Nate Bowerman and Dhynasah Cakir

CIS 677 – Project 3

**Overview**

This program implements a simulation of diffusion of a particle across a system over a given number of timesteps. The simulation we did used a single injection of the particle in a one-dimensional array, but it could be easily extended to other situations for simulation.

**Sequential Details**

A quick overview of the steps of the sequential algorithm is:

1. Input size of grid, number of timesteps, and initial concentration
2. Generate a grid with the given size and concentration of 0 in all cells
3. Set initial concentration of index 0 to the initial concentration
4. For each index greater than 0, determine the concentration of the next timestep by averaging the current concentration and the concentration of its left neighbor.
5. Repeat step 4 for the number of timesteps
6. Output final array of concentrations

It is straightforward to calculate the next concentration using the left neighbor and itself for most indexes. This means we take the average of the current and left index. And assign this new value to the second array.

The algorithm alternates between two arrays to store the new concentrations at each timestep. This is necessary because the new concentration for index i uses the concentration at index i-1, which is updated in the previous step of the loop. By saving all new concentrations in a different array than the one used to calculate the averages, this is no longer an issue. Whichever density array was updated last is the one that the program outputs.

**Parallelizing**

Steps 1-3 are the same as in the sequential version. Step 4 and on is where the process diverges.

Calling the kernel

The Kernel is run. When the function simulate() is ran, it is called from the host to run the cu code. The arrays, Density1 and Density2 are 2 of the parameters. The size of the array, the time steps and the initial concentration are also parameters in this function.

Cu file code

In the cu file simulate is designated as extern void. First, space is allocated in the device for 2 arrays, Density1\_d and Density2\_d with the size of the Density1 and Density2 arrays. In this file, the arrays that are passed through the function when called in the main function are copied to mirror arrays.

Next, the execution configuration is set. dimBlock is set to the block size. Dimgrid is set to the size of the array divided by the block size. Time steps is then incremented in a for loop. For every time step the function where the density is calculated in then updated. At the same time index 0 is kept constant to simulate a constant drip, or flow into the system.

Once this process is finished the arrays are copied back to the host. And the allocated space is then released.

**Visualization**

This visualization creates a 5x5 inch image with the width of 5X300, a height of 2x300, resolution of 300, and a point size of 8. (plotted in R, code given in appendix). The color here indicates the how the pollution disperses throughout the cylinder while there is a constant stream from one location. Based on this format its difficult to notice a gradual change.

**Speedup Comparison**

In this speed-up analysis we noticed that there was not a significant speedup. Which was not expected. Even though we increased the block size significantly, the speed remained consistent. We were uncertain why we were getting this outcome. One of the reasons we predicted is the way the memory is allocated on the GPU. Since the allocating the space is done manually rather than dynamically it can contribute to the length of time the program runs.

Another reason for our unfavorable outcomes could be that the arrays are created on the CPU first then transferred over to the GPU to do the calculations. If the array was created on the GPU this might increase speed up because as said previously, transferring large data into the GPU is costly and contributes to the time it takes to complete the simulation.

Even with these two factors we should still not see this significant decrease in time or no speed up at all. Which means something else is going extremely wrong and we don’t know exactly what it is. We would need to do a more lengthy and analysis of the code to determine the exact problem.

From the graph we can see that the speed was consistent after an increase to 64 block sizes. Where as the sequential code performed at approximately 55000 micro seconds, the parallel code performed around 40000 micro seconds consistently.