Assignment 4
1-D Game of Live
Due date: Sunday, December 5, 11:59 pm

Description

Write an OpenCL program in C that implements the 1-D Game of Life cellular automaton described by J. Millen.

This assignment can be done in groups of 1-4 students.

1-D Game of Life

This algorithm creates an image by calculating one row at a time based upon the previous row. The image is represented by a two dimensional array.

The top row of the image is defined before the automata algorithm is executed. Each of the rows after the first is defined using the row above and the algorithm. The program runs until the last row is calculated. When the program ends there will be an image stored in the array.

Each element in the array can be either occupied or empty. Each element in a row is calculated using the five adjacent cells on the line above it. Depending on the pattern in the five neighbours the element in the next row will be either occupied or empty. The algorithm for this particular 1-D Game of life is described here: http://jonmillen.com/1dlife/index.html

Do not perform all calculations in global memory. Calculate each new line in the array in local or private memory. After all kernels calculate their section of a row the the results can be copied to global memory.

Don't needlessly pass information between the kernel and the application. Pass the initial array to the kernels and then read the completed array back into the C program when all of the calculations for all rows are complete.

Empty cells in the array should contain a blank space. Occupied spaces should contain the rank of the kernel which filled that space. This will create a final array which is divided into columns based upon which kernel calculated the spaced. In the following example there are four kernels used to calculate the results and the array is divided into four columns.

```
-------------------------
|rank 0|rank1|rank2|rank3|
|      |     |     |     |
|      |     |     |     |
|      |     |     |     |
|      |     |     |     |
-------------------------
```

The number of columns which each kernel is responsible for is determined by dividing the number of columns by the number of kernels. If the number of columns is not evenly divided by the number of kernels then the last kernel will be responsible for fewer columns.

The following example shows a row which contains spaces that are filled by kernels with ranks and 1.

```
   0001111
  00 011 11
 0 0  1  1 1
  0 00 11 1
   000 111
```

Note that all of the columns to the left are filled by kernel 0 and those on the right are filled by kernel 1. If the same automata were run using four kernels then it would look something like this:

```
   0111222
  00 112 23
 0 0  1  2 3
  0 11 22 3
   011 222
```

The leftmost columns were calculated by kernel rank 0, the second set of columns - by kernel rank 1, etc..

Print the final image to the screen using ASCII characters. Print the results only once the entire array has been calculated - do not print any intermediate results. Use the 1-D GoL URL (http://jonmillen.com/1dlife/index.html) to verify the correctness of your implementation.

In the printed results, display the kernel numbers. For empty cells, display the `.` character, as you did in A3. Also, make sure you center the initial configuration pattern in the middle of the first row. If the initial configuration has an odd number of starting live cells (e.g. Face), there will be two ways of entering is (see below).

So if you populate a 12x12 array with the Face as the starting configuration and use 4 kernels, the first 5 lines of your output should be:

```
...0111222..
..00.112.23.
.0.0..1..2.3
..0.11.22.3.
...011.222..
```

or

```
..0111222...
.00.112.23..
0.0..1..2.3.
.0.11.22.3..
..011.222...
```

Command Line Arguments

The command line arguments are:

`-n #` - the number represents the number of kernels. NOTE: if `n > 10` (i.e. we have more than 10 kernels), we run out of digits. So instead of displaying the kernel number in the output, just display the `X` character. So the output for `n > 10` would look something like this:

```
..XXXXXXX...
.XX.XXX.XX..
X.X..X..X.X.
.X.XX.XX.X..
..XXX.XXX...
```

Use n ≤ 10 when debugging your code, so you can see how the work is distributed between kernels. But remember to test it with n > 10.

`-s #` - the number represents both the height and width of the array
`-i #` - the number indicates which initial configuration will be used in the first row of the array

The default number of kernels is 1. The default size of the array is 20. The initial configuration is 0 (random).

The initial configuration patterns and associated command line numbers `i` for the array are:
- `0`: random pattern. Each element in the row has a 50% chance of being occupied.
- `1`: FlipFlop. Pattern is `X XX`
- `2`: Spider. Pattern is `XXXXXX`
- `3`: Glider. Pattern is `X XXX`
- `4`: Face. Pattern is `XXXXXXX`

The sample output for the above initial configurations is available here: http://jonmillen.com/1dlife/index.html

OpenCL

The program must run on the `linux.socs.uoguelph.ca` server. Use `oclgrind` to run the OpenCL emulator. Run your compiled C program which will run the OpenCL kernel using:

```
oclgrind ./a4 <-n #> <-s #> <-i #>
```

For example:

`oclgrind ./a4 -n 80 -s 80 -i 2` will run the program using 80 kernels, using an array that is 80x80 units, and using the initial row configuration two, which means the first row will be a Spider.

`oclgrind ./a4 -n 5 -s 20 -i 0` will run the program using five kernels, using an array that is 20x20 units, and using the initial row configuration zero, which means the first row will be

random.

Documentation

Include a `readme.txt` contains your name and student number. If anything does not work correctly then include a description of the problem in the `readme.txt` file.

Submission and Evaluation

Submit the assignment using Moodle. Submit only the source code (both the C code and the `.cl` code) and the makefile. Bundle the code in a zip file.

Please submit one assignment per group. Make sure your submission includes a README file that contains the list of all group members.

The assignments will be marked on the `linux.socs.uoguelph.ca` server. If you develop your code on a different platform, then it is a good idea to test your program on the `linux.socs.uoguelph.ca` server!

The TA will unpack your code and type "`make`". They will then try to run executable named `a4` using:

```
oclgrind ./a4
```

If the makefile is missing, the make command does not work, or the program does not execute then you will lose a substantial number of marks.

It is always a good idea to unpack and test the file you are submitting to be sure that what you submit actually compiles.