Introduction

The aim of the assignment is to test your understanding and technical ability of implementing efficient code on the GPU with CUDA. You will be expected to benchmark and optimise the implementation of a simple rule based simulation. You will start by implementing a serial CPU version, you will then parallelise this version for multi-core processors using OpenMP, before finally implementing the same simulation in CUDA. The emphasis of the assignment is on how you have optimised and progressively improved your code in order to implement the simulation efficiently. In order to demonstrate this you are expected to document the processes of benchmarking and improving your code by producing a written report. The written report should show the performance improvements of your code and demonstrate that you understand what limiting factors affect the performance at each stage of your implementation. Handing in just a piece of code with excellent performance will not score highly in the assessment unless you have demonstrated in understanding in the written report of how you have progressively refined your implementation to achieve the final solution.

The Task

Conway's Game of Life[1] is a simple cellular automaton which describes a rule based game in which a player describes only the initial system state. A cellular automaton is a discrete model which consists of a population of regularly spaced cells each with a number of states. By communicating with neighbouring cells, a new generation of the population (at $t + 1$) can be generated by applying a set of rules which use the cell's current state and the cells neighbouring states to determine the next state of the current cell. With respect to the Game of Life, the specific rules of the game are very simple. A cell within the population is either in an *alive* or *dead* state. At each time step, a cell considers its Moore neighbourhood of immediately adjacent neighbours, including diagonal neighbours, of which there are 8 in total (see Figure 1).[2]

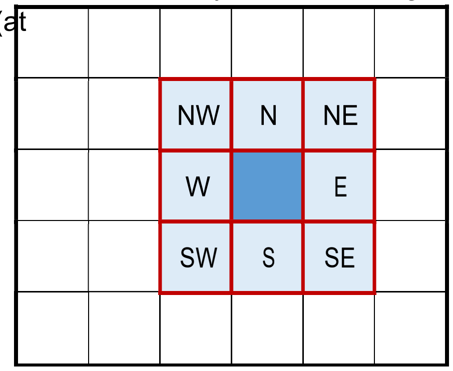

Figure 1 - Game of Life cell has a Moore Neighbourhood of exactly 8 neighbours

For any live cells the following rules are applied in determining the state of the cell at $t + 1$;

1) If there are fewer than two live neighbours; the cell dies of loneliness.
2) If there are two or three live neighbours; the cell remains alive.
3) If there are four or more live neighbours; the cell will die due to overcrowding.

For any dead cells the following rules are applied in determining the state of the cell at t+1;

[1] https://en.wikipedia.org/wiki/Conway's_Game_of_Life
[2] https://en.wikipedia.org/wiki/Moore_neighborhood

1) If there are exactly three live neighbours; the dead cell becomes alive due to reproduction.
2) For any other configuration of neighbours the cell remains dead.

The implementation of the task is split into two parts. The first part of the assignment requires implementing the game of life simulation. The second part of the assignment considers how to count a number of patterns observed during simulation.

Part 1 Requirements: You should start from the provided source code and complete the `TODO` sections. Your program should accept the arguments described by the existing `print_help()` function. The file format for Game of Life files (both input and output) should use plain text with a line *width* and *number of lines* matching the program arguments *W* and *H* respectively. Each line should be terminated with a carriage return ('\n'). A single character should be used to encode the state of each grid cell. If the cell is dead the character should have a space value (i.e. character `' '` with ASCII code `#32`). If the cell is alive the character should have a value of `'#'` (i.e. ASCII code `#35`). Using this plain text format you can visually check the state of any Game of Life file.

You should implement a serial C version of the code and make this as efficient as you can. Once implemented your serial version should act as a baseline to measure potential performance speedup of your parallel versions. *Note: It is not permitted to use data structures such as trees to spatially reduce the grid. You code must explicitly store a 2D set of state data for each cell and progress the simulation by considering the Moore neighbourhood.*

You should implement a multi core parallel version of the same simulation which uses OpenMP.

Finally, you should implement a CUDA version of the simulation. You should consider how using different approaches for memory caching affect performance over a range of problem sizes. You should also consider how boundary conditions (overlapping regions between thread blocks) are handled and give an overview of any approaches you have applied to improve boundary conditions performance.

For all versions of the implementation the environment should have wrapping bounds. This means that the environment should be 2D but form a 3D torus where cells on the extreme right side of the environment are neighbours to cells on the extreme left, and cells on the extreme top side of the environment are neighbours to cells on the extreme bottom (Figure 2).

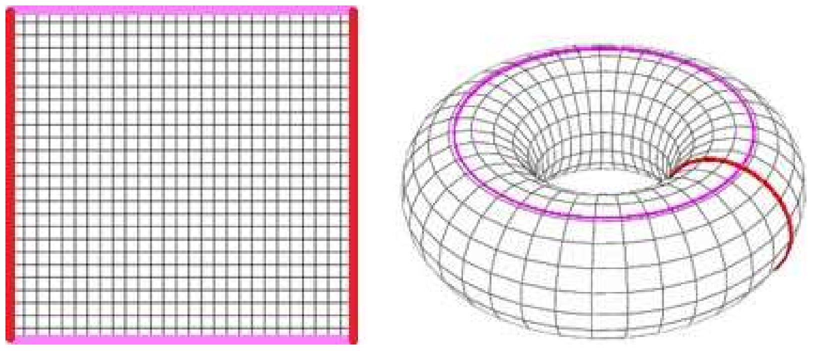

Figure 2 - Wrapping the bounds of a 2D environment to form a 3D continuous torus.

Part 1 Expected Outputs: For part 1 the expected output (via console or plain text file) is that your code should provide the execution time. If the execution mode is *CUDA* then the output timings should include both timing data for just the kernel execution and a separate timing value for total execution time including the cost of any data movement to or from the device. If the execution mode is *ALL,* timings for each case should be displayed to the console or saved to a local file in plain text. The timing value of reading and writing to disk should be ignored in all cases.

It is important that you ensure your code produces the correct result for each mode by validating the output. A compiled executable is provided as part of the assignment hand-out which is able to provide the expected outputs given a starting state and number of simulation loops. It is expected that your program passes a number of blind tests during marking and assessment. These tests are blind in that you do not know what they will be other than they will consist of a number of start states, grid sizes and a pre-defined number of simulation loops. For each blind test there is an expected correct terminal state of all cells which your implementation should produce by executing the deterministic rules.

Part 2 Requirements: It is required that you update your three implementations (serial, OpenMP and CUDA) to be able to provide analysis of the simulation by recognising common patterns. For example, there are a number of patterns which once created in the Game of Life will remain static unless their immediate neighbours are modified. These static patterns are often referred to as a still life patterns. For the purposes of the assignment it is required to know the total number of still life *Cross* patterns (Figure 3) for any given iteration.

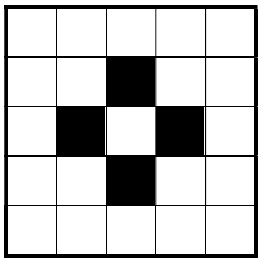

Figure 3 - The Cross still life pattern.

The Game of Life can also exhibit oscillating patterns. These oscillating patterns are classed by the period (number of iterations) which is required for them to repeat. For the purposes of the assignment it is required to know the total number of Blinker patterns (period 2) which are exhibited for any given iteration. See Figure4.

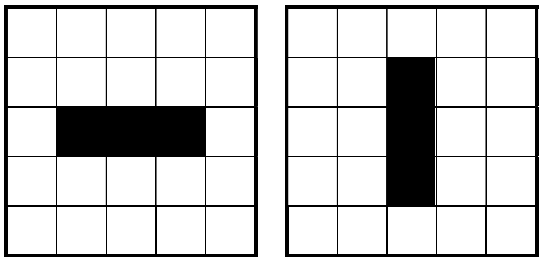

Figure 4 - The oscillating Blinker Patter. On the left the blinker at iteration 1, 3, 5.... On the right the Blinker pattern at iteration 2, 4,6.....

A Glider is a special case of an oscillating pattern which migrates across the screen as it oscillates. For the purposes of the assignment it is required to know the total number of Gliders which are exhibited for any given iteration. Figure 5 shows a the 4 patterns that are exhibited by a Gliders over 4 iterations when travelling in a south east direction. You are required to calculate the number of gliders heading in <u>any</u> direction. In total there are 16 unique patterns which can be obtained by rotating each of the patterns in Figure 5 by 90, 180 and 270 degrees.

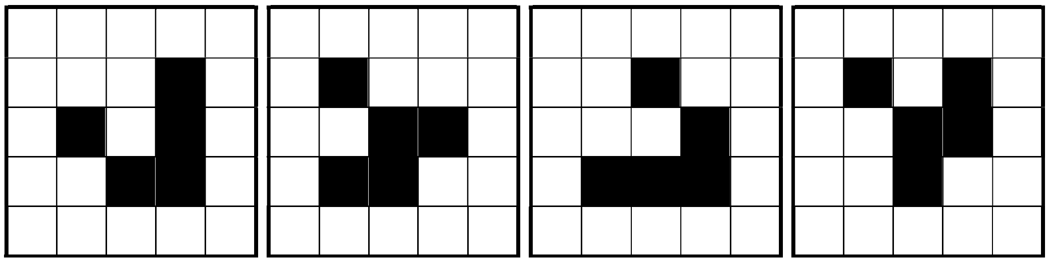

Figure 5 - The four Glider patterns for a glider travelling in a south east direction. Each pattern is shown centralised within a 5x5 grid at periods 1 through to 4. There are 16 total possible combinations of the glider patter depending on the direction of movement.

Note: that for all pattern recognition tasks (Cross, Blinkers and Gliders), it is important that the pattern and surrounding *5x5* neighbours match those shown in the figures exactly. For example when matching a pattern both the live and dead cells must represent the 5x5 patterns exactly. I.e. there must be 25 exactly matching cells. *Figure 6* shows some examples of patterns which should not be matched.

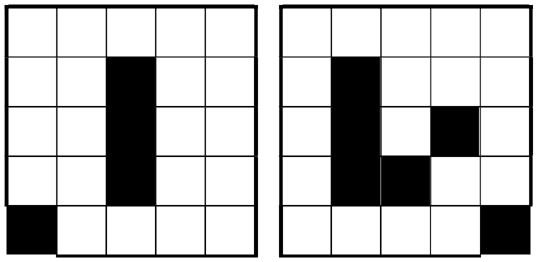

Figure 6 - A non-matching Blinker pattern (left) and non-matching Glider configuration (right). Both have interfering neighbours where the 5x5 neighbourhoods do not match exactly.

In order to implement counting of patterns you will be required to implement a form of 2D parallel reduction. You should consider different approaches (e.g. recursion, shared memory and warp shuffling) for implementing this reduction and describe how the performance of each suits the technique you have implemented.

Note: Patterns which span the boundaries of the environment should still be recognised and counted.

Part 2 Expected Output: For part 2 the expected output is that for each iteration you should output either to the console (or a plain text file) the counts of each of the 3 patterns in the following format. This will require that you implement the counting analysis technique for all three (serial, OpenMP, CUDA) versions of your code.

```
"Iteration %d, Crosses=%d, Spinners=%d, Gliders=%d\n"
```

Project Hand In

You should hand in your program code via OL with the documentation as a single pdf within a single zip file. You should also include the visual studio solution (or makefile) and any project files. Your code should build in Visual Studio release mode configuration without errors or warnings (or for Linux with an appropriate Makefile). You should submit whatever you have completed even if you have not completed the entire assignment. Your code should not rely on any third party libraries or tools.

If you use un-modified code from any of the lab solutions you should make it clear that the code is not your own with comments.

Marking

The marks for part 1 of assignment will be distributed as follows:

- *50%* of the assignment is for the coding aspect.
    - *50% of the coding marks* are for the quality of the programming and performance of your code.
    - *50% of the coding marks* are for satisfying the requirements.
- *50%* of the assignment is for the production of a document describing the processes you have undertaken to implement and optimise your code. This should include benchmarking and iterative refinement of approaches as described in the documentation requirements.

For each of parts 1, 2 and 3 the same 50:50 split of coding and documentation will be used.

In making assessment, the following requirements will be considered.

1. Is the code functionally correct for a set of test cases? I.e. does it produce the correct terminal state (output) given a specific input? Is the correct count returned for each pattern at every iteration?
2. Has iterative improvement of the code yielded sufficiently optimised code for each of the CPU, OpenMP and CUDA versions?
3. Does the code make good use of memory bandwidth (on CPU and GPU)? Is caching used effectively through data access patterns and use of specialised caches (on GPU)?
4. Does the OpenMP and CUDA implementation avoid race conditions (Especially when reducing values)?
5. Are the OpenMP variables correctly scoped? Is their excessive/unnecessary use of shared variables?
6. Have you managed to use GPUs appropriately ensuring that the device has sufficient levels of parallelism?
7. Are boundary conditions handled elegantly and efficiently?
8. Are there any compiler warnings or dangerous memory accesses (beyond the bounds of the memory allocated)? Does your program free all memory which is allocated?
9. Is your handling of input files correct and robust? Does the program deal with incorrect input file formatting elegantly (e.g. raising an error and exiting rather than crashing)?
10. Is your code structured clearly and well commented?

In assessing your documentation, the following will be considered and should act as a guideline for discussing incremental improvements to your code.

1. Description of the technique and how it is implemented. Is a good justification given for the choice of parallelisation method?
2. Have appropriate investigations been made into using a good memory access pattern and suitable caching technique? Are good explanations given for the benchmarking results?
3. Does your document describe optimisations to your code and show the impact of these?
4. Is there benchmarking and discussion about the performance difference between all three versions of the code?

Tips for Developing Your Code and Documentation

You should start with a simple serial implementation and describe how you iteratively improve the performance in your documentation. If you are unable to complete some specific part (for example parallel reduction on the GPU) then you should default back to using the simple CPU version for that part so that your code correctly builds and executes producing the correct result. Similarly if you apply a technique and find it does not improve performance you should include this in your documentation and explain why it did not work as expected. You can use `#define` to allow your code to be built in different versions to make a comparison of techniques more straight forwards.

You should comment your code to make it clear what you have done. You should test your code to make sure that it works for ALL grid sizes. For grid sizes which may fail (for example a grid width of negative 10 or an input file which does not match the specified grid width and height) your code should exit elegantly with an error message. Your code should never read or write beyond allocated memory on either the host or the device.

When benchmarking the GPU aspect of your code you should pick thread block sizes which either maximise occupancy or use a progressive search of the kernel launch parameters to find the best launch configuration. You should show via graphs or tables what launch configurations are best and give explanations in each case as to why. You can include profiling information to help with this.