must be at least 1024, You can change the height of the graph, but it must be at least the $W^{1.5}$ where W is the width. You can change the interface between the random-bit generation kernel and the percolation simulation kernel. You can use a different decomposition of the problem. You do need to compute correct results – make sure that the critical probability makes sense.

   Your solution should consist of two parts: `hw4x.cu`, you source code; and a description of your optimizations, their impact on performance, and your final result. This description should be in `hw4.txt` or `hw4.pdf`. To make this fun, feel free to send me ideas that you've implemented with a description of how much performance boost you got from them – private posts on piazza are fine. If I think it's an idea worth building on by others; I'll post a description to everyone. You'll get the credit. That way, we can collectively see how many performance-boosting techniques we can uncover.

   Have fun – the number of XC points isn't enough to bother unless you're enjoying it!