Developing and Refactoring MASS CUDA Application Benchmarks

Omar Ahmed

CSS 497 Autumn 2023 Term Report

Professor. Munehiro Fukuda

12/13/2023

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 the Autumn 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 implements the mathematical John Conway's Game of Life. In contrast, 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-GPUs, 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 (Multi-Agent Spatial Simulation) Library is a parallel-computing library for multi-agent and spatial simulations over a cluster of computing nodes. The CUDA (Compute Unified Device Architecture) platform is a framework developed by Nvidia to expose the capabilities of GPU acceleration, specifically on Nvidia CUDA-ready 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, 4 of which have been implemented (GameOfLife, Heat2D, BrainGrid, and SugarScape), while I have worked on implementing 2 of the remaining 3 applications.

The latest implemented benchmark application is the Game of Life application which is a cellular automaton simulation of the interaction between the cells in a 2D grid that are dead or alive based on their surrounding neighbors' states. This application is modeled using Places, where each Place is a cell in the grid, but does not use any static/dynamic agents.

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 Social Network benchmark application simulates the connections/friends formed between an *n* number of people in a social network and allows for retrieval of the *x*-th degree list of friends of all people in the network. The people in the social network are represented as places and not static or dynamic agents. What distinguishes this application from the ones previously implemented is that the network of friendships in the social network is a group of Places 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 due to the computationally intensive and parallel nature of the breadth-first search algorithm that is run when retrieving each user's *x*-th degree friendships.

The Tuberculosis benchmark application is a simulation of a 2D grid that represents human lung tissue. Bacteria on Places and Macrophage Agents are then spawned. Bacteria grow at a predetermined rate on each Place, and Macrophages Agents eat and react to the bacteria and other Macrophages if they are infected. At each time step, Macrophages are spawned from four entry points on the grid that represent blood vessels. At a predetermined time step, T-cell Agents enter the simulation via the blood vessels and react to the macrophages, activating the infected ones and bursting the chronically infected. The simulation computes and visualizes the state of the grid for *x* 'total days' when the simulation is terminated. What distinguishes this application is its use of a 2-dimensional grid where each cell is a place that warrants two differently behaving agents to migrate and eliminate bacteria. MASS CUDA's programming model

applies to this benchmark application as it relies on the agent-based model developed in the library which is highly parallelizable.

3. Implementation

3.1. Social Network

3.1.1. SocialPlace as Places

SocialPlaceState

The SocialPlace class represents a Person in the Social Network simulation and is responsible for tracking the Person's *k* first degree friends and computing their *x*-th degree circles of friends. Since the Social Network application is modeled as a bidirectional graph that represents all friendships between different People, each Person must track their first degree of friends from an adjacency list which includes all People's first-degree friends. The 'SocialPlaceState.h' class inherits the PlaceState class and consists of a userId to track the index of the place in the places array when instantiated as well as an integer array of the People's first-degree friends' userIds. A queue and set data structures are used to traverse the graph through breadth-first search and track visited places, then an int array 'currentLevel' is used for outputting friendships at each degree. An excerpt of that code is shown in Appendix Code Snapshot A:

SocialPlace Initialization:

SocialPlaces is meant to be a 1D array of SocialPlace classes that each contain a list of adjacencies. The code in Appendix Code Snapshot B shows how SocialPlaces is created. The basis of this adjacency list model is a 2D array that is as long as the *n* number of users, represented as places, specified on host memory through user input and is as wide as the *k* 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 Place sets their userId using an invocation to 'setUserId'. Then, a place receives their list of first-degree friends which is passed through the 'setFriends' function. The callAll MASS function is called to 'setFriends' with an argument of a 1D contiguous representation of the 2D adjacency list which is nPeople * nFriends big, then each SocialPlace retrieves the 1D array relevant to them. While each place is retrieving their int array of first-degree friends, their neighborOffsets array is being populated.

Using the MASS function ‘exchangeAll’, each Place’s actual neighbors’ array is then populated with Place pointers using their dynamic neighborOffsets array.

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 *x*-th degree friend by using a modified parallel breadth-first search algorithm that outputs and retrieves each Place’s next degree of friends. Through populating their neighborOffsets array all places exchange neighbors using the ‘exchangeAll’ function.

3.1.2. Simulation

The simulation starts with initializing each place with its index in the places array using the MASS ‘callAll’ function. Then, the 1D 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 *x*-th degree friendships. An excerpt of the almost complete functionality can be seen in Appendix Code Snapshot C

3.2. Tuberculosis

3.2.1. EnviromentPlace as Places

EnviromentPlaceState:

The EnviromentPlace class manages each of the grid places’ state throughout the TB simulation, including bacteria growth and the functions/logic to control macrophage agents that decay by 1 level per day releasing chemokine and T-cell agents responding. The state of each place is contained in ‘EnviromentPlaceState.h’ and contains the state of agents on this place as each place is limited to one macrophage agent and one T-cell agent residing on it at a time. An excerpt of this can be found in Appendix Code Snapshot D:

EnvironmentPlace initialization:

All places in the simulation representing EnviromentPlace classes are initialized using code that can be found in Appendix Code SnapShot E:

A randomization of the initial Places with bacteria on the grid occurs on CPU host memory and then sent via callAll() to the GPU device memory for processing. The blood vessels in all four quadrants are also determined based on the size of the simulation. The dimension of the grid is always set to two but the placesSize is size * size to be an n * n grid.

EnviromentPlace Functions:

When the simulation is initialized and the randomized initial state of the simulation is determined, each agent will call a place's 'getHighestChemokine' function to determine where to migrate within its cardinal neighbors.

For every simulation cycle/day, 'chemokineDecay' is called to decay the chemokine levels by 1 and if a macrophage is present with it then it will decay the macrophage's chemokine level to 2 as well.

After the first 10 days, agents can spawn deciding on location using the 'cellRecruitment' function. Then, every 10 days after that the 'bacteriaGrowth' function is invoked which causes places containing bacteria to spread it to their neighbors.

3.2.2. Macrophage as Agents

MacrophageState

The macrophage dynamic agents represent human immune system macrophage cells that can be resting, infected, activated, chronically infected, or dead. It also tracks its internal bacteria and infection times. An excerpt can be found in Appendix Code Snapshot F

Macrophage Initialization

The initial spawned amount of macrophages is determined by init_macro_num plus the four spawner points in each quadrant which results in the totalMacro. Each of these macrophages is assigned a random place and the spawners are assigned to the blood vessels. They are initialized using the code below:

```
Agents* macrophages = Mass::createAgents<Macrophage, MacrophageState> (1, NULL, sizeof(double), 0, 0);
```

Macrophage Functions

Every simulation cycle, the 'migrate' function is invoked in the macrophages. The agent's state is then updated by invoking the 'updateState' Agent function. Spawners will invoke 'spawnMicro' if the place decided that it should spawn a macrophage.

3.2.3. T-cells as Agents

TcellState

The T-cell state only tracks if it is a spawner.

T-cell Initialization

T-cell spawners are created at the beginning of the simulation, below is an excerpt of the code:

```
Agents* tcells = Mass::createAgents<TCell, TCellState> (2, NULL, sizeof(double), 0, 0);
```

T-cell Functions

Using the get ‘getHighestChemokine’ Places function, T-cells decide where to spawn. Spawners also spawn a T-cell using ‘spawnTCell’ if a Place decides to spawn one.

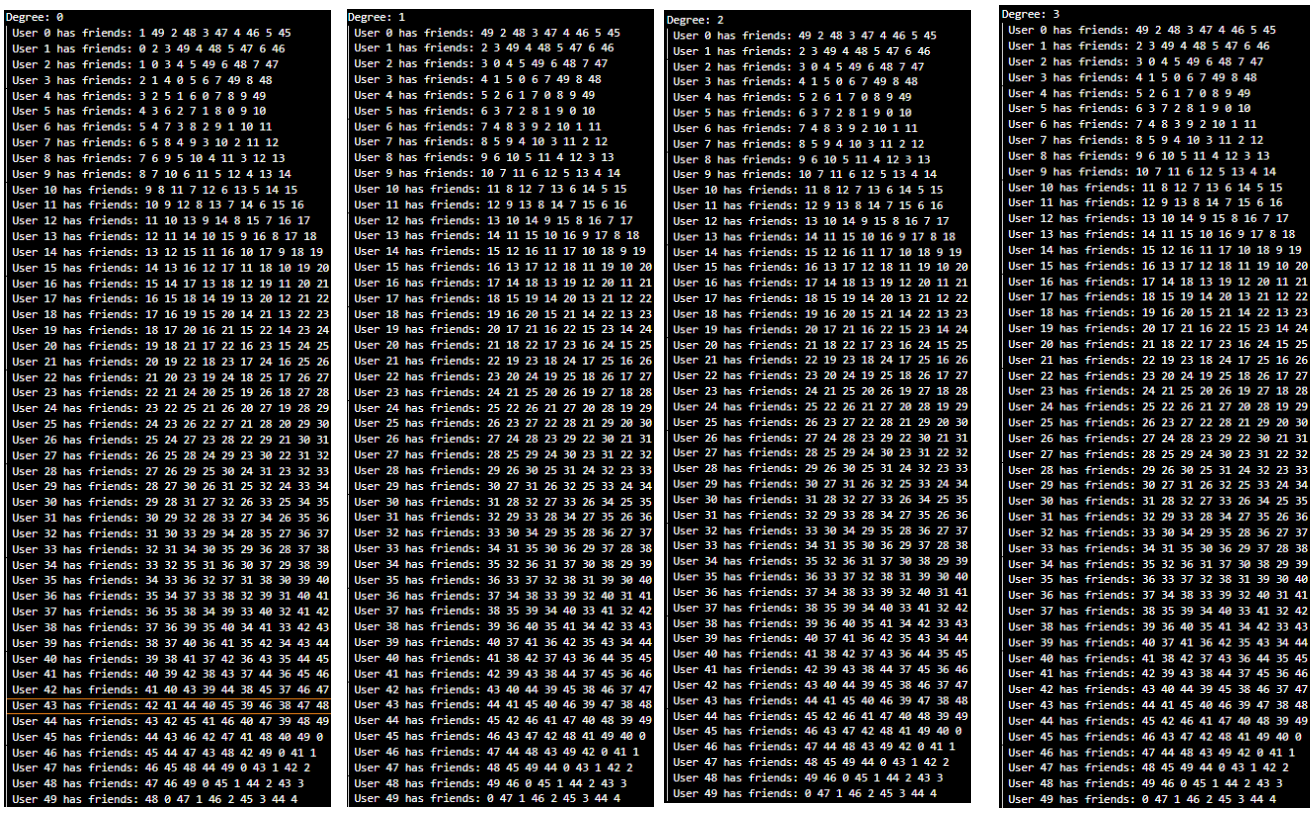
3.2.4. Simulation

This implementation uses a vector of all cardinal neighbors which is created and allows for passing of bacteria and agent state from current place to neighbors through exchangeAll() for places. The simulation is terminated when the desired number of totalDays is achieved. See code in Appendix Code Snapshot G

4. Verification

4.1. Social Network

Social Network Visualization



This is a snapshot of a representative output produced by the Social Network simulation. The parameters of this simulation are 50 people in the network, each person has 10 friends, and the goal is to retrieve their 4th degree of friends. At each degree of friendship, all people’s relevant friends for that degree are listed.

Social Network Results

The Social Network application is functioning correctly, bug-free, and safe within CUDA memory. However, the following bugs addressed are existent or potentially existent problems.

During the development of the Social Network benchmark application, I discovered several bugs within the MASS CUDA Library. Most importantly, is the lack of a method to invoke an exchange of neighbors based on offsets that are relative, not predetermined, and dynamic to places rather than predetermined offsets. A new methodology of setting neighborOffsets specific to the functionality of each application the populating a list of neighboring Places based on those dynamic offsets was implemented in a new exchangeAll() function.

Another issue I encountered is the lack of C++ standard library or NVCC Thrust library support for native data structures. This led to my implementation of necessary data structures such as queues and sets. This also birthed a new issue of non-dynamic memory allocation on a CUDA kernel which results in over-allocation of memory, and this can, in cases of large-size simulations, exhaust all available memory on the GPU device and crash the application.

Lastly, in the case of host-device memory transfer, both the Social Network and Game of Life programs have proven that MASS CUDA is inefficiently handling function invocation and control transfer between host and device which causes higher than expected execution time.

4.2. Tuberculosis

Tuberculosis Results

As Tuberculosis is now functional, it is still not producing the desired output due to incorrect agent behavior. The Agent class in MASS should allow for migration of agents to any part of the grid, while it can only migrate to direct neighbors of a Place. This has been fixed in version 0.6.2 in the new dynamic migration, termination, and spawn features implemented. As a result, Tuberculosis runs with no errors but behaves incorrectly.

4.3. Execution Performance

Social Network

After running ‘make test’, the execution performance for the Social Network application according to the 7 test simulations is as follows:

Users	Friends	Degrees (iterations)	Times (ms)
10	2	5	344
20	5	5	423
50	10	5	384
100	10	5	384

200	10	5	377
300	10	5	383
300	30	1	189

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

5. Conclusion

5.1. Summary

Over the past Autumn quarter of development, I have successfully implemented the Social Network benchmark application and have validated my implementation using 7 different valid test cases for simulation runs. This validation exercise also produced 7 different visualizations of the *x*-th degree of friends. This implementation in MASS CUDA reveals better performance metrics in both programmability and execution performance.

I have also implemented and ported a sizable portion of the Tuberculosis 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 Tuberculosis application will be as impressive as the Game of Life and Social Network figures.

Appendix

Code Snapshots:

(A)

```
class SocialPlaceState : public mass::PlaceState {
public:
    MASS_FUNCTION ~SocialPlaceState(){
        delete q;
        delete visited;
        delete[] firstDegree;
        delete[] currentLevel;
    }
    int userId;
    int nFriends;
    Queue* q;
    int* firstDegree;
    int* currentLevel;
    int currentLevelSize;
    Set* visited;
};
```

(B)

```
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
);
```

(C)

```
places->callAll(SocialPlace::SET_USER_ID, &nFriends, sizeof(nFriends));
vector<int> friendShipList((nPeople * nFriends), -1);

createKRegularGraph(friendShipList, nPeople, nFriends);

int myArray[friendShipList.size()];
for(int i = 0; i < friendShipList.size(); i++){
  myArray[i] = friendShipList[i];
}

Timer timer;
timer.start();

places->callAll(SocialPlace::SET_FRIENDS, &myArray, sizeof(myArray));
places->exchangeAll(nFriends);

// Call all places to calculateXth neighbor
for(int degree = 0; degree < xthDegree; degree++){
  places->callAll(SocialPlace::CALCULATE_XTH_FRIENDS);
  places->updateNeighborhoodWithLocalOffsets(MAX_NEIGHBORS);
```

(D)

```
class EnvironmentPlaceState : public mass::PlaceState {
public:
  // Whether the bacterium exists.
  bool bacteria;

  // Whether the current EnvironmentPlace is an entry point for simulation agents.
  bool bloodVessel;

  // The chemokine level.
  int chemokine;

  // The current Macrophage occupying the EnvironmentPlace.
  Macrophage* currentMacrophage;

  // The current T-Cell occupying the EnvironmentPlace.
  TCell* currentTCell;

  // The EnvironmentPlace of a migratation destination.
  EnvironmentPlace* migrationDest;

  // The index of the relative migratation destination.
  int migrationDestRelativeIdx;
};
```

(E)

```
const int nDims = 2, totalCells = size * size, agentDistribution = 10;
// const int bloodVessels = 4;
int placesSize[nDims] = {size, size};

Places* places = Mass::createPlaces<EnvironmentPlace, EnvironmentPlaceState>(0, NULL, sizeof(double), nDims, placesSize);
places->callAll(EnvironmentPlace::INIT_BACTERIA);
```

(F)

```
// The current state, number of internal bacteria, the number of days since
// the macrophage was first infeceted, and destination index.
int state, internalBacteria, infectedTime, destinationIdx;
```

(G)

```
vector<int*> neighbors;   // {x, y}
int northwest[2] = {-1, 1};
neighbors.push_back(northwest);
int north[2] = {0, 1};
neighbors.push_back(north);
int northeast[2] = {1, 1};
neighbors.push_back(northeast);
int west[2] = {-1, 0};
neighbors.push_back(west);
int east[2] = {1, 0};
neighbors.push_back(east);
int southwest[2] = {-1, -1};
neighbors.push_back(southwest);
int south[2] = {0, -1};
neighbors.push_back(south);
int southeast[2] = {1, -1};
neighbors.push_back(southeast);

for (int i = 0; i < totalDays; i++) {
  places->exchangeAll(&neighbors, EnvironmentPlace::CHEMOKINE_DECAY, NULL, 0);

  if (i > 0 && i % 10 == 0)   // grows every 10 days
    places->exchangeAll(&neighbors, EnvironmentPlace::BACTERIA_GROWTH, NULL, 0);

  // int recruitment[bloodVessels];
  // for (int j = 0; j < bloodVessels; j++) {
  //   recruitment[i] = i < 10 ? rand() % 2 : rand() % 3;
  // }
  // places? agents?

  macrophages->callAll(Macrophage::MIGRATE);
  macrophages->manageAll();
  macrophages->callAll(Macrophage::UPDATE_STATE);

  tcells->callAll(TCell::MIGRATE);
  tcells->manageAll();
}
```

Location

The code location for Social Network and Tuberculosis 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 SocialNetwork is found under the SocialNetwork directory. My implementation of Tuberculosis is found under the Tuberculosis_dev directory.

How to Run

Social Network & Tuberculosis

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’