CSCI-4320/6360 - Assignment 3: Hybrid Parallel Conway's Game of Life 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

March 3, 2020

DUE DATE: 11:59 p.m., Sunday, March 15th, 2020

1 Overview

As indicated in Lecture 11, you are to extend your CUDA implementation of Conway's *Game of Life* 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 For Review – Game of Life Specification

The Game of Life 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 live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by over-population.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Recall from 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 seedbirths 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.

1.2 Template and 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.

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.
         }

      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 initalized 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.
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 128 columns in 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.

2 Running on AiMOS

2.1 Building Hybrid MPI-CUDA Programs

You will need to break apart your code into two files. The `gol-main.c` file contains all the MPI C code. The `gol-with-cuda.cu` contains all the CUDA specific code including the world init routines. You'll need to make sure those routines are correctly "extern". Next, create your own Makefile with the following:

```
all: gol-main.c gol-with-cuda.cu
        mpicc -g gol-main.c -c -o gol-main.o
        nvcc -g -G -arch=sm_70 gol-with-cuda.cu -c -o gol-cuda.o
        mpicc -g gol-main.o gol-cuda.o -o gol-cuda-mpi-exe -L/usr/local/cuda-10.1/lib64/ -l
```

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.$SLURM_JOB_ID >/tmp/tm
mv /tmp/tmp.$SLURM_JOB_ID /tmp/hosts.$SLURM_JOB_ID

module load gcc/7.4.0/1
module load spectrum-mpi
module load cuda

mpirun -hostfile /tmp/hosts.$SLURM_JOB_ID -np $SLURM_NPROCS /gpfs/u/home/SPNR/SPNRcaro/Game

rm /tmp/hosts.$SLURM_JOB_ID
```

Next, please follow the steps below:

1. Login to CCI landing pad (`lp01.ccni.rpi.edu`) using SSH and your CCI account and password information. For example, `ssh SPNRcaro@lp03.ccni.rpi.edu` and at prompt type in password.
2. Login to *AiMOS* front end by doing `ssh PCP9yourlogin@dcsfen01`.
3. (Do one time only if you did not do for Assignment 1 or 2). Setup ssh-keys for passwordless 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 gcc/7.4.0/1 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 `make` command using previous described `Makefile`.
6. Get a single node job by issuing: `sbatch -N 1 ----ntasks-per-node=6 --gres=gpu:6 -t 30 ./slurmSpectrum.sh` which will create an MPI job on a single compute node with 6 MPI ranks using only 6 GPUs for 30 mins. Here, each MPI rank would use a separate CUDA device. The max time for the class is 30 mins per job.
7. Use the `squeue` to view the status of this job. When there is no job, it should be complete. The job output will be in `slurm-JOBID.out`.

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 with 1024 CUDA thread block size and pattern 3.
- 1 node, 2 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.
- 1 node, 3 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.
- 1 node, 4 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.
- 1 node, 5 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.
- 1 node, 6 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.

- 2 nodes, 12 GPUs/MPI ranks, 16Kx16K world size each MPI rank with 1024 CUDA thread block size and pattern 3.

Determine which 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. 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. 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.