# CSCI-4320/6360 - Assignment 2: Parallel HighLife Using 1-D Arrays in CUDA

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180-3590

January 25, 2024

**DUE DATE: 11:59 p.m., Friday, February 9th, 2024**

## 1 Overview

For this assignment, you are to construct a parallel C program that specifically implements *HighLife* (modification to the Game of Life) but with the twist of using ****only**** 1-D arrays in CUDA. Additionally, you will run this C-program on the *AiMOS* supercomputer at the CCI in parallel using a single CUDA enabled GPU.

Don't worry however, version of *HighLife* in serial C code will be provide as a template on which to base you parallel CUDA implementation.

**Note, this is an individual assignment and is not a group assigment.**

### 1.1 Review of HighLife Specification

The *HighLife* is an example of a Cellular Automata where universe is a two-dimensional orthogonal grid of square cells (with WRAP AROUND FOR THIS ASSIGNMENT), each of which is in one of two possible states, *ALIVE* or *DEAD*. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur at each and every cell:

- Any LIVE cell with FEWER THAN two (2) LIVE neighbors DIES, as if caused by under-population.
- Any LIVE cell with two (2) OR three (3) LIVE neighbors LIVES on to the next generation.
- Any LIVE cell with MORE THAN three (3) LIVE neighbors DIES, as if by over-population.
- Any DEAD cell with EXACTLY three (3) or six (6) LIVE neighbors becomes a LIVE cell, as if by reproduction/birth. *Note, the addition of the six LIVE neighbors in this rule make HighLife different from the original Game of Life.*

The world size and initial pattern are determined by an arguments to your program. A template for your program will be provide and more details are below. The first generation is created by applying the above rules to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a "tick" or iteration. The rules continue to be applied repeatedly to create further generations. The number of generations will also be an argument to your program. Note, an iteration starts with $Cell(0, 0)$ and ends with $Cell(N - 1, N - 1)$ in the serial case.

## 1.2 C-template for Assignment 2

The world size and initial pattern are determined by an arguments to your program. The first generation is created by applying the above rules to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a "tick" or iteration. The rules continue to be applied repeatedly to create further generations. The number of generations will also be an argument to your program.

## 1.3 Template and Implementation Details

Extend the provided template, *highlife.c*, to a CUDA parallel implementation in *highlife.cu* which supports the processing of the world gid. Here is what you should look out for:

- **new headers:**

```
#include <cuda.h>
#include <cuda_runtime.h>
```

- **init routines:** Replace all the `calloc` calls with `cudaMallocManaged`. For example:
  `cudaMallocManaged( &g_data, (g_dataLength * sizeof(unsigned char)));`
  will allocate the right amount of space for the `g_data` pointer in the template. Also, make sure you correctly ZERO out all the data elements as `cudaMallocManaged` will not does this for you like `calloc` does.

- **HL_kernelLaunch**: this function is called from `main` and looks like:

```
bool HL_kernelLaunch(unsigned char** d_data,
                     unsigned char** d_resultData,
                     size_t worldWidth,
                     size_t worldHeight,
                     size_t iterationsCount,
                     ushort threadsCount)
```

  This function computes the world via a CUDA kernel (described below) and swaps the new world with the previous world to be ready for next iteration. It should invoke the **HL_kernel** CUDA kernel function which will advance the whole world one step or iteration. Last, it will need to invoke `cudaDeviceSynchronize()` between iterations before `HL_swap` and prior to returning to the `main` routine.

- **HL_kernel**: is the main CUDA kernel function. This function will look like:

```
__global__ void HL_kernel(const unsigned char* d_data,
                          unsigned int worldWidth,
                          unsigned int worldHeight,
                          unsigned char* d_resultData)
```

  This function will iterate starting with:

  `index = blockIdx.x *blockDim.x + threadIdx.x`

  and increment by

  `index += blockDim.x * gridDim.x`

  provided

  `index < worldWidth*worldHeight`

  .

  Using current thread `index` you'll need to compute the `x0, x1, x2, y0, y1` and `y2` offsets as the serial template does using the `worldWidth` and `worldSize` variables.

  **Note, you don't need to use any shared memory in your implementation. That's an advanced optization you could consider as part of a Cellular Automata group project if interested.**

  Once you have those offset values, count the number of alive cells among the 8 neighbors in the worlds.

To compile and execute the `highlife` program, do:

- modules: `module load xl_r spectrum-mpi cuda`
- compile: `nvcc -g -G -gencode arch=compute_70,code=sm_70 highlife.cu -o highlife` – this will invoke the GCC compiler with debug and all warnings turned on. For performance runs, remove the `-g` and `-G` options and replace with `-O3`.
- Allocate one compute-node: See `salloc` command below in the "Running on AiMOS" section. You may also use `sbatch` an execute a series of runs a job.

For this assignment, there are the following 5 patterns:

- Pattern 0: World is ALL zeros.
- Pattern 1: World is ALL ones.
- Pattern 2: Streak of 10 ones in about the middle of the World.
- Pattern 3: Ones at the corners of the World.
- Pattern 4: "Spinner" pattern at corners of the World.
- Pattern 5: Replicator pattern starting in the middle.

# 2 Running on AiMOS

Please follow the steps below:

1. Login to CCI landing pad (`blp01.ccni.rpi.edu`) using SSH and your CCI account and PIC/Token/password information. For example, `ssh SPNRcaro@blp03.ccni.rpi.edu`.
2. Login to *AiMOS* front end by doing `ssh PCPEyourlogin@dcsfen01`.
3. (Do one time only if you did not do for Assignment 1). Setup ssh-keys for password-less login between compute nodes via `ssh-keygen -t rsa` and then `cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys`.
4. Load modules: run the `module load xl_r spectrum-mpi cuda` command. This puts the correct GNU C compiler along with MPI (not used) and CUDA in your path correctly as well as all needed libraries, etc.
5. Compile code on front end by issuing the `nvcc -g -G -gencode arch=compute_70,code=sm_70 highlife.cu -o highlife` for debug or `nvcc -O3 -gencode arch=compute_70,code=sm_70 highlife.cu -o highlife` for optimized code/performance.
6. Get a single node allocation by issuing: `salloc -N 1 --partition=el8-rpi --gres=gpu:4 -t 30` which will allocate a single compute node using 4 GPUs for 30 mins. The max time for the class is 30 mins per job. Your `salloc` command will return once you've been granted a node. Normally, it's been immediate but if the system is full of jobs you may have to wait for some period of time.
7. *Note, that the* `el8-rpi` *compute nodes only have GPUs with 16GB of RAM vs. the other larger* `el8` *partition which has GPUs with 32 GB. Those higher GPU memory compute nodes maybe needed to complete the 64Kx64K grid runs depending on your implementation.*
8. Use the `squeue` to find the `dcsXYZ` node name (e.g., `dcs24`).
9. SSH to that compute node, for example, `ssh dcs24`. You should be at a Linux prompt on that compute node.
10. Issue run command for HighLife. For example, `./highlife 4 64 2 32` which will run *High-Life* using pattern 4 with a world size of 64x64 for 2 iterations using a 32 thread blocksize.
11. If you are done with a node early, please `exit` the node and cancel the job with `scancel JOBID` where the JOBID can be found using the `squeue` command.

# 3 Parallel Performance Analysis and Report

First, make sure you disable any "world" output from your program. The worlds produced in these performance test will be too large for effective human analysis without additional post-processing of the data. Using the `time` command, execute your program across the following configurations and collect the total execution time for each run.

**Use the "Replicator" pattern #5 for all runs below.**

- Blocksize is 8 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.
- Blocksize is 16 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.
- Blocksize is 32 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.
- Blocksize is 128 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.
- Blocksize is 512 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.
- Blocksize is 1024 threads. Execute over world height/width of 1024, 2048, 4096, 8192, 16384, 32786 and 65536 for 1024 iterations each.

For each world height/width, plot the execution time in seconds (y-axis) of each different thread blocksize (x-axis). Note you will have 7 plots. Determine for each world size, what thread configuration yields the fastest execution time. Why do you think that is the case? What did you notice when the blocksize was less than or equal-to the CUDA hardware "warp size" of 32 threads?

Use the `time` command to support your reasoning. Include the output from an `time ./highlife <args>` run in your report.

Next, determine which thread and worldsize configuration yields the fastest "cells updates per second" rate. For example, a world of $1024^2$ that runs for 1024 iterations will have $1024^3$ cell updates. So the "cell updates per second" is $1024^3$ divided by the total execution time in seconds for that configuration. For this, please create a table of your data - there will be 42 entries - one for each thread blocksize / worldsize configuration with it's corresponding "cell updates per second" rate.

For the fastest "cells updates per second" rate configuration, how much faster (or slower) is it compared to the C-code serial version of *highlife.c* using just the POWER9 processor (one core only).

Last, please provide instructions on how to run your `highlife` program if it differs from using 3 arguments. For example, you might find it easier to run all your jobs if you provide the thread blocksize as a 4th argument and modify the `argc, argv` processing code in `main` accordingly.

You can use any graphing software so long as your report is in a PDF file. Please make sure to label each figure and put a proper caption that describes what the figure/plot is.

# 4 HAND-IN and GRADING INSTRUCTIONS

Please submit your CUDA/C-code `highlife.cu` and PDF report (yes it must be PDF format) with performance plots to the `submitty.cs.rpi.edu` grading system. All grading will be done manually because Submitty currently does not support GPU programs. A rubric will be posted which describes the grading elements of the both the program and report in Submitty. Also, please make sure you document the code you write for this assignment. That is, say what you are doing and why.