Refactoring and Porting MASS CUDA Application Benchmarks

Omar Ahmed

CSS 497 Summer 2023 Term Report

Professor. Munehiro Fukuda

8/21/2023

Table of Contents

1. Introduction

1.1. Motivation

The motivation behind my work on both porting and refactoring benchmark applications for MASS CUDA is to get closer to implementing a set of standard applications on the CUDA version of the MASS library. This set of applications, which is also implemented on MASS C++ and Flame, is used to benchmark the performance and efficiency of the MASS CUDA library, and compare that to other implementation performances for agent parallelization.

During Summer quarter, I focused my efforts on correctly implementing the two MASS CUDA benchmark applications, Game of Life and The Social Network. The Game of Life benchmark application is an implementation of the mathematical John Conway's Game of Life, while The Social Network benchmark application implements a specification for a graph of people who are connected as friends and aims to optimally retrieve their connections. Please refer to sections 2.2 and 2.3 for a detailed explanation.

1.2. Project Goal

The goal of this project is to create standardized applications to be used in measuring the execution performance and programmability of the MASS CUDA library on a multi-GPU setup against the performance of other single-GPU, different architecture, or different library approaches. This project also serves as a metric for the usability of the MASS CUDA library itself for developers and to expose any library errors.

The completion of both Game of Life and Social Network will contribute to the complete and whole measurement of the MASS CUDA library when compared to FLAME GPU and MASS C++ as all agent-based parallelization libraries.

2. MASS CUDA Background

2.1. Context

The MASS, short for Multi-Agent Spatial Simulation, library is a parallel-computing library for multi-agent and spatial simulations over a cluster of computing nodes. The CUDA, short for Compute Unified Device Architecture, platform is a framework developed by Nvidia to expose the capabilities of GPU acceleration, specifically on Nvidia CUDA GPUs, and allow for intuitive and optimized general purpose GPU (GPGPU) programming.

The MASS CUDA library takes advantage of Nvidia CUDA's programming model, nvcc compiler, and parallelization to implement its Place and Agent based simulation model on the GPU cluster, Juno machines.

2.2. Previous Applications

The goal number of MASS CUDA benchmark applications is 7 applications, 3 of which have been implemented (Tuberculosis, Heat2D, BrainGrid, and SugarScape), while I have worked on implementing 2 of the remaining 4 applications.

The latest implemented benchmark application is the Tuberculosis application which is a complex simulation of the interaction between the immune system in the human lungs and Mycobacterium Tuberculosis. It applies the place-agent model to using two different agents and one place to simulate the environment.

The Heat2D benchmark application simulates the dispersion of heat across a metal sheet as a 2D grid starting from an initial point. It applies the place-agent model using a place to simulate each of the cells in the 2D grid but does not use any static/dynamic agent.

The BrainGrid benchmark application simulates a human neural network that generates axons and dendrites to form the network. It applies the place-agent model using a place to simulate each of the cells in the 2D grid but doesn’t involve any static/dynamic agent.

The SugarScape benchmark application simulates interactions of ants in a grid of different sugar mounds attempting to collect as much sugar as possible. It applies the place-agent model using dynamic ant agents that use agent migration to transfer places and uses a place to simulate a grid with sugar mounds.

2.3. Reasons for Selected Applications

The Game of Life benchmark application simulates the mathematical game by John Conway. The rules of the game are that any cell with fewer than two neighbors die of underpopulation, any cell with two or three neighbors survives, any cell with more than three neighbors dies of overpopulation, and any dead cell with exactly three neighbors becomes alive. This game is a cellular automaton with two states, dead or alive, that describes how an entity could be responsible for creating another of itself as John Von

Neumann stated, also how the evolution of the state of the game is solely dependent upon the initial state of creation. Since it is a zero-player game that can stretch to infinity, the game is terminated after a certain count of iterations has completed or the game evolved to contain no more living cells. In MASS CUDA, The Game of Life is simulated using an orthogonal grid of cells where each cell is represented by a place in the library. Each place/cell contains information about its neighboring cells' state and the game is iterated based on each place/cell calculating its own state using the states of neighboring cells. This iterative, yet finite, calculation of each cell's state is a computationally intensive but parallelizable problem and is why MASS CUDA's programming model has easily been applied to the Game of Life with success. The purpose of the game is to determine the state of the grid/environment after x generations, known as ticks, given a random initial state of the grid. The Game of Life application uses places for cells not any static/dynamic agents to compute the simulation state, but unlike other previously implemented applications, Game of Life is the only application simulating cellular automaton with a possibility of an infinitely long simulation.

The Social Network benchmark application simulates the connections/friends formed between a number of users in a social network and allows for retrieval of the Xth degree list of friends of a particular user. The users in the social network are represented using places and not static or dynamic agents. What distinguishes this application from the ones previously implemented is that the social network of friendships in the social network is a group of users connected to each other using a bidirectional graph through an adjacency list stored at each place. MASS CUDA's programming model applies greatly to this benchmark application because the computationally intensive and parallelize workload of the breadth-first search that is run when retrieving each user's xth degree friendships.

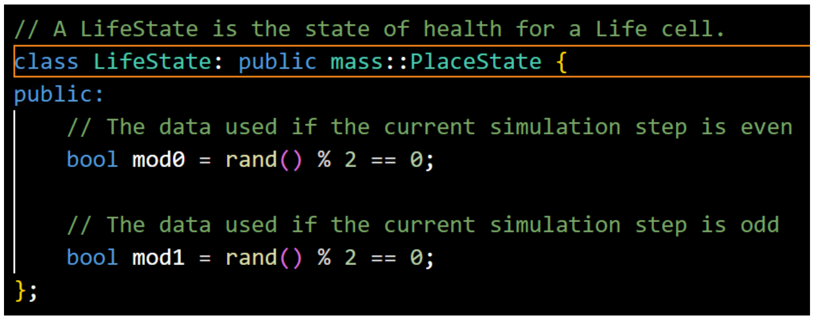
3. Implementation

3.1. Game of Life

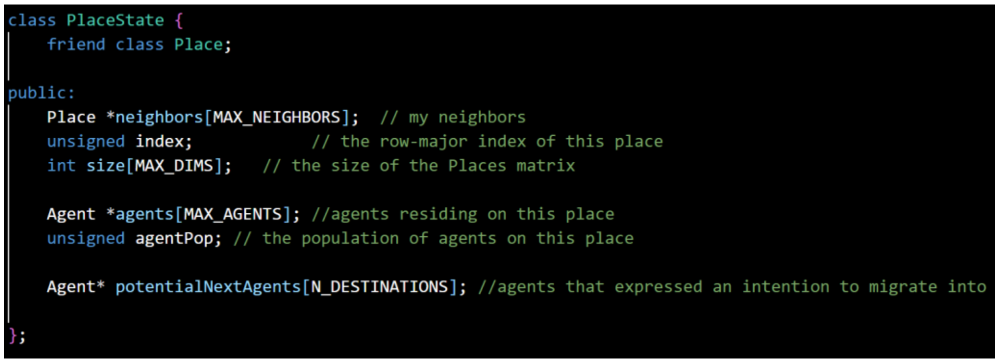
3.1.1. Life Cells as Places

Life State:

The Life class manages each of the grid places' state throughout the GoL simulation, including the health state of the cell and the functions/logic to compute the next state of the cell. The state of the cell is contained in 'LifeState.h' which includes an array of the 8-directional neighbors of the cell, a value 'mod0' containing the state of the cell if the simulation tick is even, and a value 'mod1' containing the state of the cell if the simulation tick is odd. Below is an excerpt of this:



```
// A LifeState is the state of health for a Life cell.
class LifeState: public mass::PlaceState {
public:
    // The data used if the current simulation step is even
    bool mod0 = rand() % 2 == 0;

    // The data used if the current simulation step is odd
    bool mod1 = rand() % 2 == 0;
};
```

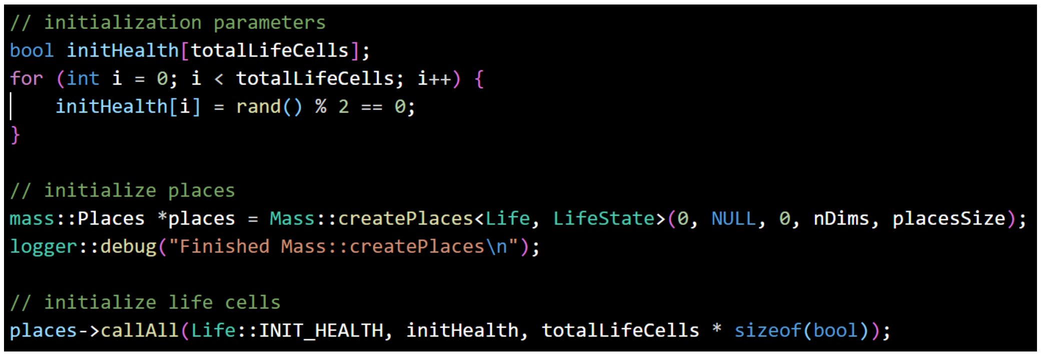


```
class PlaceState {
    friend class Place;

public:
    Place *neighbors[MAX_NEIGHBORS];  // my neighbors
    unsigned index;            // the row-major index of this place
    int size[MAX_DIMS];   // the size of the Places matrix

    Agent *agents[MAX_AGENTS]; //agents residing on this place
    unsigned agentPop; // the population of agents on this place

    Agent* potentialNextAgents[N_DESTINATIONS]; //agents that expressed an intention to migrate into

};
```

In comparison to MASS C++, this is a decoupling of the functionality of the Life class since state-related attributes of each Life cell in C++ stored as attributes of the Life class itself as a vector of cardinal neighbors and current state of cell. This difference in structure is due to a difference in global memory management by CUDA and how the state of each cell must be associated with a PlaceState pointer for CUDA to store and access.

Life Places initialization:

All places in the simulation representing Life classes are initialized using the following code:



```
// initialization parameters
bool initHealth[totalLifeCells];
for (int i = 0; i < totalLifeCells; i++) {
    initHealth[i] = rand() % 2 == 0;
}

// initialize places
mass::Places *places = Mass::createPlaces<Life, LifeState>(0, NULL, 0, nDims, placesSize);
logger::debug("Finished Mass::createPlaces\n");

// initialize life cells
places->callAll(Life::INIT_HEALTH, initHealth, totalLifeCells * sizeof(bool));
```

A randomization of the initial grid state happens on CPU host memory and then sent via callAll() to the GPU device memory for processing. The dimension of the grid is always set to two but the placesSize is size * size to be an n * n grid.

Life Place Functions:

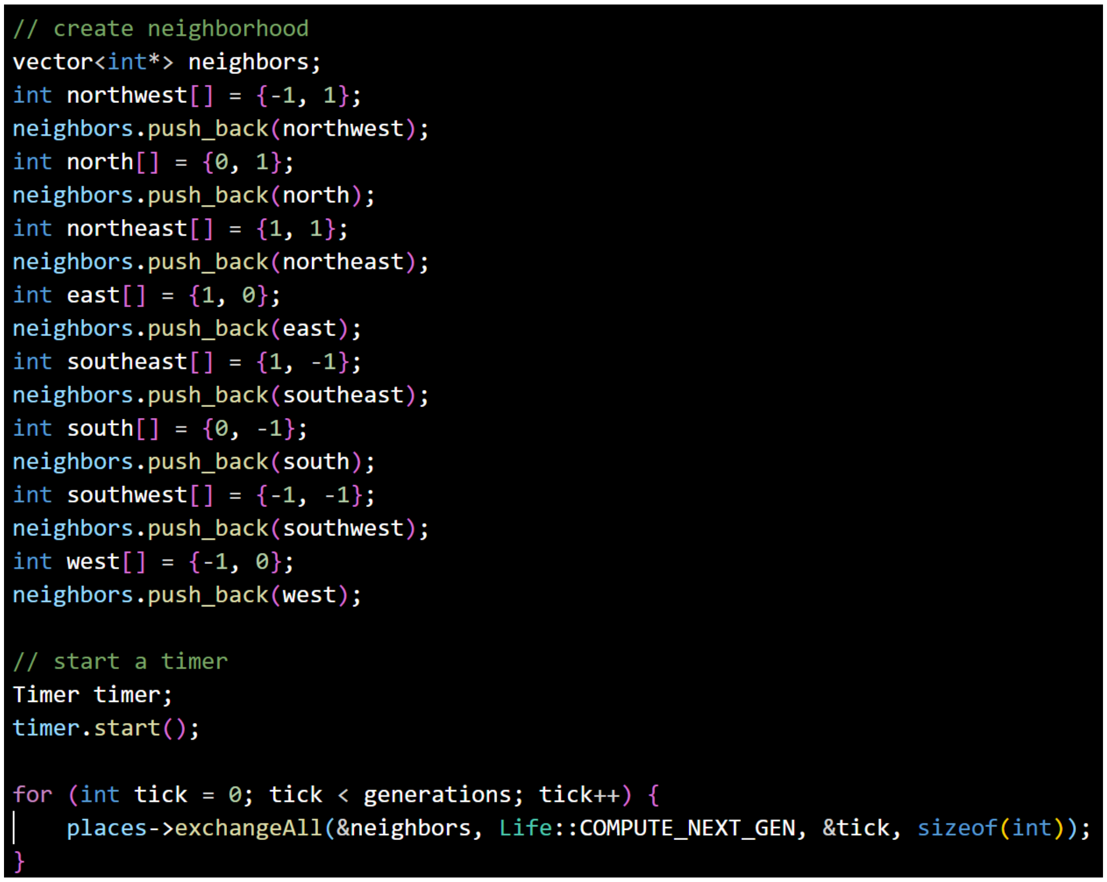
When the simulation is initialized and the randomized initial state of the simulation is determined, each place calls the ‘initalizeHealth’ function storing the initial health state in the mod0 attribute by retrieving its place index.

When the simulation is terminated and the desired nth state of the grid is achieved, the grid must be displayed by retrieving the health state of each place using a public member. This is done through the function ‘getHealth’ which returns the mod0 attribute.

When the entire grid state must be computed at each tick/iteration of the simulation, the ‘computeDeadOrAlive’ function is called upon by the exchangeAll() function to allow for each place to send message to each of its neighbors with its updated state then each place gets the most updated state by accessing each of its neighbors’ mod0/mod1 state and updating its own based on that.

3.1.2. Simulation

This implementation uses a vector of all cardinal neighbors which is created and allows for passing of health state from current place to neighbors through exchangeAll() for places. The simulation is terminated when the desired number of generations is achieved. See below for code:



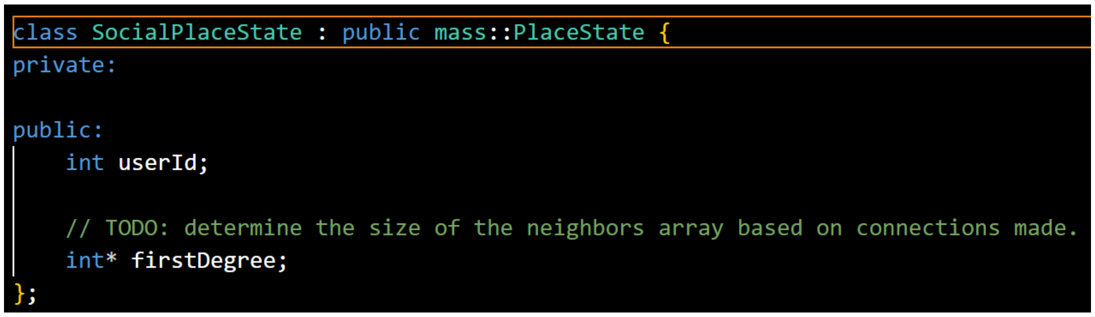
```
// create neighborhood
vector<int*> neighbors;
int northwest[] = {-1, 1};
neighbors.push_back(northwest);
int north[] = {0, 1};
neighbors.push_back(north);
int northeast[] = {1, 1};
neighbors.push_back(northeast);
int east[] = {1, 0};
neighbors.push_back(east);
int southeast[] = {1, -1};
neighbors.push_back(southeast);
int south[] = {0, -1};
neighbors.push_back(south);
int southwest[] = {-1, -1};
neighbors.push_back(southwest);
int west[] = {-1, 0};
neighbors.push_back(west);

// start a timer
Timer timer;
timer.start();

for (int tick = 0; tick < generations; tick++) {
    places->exchangeAll(&neighbors, Life::COMPUTE_NEXT_GEN, &tick, sizeof(int));
}
```

3.2. Social Network

3.2.1. SocialPlace as Places

SocialPlace State

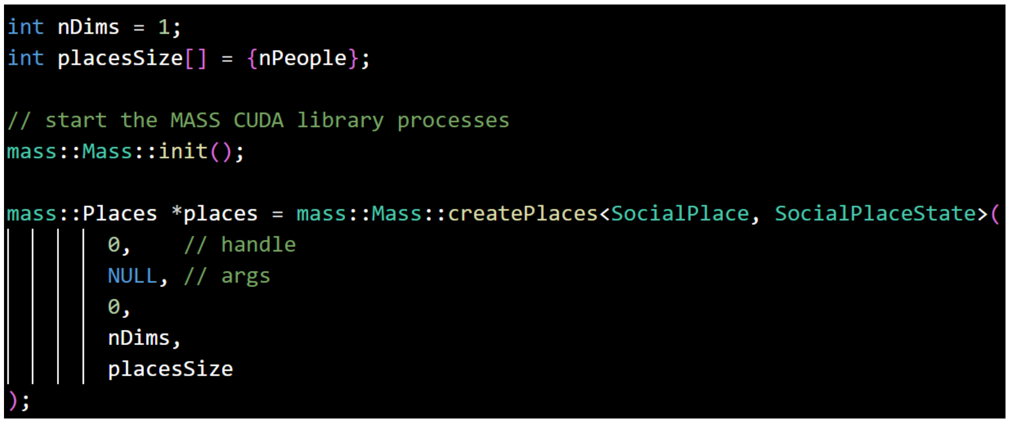
The SocialPlace class represents a user in the Social Network simulation and is responsible for tracking the user's first degree friends and computing their xth degree circles of friends. Since the Social Network application is modeled as a bidirectional graph that represents all friendships between different users, each user must form and adjacency list which includes its first degree friends. The 'SocialPlaceState.h' class consists of a userId to track the index of the place/user in the places array when instantiated as well as an integer array of the user's first degree friends' userIds. An excerpt of that code is shown below:



```
class SocialPlaceState : public mass::PlaceState {
private:

public:
    int userId;

    // TODO: determine the size of the neighbors array based on connections made.
    int* firstDegree;
};
```

SocialPlace Initialization:

SocialPlaces is meant to be a 1D array of SocialPlace classes that each contain a list of adjacencies. The following code shows how SocialPlaces is created:



```
int nDims = 1;
int placesSize[] = {nPeople};

// start the MASS CUDA library processes
mass::Mass::init();

mass::Places *places = mass::Mass::createPlaces<SocialPlace, SocialPlaceState>(
                0,    // handle
                NULL, // args
                0,
                nDims,
                placesSize
);
```

The basis of this adjacency list model is a 2D array that is as long as the number of users, represented as places, specified on host memory through user input and is as wide as the number of first-degree connections each user should have. Each user has k = n / m number of friends, where n is the number of users in the entire simulation and m is the fraction of the total group and this is done to create a k-regular bidirectional graph.

SocialPlace Functions:

When the simulation is initialized and a randomized k-regular graph is created, each of users the receives their list of first- degree friends which is passed through the ‘setFriends’ function. The callAll function is called to setFriends with an argument of a 2D array which is nPeople * nFriends big, then each user/SocialPlace retrieves the 1D array relevant to them.

When each place is prompted by its neighboring place to exchange their first-degree friend list using ‘exchangeAll’, each user returns their predefined friendship list using the ‘getFriends’ function.

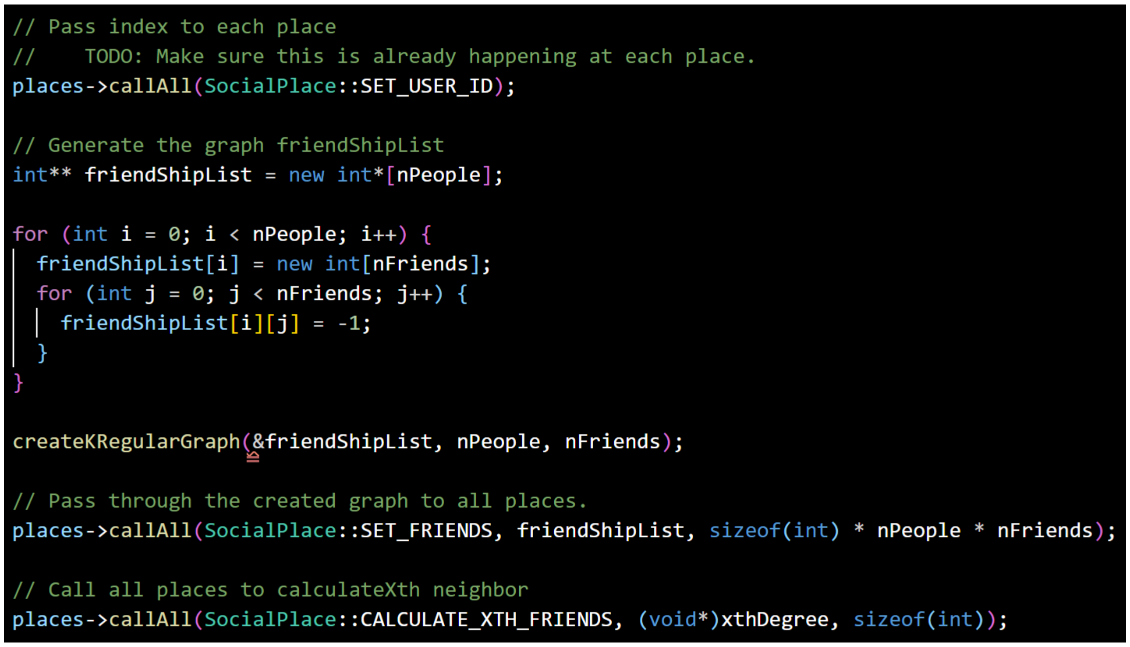
After all places are initialized, the ‘setUserId’ function is called to before calling the ‘setFriends’ function so each user has a populated index from their place in the Places array in the ‘userId’ property in SocialPlaceState.

The ‘getUserId’ function is used to retrieve the place’s index in the places array when needed for displaying results, etc.

When the simulation is initialized and each user/place has populated their first-degree friendship list, the simulation then uses the ‘calculateXthFriends’ function to retrieve each user’s xth degree friend by using the ‘exchangeAll’ function. The ‘calculateXthFriends’ function uses breadth-first search to traverse the graph and compile the friends at each degree up to the xth degree.

3.2.2. Simulation

The simulation starts with initializing each place with its index in the places array using the ‘callAll’ function. Then, the 2D array ‘friendShipList’ is initialized and randomly created to be a k-regular graph to then be passed to each place for populating their friends list. The simulation ends after iterating through the array ‘places’ and returning up to their xth degree friendships. An excerpt of the almost complete functionality is seen below:



```
// Pass index to each place
//    TODO: Make sure this is already happening at each place.
places->callAll(SocialPlace::SET_USER_ID);

// Generate the graph friendShipList
int** friendShipList = new int*[nPeople];

for (int i = 0; i < nPeople; i++) {
  friendShipList[i] = new int[nFriends];
  for (int j = 0; j < nFriends; j++) {
    friendShipList[i][j] = -1;
  }
}

createKRegularGraph(&friendShipList, nPeople, nFriends);

// Pass through the created graph to all places.
places->callAll(SocialPlace::SET_FRIENDS, friendShipList, sizeof(int) * nPeople * nFriends);

// Call all places to calculateXth neighbor
places->callAll(SocialPlace::CALCULATE_XTH_FRIENDS, (void*)xthDegree, sizeof(int));
```

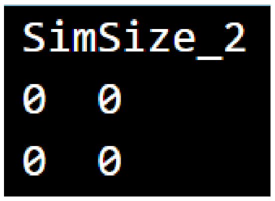
4. Verification

4.1. Game of Life

Game of Life Visualization

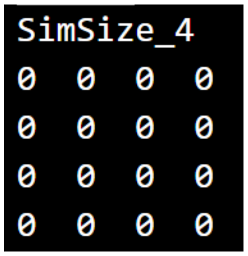
For the Game of Life simulation, there are 5 tests that run for 10 generations each and all start at a randomized state. These 5 unit tests are found in the file 'GameOfLife/GameOfLife_dev/test/main.cu' where the tests are executed with the command 'make test', 'make test' in this case does not receive any command line arguments to specify simulation size but is predetermined for each test when executed at simulation size 2, 4, 8, 16, 32. Below are all the visualized results of the 5 simulations run.

- For a simulation of size 2 (2 * 2)



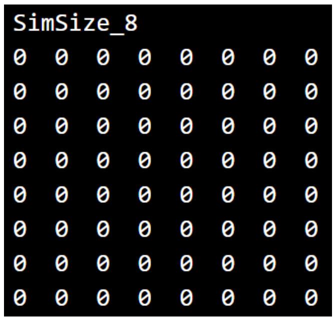
```
SimSize_2
0  0
0  0
```

- For a simulation of size 4:



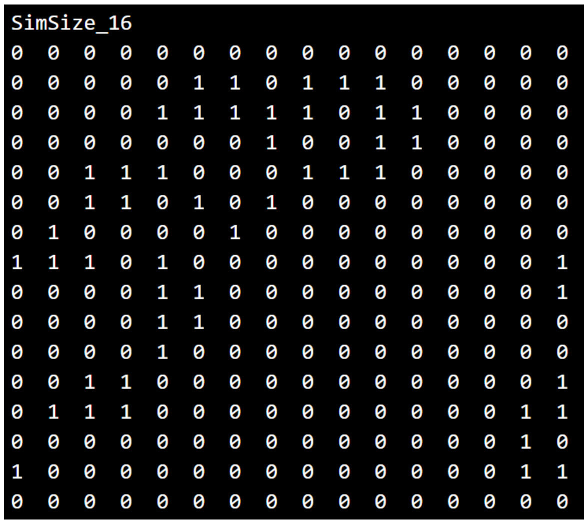
```
SimSize_4
0  0  0  0
0  0  0  0
0  0  0  0
0  0  0  0
```

- For a simulation of size 8:



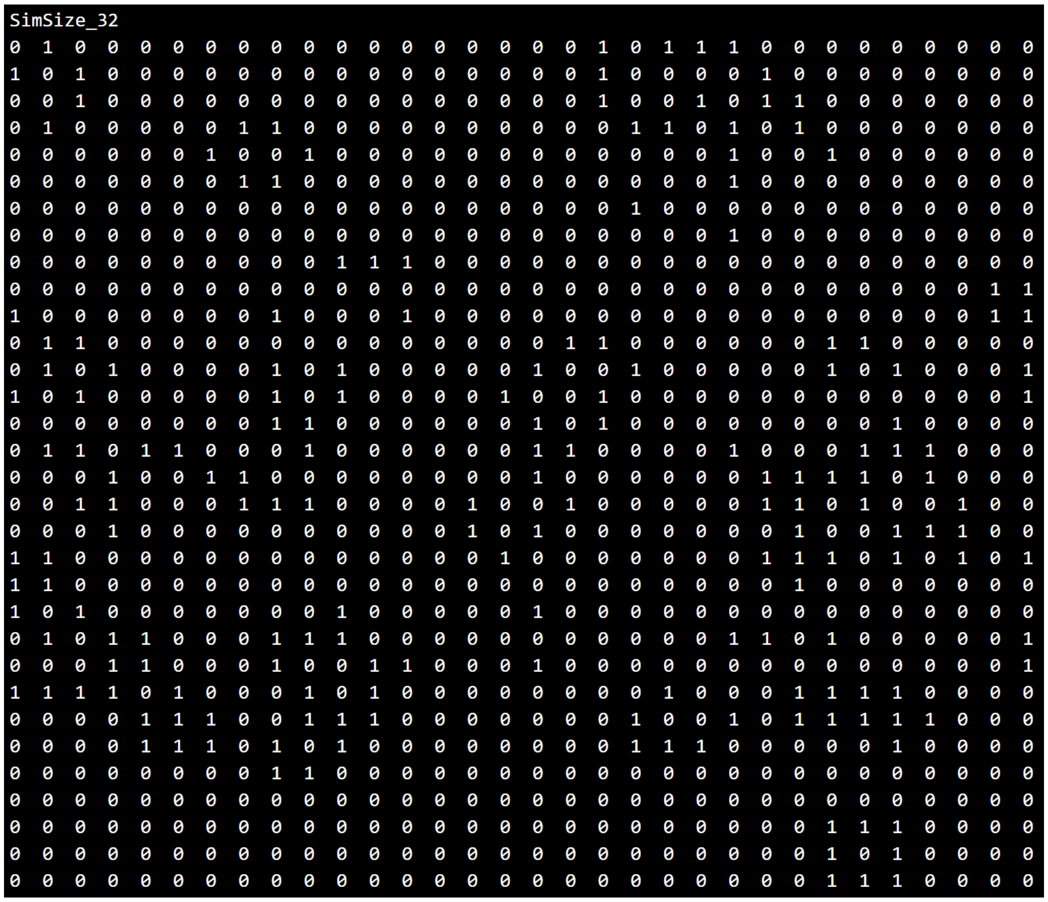
```
SimSize_8
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
```

- For a simulation of size 16:



```
SimSize_16
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  1  1  0  1  1  1  0  0  0  0  0
0  0  0  0  1  1  1  1  1  0  1  1  0  0  0  0
0  0  0  0  0  0  0  1  0  0  1  1  0  0  0  0
0  0  1  1  1  0  0  0  1  1  1  0  0  0  0  0
0  0  1  1  0  1  0  1  0  0  0  0  0  0  0  0
0  1  0  0  0  0  1  0  0  0  0  0  0  0  0  0
1  1  1  0  1  0  0  0  0  0  0  0  0  0  0  1
0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  1
0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0
0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  1
0  1  1  1  0  0  0  0  0  0  0  0  0  0  1  1
0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0
1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

- For a simulation of size 32:



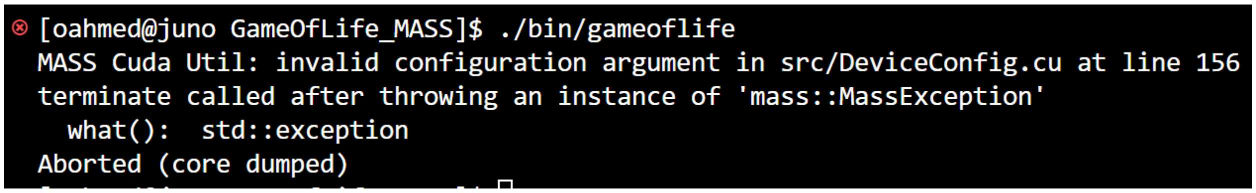
```
SimSize_32
0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  1  1  0  0  0  0  0  0  0  0  0
1  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  0  0
0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  1  1  0  0  0  0  0  0  0
0  1  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  1  1  0  1  0  1  0  0  0  0  0  0  0
0  0  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  1  0  0  0  0  0  0
0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
1  0  0  0  0  0  0  0  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1
0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  1  1  0  0  0  0  0
0  1  0  1  0  0  0  0  1  0  1  0  0  0  0  0  1  0  0  1  0  0  0  0  0  1  0  1  0  0  0  1
1  0  1  0  0  0  0  0  1  0  1  0  0  0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  1
0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  0  0  1  0  0  0  0
0  1  1  0  1  1  0  0  0  1  0  0  0  0  0  0  1  1  0  0  0  0  1  0  0  0  1  1  1  0  0  0
0  0  0  1  0  0  1  1  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  1  1  1  0  1  0  0  0
0  0  1  1  0  0  0  1  1  1  0  0  0  0  1  0  0  1  0  0  0  0  0  1  1  0  1  0  0  1  0  0
0  0  0  1  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0  0  0  0  1  0  0  1  1  1  0  0
1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1  1  1  0  1  0  1  0  1
1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0
1  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  1  0  1  1  0  0  0  1  1  1  0  0  0  0  0  0  0  0  0  0  0  1  1  0  1  0  0  0  0  0  1
0  0  0  1  1  0  0  0  1  0  0  1  1  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
1  1  1  1  0  1  0  0  0  1  0  1  0  0  0  0  0  0  0  0  1  0  0  0  1  1  1  1  0  0  0  0
0  0  0  0  1  1  1  0  0  1  1  1  0  0  0  0  0  0  0  1  0  0  1  0  1  1  1  1  1  0  0  0
0  0  0  0  1  1  1  0  1  0  1  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0  0  1  0  0  0  0
0  0  0  0  0  0  0  0  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  0  0  0  0
```

As observed by the displayed results of each of the 5 different sized simulations, all simulations of size 8 or less are all resolved to no living cells after only 10 generations/iterations. The complexity of simulations of size 16 and 32, however, is much higher and much more unpredictable.

Game of Life Results

Since the Game of Life benchmark application is not complex in implementation, without any agents or concurrency problems, the implementation and results for the application were fairly straightforward.

One minor bump in the road was a mis-implementation of the application using MASS CUDA without using the correct app template. Due to many directory, import, and segmentation issues, the original implementation would produce an invalid configuration argument error. Below is a snippet of the error:



```
[oahmed@juno GameOfLife_MASS]$ ./bin/gameoflife
MASS Cuda Util: invalid configuration argument in src/DeviceConfig.cu at line 156
terminate called after throwing an instance of 'mass::MassException'
  what():  std::exception
Aborted (core dumped)
```

I reported this issue to Brian, but found that an easier solution was the restructuring of the Game of Life application all together.

4.2. Social Network

Social Network Results

At this current time, the Social Network benchmark application is not in a state that can be run or tested. However, there is an issue referenced in section 4.4 with the MASS CUDA library implementation that was exposed through the Social Network's use of 'exchangeAll'.

4.3. Execution Performance

Game of Life

After running 'make test', the execution performance for the Game of Life application according to the 5 test simulations is as follows:

- Simulation size 2: 211 ms
- Simulation size 4: 459 ms
- Simulation size 8: 441 ms

- Simulation size 16: 437 ms
- Simulation size 32: 440 ms

This optimality in execution performance is a great indicator of how the MASS CUDA library can leverage the parallelizable potential of Game of Life since the execution time remained near constant as simulation sizes grew.

4.4. Current Problems

Social Network

Due to the 'exchangeAll' function requiring an argument of neighbors, the 'neighbor' member of PlaceState is not sufficient for the use case of Place and PlaceState in the Social Network application. Brian has been notified of this bug and it should be fixed and ready for use through my next quarter of work on Social Network.

5. Conclusion

5.1. Summary

Over the past summer quarter of development, I have successfully implemented the Game of Life benchmark application and have validated my implementation using 5 different valid test cases for simulation runs. This validation exercise also produced 5 different visualizations of the state of the grid after 10 generations. Although I am not familiar with the execution performance of Game of Life on MASS C++, I imagine that the implementation on MASS CUDA reveals better performance metrics.

I have also implemented a sizable portion of the Social Network benchmark application which is ported from its MASS C++ implementation to MASS CUDA. While I have not produced execution performance figures after my brief time of work on the application, I am confident that once my implementation is complete through Fall quarter, the performance metrics for the Social Network application will be as impressive as the Game of Life figures.

5.2. Future Development

My future development plans are to continue with my efforts to implement and validate the Social Network benchmark application through Fall quarter. My validation work will include producing some visualization output for reference. I will then be working primarily on porting and implementing the VDT benchmark application from its MASS C++ implementation to a new MASS CUDA implementation.

By achieving my end goal, there will only be 3 more benchmark applications implemented using the MASS CUDA library.

Appendix

Code Location

The code location for Game of Life and Social Network applications is located under the 'oahmed_develop' branch at following repo location:

https://bitbucket.org/mass_application_developers/mass_cuda_appl/src/oahmed_develop/

My implementation of Game of Life is found under the GameOfLife directory in 'GameOfLife_dev'.

How to Run

Game of Life

Code There is a reference to the Game of Life running instructions in the README.md file in the 'GameOfLife_dev' directory, but the two steps to reproduce my output are:

- 'make build'
- 'make test'

Social Network

The Social Network application is still under development at this time and cannot be run!