Lab Session 8

Submission deadline: April 17, 11:59pm

Description:

In this lab, you are going to implement a simple biological system simulation based on CUDA. Some biological systems evolve over time and can be modeled using cellular automata. A biological system can be partitioned into a number of cells, each of which is a finite state machine. Each cell's evolution is computed in steps of time; each cell makes one state change, then makes the next state change and so on.

Each cell, represented by an (i,j) element in an MxN matrix, contains an organism that is either dead or alive (represented by a 0 or 1). At each step of the program, a cell's sate changes based on its current state and the state of its neighbors. The following are rules for a cell's state transition at each step:

1. A live cell with zero or one live neighbor dies from loneliness.
2. A live cell with four or more live neighbors dies due to overpopulation.
3. A dead cell with two or three live neighbors becomes alive.
4. Otherwise, a cell's state stays unchanged.

At each step, each cell needs to exchange information with its neighbors. Keep in mind that a cell in the middle has eight neighbors, and a cell in a corner has only three neighbors, and a cell on an edge has five neighbors. Make sure that at each iteration *i*, each CUDA thread gets values from each of its neighbors corresponding to their ith state, and that you solution is deadlock free.

Write a program that takes three command line arguments: two dimension values (M and N) for the matrix; and K the number of iterations (steps) as input. You can have one CUDA thread for each cell in the MxN matrix. You are welcome to try another way of dividing up the matrix if you'd like (either way is fine). No matter how you decide to distribute the matrix, you should choose one CPU process to be the master and CUDA threads to be the slaves. The master should be responsible for sending slaves their original values and receiving the final result from the slaves and printing it to stdout.

You should also have a debug mode where each slave sends the master its value at the end of a round, and the master after receiving values from all slaves, prints out the MxN matrix of round i values. This will help you debug your solution. When your program is not running in debug mode, slaves should not send the master their intermediate results.

Here is some sample output from a run of a program with a 6x4 matrix for two rounds, with debugging turned on.

```
Start:
------
0 0 0 1
0 0 0 1
0 1 0 0
1 1 1 0
0 1 0 0
0 1 0 0
```

```
Round 0:
-------
0 0 1 0
0 0 1 0
1 1 0 1
1 0 1 0
0 0 0 0
1 0 1 0

Round 1:
-------
0 1 0 1
1 0 1 1
1 0 0 1
1 0 1 1
1 0 1 1
0 1 0 0
```

Once your program works, you should evaluate its performance for different sized matrices (two or three different sized MxN matrices should be fine). Measure total execution times, and describe

1. how you divided cells up per processes,
2. your tests,
3. your results,
4. and what, if any, conclusions you can draw from your results.

Hints: Consider to use shared memory to improve performance.

Hand in

1. Your solution source code and Makefile;
2. Performance data with different input combinations;
3. A report (up to 2 pages);