1 Introduction

1.1 Motivation

Although Moore’s Law continues, CPU clock speeds can no longer grow due to physical and power limitations. For this reason, hardware manufacturers have begun to build processors with several cores and parallelization constructs. Consequently, the burden is placed on the programmers to make use of these several cores and parallelization tools. The ability to program parallel architectures and to organize code in a parallelizable way is more important than ever.

To this end, this project provided us with a working introduction to writing parallel software and taking advantage of these tools. We were also able to measure the degree to which parallelized code performs better than unparallelized code. Several frameworks exist to facilitate programming general parallelized code; for example, Pthreads. Additionally, specialized frameworks exist to facilitate parallel programming using GPU hardware. These frameworks include CUDA and OpenCL.

This project examines the improvements in performance achievable by applying several different methods of parallelization to the well-known cellular automaton, Conway’s Game of Life. The methods used are single-threaded updates, multithreaded updates with varying numbers of threads, and finally GPU computation of the updates.

1.2 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing framework developed by Nvidia which leverages the GPU for the actual processing. Nvidia additionally provides a variant of the C programming language called C for CUDA, which is the most common language through which CUDA is programmed.

1.3 OpenCL

OpenCL (Open Computing Language) is an alternative language binding to CUDA which is compatible with hardware produced by ATI and Intel as well as Nvidia. OpenCL exposes the same CUDA functionality as C for CUDA, but with a slightly different syntax. OpenCL was originally developed by Apple, but is now managed by the nonprofit Khronos Group.

1.4 Cellular Automata

Cellular automata are a class of computable structures that consist of a regular grid of ‘cells’. In the case of Conway’s Game, the grid is a two-dimensional rectangular matrix, but automata have been formulated for three or more dimensions. Each cell in an automaton has some internal state. In the case of Conway’s Game, the state is a single bit, representing ‘alive’ or ‘dead’ for that cell.

Beginning from some initial state (said to be the time $t=0$), the state of each cell in the grid is simultaneously updated based on some rule. This rule is applied in the same way for each cell, updating the state of all cells in the automaton at each time step.
2 Conway’s Game of Life

2.1 Overview

Conway’s Game of Life is a simple cellular automaton governed by four simple rules. The computation involved lends itself to parallelization because each cell’s next state depends only on the state of its immediate neighbors. Thus, each cell can be computed individually and thus distributed into arbitrary partitions for execution on separate threads with minimal transfer of data. The rules of Conway’s Game are as follows (per Wikipedia):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

2.2 Emergent Behavior

Although the rules are individually very simple, the Game of Life is well-known for the unpredictable and emergent behavior that its states exhibit in certain conditions. Although most simple structures of cells (such as boxes, lines, and circles) will divide, scatter apart, and die out, there are some structures capable of more interesting behavior.

Structures consisting of only a few dozen live cells, such as the Gosper’s Glider Gun (shown at left) are capable of indefinite production of new structures. In this case, the Glider Gun oscillates while remaining in a 10x40 area of the grid. With each oscillation, the gun produces a ‘glider’, a five-cell structure that moves diagonally until hitting another live cell.

Many larger and more complex structures have been discovered. One such structure, shown in the middle, is a moving reproductive pattern that creates Glider Guns in its wake. Some very complex patterns are capable even of self-reproduction. These intriguing behaviors have led many to study the computational aspects of Conway’s Game and other cellular automata.

2.3 Implementation

Our implementation of Conway’s Game is written in C, with the GPU interface written in OpenCL. The Game of Life automaton is represented in our program as a two-dimensional grid of unsigned integer values. Each integer holds one of two values: ‘alive’ or ‘dead’. The grid is initialized with a sequence of randomly-generated Boolean values. For the tests run in this case, each cell was initialized with a 50% chance of being alive, and 50% of being dead. This left a large enough number of live cells for the automaton to
continue exhibiting interesting behavior for the duration of the test. The same random seed was used to initialize each grid.

At each time step, an update function applies the Game’s rules to each cell, and that cell’s new state is written to a separate buffer. This buffer is switched with the original grid after each update. The automaton can be run for as many iterations as required. To show a reasonable balance between overhead and runtime, the data collected for this report consists of runtimes for 100 iterations, on different grid sizes.

For the single-threaded and CPU multi-threaded versions of the program, the update function uses a helped function that finds the number of live cells surrounding any particular cell. This value is then used to determine whether the cell in question should become live or dead.

```c
// Given a specific location in a 2D cellular grid, this function
// returns the sum of the 8 cells that compromise that location's
// Moore neighborhood
// The function 'wraps around' by treating the far right column
// as adjacent to the far left, and top as adjacent to bottom.
uint sum_moore_neighbors(Grid &state, uint row, uint col) {
    int left, right, top, bot;
    left = (col-1); if (left < 0) left += state.nc();
    right = (col+1); if (right == state.nc()) right = 0;
    top = (row-1); if (top < 0) top += state.nr();
    bot = (row+1); if (bot == state.nr() ) bot = 0;

    uint total = 0;
    total += uint(state[top][left] != DEAD);
    total += uint(state[top][col] != DEAD);
    total += uint(state[top][right] != DEAD);
    total += uint(state[row][left] != DEAD);
    total += uint(state[row][right] != DEAD);
    total += uint(state[bot][left] != DEAD);
    total += uint(state[bot][col] != DEAD);
    total += uint(state[bot][right] != DEAD);

    return total;
}
```

### 2.4 Single Threaded Implementation

On a single thread, the function to calculate the next states for all cells is a simple nested for loop. The entire grid is updated sequentially, row by row and column by column. The logical expression used in the state.out call enforces the four rules given above as the Game’s definition.

```c
// Given an input cellular automaton grid, apply the rules of one
// iteration of Conway's Game of Life. Output the new state to the
// given output grid
void next_state(Grid &state_in, Grid &state_out) {
    uint neb, val;
    for (int j = 0; j < state_in.nr(); j++) {
        for (int i = 0; i < state_in.nc(); i++) {
            val = state_in[j][i];
            neb = sum_moore_neighbors(state_in, j, i);
            state_out[j][i] = (neb == 3 || neb == 2 && val != DEAD) ? ALIVE : DEAD;
        }
    }
}
```

### 2.5 Multithreaded CPU Implementation

In order to update parts of the grid in parallel, a new update function was written, one that takes parameters specifying which rows of the grid to update. The function uses the same sum-neighborhood helper, but...
only updates its given part of the grid.

```c
void next_state_partial(Grid &state_in, Grid &state_out, uint row_start, uint row_end) {
    uint neb, val;
    for (uint j = row_start; j < row_end; j++) {
        for (int i = 0; i < state_in.nc(); i++) {
            val = state_in[j][i];
            neb = sum_moore_nebbers(state_in, j, i);
            state_out[j][i] = cell((neb == 3 || neb == 2 && val != DEAD) ? ALIVE : DEAD);
        }
    }
}
```

In order to calculate the full state, the grid is subdivided into a number of equally-sized parts, and each thread is dispatched to update one part, using the function given above. The multithreaded CPU implementation uses Dlib’s Threads library, which acts as a wrapper around pthreads or the Windows equivalent. Given a command-line specified number of threads, a thread pool is created. From that pool, each thread is assigned a function to complete, in this case, the partial update function with parameters specifying a part of the grid. The main thread sleeps until each of the dispatched pool threads have completed their allotted task.

```c
dlib::thread_pool tp(num_threads);
dlib::swap(flip,flop);
while (run_time-- > 0) {
    for (uint i = 0; i < num_threads; i++) {
        // Each thread is allotted a task using the following lambda function
        tp.add_task_by_value([&flip, &flop, height, num_threads, i]() {
            next_state_partial(
                flip, flop,
                (height * i) / num_threads,
                (height * (i+1)) / num_threads );
        });
    }
    tp.wait_for_all_tasks();
dlib::swap(flip,flop);
}
```

### 2.6 Parallel GPU Implementation

A large portion of the runtime involved in the OpenCL implementation is taken by memory access between the CPU and the GPU. Any memory access between the CPU and GPU requires a large amount of overhead. Once data has been transferred to the GPU, however, its massively-parallel processors can apply the update function much faster than the CPU. For this reason, the GPU performs better as the number of time steps to be calculated increases.

A large amount of code is required to create an OpenCL context object, bind it to a GPU, and transfer data and kernel code to the GPU. This overhead code is documented in the appendix. Following is the kernel function applied to each element of the grid at each time step.

```c
__kernel void conway_game(
    __global int* in,
    __global int* out,
    int rows, int cols)
{
    int left, right, top, bot, row, col, val;
    row = get_global_id(0);
    col = get_global_id(1);
    // Bounds check
    if (row >= rows || col >= cols)
    { return;
    }
```
3 Results

After writing the program, we ran varying sizes of grids for each of four execution modes. First, the automaton was run using single-threaded updates, using two concurrent threads, using four concurrent threads, and using the GPU.

3.1 Test Executions

At the bash prompt, the program was executed for various grid sizes using the Unix Time command. Due to its much lower runtime at large grid sizes, the GPU was tested for a larger range of grids than the other update methods.

```
$ for i in 100 200 300 400 500 600 700 800 900 1000 1500 2000 3000 4000; 
  do time ./conway_life -w 1000 -h $i -t 100; done

$ for i in 100 200 300 400 500 600 700 800 900 1000 1500 2000 3000 4000; 
  do time ./conway_life -w 1000 -h $i -t 100 -n 2; done

$ for i in 100 200 300 400 500 600 700 800 900 1000 1500 2000 3000 4000; 
  do time ./conway_life -w 1000 -h $i -t 100 -n 4; done

$ for i in 100 200 300 400 500 600 700 800 900 1000 1500 2000 3000 4000 5000 10000 15000 20000 30000; 
  do time ./conway_life -w 1000 -h $i -t 100 -g; done
```

3.2 Results

The following table shows the parameters and results of each run. At left is the number of cells comprising each grid, in thousands. After that is the runtime, in seconds, for 100 iterations of the Game of Life. All tests were performed on a desktop PC running a four-core AMD Phenom II at 3.4GHz, and an AMD Radeon HD 5770 GPU.
<table>
<thead>
<tr>
<th>Cells(K)</th>
<th>Serial</th>
<th>2-Thread</th>
<th>4-Thread</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.24</td>
<td>0.15</td>
<td>0.12</td>
<td>0.66</td>
</tr>
<tr>
<td>200</td>
<td>0.36</td>
<td>0.18</td>
<td>0.13</td>
<td>0.62</td>
</tr>
<tr>
<td>300</td>
<td>0.43</td>
<td>0.25</td>
<td>0.16</td>
<td>0.59</td>
</tr>
<tr>
<td>400</td>
<td>0.56</td>
<td>0.3</td>
<td>0.24</td>
<td>0.62</td>
</tr>
<tr>
<td>500</td>
<td>0.69</td>
<td>0.37</td>
<td>0.23</td>
<td>0.64</td>
</tr>
<tr>
<td>600</td>
<td>0.85</td>
<td>0.44</td>
<td>0.28</td>
<td>0.65</td>
</tr>
<tr>
<td>700</td>
<td>0.95</td>
<td>0.5</td>
<td>0.38</td>
<td>0.62</td>
</tr>
<tr>
<td>800</td>
<td>1.13</td>
<td>0.57</td>
<td>0.34</td>
<td>0.64</td>
</tr>
<tr>
<td>900</td>
<td>1.22</td>
<td>0.65</td>
<td>0.41</td>
<td>0.65</td>
</tr>
<tr>
<td>1000</td>
<td>1.35</td>
<td>0.69</td>
<td>0.42</td>
<td>0.66</td>
</tr>
<tr>
<td>1500</td>
<td>2</td>
<td>1.05</td>
<td>0.61</td>
<td>0.75</td>
</tr>
<tr>
<td>2000</td>
<td>2.65</td>
<td>1.37</td>
<td>0.84</td>
<td>0.78</td>
</tr>
<tr>
<td>3000</td>
<td>3.93</td>
<td>2.17</td>
<td>1.24</td>
<td>0.83</td>
</tr>
<tr>
<td>4000</td>
<td>5.22</td>
<td>2.7</td>
<td>1.51</td>
<td>0.94</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td></td>
<td></td>
<td>1.06</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td></td>
<td>1.44</td>
</tr>
<tr>
<td>15000</td>
<td></td>
<td></td>
<td></td>
<td>1.89</td>
</tr>
<tr>
<td>20000</td>
<td></td>
<td></td>
<td></td>
<td>2.32</td>
</tr>
<tr>
<td>30000</td>
<td></td>
<td></td>
<td></td>
<td>3.3</td>
</tr>
</tbody>
</table>

Although it starts with a much higher apparent overhead cost, the GPU becomes far faster than alternatives after more than a few thousand cells must be computed at once. The approximately 0.5 seconds required for the GPU to load and compile a new OpenCL kernel is amortized for very large grids, and extrapolation shows that the single-threaded version of the program would probably require close to a minute to run at 30 million cells, compared to the GPU’s performance of 3.3 seconds.
3.3 Interpretation

The data shows that spawning OS threads for the computation unambiguously improves performance, even at relatively small grid sizes. It also shows that after crossing a threshold where the GPU speedup becomes better than the overhead, GPU acceleration is dramatically more performant.

4 Full C++ Source

```cpp
#include "dlib/console_progress_indicator.h"
#include "dlib/logger.h"
#include "dlib/image_io.h"
#include "dlib/image_transforms.h"
#include "dlib/gui_widgets.h"
#include "dlib/cmd_line_parser.h"
#include "dlib/threads.h"

#include "CL/opencl.h"

#include <vector>

using namespace dlib;
using namespace std;
```
// Typedefs for convenience
typedef unsigned int uint;
typedef uint cell;
typedef cell data_t;

// Just as in C, we define our grid to be of unsigned 32-bit ints
typedef dlib::array2d<cell> Grid;

// This is the full-featured version of the dlib command line parser
typedef dlib::cmd_line_parser<char>::check_1a_c cl_parser;

// Let's define the cell states with an enumeration
define CellState {
    DEAD = 0,
    ALIVE = 100
};

// Given a specific location in a 2D cellular grid, this function
// returns the sum of the 8 cells that compromise that location's
// Moore neighborhood (that is to say, all cells with a Chebyshev
// distance of one, or in real person terms "the ones next to it")
// The function 'wraps around' by treating the far right column
// as adjacent to the far left, and top as adjacent to bottom.
uint sum_moore_nebbers(Grid &state, uint row, uint col) {
    int left, right, top, bot;
    left = (col-1); if (left < 0) left += state.nc();
    right = (col+1); if (right == state.nc()) right = 0;
    top = (row-1); if (top < 0) top += state.nr();
    bot = (row+1); if (bot == state.nr()) bot = 0;

    uint total = 0;
    total += uint(state[top][left] != DEAD);
    total += uint(state[top][col] != DEAD);
    total += uint(state[top][right] != DEAD);
    total += uint(state[row][left] != DEAD);
    total += uint(state[row][right] != DEAD);
    total += uint(state[bot][left] != DEAD);
    total += uint(state[bot][col] != DEAD);
    total += uint(state[bot][right] != DEAD);

    return total;
}

// Given an input cellular automaton grid, apply the rules of one
// iteration of Conway's Game of Life. Output the new state to the
// given output grid
void next_state(Grid &state_in, Grid &state_out) {
    uint neb, val;
for (int j = 0; j < state_in.nr(); j++) {
    for (int i = 0; i < state_in.nc(); i++) {
        val = state_in[j][i];
        neb = sum_moore_nebbers(state_in, j, i);
        state_out[j][i] = (neb == 3 || neb == 2 && val != DEAD) ? ALIVE : DEAD;
    }
}

// Here we can set a deterministic pseudorandom initial state
// based on the value of the 'seed' parameter and a given rate
// Because dlib::rand uses a deterministic random generator
// (the Mersenne Twister), we can expect a given seed and rate
// to produce a unique state, on any platform we run on
void generate_state(Grid &state_in, string seed, float life_rate) {
    dlib::rand rand_gen;
    rand_gen.set_seed(seed);
    for (int i = 0; i < state_in.nr(); i++)
        for (int j = 0; j < state_in.nc(); j++)
            state_in[i][j] =
                (rand_gen.get_random_float() < life_rate)? ALIVE : DEAD;
}

// We save in the same format as before.
// 3 bytes: The ASCII letters "CEL"
// 4 bytes: The grid’s width, unsigned int
// 4 bytes: The grid’s height, unsigned int
// w*h*4 bytes: Row major grid values
void save_state_to_file(Grid &flip, string fname) {
    FILE *fp;
    char buf[4] = "CEL";
    int width = flip.nc(), height = flip.nr();

    fp = fopen(fname.c_str(), "w");
    if (fp == 0)
        throw new exception("Could not save file");
    fwrite(buf, sizeof(char), 3, fp);
    fwrite(&width, sizeof(uint), 1, fp);
    fwrite(&height, sizeof(uint), 1, fp);
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            fwrite(&flip[i][j], sizeof(cell), 1, fp);
    fclose(fp);
}

// Check the CEL code, set values of width and height
// Notice the set_size() call- easier than malloc’ing
// like we needed to in the C version
void load_state_from_file(Grid &flip, uint& width, uint& height, string fname) {
FILE *fp;
char buf[4];

fp = fopen(fname.c_str(), "r");
if (fp == NULL)
    throw new exception("Could not load file");
fwrite((void*)buf, sizeof(char), 3, fp);
    throw new exception("Loaded Invalid CEL file");
fwrite(&width, sizeof(uint), 1, fp);
fwrite(&height, sizeof(uint), 1, fp);
flip.set_size(height, width);
for (uint i = 0; i < height; i++)
    for (uint j = 0; j < width; j++)
        fread(&flip[i][j], sizeof(cell), 1, fp);
fclose(fp);
}

// Given an input cellular automaton, compute one portion of it
// starting at row_start and continuing through row_end-1
void next_state_partial(Grid &state_in, Grid &state_out,
    uint row_start, uint row_end)
{
    uint neb, val;
    for (uint j = row_start; j < row_end; j++) {
        for (int i = 0; i < state_in.nc(); i++) {
            val = state_in[j][i];
            neb = sum_moore_nebbers(state_in, j, i);
            state_out[j][i] = cell((neb == 3 || neb == 2 && val != DEAD) ?
                ALIVE : DEAD);
        }
    }
}

// Given an input cellular automaton grid, run a GPU kernel
// with OpenCL
void gpu_run_for_n_states(Grid const &in_img, Grid &out_img, uint run_count) {
    const char* src_conway =
"__kernel void conway_game( 
    __global int* in, 
    __global int* out, 
    int rows, int cols) 
{
    int left, right, top, bot, row, col, val;
    row = get_global_id(0);
    col = get_global_id(1);
    // Bounds check
    if (row >= rows || col >= cols)
    {
";
return;

left = (col-1); if (left < 0) {left += cols;}
right = (col+1); if (right == cols) {right = 0;}
top = (row-1); if (top < 0) {top += rows;}
bot = (row+1); if (bot == rows) {bot = 0;}
top *= cols;
row *= cols;
bot *= cols;
int total = 0;
total += (in[top + left] > 0);
total += (in[top + col] > 0);
total += (in[top + right] > 0);
total += (in[row + left] > 0);
total += (in[row + right] > 0);
total += (in[bot + left] > 0);
total += (in[bot + col] > 0);
total += (in[bot + right] > 0);
val = in[row + col];
val = 100*(total == 3 || (total == 2 && val > 0));
// copy the array
out[row + col] = val;

// Number of rows (img height), number of columns (width)
int nr = in_img.nr();
int nc = in_img.nc();

// Images in dlib array2d are stored row-major with row padding.
// This gives us the total length in memory of each row, so we can
// later ignore the padding when transferring to the GPU
int row_step = in_img.width_step();

cl_int errcode; // C-style error code
cl_platform_id platform_id; // Platform specifies the vendor
cl_device_id device_id; // Device specifies the card
cl_context context; // A Context manages all devices
cl_command_queue comm_q; // Commands are queued to each device
cl_program prog_conway; // An OpenCL kernel source prog
cl_uint num_platforms, num_devices;
char device_name[256];

// Fetch the Platform and Device IDs
errcode = clGetPlatformIDs(1, &platform_id, &num_platforms);
errcode = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &num_devices);

// Get the name of the GPU- eg. my amd 5770 is a 'Juniper'
errcode = clGetDeviceInfo(device_id, CL_DEVICE_NAME, 256, device_name, NULL);
// Other info that might be useful for dividing the work
// clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, ...  
// clGetDeviceInfo(device_id, CL_DEVICE_MAX_CLOCK_FREQUENCY, ...  
// clGetDeviceInfo(device_id, CL_DEVICE_GLOBAL_MEM_SIZE, ...)  

// cl_context_properties is a zero-terminated list of pairs  
// where each pair is (property name, value)  
// Note that nVidia's OpenCL requires CL_CONTEXT_PLATFORM  
cl_context_properties properties[] =  
    {CL_CONTEXT_PLATFORM, (cl_context_properties)platform_id, 0};

// We make a context that manages one device: the device_id  
// that we picked up earlier. If we want to use multiple GPU’s,  
// we would use an array of device_id’s instead of one  
context = clCreateContext(  
    properties, // List containing one property, the platform id  
    1, // Just 1 device  
    &device_id, // One-element array of device ID’s  
    NULL, // Optional error handler callback function  
    NULL, // Ptr to data to pass to the error handler  
    &errcode);

// Commands are issued per-device in a (usually) non-blocking queue  
comm_q = clCreateCommandQueue(  
    context, // For a single GPU, this is the only queue  
    device_id, // The single device we want to play with  
    0, // Properties list, like context_properties  
    &errcode);

cout << "Created CL command queue for device '" << device_name << "'");

// Compile our openCL kernel  
prog_conway = clCreateProgramWithSource(  
    context, // We use the context for everything  
    1, // Our program is stored in 1 string  
    &src_conway, // So, 1-element array of source strings  
    0, // Optional array of sizes of source strings  
    &errcode);

cout << "Loaded kernel source with status " << errcode << endl;

// Now we build our compiled openCL kernel  
errcode = clBuildProgram(  
    prog_conway, // Compiled program object  
    0, // Number of devices to build this for  
    NULL, // Array of device ID’s (defaults to 'all devices')  
    NULL, // Ptr to string of build options  
    // Try "-cl-mad-enable" for faster mult-adds  
    NULL, // Callback fn to be run when the build is complete  
    // If specified, this fn is non-blocking
// Argument to pass to the callback

if (errcode != CL_SUCCESS)
    cout << "Error " << errcode << " while building OpenCL kernel!" << endl;

// create kernel
cl_kernel kernel = clCreateKernel(
    prog_conway, // The built kernel program
    "conway_game", // Name of the kernel function
    &errcode);

cout << "Image is stored in memory as a " << nc << "x" << nr
    << " with " << row_step-nc << " padding" << endl;

// Allocate two image-sized buffers in GPU memory
cl_mem a_mem_obj = clCreateBuffer(
    context,
    CL_MEM_READ_WRITE, // This mem is read-only to the kernel
    nr * nc * sizeof(data_t), // Buffer Size
    NULL, // Optionally fill the buffer with BufferSize bytes
        // copied from this pointer. (we'll do that ourself)
    &errcode);

cl_mem b_mem_obj = clCreateBuffer(
    context,
    CL_MEM_READ_WRITE, // This is a write-only output buffer
    nr * nc * sizeof(data_t), // Same size
    NULL, // No need to fill it
    &errcode);

if (errcode != CL_SUCCESS)
    cout << "Error " << errcode << " while allocating buffers" << endl;

// Copy the image into buffer A on the GPU
// Note that in host mem, the image is stored with padding
// We need to ignore the extra (row_step-nc) padding that
// comes after each row of the image when we copy
size_t buffer_origin[3] = {0, 0, 0};
size_t host_origin[3] = {0, 0, 0};
size_t read_rect[3] = {nc*sizeof(data_t), nr, 1};

void *data_in = (void*)&in_img[0][0];

// Copy nc floats from each row, where rows are row_step apart
errcode = clEnqueueWriteBufferRect(
    comm_q,
    a_mem_obj, // 'a' stores the input to the kernel
    CL_TRUE, // Blocking? For now we will block here
    buffer_origin, // Start (x,y,z) write location
    host_origin, // Start (x,y,z) read location
    read_rect, // (width,height,depth) region to copy
    nc * sizeof(data_t), // GPU (dest) row width
    ...)
0, // GPU (dest) total size
cn * sizeof(data_t), // Host (src) row width
0, // Host (src) total size
data_in,
0, // Size of the event-wait-list
NULL, // Wait list array; delay execution until
// every event in this array has completed
NULL); // Return an event object for this copy
if (errcode != CL_SUCCESS)
  cout << "Error " << errcode << " while writing to GPU" << endl;

// Each pixel (nr*nc) is its own task. Divide the tasks into
// workgroups of 10x10 pixels each
size_t localWorkSize[2], globalWorkSize[2];
localWorkSize[0] = 10;
localWorkSize[1] = 10;
globalWorkSize[0] = nr;
globalWorkSize[1] = nc;

// Repeat the
void *buffer_a = (void *)&a_mem_obj;
void *buffer_b = (void *)&b_mem_obj;
while (run_count--) {
  // Argument vector to the kernel
  clSetKernelArg(kernel, 0, sizeof(cl_mem), buffer_a);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), buffer_b);
  clSetKernelArg(kernel, 2, sizeof(int), (void*)&nr);
  clSetKernelArg(kernel, 3, sizeof(int), (void*)&nc);

  // create N-D range object with work-item dimensions and execute kernel
  errcode = clEnqueueNDRangeKernel(
    comm_q,
    kernel,
    2,
    NULL, globalWorkSize,
    localWorkSize,
    0, NULL, NULL);
  if (errcode != CL_SUCCESS)
    cout << "Error " << errcode << " while running kernel" << endl;

  // Now reverse it, copy the next iteration from flop to flip
  swap(buffer_a, buffer_b);
}

// Woo! Now we need to copy the result back into CPU land
// This time, we copy a contiguous GPU memory chunk into
// a row-padded CPU memory array
void* data_out = (void*)&out_img[0][0];
errcode = clEnqueueReadBufferRect(comm_q,
b_mem_obj,
CL_TRUE,
buffer_origin,
host_origin,
read_rect,
nc * sizeof(data_t), // GPU (src) row width
0, // GPU (src) total size (default)
nc * sizeof(data_t), // Host (dest) row width
0, // Host (dest) total size
data_out, // Write to here
0, NULL, // Wait list array;
NULL);
if (errcode != CL_SUCCESS)
cout << "Error " << errcode << " while reading from GPU" << endl;

cout << "Copied result back to memory, status " << errcode << endl;
// Now the data has been copied back from the GPU!
}

// Parse the command line arguments, and create two Grid type buffers
// Then load a given file name, or generate a state based on user params
// Finally, run for the given number of iterations and save a file
// For interactive mode, display images instead of saving
int main(int argc, char **argv)
{
    string load_file = "";
    string save_file = "";
    string random_seed = "1";
    float life_perct = 0.5f;
    uint width = 20;
    uint height = 12;
    uint run_time = 0;
    uint num_threads = 0;

    cl_parser par;
    par.add_option("l","Begin by loading the initial state from given file.", 1);
    par.add_option("s","After running, save resulting state to given file.", 1);
    par.add_option("r","Begin by generating a random state with the given seed", 1);
    par.add_option("p","Set the percentage of live cells randomly generated", 1);
    par.add_option("w","Set the width of the initial random state to.", 1);
    par.add_option("h","Set the height of the initial random state", 1);
    par.add_option("t","Run the Game for this many time steps before ending", 1);
    par.add_option("n","Run in multithread mode with this many threads", 1);
    par.add_option("g","Run as a GPU kernel using OpenCL");
    par.add_option("d","Display the final automaton result in a window");
    par.add_option("?", "Help");
    par.parse(argc,argv);

    if (par.option("l"))
        load_file = par.option("l").argument();
if (par.option("s"))
    save_file = par.option("s").argument();
if (par.option("r"))
    random_seed = par.option("r").argument();
if (par.option("p"))
    life_perct = 0.01f * dlib::string_cast<float>(par.option("p").argument());
if (par.option("w"))
    width = dlib::string_cast<uint>(par.option("w").argument());
if (par.option("h"))
    height = dlib::string_cast<uint>(par.option("h").argument());
if (par.option("t"))
    run_time = dlib::string_cast<uint>(par.option("t").argument());
if (par.option("n"))
    num_threads = dlib::string_cast<uint>(par.option("n").argument());

// Display help if -? is input
if (par.option("?"))
{
    cout << "Conway’s Game of Life Emulator" << endl<< "Runs the Game of Life cellular automaton. Each cell is represented as a" << endl<< "Boolean variable, 1 for alive and 0 for dead, in a 2D grid. The program" << endl<< "follows the rules listed at en.wikipedia.org/wiki/Conway's_Game_of_Life" << endl<< "and cells wrap around right-to-left and bottom-to-top.";
    // This function prints out a nicely formatted list of
    // all the options the parser has
    par.print_options(cout);
    cout << endl;
    return 0;
}

try {
    // We use two buffers and alternate between them.
    Grid flip(height, width), flop(height, width);

    // If no file is to be loaded, generate a state from seed
    if (load_file.empty())
        generate_state(flip, random_seed, life_perct);
    else {
        load_state_from_file(flip, width, height, load_file);
        flop.set_size(height, width);
    }

    // For batch mode, we support three methods of execution:
    // -A single-threaded mode, same as the C implementation
    // -A multithreaded CPU mode using dlib’s pthreads wrapper
    // -A GPGPU implementation using OpenCL
    if (run_time != 0) {
        // The GPGPU version has large overhead, so the entire run is done in
        // the gpu_run function. Swapping is performed onboard the GPU
        if (par.option("g")) {
gpu_run_for_n_states(flip, flop, run_time);
}
// The threaded version creates several threads, and waits to join after
// each iteration of the Game.
else if (num_threads > 0) {
    dlib::thread_pool tp(num_threads);
    while (run_time-- > 0) {
        for (uint i = 0; i < num_threads; i++) {
            tp.add_task_by_value([&flip, &flop, height, num_threads, i]() {
                next_state_partial(
                    flip, flop,
                    (height * i) / num_threads,
                    (height * (i + 1)) / num_threads
                );
            });
        }
        tp.wait_for_all_tasks();
        dlib::swap(flip, flop);
    }
}
// The single-threaded version runs on only one core
else {
    while (run_time-- > 0) {
        next_state(flip, flop);
        dlib::swap(flip, flop);
    }
}
// After running, optionally display windows
if (par.option("d")) {
    image_window img_disp(flip);
    img_disp.set_title("Original Automaton State");
    image_window img_disp2(flop);
    img_disp2.set_title("Final Automaton State");
    img_disp.wait_until_closed();
}
// Otherwise, we run in interactive mode
else {
    cout << "Conway’s Game of Life simulation: "
         << "Press Enter to Continue, q to quit" << endl;

dlib::image_window img_disp(flip);
img_disp.set_title("Conway’s Game of Life");
img_disp.set_size(400, 400);

while (true) {
    if (getchar() == 'q') break;
    next_state(flip, flop);
    dlib::swap(flip, flop);
    img_disp.set_image(flip);
}
// Save the file if the user specified so
if (!save_file.empty())
    save_state_to_file(flip, save_file);

catch (exception& e)
{
    cerr << "Exception Thrown: " << e.what() << endl;
}

return 0;