Name: Brijesh Mavani
CWID: A20406960
University: Illinois Institute of Technology
Course: Parallel and Distributed Processing
Assignment: 5 - Design Report

There are two codes written for implementing Conway's Game of life.

1) Serial code in C.

2) Parallel code in Cuda.

Implement details are as follows for both the implementation:

1. Serial code:
   The Main function will take two command line arguments. The first argument will be number of iterations to consider and second will be Matrix size. I have considered matrix as a square matrix. I have created a matrix in a form of 1D vector for simplicity. The matrix will be initialized with random values of 1 and 0 as per the requirement. I have used Rand () %2 operations to initialized matrix with 1s and 0s. After creating a main matrix, another matrixN is created to hold the intermediate results of each iteration. Two memory pointers (cmatrix and nmatrix) are created to point the initial main matrix and newly created matrixN. This will allow me to operate them directly without passing an entire matrix for each iteration. At the initial stage, the cmatrix will point to initial matrix and nmatrix pointer will point to matrixN. At every iteration the game rules will be applied and corresponding results will be stored in another matrix as described below:
   1st iteration will work on main matrix and store result in a matrixN. At second iterations pointers cmatrix and nmatrix will be changed to point matrixN and the main matrix (initial matrix) respectively. By doing this I am assuring that no two iterations work and store result in the same matrix. Additionally, by storing result in different matrix at each iteration and changing the pointers as well will guarantee that results will be computed correctly. Also, for debugging, I added print matrix statements after 1st, 2nd and 3rd iterations and checked results manually for the correctness.
   As per the requirement, I printed the resulting matrix after 10, 100 and 1000 iterations. After all iterations are executed final resulting matrix will be printed for a reference. The timing is also calculated for timing analysis. The timer will be started just before starting iteration loop and stopped after all iterations are executed. At end total elapsed time will be printed.

2. Parallel code in Cuda:
   The implementation is quite similar with serial implementation in C. There are few changes to make it run in parallel as well as Cuda functions and implementation guidelines has been used. The Main function will take two command line arguments. The first argument will be number of iterations to consider and second will be Matrix size. I have considered matrix as a square matrix. As we need to copy the matrix from CPU to GPU and GPU to CPU repeatedly, I have

created a matrix in a form of 1D vector for simplicity and reduce the copying time. This matrix will be created in CPU and will be initialized with random values of 1 and 0 as per the requirement. I have used Rand () %2 operations to initialized matrix with 1s and 0s. After creating an initial matrix in CPU, another two matrices currentmatrix and nextmatrix will be created in the GPU.
The initial matrix from the CPU will be copied to currentmatrix in GPU and nextmatrix will be initialized with all 0s.
Two matrices are created in GPU to hold the intermediate results of each iteration. The Two memory pointers (cmatrix and nmatrix) are created to point the initial main matrix and newly created matrixN. This will allow me to operate them directly without passing an entire matrix for each iteration. At the initial stage, the cmatrix will point to initial matrix and nmatrix pointer will point to matrixN. The cuda kernel function will be called with num of threads as 16 and block as input matrix size/16. The Cuda kernel function will be called for N times as per the input. At every iteration the game rules will be applied and corresponding results will be stored in another matrix as described below:
1st iteration will work on the currentmatrix and store result in nextmatrix. At second iterations pointers cmatrix and nmatrix will be changed to point nextmatrix and currentmatrix respectively. By doing this I am assuring that no two iterations work and store result in the same matrix. As the output matrix is different, input matrix will be read only for each thread. Both these matrices will be accessible to all threads as they reside in GPU memory and accessed via memory pointer. This indicates no dependency between the data. Additionally, by storing result in different matrix at each iteration and changing the pointers as well will guarantee that results will be computed correctly. Also, for debugging, I added print matrix statements after 1st, 2nd and 3rd iterations and checked results manually for the correctness

As per the requirement, I printed the resulting matrix after 10, 100 and 1000 iterations. After all iterations are executed final resulting matrix will be printed for a reference. The timing is also calculated for timing analysis. The timer will be started just before starting iteration loop and stopped after all iterations are executed. At end total elapsed time will be printed.

The evaluation of the matrix cells is very heavy on memory I/O and not so much on arithmetic operations. The GPU has very fast memory chips with much higher memory throughput compared to the CPU's RAM. Due to this, the GPU is significantly faster than the CPU when computations are memory intensive. This superiority of the GPU over CPU plays a critical part in giving better speed up.

When to use GPU: A CPU consists of a few cores optimized for sequential serial processing, while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. The GPU has very fast memory chips with much higher memory throughput compared to the CPU's RAM. So, they are better performer when the code requires to execute same instructions on huge data in parallel. They also provide better results when computations are memory intensive as seen in this experiment.

- Efficiency
    - ✓ 1024x1024 Matrix using 4 GPU nodes and 1000 iterations: 5.70
    - ✓ 10000x10000 Matrix using 4 GPU nodes and 1000 iterations: 6.20

- Speed up
    - ✓ 1024x1024 Matrix with 1000 iterations: 22.812.
    - ✓ 10000x10000 Matrix with 1000 iterations: 24.786.