CSCI-4320/6360 - Assignment 3: Hybrid Parallel *HighLife* Using 1-D Arrays in CUDA and MPI

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

February 26, 2021

DUE DATE: 11:59 p.m., Friday, March 12th, 2021

1 Overview

For Assignment 3, you are to extend your CUDA implementation of *HighLife* that uses ***only*** 1-D arrays to running across multiple GPUs and compute nodes using MPI. You will run your hybrid parallel CUDA/MPI C-program on the *AiMOS* supercomputer at the CCI in parallel using at most 2 compute nodes for a total of 12 GPUs.

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.

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 Implementation Details

For the MPI parallelization approach, each MPI Rank will perform an even "chunk" of rows for the Cellular Automata universe. Using our existing program, this means that each MPI rank's sub-world will be **stack on-top of each other**. For example, suppose you have a 1024x1024 sub-world for each MPI rank to process, each Rank will have 1024x1024 cells to compute. Thus, Rank 0, will compute rows 0 to 1023, Rank 1 computes rows 1024 to 2047 and Rank 2 will compute rows 2048 to 3071 and so on. Inside of each rank, the *HighLife* CUDA kernel will be used to process each iteration.

For the Cellular Automata universe allocation each MPI Rank only needs to allocate it's specific chunk plus space for "ghost" rows at the MPI Rank boundaries. The "ghost" rows can be held outside of the main MPI Rank's Cellular Automata universe.

Now, you'll notice that rows at the boundaries of MPI Ranks need to be updated / exchanged prior to the start of computing each tick. For example, with a universe and 16 MPI Ranks example, MPI Rank 0 will need to send the state of row 0 (all 1024 entries) to MPI Rank 15. Additionally, Rank 15 will need to send row 1023 to Rank 0. Also, Rank 0 and Rank 1 will do a similar exchange.

For these row exchanges, you will use the `MPI_Isend` and `MPI_Irecv` messages. You are free to design your own approach to "tags" and how you use these routines except your design should not deadlock.

So the algorithm becomes:

```
main(...)
   {
      Setup MPI
      Set CUDA Device based on MPI rank.
      if Rank 0, start time with MPI_Wtime.
      Allocate My Rank's chunk of the universe per pattern and
         allocate space for "ghost" rows.

      for( i = 0; i < number of ticks; i++)
         {
            Exchange row data with MPI Ranks
               using MPI_Isend/Irecv.

            Do rest of universe update as done
               in assignment 2 using CUDA HighLife kernel.
           // note, no top-bottom wrap except for ranks 0 and N-1.
         }

      MPI_Barrier();
```

```
        if Rank 0,
           end time with MPI_Wtime and printf MPI_Wtime
           performance results;

        if (Output Argument is True)
           {
              Printf my Rank's chunk of universe to separate file.
           }

        MPI_Finalize();
     }
```

To init MPI in your `main` function do:

```
// MPI init stuff
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 MPI_Comm_size(MPI_COMM_WORLD, &numranks);
```

To init CUDA (after MPI has been initialized in `main`) in your "init master" function, do:

```
 if( (cE = cudaGetDeviceCount( &cudaDeviceCount)) != cudaSuccess )
   {
     printf(" Unable to determine cuda device count, error is %d, count is %d\n",
            cE, cudaDeviceCount );
     exit(-1);
   }

 if( (cE = cudaSetDevice( myrank % cudaDeviceCount )) != cudaSuccess )
   {
     printf(" Unable to have rank %d set to cuda device %d, error is %d \n",
            myrank, (myrank % cudaDeviceCount), cE);
     exit(-1);
   }
```

Note, `myrank` is this MPI rank value.

As noted in the pseudo-code comments, you will need to modify your CUDA *HighLife* kernel to account for the lack of top and bottom edge "world wrapping" for ranks other than rank 0 and rank N-1 where N is the number of ranks used in the parallel job. All ranks will need to support left and right edge "world wrapping".

Moreover, for this assignment, you'll need to modify some of the initialization 5 patterns per below:

- **Pattern 0**: World is ALL zeros. Nothing todo here.
- **Pattern 1**: World is ALL ones. Nothing todo here.

- **Pattern 2**: Streak of 10 ones at column 128 and appears at the last row of each MPI rank. Note the smallest configuration we would run is 1024x1024 for each MPI rank.
- **Pattern 3**: Ones at the corners of the World. Here, the corners are row 0 of MPI rank 0 and last row of the last MPI rank.
- **Pattern 4**: "Spinner" pattern at corners of the World. Solution is modify current function so that only Rank 0 does the init of their local world.
- **Pattern 5**: Replicator pattern starting in the middle. Nothing todo here.

2 Running on AiMOS

2.1 Building Hybrid MPI-CUDA Programs

You will need to break apart your code into two files. The `highlife-mpi.c` file contains all the MPI C code. The `highlife-cuda.cu` contains all the CUDA specific code including the world init routines. You'll need to make sure those routines are correctly "extern". Because `nvcc` is a C++ like compiler, you'll need to turn off name mangling for any kernel launch C functions via `extern ''C'' { function dec }` that are called from `highlife-mpi.c` and defined in `highlife-cuda` Next, create your own Makefile with the following:

```
all: highlife.c highlife-cuda.cu
        mpixlc g highlife-mpi.c -c -o highlife-mpi.o
        nvcc -g -G -arch=sm_70 highlife-cuda.cu -c -o highlife-cuda.o
        mpicc -g highlife-mpi.o highlife-cuda.o -o highlife-exe \
          -L/usr/local/cuda-10.2/lib64/ -lcudadevrt -lcudart -lstdc++
```

2.2 SLURM Submission Script

The create your own `slurmSpectrum.sh` batch run script with the following:

```
#!/bin/bash -x

if [ "x$SLURM_NPROCS" = "x" ]
then
  if [ "x$SLURM_NTASKS_PER_NODE" = "x" ]
  then
    SLURM_NTASKS_PER_NODE=1
  fi
  SLURM_NPROCS=‘expr $SLURM_JOB_NUM_NODES \* $SLURM_NTASKS_PER_NODE‘
else
  if [ "x$SLURM_NTASKS_PER_NODE" = "x" ]
  then
    SLURM_NTASKS_PER_NODE=‘expr $SLURM_NPROCS / $SLURM_JOB_NUM_NODES‘
  fi
fi
```

```
srun hostname -s | sort -u > /tmp/hosts.$SLURM_JOB_ID
awk "{ print \$0 \"-ib slots=$SLURM_NTASKS_PER_NODE\"; }"\
/tmp/hosts.$SLURM_JOB_ID >/tmp/tmp.$SLURM_JOB_ID
mv /tmp/tmp.$SLURM_JOB_ID /tmp/hosts.$SLURM_JOB_ID

module load xl_r spectrum-mpi cuda/10.2

mpirun -hostfile /tmp/hosts.$SLURM_JOB_ID -np $SLURM_NPROCS\
 /gpfs/u/home/SPNR/SPNRcaro/barn/PARALLEL-COMPUTING/HighLife-CUDA/highlife-with-cuda 5 1638

rm /tmp/hosts.$SLURM_JOB_ID
```

Next, 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 PCPAyourlogin@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/10.2` command. This puts the correct IBM XL compiler along with MPI and CUDA in your path correctly as well as all needed libraries, etc.
5. Compile code on front end per directions above.
6. Get a single node allocation by issuing: `salloc -N 1 --partition=rcs --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. Use the `squeue` to find the `dcsXYZ` node name (e.g., `dcs24`).
8. SSH to that compute node, for example, `ssh dcs24`. You should be at a Linux prompt on that compute node.
9. Issue run command for HighLife. For example, `./highlife 5 16384 128 256` which will run Highlife using pattern 5 with a world size of 16Kx16k for 128 iterations using a 256 thread blocksize.
10. 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.
11. Use the example Sbatch script above and `sbatch` command covered in lecture 9 to run across multiple nodes.

3 Parallel Performance Analysis and Report

First, make sure you disable any "world" output from your program to prevent extremely large output files. Note, the arguments are the same as used in Assignment 2. Using the `MPI_Wtime` command, execute your program across the following configurations and collect the total execution time for each run.

- 1 node, 1 GPU, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 1 node, 2 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 1 node, 3 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 1 node, 4 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 1 node, 5 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 1 node, 6 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.
- 2 nodes, 12 GPUs/MPI ranks, 16Kx16K world size each MPI rank, 128 iterations with 256 CUDA thread block size and pattern 5.

Determine your maximum speedup relative to using a single GPU and which configuration yields the fastest "cells updates per second" rate as done in Assignment 2. Explain why you think a particular configuration was faster than others.

4 HAND-IN and GRADING INSTRUCTIONS

Please submit your C-code and PDF report with performance data/table to the `submitty.cs.rpi.edu` grading system. All grading will be done manually because Submitty currently does not support GPU programs. We will test against a smaller world size (e.g., 32x32 per rank) for correctness. A rubric will be posted which describes the grading elements of 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.