CIS 372 INTRODUCTION TO PARALLEL PROGRAMMING Spring 2018
Lab 6 CUDA Individual or Pair Lab Assignment
Due Date: Tuesday, April 17, 2018 at 11:55PM

Objectives

- Apply techniques for decomposition and multi-threading to ensure accurate behavior
- Apply concurrency techniques to shared data structures

The Problem: N-Body simulation

Molecular Dynamics are an important general simulation environment for problems in physics, biology, and chemistry. MD problems involve simulating the interactions of molecules with each other. This problem belongs to a general class of N-Body simulation problems that involve dynamic particles usually under the influence of physical forces. In this lab, we will simulate a very simplified version of a 2-dimensional particle system.

The Setup:

- Download provided code from your github classroom account

- We will have N particles that each have:
    - the exact same mass, density, spherical shape and radius
    - velocity as a tuple in 2-dimensions (magnitude is a constant)
    - position as a tuple in 2-dimensions
- The basic model computes elastic collisions using Newtonian equations (http://en.wikipedia.org/wiki/Elastic_collision)
- Energy might not be conserved due to rounding.
- The world size is NxN, each sphere has diameter of 1 and initial velocity magnitude of 1
- Collisions are detected with the bounds of the world and with other particles
- Your parallel solution results must match the original sequential implementation **(you may not make improvements that change the accuracy of the result)**

The basic implementation:

The basic model intializes a random NxN world with N particles (one per row to start) and then uses a step-based simulation similar to Conway's game of life. On each iteration, the simulation does 3 things:

- update_particle_velocities_boundary_collisions
    - if either the x or y component of the position is over a boundary, the corresponding x or y component of the velocity is sign flipped. Note that to avoid race conditions the velocities array is twice N so that we can output to the second half of the array.
- update_particle_velocities_nbody_collisions
    - for each particle
        - check to see if it collided with any other particle, and if so use elastic collision formula in 2D to transfer velocity from the other unit to the

particle. Collision is based on distance between two particles being < 1. Again, to avoid race conditions we only update the particle (not its collision particle) and we apply the update to the first half of the array
    - if no collision happened, we copy the value from the second half of the array to the first so that we start each iteration reading from the first half of the array
- update_particle_position
    - the updated velocities are applied to the particle positions using the step time

The Task:

1) Convert the sequential code into a naive parallel version. The decomposition here is straight forward -- each thread should compute a single particle. Notes:
    - You will need to tag the three kernels as __global__ so that they can be called from the host (one for each of the above methods).
    - You will need to tag the other helper functions called on the GPU side as __device__.
    - You will need to add a global variable, THREADS_PER_BLOCK, and the following code to the main function to support the GPU:

```
// d_N = n; // comment this line out and replace with:
 if (argc > 6) {
   THREADS_PER_BLOCK = atoi(args[6]);
 }
 cudaMemcpyToSymbol(d_N, &h_N, sizeof(int));
```

    - Almost all of your modifications will go in the simulate function (do not parallelize the initialization of the random numbers)
    - You will need to add code to transfer the positions and velocities to the GPU and transfer the positions back from the GPU at the end
    - You do not need to transfer the data back and forth inside of the loop (except to help debug -- so you can do this within the if debug block)
2) Look at the last parameter from the command line arguments, if set to 0 it will print only the final time; if set to 1 it will print the final state of the positions. You should be able to run the sequential version, get the final position state output and compare that to your GPU version to ensure they are the same. To automate this, we have provided a Makefile target to test correctness:

    make run_test

3) Generate two graphs:
    - one with x-axis as different values for N (1024, 2048, 4096, 8192, 16384, 32768, 65536) and y-axis as time using 128 threads per block
    - one with x-axis as different values for threads per block (1, 16, 32, 64, 128, 256, 512) and y-axis as time using N of 16384
    - To run with these different settings, your code needs to invoke the kernels with correct settings. For example with N of 256 and threads per block of 128 you would have 2 blocks, giving you kernel invocations of <<<2, 128>>>; with 1 thread per block and N of 16384 you would have <<<16384, 1>>>.

- To automate this, we have provided a Makefile target to generate these data points:

  make run_performance

- Your naive implementation should give a time of approximately 0.08 seconds for the first test (N=1024, 128 threads) and 1.50 seconds for the last test (N=16384, 512 threads. If your times are significantly off from these you probably have not implemented a correct parallelization of the code.

4) Create a writeup that discusses/explains your results. Make sure to discuss the trends you see in both graphs and try to connect these to ideas and concepts of CUDA performance discussed in lecture (occupancy, register usage).

5) **[Optional bonus credit]** Improve performance of the simulation. One way to do this is by keeping track of groups of particles. Right now we perform NxN computations to detect collisions. This is certainly overkill, but to do better we must track regions / groupings of particles. Since collisions only happen when two particles are really close to each other, this could significantly improve the performance.
   - Once you have completed an improved version, run the tests from before, generate a new graph, and discuss/explain your results.
   - You are welcome to implement any additional performance improvements as long as the accuracy of the result does not change

Grading:
This lab is worth 100 points. They will be broken down as follows:

70 points = A correct naive parallel implementation
10 points = Performance graphs
20 points = Writeup discussion
up to 50 points = Improvements in step #5