Agent-Based Models Library Over Multiple GPUs: Term Report

Warren Liu

School of STEM, Computer Science & Software Engineering

University of Washington

CSS 595: Master’s Project, Autumn 2023

12/12/2023

Project Committee:

Professor Munehiro Fukuda, Committee Chair

Professor Kelvin Sung, Committee Member

Professor Clark Olson, Committee Member

TABLE OF CONTENTS

1 Introduction 2
2 Overview 2
3 Goals 3
4 Achievements This Quarter 3
4.1 Places 4
4.1.1 Update of Neighbor's Array Based on Relative Index 4
4.1.2 ExchangeAll() 5
4.2 Agents 5
4.2.1 Spawn 6
4.2.2 Migration 9
4.2.3 Termination 10
5 Results 11
6 Next Quarter's Plan 11
7 Summary 12
8 Appendix 12
8.1 Function Implementation 12
8.2 How to Run 15
References 16

1 Introduction

This term report summarizes the progress made on my capstone project during the Autumn 2023 quarter. It marks the first quarter where I began implementing the MASS CUDA library. My work involved familiarizing myself with the library's implementation, identifying issues within the single-GPU framework, testing and providing solutions, refactoring the implementation, and building a fully functional single-GPU version of the MASS CUDA library.

2 Overview

MASS is a parallel computing library for **M**ulti-**A**gent **S**patial **S**imulation. As implied by its name, the foundational design principle of MASS revolves around the concept of multi-agents, each acting as an individual simulation entity within a designated virtual space [1].

At the heart of the MASS library are two fundamental components: “Places” and “Agents” as shown in Figure 1. “Places” refers to a matrix of elements distributed across multiple GPUs within a single machine, where each element, known as a “Place”, is identifiable by matrix indices and capable of engaging in information exchange with other places. In contrast, “Agents” represent a collection of execution instances. These agents are not only able to inhabit a place but also possess the mobility to migrate across different places, potentially replicating themselves in the process. Moreover, agents have the capability to interact with both other agents and various places simultaneously [1]. In our current implementation of MASS using CUDA, the library is operational on a single GPU.

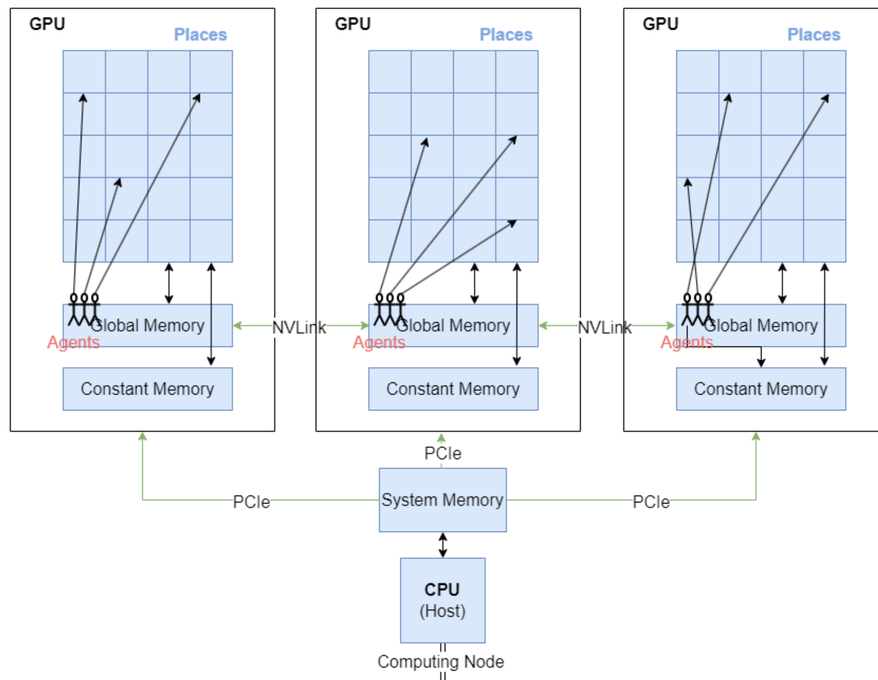

Figure 1 MASS CUDA Architecture

My primary goal was to extend the MASS CUDA library's functionality from a single GPU to multiple GPUs. The plan for this quarter was to analyze the existing code base, design a new architecture, and begin the implementation phase. However, numerous challenges arose.

The first major challenge was the lack of documentation. The source code had few comments, and the only developer guide, written in 2014, did not reflect the actual implementation. Consequently, understanding the interaction of files and functions took considerably longer than anticipated.

Secondly, the existing source code differed from its intended design. Previous students' term reports and theses, particularly this work on MASS CUDA [2], suggested functionalities that were absent in the actual code. The real issue was that the MASS CUDA library was not fully operational on a single GPU, particularly in terms of Agent functionalities, which were only partially working and bug-ridden.

As a result, my focus shifted from developing a multi-GPU version to first establishing a fully functional single-GPU version, benchmarking it, identifying potential issues, and exploring performance improvements. Despite these challenges, I view this experience as a valuable opportunity to deepen my understanding of the MASS CUDA library, which will benefit future performance tuning and multi-GPU development.

3 Goals

My goals for this quarter were divided into three main categories:

1. Become familiar with the code base and understand the library.
2. Refactor the code base for full functionality on a single GPU:
    a. Refactor Place-related code for consistent function naming across different MASS library implementations and to support new functionalities.
    b. Implement Agent-related code, including termination, migration, and spawning.
    c. Address any issues that arise during the implementation process.
3. Analyze potential issues and prepare for implementing the multi-GPU version:
    a. Examine code related to GPU usage.
    b. Identify code that hinders the library's operation on multiple GPUs.
    c. Seek potential performance improvements.

4 Achievements This Quarter

Despite minor issues, there were several major achievements:

4.1 Places

The MASS library is characterized by two essential functions within its Places component: *Places::callAll()* and *Places::exchangeAll()*. These functions are integral to the library's operation, offering parallel processing capabilities that significantly enhance simulation efficiency.

The *Places::callAll()* function enables users to execute specific functions across all Places in parallel. By passing a user-defined function and optional arguments to *Places::callAll()*, each Place executes this function concurrently.

In contrast, the *Places::exchangeAll()* function is designed for updating neighbor information of Places in a parallel manner. Neighbor information is vital in the MASS library, facilitating information exchange between Places and enabling Agents to migrate to neighboring Places.

4.1.1 Update of Neighbor's Array Based on Relative Index

In MASS CUDA, *Places::ExchangeAll()* enables users to update neighbors' information, specifically, to define neighboring Places. Users input an integer vector to specify the relative index of neighbors. For instance, in a row-major index system, [-1, 0] indicates a relative index "top", and [0, 1] indicates "right", as illustrated in Figure 2 on the left.

-1, -1	-1, 0	-1, 1
0, -1	0, 0	0, 1
1, -1	1, 0	1, 1

-1, -1	0, -1	1, -1
-1, 0	0, 0	1, 0
-1, 1	0, 1	1, 1

Figure 2 Relative Index in Row-major (left) and Column-major (right)

However, upon implementing test cases and reviewing the source code of existing tests and applications built using the MASS CUDA library, I discovered that the library defaults to using column-major indexing. This means [-1, 0] translates to a relative index of "left" and [0, 1] to "bottom", as shown in Figure 2 on the right. Consequently, some tests and applications input column-major indices for *Places::ExchangeAll()*. Developers in later projects created their own functions to convert "relative index to real offset" before passing it to *Places::ExchangeAll()*.

To address this, I implemented an override of the *MASS::createPlaces()* function, which includes an additional parameter for specifying the indexing major. When users specify this major, the Places object and subsequent relative index to offset translations use the specified major. If the major is not specified, column-major is used by default. This override not only preserves the functionality of existing applications using the MASS CUDA library but also provides future users with more development flexibility.

The modified function signatures are as follows:

```
template <typename PlaceType, typename PlaceStateType>
        static Places *createPlaces(int handle, void *argument, int argSize,
                                    int dimensions, int size[], int majorType);
```

```
template <typename PlaceType, typename PlaceStateType>
        static Places *createPlaces(int handle, void *argument, int argSize, dimensions, int size[]);
```

4.1.2 ExchangeAll()

As previously mentioned, the *Places::ExchangeAll()* function allows users to update the neighbor information of Places. Originally, this function was designed to accept relative indices in integer vector format. However, some applications prefer using the real offset of Places instead of relative indices. In earlier solutions, Brian Luger implemented the *Places::updateNeighborhoodWithLocalOffsets()* function to handle offsets [3]. This, however, resulted in inconsistencies in function naming across all MASS libraries and confusion regarding the usage of different functions. To resolve this, I integrated the functionalities of *Places::updateNeighborhoodWithLocalOffsets()* into two new overrides of the *Places::ExchangeAll()* function. This approach not only provides users with more options but also maintains consistency in function naming:

```
void exchangeAllPlaces(int placeHandle, std::vector<int*> *destinations);
void exchangeAllPlaces(int handle, std::vector<int*> *destinations, int functionId,
        void *argument, int argSize);
void exchangeAllPlaces(int handle, int nNeighbors);
void exchangeAllPlaces(int handle, int nNeighbors, int functionId, void *argument, int argSize);
```

4.2 Agents

In the MASS library, the Agents component is characterized by two essential functions: *Agents::callAll()* and *Agents::manageAll()*. These functions are integral in enabling parallel execution and efficient management of Agents within the simulation.

The *Agents::callAll()* function operates similarly to *Places::callAll()*, offering users the capability to pass functions and arguments that are executed by all Agents concurrently. This parallel execution is crucial for implementing uniform operations across the entire set of Agents swiftly and efficiently.

Additionally, each Agent in the MASS library comes equipped with three primary functions: migrate, spawn, and kill. The *Agent::migrate()* function allows an Agent to move from its current Place to another, adding dynamic interactions within the virtual environment. The *Agent::spawn()* function lets an Agent create offspring either at its current location or at a different Place, contributing to the complexity of the simulation. The *Agent::kill()* function serves to terminate an Agent, thereby freeing up space for new Agents and excluding the terminated Agents from future parallel operations.

To conduct batch operations like migration, spawning, or termination of Agents effectively, these functions can be utilized in conjunction with *Agents::callAll()*. Following the use of any of these functions, it is imperative to use *Agents::manageAll()*. This function commands the MASS library to execute the actual operations as per the given instructions. The detailed functionality and application of *Agents::manageAll()* is further elaborated in Section 4.2.2 and 4.2.3.

4.2.1 Spawn

When initializing Agents on the GPU, an array of Agent objects is allocated to hold the specified number of Agents. Throughout the program, users may terminate or spawn new Agents. In the existing implementation, we track the number of alive Agents. When a new Agent is spawned, the count of alive Agents increases, and vice versa. The Agent array is then updated based on this count.

An example of this implementation is depicted in Figure 3. Initially, the Agents array is populated with 5 alive Agents and a maximum capacity of 10. When Agent 2 is terminated, reducing the count of alive Agents to 4, the next new Agent should be placed at index 4. However, an alive Agent already occupies this index, revealing a significant bug.

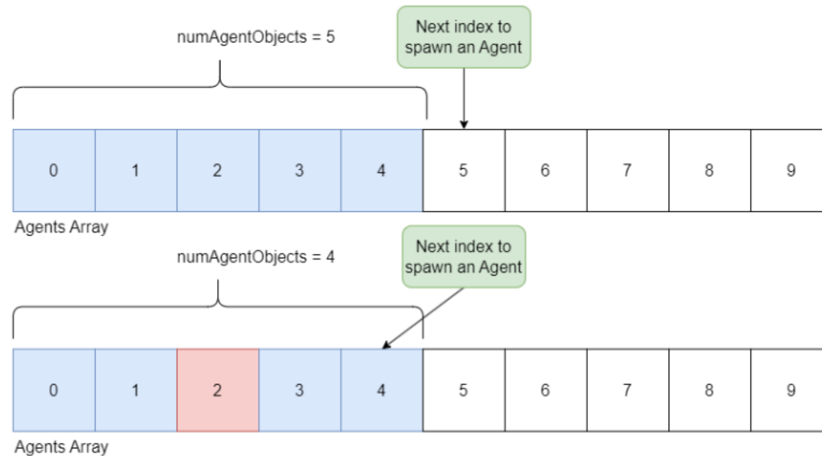

Figure 3 Existing Algorithm of Agent Spawn

After discussing this issue with Dr. Fukuda, he suggested the original design for managing the Agents array, which involved sorting the array using CUDPP's radix sort to position alive Agents at the beginning and dead Agents at the end (as shown in Figure 4), while simultaneously counting the number of alive Agents [4].

Figure 4 Original Proposed Algorithm of Agent Spawn

However, the implementation of this sorting algorithm in the existing architecture of the MASS CUDA library led to more problems than solutions. The MASS library primarily consists of Places and Agents arrays. Places are arrays of Place objects in GPU memory holding data, and Agents are arrays of Agent objects dynamically allocated in GPU memory, each assigned to different Place objects as real execution instances.

In the MASS CUDA library, unlike other MASS implementations, four arrays are used to hold Places and Agents (Figure 5). The Places array stores Place objects, which uniquely focus on housing functions rather than containing data such as indices or neighborhood information. This data is instead held in the PlaceState array, where each index aligns with a Place object in the Places array, ensuring that all relevant data is kept in a corresponding and orderly manner. Similarly, the Agents array consists of Agent objects that encapsulate the necessary functions for the agents' actions within the simulation. Complementing this, the AgentState array holds all the data for each Agent object, effectively separating the agents' functional aspects from their state data. To ensure cohesion and efficient access between these functional and state aspects, each Place is linked to its PlaceState, and each Agent to its AgentState, through pointers.

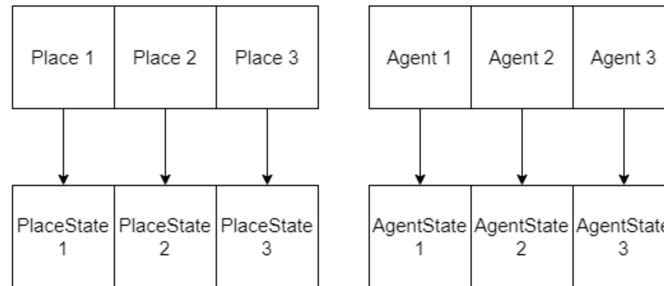

Figure 5 Place and Agent object in MASS CUDA Library

Let's consider an example where we assign Agent 1, 2, and 3 to Place 1, 2, and 3, respectively. In this scenario, each Agent pointer in the PlaceState object points to the corresponding Agent object, and each Place pointer in the AgentState object points to the corresponding Place object. This forms a link between them, as illustrated in Figure 6. For simplicity, the PlaceState object is omitted in the figure.

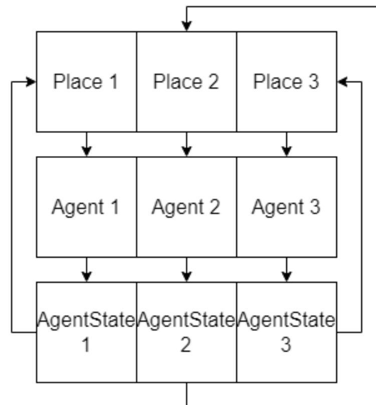

Figure 6 Linked Place and Agent Object

Now, if we decide to terminate Agent 2, as per the previously explained algorithm, we need to sort the Agent array to move Agent 2 to the end. As a result, the order in the Agents array would be Agent 1 (alive), Agent 3 (alive), and Agent 2 (dead). It's important to note that all data of an Agent are stored in the AgentState object, so the actual sorting happens in the AgentState array, not the Agents array. However, since pointers reference addresses and not the objects themselves, the Agent still points to its original index in the AgentState array. Consequently, if we search for an Agent from a Place object, the results

may not align with expectations, as shown in Figure 7. The PlaceState object is again omitted for simplicity.

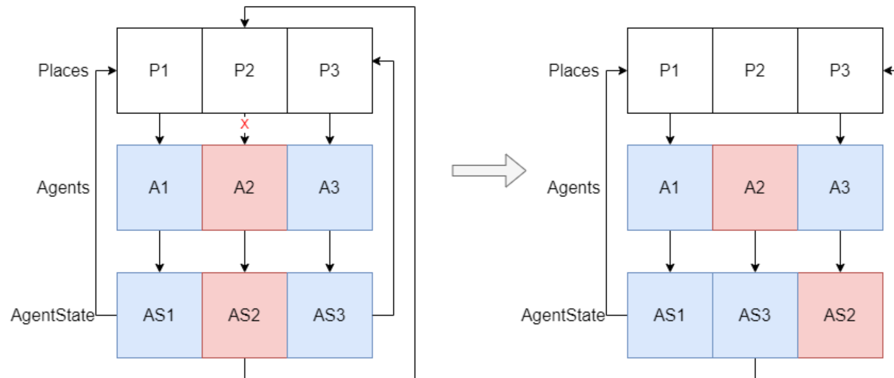

Figure 7 Sort Linked Place and Agent Object

My initial solution was to sort both the Agents and AgentState arrays. However, this did not resolve the issue of Place objects not pointing to the correct Agents. The ultimate solution seemed to involve sorting all arrays, including Place, PlaceState, Agent, and AgentState, which would have necessitated a complete library refactor.

Another solution considered was to sort the Agents array alongside the AgentState array, excluding the Place and PlaceState arrays. After sorting, we would then "re-link" Places to the correct Agents. This algorithm, however, required each Place object to loop through the entire Agent array, resulting in an $O(n^2)$ algorithm.

I then devised a new algorithm for obtaining available indexes in the Agents array. As shown in Figure 8, we created an integer array, the same length as the Agents array, initialized with -1s. Agents report their availability to this array. We then remove all -1s, leaving only available indexes.

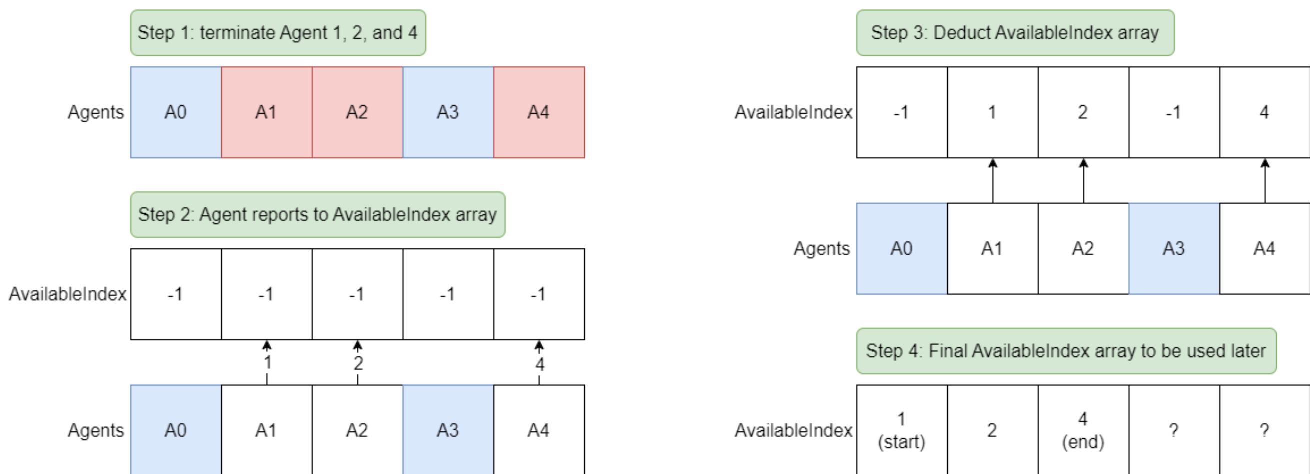

Figure 8 New Algorithm of Agent Spawn

This method allows for the direct placement of new Agents in the available indexes. The reporting step is executed as a CUDA kernel function, rendering it an O(1) operation. Additionally, the CUDA Thrust Library [5], which provides a well-developed deduction function using CUDA, is employed for the deduction process, also an O(1) operation. Although this solution may not be optimal, it is more effective than the originally proposed one.

4.2.2 Migration

When a user intends to migrate an Agent from one Place to another, the MASS CUDA library executes a series of operations as follows:

1. **In the *Agent::migrate()*:**
   - The Agent is added to the destination Place's potentialAgents array. This action signifies the Agent's intention to migrate to that Place.
2. **In the *Dispatcher::manageAll() -> Dispatcher:: migrateAgents()*:**
   - Since there is a maximum number of Agents that can reside in a Place, conflicts must be resolved first. For example, if Agents 1, 2, and 3 all wish to migrate to Place 1, but Place 1 can only accommodate two Agents, one of the Agents will not be migrated and will remain at its original Place. The current algorithm selects the Agent with the lowest N index for migration, where N represents the maximum number of Agents a Place can accommodate.
   - After resolving these conflicts, eligible Agents are removed from their source Place and added to their destination Place.

During the implementation, I identified a significant bug in the *Agent::migrate()* function. The existing implementation involved adding an Agent to the Place's local potentialNextAgents array, which stored Agents expressing interest in migrating to that Place. This process involved iterating through the potentialNextAgents array and placing the Agent in the first found empty location.

```
__device__ void Place::addMigratingAgent(Agent* agent, int relativeIdx) {
    for (int i = 0; i < N_DESTINATIONS; i++) {
        if (state->potentialNextAgents[i] == NULL) {
            state->potentialNextAgents[i] = agent;
            break;
        }
    }
}
```

This function performed adequately when executed sequentially. However, it's crucial to remember that we were implementing the library using CUDA, which meant this function would be executed in parallel by multiple Agents attempting to migrate to the same Place simultaneously. This parallel execution posed a risk of a race condition, where multiple Agents could identify the same empty location in the array at

the same time.

To mitigate this race condition, I introduced an additional local variable to the Place object, named numPotentialNextAgents. This variable tracks the number of Agents added to the potentialNextAgents array and serves as the index for adding the next Agent. By employing the atomicAdd() function provided by CUDA, this variable is made thread-safe, ensuring that only one Agent can modify it at a time. The updated code is as follows:

```
__device__ void Place::addMigratingAgent(Agent* agent, int relativeIdx) {
    int index = atomicAdd(state->numPotentialNextAgents, 1);
    state->potentialNextAgents[index] = agent;
}
```

4.2.3 Termination

In the MASS CUDA implementation, Agents possess three core functions: termination, migration, and spawn. To manage the execution of these functions on the GPU (also known as kernel functions), we have introduced a class named "Dispatcher." This class is pivotal in leveraging the GPU's capabilities to perform calculations in parallel, thereby enhancing performance.

When users employ Agent functions for different purposes, these functions cannot be executed immediately. This is because it is necessary for all Agents to receive their orders in a single step. Consequently, we set what can be termed as "pre-operation" attributes for each Agent. The Dispatcher class then takes over, executing these operations in parallel.

For instance, if there is a need to spawn children for some Agents, the first step involves using the *Agents::callAll()* function. This function sets the number of new children and the destination Place information for each Agent intended to spawn children. Following this, the *Agents::manageAll()* function is called to perform the actual spawn operation. Within this function, the specified number of new children are spawned at the destination Place. Similarly, if there is a requirement to migrate Agents to different Places, the *Agents::callAll()* function is once again utilized to set the destination Place information for each Agent. This is followed by invoking the *Agents::manageAll()* function, which facilitates the actual movement of Agents in parallel. *Agents::manageAll()* invokes Dispatcher functions directly.

However, the process of termination significantly differs from migration and spawn. In the case of termination, only a single operation step is involved: setting the status of an Agent to "False" to signify its termination. This simplicity means that no further actions are required post this setting. Therefore, the *Dispatcher::terminateAgents()* function is intentionally left blank. This approach is purposeful and aligns with the streamlined nature of the termination process.

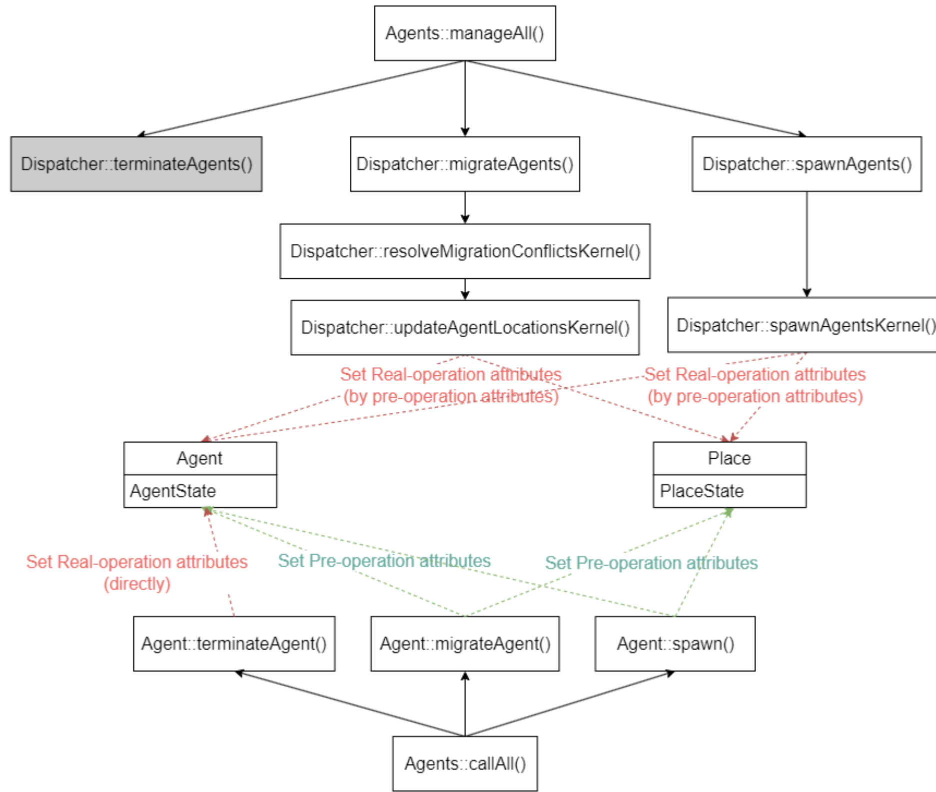

Figure 9 Relationship Between Agents and Dispatcher Classes

5 Results

All the required functions of MASS, Places, and Agents have been successfully implemented and have passed various tests. However, we currently lack new applications built using the updated MASS CUDA library. I anticipate that additional potential issues might be identified when students commence benchmarking of the library. To date, each function has undergone thorough testing, and all tests have been successfully passed.

6 Next Quarter's Plan

Despite NVIDIA's official documentation recommending performance tuning towards the end of the implementation phase to avoid repetitive work (owing to potential reversion of code changes due to performance issues), several areas are identified where performance can be significantly enhanced. These enhancements include using local memory instead of global memory in kernel functions, removing redundant code that allocates unnecessary memory (leftover from previous implementation changes), and optimizing data padding on the GPU for accelerated processing. My immediate focus, before delving into multi-GPU implementation, will be on these performance tuning and benchmarking tasks.

Furthermore, the documentation for the MASS CUDA library, last updated in 2014, does not accurately

reflect our current implementation. I plan to update this documentation concurrently with ongoing library development.

Additionally, now that the single GPU version is operational, I intend to proceed with the development of the multi-GPU version. I have identified several areas to begin this work, such as addressing parts of the code that currently prevent the library from running on multiple GPUs. If new students join to benchmark the single GPU version of the MASS CUDA library, I will also engage in debugging this version.

Moreover, all functions in the MASS CUDA library are currently defined as *MASS_FUNCTION*, which translates to *__device__ __host__* functions, indicating their capability to execute on both the device and the host. However, most of these functions are required only on the device. Thus, it's essential to revise the definition of these functions. Although this change might not directly impact function performance, it will clarify the function's usage for users and developers. Furthermore, specifying the correct and necessary execution location for each function will aid in optimizing memory space utilization within our library.

7 Summary

Although I am currently behind the initially planned schedule due to unforeseen challenges in implementation and documentation, this project has afforded me the opportunity to develop the MASS CUDA library from the ground up. This approach has been instrumental in deepening my understanding of the library's functionalities. With this invaluable experience, I am confident in my ability to implement the multi-GPU version more efficiently and with fewer potential bugs. Consequently, I am optimistic about catching up with the schedule in the near future. Reflecting on the timeline outlined in my proposal, I am confident that I can complete the forthcoming tasks on time, given my current proficiency with the MASS CUDA library.

8 Appendix

8.1 Function Implementation

Below are detailed implementations of key functions discussed in the report.

Dispatcher::spawnAgents()

Description: This function handles the spawning of new agents. It calculates available indexes and then uses these indexes to spawn agents efficiently.

Linked Section: Section 4.2.1

Implementation:

```
void Dispatcher::spawnAgents(int handle) {
    Agent **a_ptrs = deviceInfo->getDevAgents(handle);
    dim3* dims = deviceInfo->getAgentsDims(handle);
    int maxAgents = deviceInfo->getMaxAgents(handle);
```

```
    // Find the available indexes of the agents array to boost the performance of the
spawnAgentsKernel
    int* availableIndexesArray = new int[maxAgents];
    size_t size_availableIndexesArray = sizeof(int) * maxAgents;
    cudaMalloc(&availableIndexesArray, size_availableIndexesArray);
    cudaMemset(availableIndexesArray, -1, size_availableIndexesArray);
    countAvailableIndexKernel<<<dims[0], dims[1]>>>(a_ptrs, maxAgents, availableIndexesArray);
    CHECK();

    // Reduce the availableIndexesArray to get the number of available indexes
    thrust::device_ptr<int> dev_availableIndexPtr(availableIndexesArray);
    thrust::device_ptr<int> new_end;
    new_end = thrust::remove(dev_availableIndexPtr, dev_availableIndexPtr + maxAgents, -1);
    int nAvailableIndexes = new_end - dev_availableIndexPtr;
    logger::debug("Number of available indexes: %d", nAvailableIndexes);
    logger::debug("Number of max agents: %d", maxAgents);

    // Pass this information to spawnAgentsKernel to boost the performance
    int* nAliveAgents = new int (maxAgents - nAvailableIndexes);
    int* startIdx;
    int* h_startIdx = new int(0);
    CATCH(cudaMalloc(&startIdx, sizeof(int)));
    CATCH(cudaMemcpy(startIdx, h_startIdx, sizeof(int), H2D));
    int* availableIndexesArrayReduced = new int[nAvailableIndexes];
    cudaMalloc(&availableIndexesArrayReduced, sizeof(int) * nAvailableIndexes);
    cudaMemcpy(availableIndexesArrayReduced, availableIndexesArray, sizeof(int) * nAvailableIndexes,
D2D);
    CHECK();

    // Launch the spawnAgentsKernel
    spawnAgentsKernel<<<dims[0], dims[1]>>>(
        a_ptrs, maxAgents, availableIndexesArrayReduced, nAvailableIndexes, startIdx
    );
    CHECK();

    // Copy back the startIdx to calculate the new number of agents
    CATCH(cudaMemcpy(h_startIdx, startIdx, sizeof(int), D2H));
    if (*h_startIdx > nAvailableIndexes) {
        throw MassException("Trying to spawn more agents than the maximun set for the system");
    }

    deviceInfo->devAgentsMap[handle].nAgents += *h_startIdx;
    // deviceInfo->devAgentsMap[handle].nextIdx += *h_startIdx;

    // Clean up
    delete nAliveAgents;
    delete h_startIdx;
    CATCH(cudaFree(availableIndexesArray));
    CATCH(cudaFree(availableIndexesArrayReduced));
    CATCH(cudaFree(startIdx));
}

__global__ void countAvailableIndexKernel(Agent **ptrs, int maxAgents, int* availableIndexes) {
    int idx = getGlobalIdx_1D_1D();

    if(idx < maxAgents && !ptrs[idx]->isAlive()) {
        availableIndexes[idx] = idx;
    }
```

```
}

__global__ void spawnAgentsKernel(
    Agent **ptrs, int maxAgents, int* availableIndexesArray, int numAvailableIndexes, int* startIdx)
{
    int idx = getGlobalIdx_1D_1D();

    if (idx < maxAgents) {
        int numChildren = ptrs[idx]->state->nChildren;
        // If the agent is alive and has children to spawn:
        if ((ptrs[idx]->isAlive()) && (numChildren > 0)) {
            // Find a spot in Agents array:
            int idxStart = atomicAdd(startIdx, numChildren);
            if (idxStart+numChildren > numAvailableIndexes) {
                return;
            }

            for (int i=0; i< numChildren; i++) {
                int childIdx = availableIndexesArray[idxStart + i];
                // Instantiate with proper index
                ptrs[childIdx]->setAlive();
                // Link to a place
                ptrs[childIdx] -> setPlace(ptrs[idx]->state->childPlace);
                ptrs[idx]->state->childPlace -> addAgent(ptrs[childIdx]);
            }

            // restore Agent spawning data:
            ptrs[idx]->state->nChildren = 0;
            ptrs[idx]->state->childPlace = NULL;
        }
    }
}
```

Place::resolveMigrationConflicts()

Description: This function resolves migration conflicts by selecting agents with the lowest index when multiple agents attempt to migrate to the same place.

Linked Section: Section 4.2.2

Implementation:

```
MASS_FUNCTION void Place::resolveMigrationConflicts() {
    // The real algorithm is simply sort the potentialNextAgents array and
    // select the first N agents
    // @note by Warren: currently using the simply insertion sort
    // which is efficient for small array
    // if we design to allow large number of agents to reside in a place
    // we should use more efficient sorting algorithm
    for (int i = 1; i < N_DESTINATIONS; i ++) {
        Agent* key = state->potentialNextAgents[i];
        int j = i - 1;

        while (j >= 0 && (state->potentialNextAgents[j] == NULL ||
               (key != NULL && state->potentialNextAgents[j]->getIndex() > key->getIndex()))) {
            state->potentialNextAgents[j + 1] = state->potentialNextAgents[j];
            j = j - 1;
        }
```

```
            state->potentialNextAgents[j + 1] = key;
        }

        if (MAX_AGENTS == 1) {
            // If only 1 agent can reside in a place
            // we select the agent with the lowest index
            if (state->potentialNextAgents[0] != NULL) {
                addAgent(state->potentialNextAgents[0]);
            }
        }
        else {
            // If more than 1 Agent can reside in a place, we select the fist N Agents
            // with the lower index
            for (int i = 0; i < MAX_AGENTS; i++) {
                if (state->potentialNextAgents[i] != NULL) {
                    addAgent(state->potentialNextAgents[i]);
                }
                else{
                    break; // Early termination
                }
            }
        }

        // Clean potentialNextAgents array
        for (int i = 0; i < N_DESTINATIONS; i ++) {
            state->potentialNextAgents[i] = NULL;
        }
        // Reset numPotentialNextAgents
        *state->numPotentialNextAgents = 0;
}
```

8.2 How to Run

The code for the MASS CUDA library is available at Bitbucket. Detailed instructions on how to set up and run the code can be found in the README file in the repository. This includes information on installation, dependencies, and execution of the library.

References

[1] N. Hart, "MASS CUDA: Parallel-Computing Library for Multi-Agent Spatial," 2014. [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCuda.pdf. [Accessed 11 December 2023].

[2] B. Luger, "Parallelization of Agent-based Models over," 16 March 2023. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BrianLuger_wi23.pdf. [Accessed 9 December 2023].

[3] B. Luger, "Dispatcher.cu," [Online]. Available: https://bitbucket.org/mass_library_developers/mass_cuda_core/src/v0.6.0/src/Dispatcher.cu#lines-291.

[4] E. S. W. K. H. A. Munehiro Fukuda, "CDS&E:small:Agent-Based Parallelization of Micro-Simulation and Spatial Data Analysis".

[5] NVIDIA, "Thurst 12.3 documentation," NVIDIA, 10 October 2023. [Online]. Available: https://docs.nvidia.com/cuda/thrust/index.html#sorting. [Accessed 10 December 2023].