Multi-GPU Parallelized Agent-Based Modeling

Holt Ogden

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2025

Reading Committee:
Professor Munehiro Fukuda, Chair
Professor Michael Stiber
Professor Robert Dimpsey

Program Authorized to Offer Degree:
Computer Science & Software Engineering

University of Washington

Abstract

Multi-GPU Parallelized Agent-Based Modeling

Holt Ogden

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
School of STEM, Division of Computing & Software Systems Faculty

Agent-based modeling (ABM) is a method for simulating emergent behaviors in complex systems by modeling the interactions of individual agents. These simulations often require substantial computational resources, necessitating increased parallelization and spatial scalability to produce usable results. One approach is to use a computer's Graphics Processing Unit (GPU) to allow increased parallelization by utilizing the greater number of threads available on the GPU compared to the central processing unit. The Multi-Agent Spatial Simulation (MASS) library for NVIDIA's Compute Unified Device Architecture (CUDA) provides a platform that allows users to write ABM programs to run on the GPU.

However, these programs are limited in their spatial scalability by the memory available on a single graphics card. In this project, we improved the MASS CUDA library's Place object implementation by extending it to function over multiple GPUs connected via NVIDIA NVLink, increasing the potential simulation size of MASS CUDA programs. This required splitting ABM data between multiple graphics cards, ensuring memory synchronization between the cards, and recombining result data at the end of the simulation. These changes improved the runtime of some benchmark programs by 32% and increased the maximum simulation size by 77%. In addition, these changes were designed to be abstracted from the user so that minimal changes are required by ABM programs written for previous single-GPU versions of MASS CUDA. Overall, this project significantly expands the compu-

tational resources available to the MASS CUDA library, allowing the running of larger and more complex ABM programs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Agent based modeling (ABM) is a method for simulating emergent behaviors in complex systems by modeling the interactions of individual agents. Agents are autonomous objects that are programmed with a relatively simple set of rules, allowing them to interact and exchange information with their environment and each other. Using these rules, a simulation can model the large-scale emergent behavior of millions of agents starting with only a basic understanding of individual behavior.

ABM often requires a huge number of agents to produce usable data, and therefore simulating these agents demands significant computing resources. The number of agents within an agent-based model is limited by the runtime of the simulation and the available memory to perform the simulation. Increasing the size of the simulation requires improved processing speeds and increased spatial scalability of the model. To accomplish this, agent-based models often utilize strategies such as parallel computing to improve runtimes and distributed computing to improve spatial scalability.

One specific strategy is to use a computer's Graphics Processing Unit (GPU) to allow increased parallelization by harnessing the greater number of threads available on the GPU compared to the Central Processing Unit (CPU). CPUs typically support between 4 and 64 threads at once, while GPUs can have thousands of threads running simultaneously [1]. Using NVIDIA's Compute Unified Device Architecture (CUDA), programmers can access GPU resources for General Purpose computing on GPUs (GPGPU). CUDA therefore allows a simulation, such as ABM, to utilize the resources of the GPU. However, writing software to run on the GPU can be a difficult and time-consuming process that requires specialized programming knowledge. CUDA is a language extension of C++ that requires knowledge of

C programming, memory management, and GPU architecture to be used effectively.

The Multi-Agent Spatial Simulation (MASS) CUDA library provides a method for accessing the GPU's resources to run ABM without needing to understand the underlying CUDA code. Users can set up agents, their simulation environments, and simulation data through the MASS CUDA API, and then all of the code to run the simulation on the GPU is handled in the back end [2]. This allows users to focus on writing their ABM programs without worrying about the complexities of CUDA programming.

While the MASS CUDA library enables increased parallelization of ABM programs using the GPU, the scalability of these programs is still limited by the available memory on the GPU. ABM simulations that exceed the memory of a single GPU will likely crash, or may encounter severe performance limitations due to the time cost of repeatedly copying data from the CPU memory to the GPU memory. This effectively limits the size of the ABM programs to the memory size of a single GPU; for example, the NVIDIA RTX A5000 used by UW Bothell's Juno lab computers has 24 GB of available memory.

To address this problem, our work extends the MASS CUDA library to run over multiple GPUs connected via NVIDIA NVLink, increasing the potential simulation size of MASS CUDA programs. NVLink provides a direct physical connection between multiple GPUs, allowing them to exchange data more quickly between each other compared to transferring data from the GPU to the CPU memory. For instance, the UW Bothell's Juno lab computers provide approximately 32 GB/s unidirectional transfer speeds from CPU to GPU over PCIe, while their NVLink Version 4 bridge allows 56 GB/s. However, each GPU is still a separate device with its own memory, and therefore requires careful memory management to split a problem over two memory locations. Our work modifies the MASS CUDA library to split the data evenly between the GPUs, manage synchronization between them during the simulation, and recombine data at the end of the simulation. This project focused on the implementation of multi-GPU MASS CUDA for the `Place` objects within MASS CUDA; future work will provide an `Agent` object implementation.

Chapter 2
BACKGROUND

This section introduces the overall MASS library and the MASS CUDA library. It explains the basic functionality of the MASS library, the motivations for porting MASS to run using CUDA, previous work on multi-GPU MASS CUDA, and recent work on the MASS CUDA library.

2.1 MASS Library

The MASS library is designed to allow for performance-enhancement and abstraction of agent-based modeling using both distributed computing and parallel computing strategies. It is based on two key components: places and agents. A `Place` object, or place, is a location in memory that holds data or attributes, and can also host `Agent` objects, or agents. Places can exchange data between each other or allow residing agents to access their data. Agents are execution instances that are located upon a place. They contain their own data and also instructions that dictate their behavior. Agents can "jump" between places, effectively transferring an execution instance from one location to another. New agents can also be spawned or terminated during the simulation. This allows agents and their data to spawn, move between places, and terminate based on user-defined behavior.

Users of the MASS library write programs by extending the `Place` and `Agent` classes into their own subclasses and defining the behavior of these classes. For instance, in the Multi-Agent Transport Simulation (MATSim) program, `Place` is extended to `IntersectionPlace` in order to represent intersections, and `Agent` is extended to `CarAgent` to represent cars on the road. Users then define functions that are called on all places or agents simultaneously, known as `callAll()` functions. For instance, in the MATSim program a user might call a function

such as "`MoveCarsIntoIntersection`" on all `CarAgent` objects. By calling functions to run on both places and agents, users can simulate complex emergent behavior in the simulation using relatively simple commands and run functions on millions of places or agents in parallel.

The MASS library also provides tools to parallelize the ABM program. In MASS versions written in C++ and Java, the MASS library can be run on a cluster of computing nodes using multiple threads within those nodes, enhancing both spatial scalability and parallelization. The library handles data connections between nodes and data synchronization during the simulation, abstracting distributed and parallel computing elements away from the end user. This allows users to write ABM programs that run in parallel across multiple computing nodes without any knowledge of distributed computing.

2.2 MASS CUDA

The MASS CUDA library modifies the MASS library to take advantage of the performance capabilities of GPUs. In this version of MASS, places, agents, and their associated data are loaded into a GPU's memory and their `callAll()` functions executed in parallel, taking advantage of the thousands of cores within modern GPUs. The library handles copying of data between the memory controlled by the CPU (the host) and the GPU memory used by the GPU (the device). Upon execution of place or agent `callAll()` functions, each place or agent is run on a separate thread using a CUDA kernel call, allowing for massive parallelization of ABM programs [2].

MASS CUDA has been under development at the University of Washington, Bothell since 2012, with continuous improvements made since then. Early efforts in this development have shown promising results, with MASS CUDA achieving up to 3.9 times faster runtimes of benchmarks compared to sequential execution on the CPU [3]. Recent efforts have focused primarily on two areas of improvement: improvements to the base single-GPU library and attempts to implement multi-GPU MASS CUDA.

2.2.1 Recent Updates to MASS CUDA Library

The most recent incremental improvement to the MASS CUDA library was completed by Warren Liu in the Spring of 2024. He fixed issues with the agent implementation, reorganized the library, and improved documentation. Most critically for this paper, his single-GPU MASS CUDA implementation implemented performance enhancements to bring MASS CUDA closer to the performance of other ABM libraries [4].

For the purposes of this term paper, which focuses only on the implementation of places within the MASS CUDA library, the primary change made by Warren's project was the complete rewrite of how place data members are created, stored, and accessed by both the host and the device. Previously, data was stored as two separate objects: `Place` and `PlaceState`. `Place` held all the functionality of a MASS place; its accessor functions for data, functions to add and remove agents, and the `callAll()` function that can call user-defined functions. A `Place` object then contained a pointer to a `PlaceState` object, which stored any data members used by the `Place`, such as its neighbors, its index, and any agents residing there. A user implementing a MASS program would extend the `Place` and `PlaceState` objects by adding their own functions to their `UserPlace` and their own data members to their `UserPlaceState` objects. Arrays of `UserPlace` and `UserPlaceState` objects would be copied to the device and used to run MASS simulations.

However, this implementation led to performance problems due to uncoalesced memory access within the `PlaceState`. Storing arrays of `PlaceState` objects on the device meant that accessing the same data member for two adjacent places, such as their index, required accessing widely separated memory locations. Because a place's `callAll()` function operates on the same data members for all places simultaneously, each `Place` object is therefore making suboptimal memory accesses with this implementation.

To fix this problem, the MASS CUDA library was rewritten to remove the `PlaceState` object and change all data members to attributes of the `Place` object, as shown in Figure 2.1. This means that each data member is stored in a contiguous array of attributes for all places,

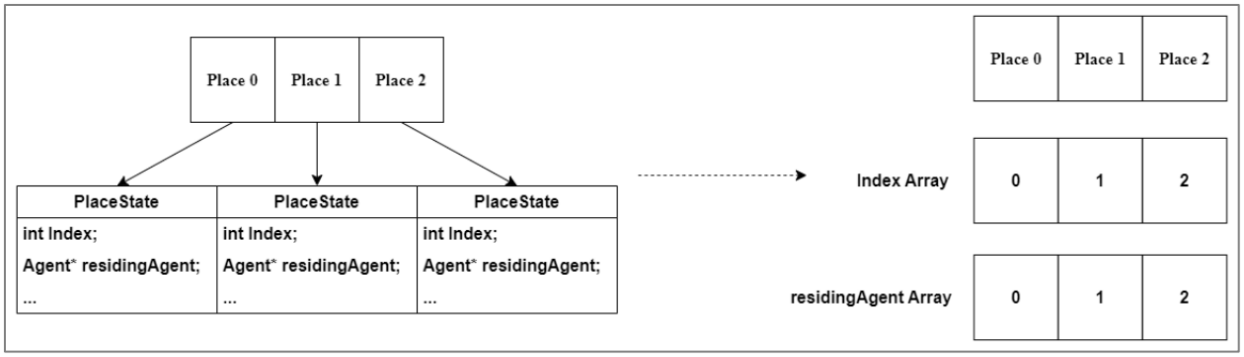

Figure 2.1: Removal of `PlaceState` and change to attribute arrays.[4]. The configuration on the left shows the previous implementation of place memory as an array of `Place` objects pointing to an array of `PlaceState` structs. The right shows the updated configuration with separate arrays for each attribute. This allows for memory for each attribute to be accessed as contiguous blocks upon kernel calls.

with attribute index `i` corresponding to the data member for `Place i`. Effectively, data was reorganized from being stored from an array of structures to a structure of arrays. This allows data to be stored and accessed more efficiently within CUDA threads. These changes led to significant improvements in MASS CUDA performance, with benchmark programs such as Game of Life taking 85% less time to run than in previous versions of MASS CUDA. However, benchmark programs such as Heat2D and SugarScape still showed slower performance than other ABM libraries, demonstrating the need for further improvements [4].

2.2.2 Previous Work on Multi-GPU MASS CUDA

Several students within the MASS lab have previously created implementations of multi-GPU MASS CUDA. These include Nathaniel Hart, Lisa Kosiachenko, and Ben Pittman [5, 6]. The most recent and complete implementation was done by Ben Pittman in the Summer of 2021, who updated MASS places and agents to run over multiple GPUs and then tested this by updating the benchmark programs Sugarscape and Braingrid to run on

his library [6]. However, the library encountered difficulties achieving the correct results with both benchmark programs, and therefore did not create a fully functional multi-GPU implementation.

The implementation of multi-GPU MASS CUDA described in this paper builds upon some elements of this previous work, such as using OpenMP to make CUDA calls on each GPU within its own thread simultaneously, and ghost places within each device's place array [6]. However, the multi-GPU implementation described in this paper is primarily built upon Warren Liu's single-GPU library. This is mostly due to the fact that the removal of the `PlaceState` and the new attributes system made much of the previous work no longer applicable to the current MASS CUDA library.

Chapter 3
RELATED WORK

There are several publicly available libraries comparable to MASS that allow users to write programs for agent-based modeling. These include Multi-Agent Simulator of Neighborhoods (MASON), NetLogo, and Recursive Porous Agent Simulation Toolkit (Repast). However, there are fewer libraries that focus specifically on ABM on the GPU; the primary competitor for MASS CUDA in the GPU-accelerated ABM space is Flexible Large-scale Agent Modeling Environment (FLAME) GPU2. This section reviews each ABM library and compares them to MASS and MASS CUDA.

3.1 CPU-Based ABM Libraries

MASON is a multi-agent simulation library developed by George Mason University. It is written in Java and designed to support large-scale, discrete-time simulations involving thousands to millions of agents. MASON is designed with the primary goals of efficiency, fault-tolerance, and scalability. It uses custom-written Java libraries to decrease runtimes and includes checkpointing procedures to save work during simulations. Unlike MASS, MASON is primarily written to run on a single high-performance computer, or even supercomputers for large jobs. However, it also has a Distributed MASON extension that allows for cluster deployment, similar to the MASS Java implementation. MASON is a mature and widely used ABM platform, with a project manual that spans more than 450 pages [7].

NetLogo is an open source agent-based modeling environment developed by Northwestern University. It is written in Java and was designed to be useful both as a teaching tool and as a powerful research platform. It is similar to MASS in that it includes both stationary agents (called patches) analogous to MASS places and mobile agents (called turtles). In addition,

it supports specialized link agents that can be used to create graphs and networks. Additional features include an interface builder, built-in language primitives, and an extensions library that supports additional languages, like Python and R. While NetLogo includes some extensions and scripts that allow parallelization, it primarily runs in a single thread on the CPU[8].

Repast Suite is a group of ABM platforms originally developed by the University of Chicago, expanded by the Argonne National Laboratory, and now maintained by the Repast Organization for Architecture and Design (ROAD). The Repast Suite includes Repast Simphony, a Java-based modeling toolkit, Repast for High Performance Computing (HPC), and Repast for Python. Repast is designed for high-performance simulations with detailed agent behavior and supports features like GIS integration, social network modeling, and batch experimentation. Repast Simphony and Repast HPC both support cluster computing [9, 10].

3.2 GPU-Based ABM Libraries

While there are several widely used platforms that support CPU or cluster-based ABM, there are considerably fewer platforms for running ABM on the GPU. This may be due to the relatively recent availability of GPGPU platforms; CUDA was first released in 2007, while MASON, NetLogo, and Repast were in development in the late 1990s and were released between 1999 and 2003 [8, 9, 10, 11].

The primary competitor for MASS CUDA in the GPU-based ABM space is FLAME GPU2, a GPU-enabled ABM developed by the University of Sheffield. It supports GPU-based ABM modeling as part of the core library, including abstractions to allow inexperienced programmers to utilize the GPU without understanding parallel programming. Previous capstone projects within the MASS lab have used FLAME GPU2 as a comparison to test MASS CUDA against another real-world project. At this time FLAME GPU2 does not support multi-GPU ABM, but there are plans to support it in the future [12].

3.3 Multi-GPU ABM

Currently, no libraries could be identified that provide direct platform support for multi-GPU ABM, but there have been individual multi-GPU implementations applied to specific ABM scenarios. Individual researchers have implemented multi-GPU support for the MASON library, achieving 187 times faster runtime performance of the Boids simulation compared to CPU-based MASON [13]. However, this paper only compared multi-GPU implementations to CPU-based implementations and was a one-off test for a specific problem instead of generalized library support. In another example, researchers from Virginia Commonwealth University implemented agent-based blood coagulation simulations using NetLogo, Repast, non-parallelized C code, and CUDA code to compare runtime executions, showing 10-300 time runtime speedups for CUDA implementations compared to ABM and non-parallelized implementations [14]. However, this simulation used ABM and multi-GPU modeling but did not combine them together.

3.4 Motivation

While there are several libraries that support CPU-based or distributed Agent-Based Modeling, we have not been able to identify any current implementations of a general multi-GPU ABM library. Therefore, the multi-GPU MASS CUDA presented in this report presents an opportunity to develop a new approach to increasing the performance of ABM models. Multi-GPU MASS CUDA offers a new general method to increase the spatial scalability of ABM simulations using multiple GPUs, without requiring the user to understand CUDA coding. It therefore can enable users to create larger, more complex agent-based models using the existing MASS CUDA platform.

Chapter 4

IMPLEMENTATION

This section contains an overview of the MASS CUDA library structure and the key functions used to run the ABM simulation. It reviews the functions that are used to initialize MASS CUDA, create places, and utilize attributes, looking specifically at how these functions were modified to support multiple devices. In addition, it explains the use of ghost places to ensure data synchronization between devices.

4.1 System Overview

The MASS CUDA library is structured with a command-dispatcher model, such that all classes pass their main function calls through the `Dispatcher` object. This functionality is shown in the class diagram in Figure 4.1. This model ensures that all classes have access to the required place data, and also allows the `Places` and `Mass` classes to have their functionality abstracted from the underlying CUDA implementation. A further level of abstraction is defined in the `DeviceConfig` object. The `DeviceConfig` stores information on the available GPUs and their settings, and manages the background infrastructure related to splitting places across multiple devices. To do this, it stores a struct called a `PlaceArray` for each `Places` object. This struct manages all the pointers to the devices and also the data required to track ghost places.

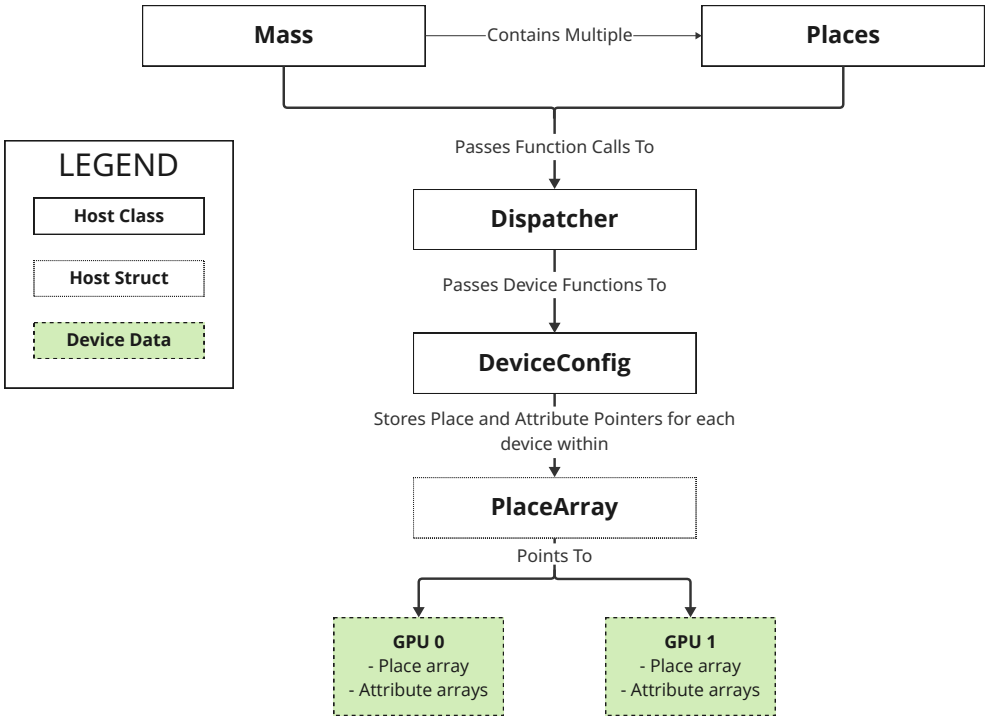

Figure 4.1: Multi-GPU MASS CUDA class diagram. Shows how all functions from `Mass` and `Places` classes are passed to the `Dispatcher`, which manages all MASS function calls. `Dispatcher` passes any device-related functions to the `DeviceConfig`, which manages the GPUs using `PlaceArray` structs to store data. These structs contain data and pointers for MASS places and attributes for each device.

4.2 Initialization

When the MASS library is initialized, the following steps are followed:

1. Determine the number of devices using `cudaGetDeviceCount(int* count)`. This returns the total number of devices with compute capability (CC) greater than or equal to 2.0. Compute capability is a version label that defines the CUDA features and capabilities for GPU hardware.

2. Create a `devices` vector to store each device ID in the form of an int.

3. Determine which device has the highest compute capability by calling the code shown in Listing 4.1.

Listing 4.1: Determination of highest CC device. This gets the properties of each device using `cudaGetDeviceProperties()`, then gets the `prop.major` and `prop.minor` properties to determine CC.

```
1   int bestDevice = 0;
2   int bestCC = 0;
3
4   for (int i = 0; i < gpuCount; i++) {
5       cudaDeviceProp prop;
6       cudaGetDeviceProperties(&prop, i);
7       int cc = prop.major * 10 + prop.minor;
8       if (cc > bestCC) {
9           bestCC = cc;
10          bestDevice = i;
11      }
12  }
```

4. Add the device ID of the highest compute capability device to the `devices` vector. If there is only one device, this device will be added to the vector by default.

5. For every other device, check if that device has the capability to access the default device, and vis-versa using `cudaDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice)`. If a device can access the default device, add it to the `devices` vector.

6. Enable bi-directional peer memory access between every device in the `devices` vector. For each device, call `cudaDeviceEnablePeerAccess(int peerDevice, unsigned int flags)` with every other peer device using nested loops.

7. Set up OpenMP access to call CUDA functions in parallel by assigning each device its own host thread using the code shown in Listing 4.2 .

Listing 4.2: Setup of OpenMP and mapping of OpenMP threads to devices. This ensures that there is one host OpenMP thread for each device and that each host thread can check what its current device is by calling `cudaGetDevice()`.

```
1  omp_set_dynamic(0);
2  omp_set_num_threads(devices.size());
3  #pragma omp parallel
4  {
5      int gpu_id = -1;
6      const int thread_id = omp_get_thread_num();
7      CATCH(cudaSetDevice(thread_id));
8      CATCH(cudaGetDevice(&gpu_id));
9  }
```

After this setup is complete, any subsequent device functions, such as launching kernels or copying data to the device, can be wrapped in `#pragma omp parallel` segments, and then `cudaGetDevice(&gpu_id)` can be used to determine which device is being managed by the current thread. In addition, the host can now copy data directly between the devices using `cudaMemcpyPeerAsync(void* dst, int dstDevice, const void* src, int srcDevice, size_t count, cudaStream_t stream = 0)`.

4.3 Place Creation

To create places on the device, the user calls the `Mass::createPlaces(int handle, int dimensions, int size[], int majorType)` function. This function creates an overall array of places of size M x N, as defined in the `size` array parameter. This place array must be evenly split between two devices. However, this split must first define the boundary between the two devices and the method for communicating between GPUs at that boundary.

While the two GPUs have separate memory and run separate CUDA kernels, their places must be able to communicate with each other across the GPU boundary. There are several ways to do this. One method would be to simply wait until a place needs to access another place's data, check if that data is on another GPU, and then run `cudaMemcpyPeerAsync()`

to copy the data across devices. However, copying data repeatedly for each place would lead to thousands of tiny data transfers in each simulation step, leading to inefficient movement of data.

To solve this issue, we use ghost places to ensure synchronization and memory access between devices. This approach creates a boundary layer of ghost places between GPUs which are copies of the base (non-ghost) places on adjacent GPUs. This can be seen as the green places shown in Figure 4.2. These ghost places hold the same data as their complimentary base places, but do not run `callAll()` functions. The purpose of these ghost places is that when a base place that is on the boundary of its GPU wants to exchange information with an adjacent place on another GPU, it can directly access the ghost place on its GPU that stores all that data, rather than having to access another GPU. The data within the ghost place is kept updated using the boundary communication described in Section 4.9.

The `createPlaces()` function must calculate the number of base and ghost places and save this data so the place array can be recombined later. To do this, `DeviceConfig` creates a new `PlaceArray` struct to store all data and pointers for places with this handle. Once this is setup, the function determines the number of base places and ghost places for each device using the formulas shown in Listing 4.3, which are also illustrated in Figure 4.2. The number of base places, called `basePlacesPerStride`, is determined by dividing the total number of rows in the original place array by the number of GPUs then multiplying by total columns in the original place array, such that `basePlacesPerStride = (total place rows / numDevices) * total place columns`. This can be seen in line 1 of Listing 4.3.

Listing 4.3: Calculation of base and ghost place quantities. First, the `basePlacesPerStride` is determined by dividing the number of rows by the number of devices. Next, the `maxTravel` is found by taking the maximum of `MAX_PLACE_EXCHANGE` and `MAX_AGENT_TRAVEL` values set by the user. Then, for each device the number of ghost places before and after is calculated taking the number of ghost rows (which equals `maxTravel`) and multiplying by the number of columns in the place array. If a device is the first device, it has no ghost places before, while the last device has no ghost places after. Finally, the ghost places before, base places, and ghost places after are added to get the total `placeStride`, or number of places on this device.

```
1   int basePlacesPerStride = (p.dims[0] / activeDevices.size()) * p.dims[1];
2
3   int maxTravel = std::max(MAX_PLACE_EXCHANGE, MAX_AGENT_TRAVEL);
4
5   #pragma omp parallel
6   {
7       int gpu_id = -1;
8       CATCH(cudaGetDevice(&gpu_id));
9
10      if (gpu_id == 0) {
11          p.ghostPlacesBefore[gpu_id] = 0; // First device will have no ghost places at start
12      } else {
13          p.ghostPlacesBefore[gpu_id] = maxTravel * p.dims[1];
14      }
15
16      if (gpu_id == activeDevices.size() - 1) {
17          p.ghostPlacesAfter[gpu_id] = 0; // Last device will have no ghost places at end
18      } else {
19          p.ghostPlacesAfter[gpu_id] = maxTravel * p.dims[1];
20      }
21
22      p.placeStride[gpu_id] = p.ghostPlacesBefore[gpu_id] + basePlacesPerStride + p.
            ghostPlacesAfter[gpu_id];
23  }
```

Next, the function determines the number of ghost places before and after each device's base places. This is based on two numbers: the maximum number of places an agent can move across in one simulation step, and the maximum rows away a place can exchange information.

These values are defined by the user as `MAX_AGENT_TRAVEL` and `MAX_PLACE_EXCHANGE` in the settings.h file. For instance, if the user's simulation allows an agent to move three places away at once, and allows places to exchange data up to four rows away, then `createPlaces()` will take the maximum of these two values and create four ghost rows of ghost places between each device. This is seen in line 3 of Listing 4.3.

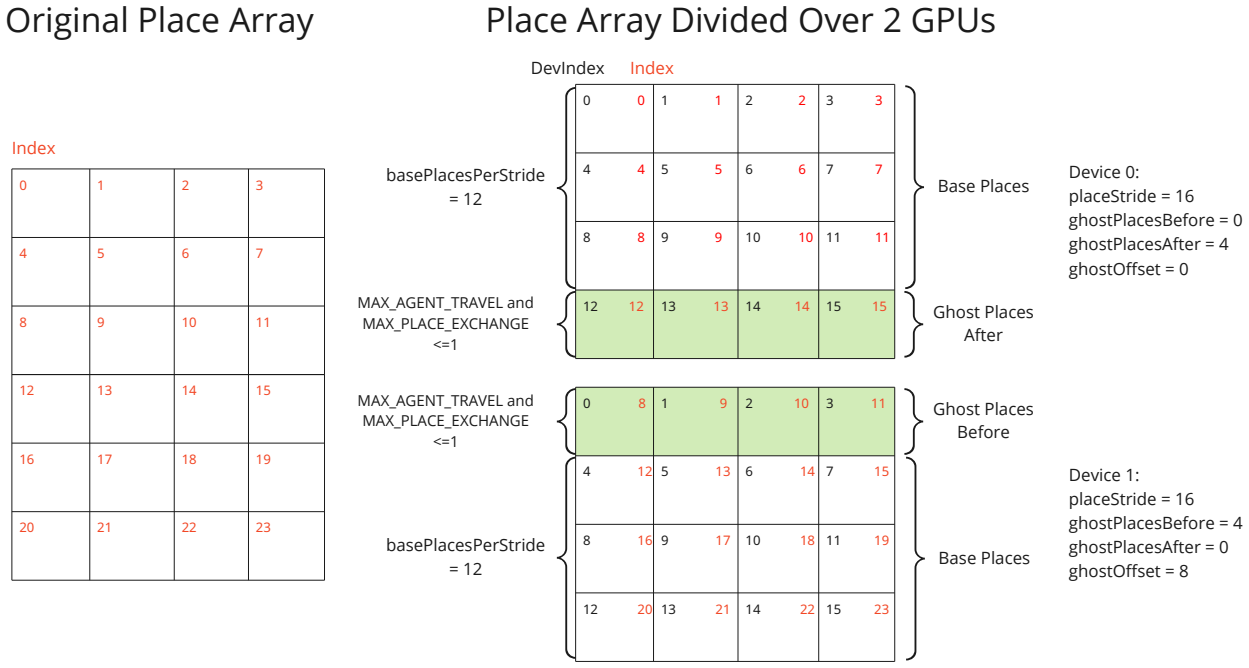

Figure 4.2: Creation of Place array distributed over two GPUs. The `basePlacePerStride` value represents the base place on each device; if there are 24 initial base places, each device will have 12 base places. Ghost places are defined by the `MAX_AGENT_TRAVEL` and `MAX_PLACE_EXCHANGE` values; if both have a value of one then there will be one row of ghost places before and after (except the first and last devices). The `placeStride` stores the total places per device of both types. Finally, `ghostOffset` represents the difference between the `DevIndex` (in black text on each place) and the `Index` (in red text).

For each device in parallel, the function then calculates the number of ghost places before the base places (0 if the first device) and the ghost places after (0 if the last device), as

seen in lines 5-20 of Listing 4.3, and on the right side of Figure 4.2. These are added to the `basePlacesPerStride` to get the total places on each device, as seen in line 22 of Listing 4.3. In addition, a `ghostOffset` value is calculated for each device. This value determines the difference between each place's original `index` in the base place array and its new CUDA thread `idx` within the device it is assigned to, called `devIndex`. Using this offset value, when the place is initiated it can calculate its original `index` in the base array and store this value. This is seen in Figure 4.2, where the original `index` is in red for each place, the `devIndex` is in black, and the offset is the difference between them shown for each device. The total number of places on a device is called its `placeStride` and equals the `ghostPlacesBefore + base places + ghostPlacesAfter`. This is equivalent to the overall place count value in a single-GPU operation and is used within kernel operations to ensure that threads whose `idx` is greater than the `placeStride` value do not perform any operations. In addition, the kernel dimensions to determine the size of blocks and threads when running kernels are based on the `placeStride` of the devices.

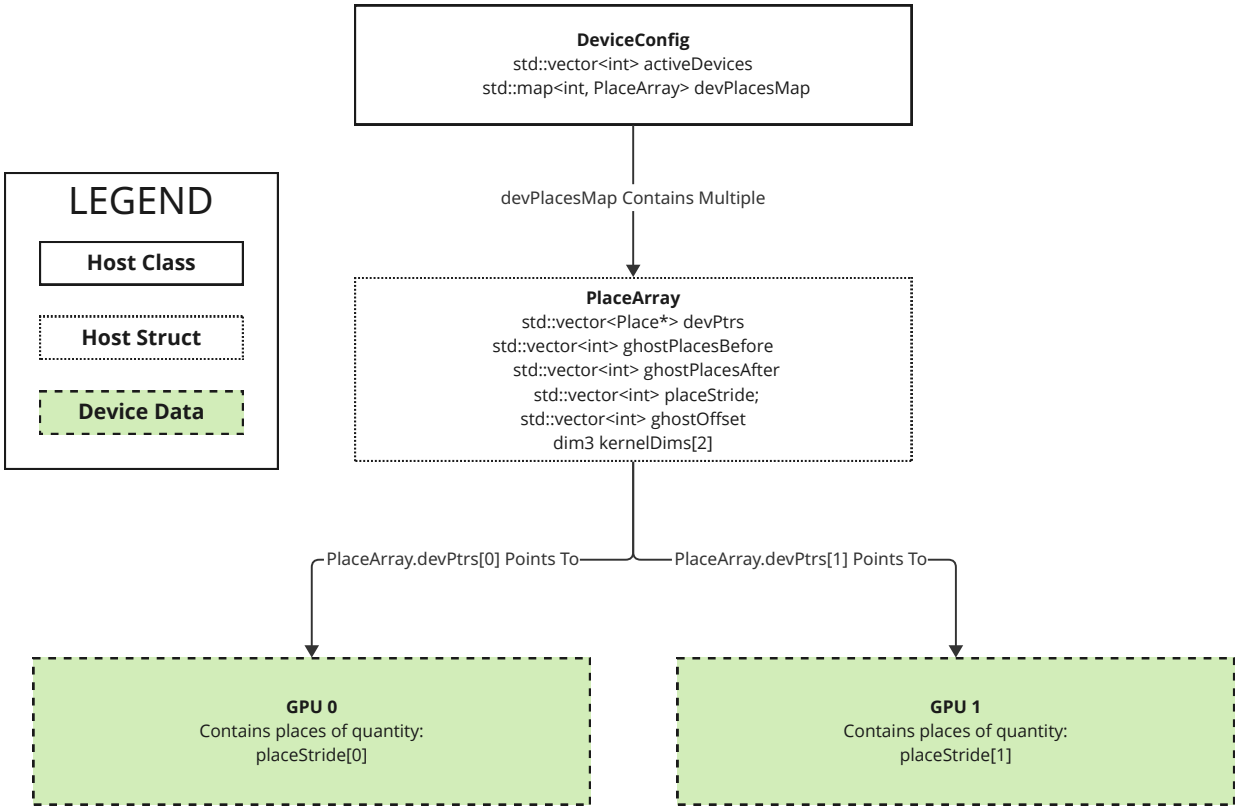

Figure 4.3: PlaceArray creation and key data members. This shows an expanded view of Figure 4.1 with key data members included. Each `DeviceConfig` stores a vector containing a list of active devices, as well as a map that links place handles to its corresponding `PlaceArray`. Each `PlaceArray` stores a series of vectors, where each index in that vector represents the values calculated in Listing 4.3 for device i. The `devPtrs` vector contains pointers to the `Place` arrays allocated on each device.

Once the numbers of places are determined, the `createPlaces()` function allocates memory on each device and initializes `Place` objects on them. Figure 4.3, which represents a more detailed view of the class diagram in Figure 4.1, shows how `PlaceArray` was modified to contain vectors of pointers to the device memory instead of a single pointer; each device is represented by an index in this vector. Similarly, the ghost places and offsets that were

previously calculated are stored for each device. Using multiple threads, the function then allocates memory on each device `N` of `size PlaceType * placeStride[N]`. Next each device calls a kernel function that creates a new `Place` at each location. This is seen in Listing 4.4, which gets the `idx` of the current thread, checks if the `idx` is less than the `placeStride`, then determines if the place at this `idx` is a ghost or base place. If it is a ghost place, then the function creates a new `Place` with the data member `basePlace` set to false, while base places will be set to true. Both base and ghost places are created with an `Index` and a `devIndex` value. While ghost places did not have an `Index` within the original place array, they receive an `Index` that corresponds to the `Index` of the base place they are duplicating.

Listing 4.4: Place creation kernel. This kernel creates Place objects in device memory, separating them into base and ghost places using the `idx` (which corresponds to the `DevIndex`), `placeStride`, `ghostPlaceBefore`, and `ghostPlaceAfter` values.

```
1   unsigned idx = getGlobalIdx_1D_1D();
2   if (idx < placeStride) {
3       // Place is a ghost Place
4       if (idx < ghostPlaceBefore || idx >= placeStride - ghostPlaceAfter) {
5           new (&places[idx]) PlaceType(idx + offset, idx, false);
6       }
7       // Place is a base Place
8       else {
9           new (&places[idx]) PlaceType(idx + offset, idx, true);
10      }
11  }
```

Upon completion of `createPlaces()`, each device will have an array of `Place` objects of size `placeStride` initialized where each `Place` stores its Index and whether it is a base place or a ghost place. In addition, `DeviceConfig` now has pointers to each array of places stored on each device.

4.4 Attribute Creation

Since places are now split over multiple devices, the attributes associated with those places must also be split. Similar to the previous single-GPU MASS CUDA code, creating attributes for places is a two-step process. First, the user calls `Places::setAttribute()` for each attribute they intend to use. This allocates memory for those attributes on the device and can also set a default value if provided by the user. The second step is calling `finalizeAttribute()`, which transfers the array of attribute pointers to the device and provides each Place object with pointers to its attributes. A detailed description of `finalizeAttribute()` is described in Section 4.5.

Data for attributes is stored within the corresponding `Places` object as three vectors: `vector<int> attributeTags`, `vector<void**> attributeDevPtrs`, and `vector<size_t attributePitch`, as shown in Figure 4.4. Each index within these vectors represent a different attribute, with the tag storing a user-enumerated `int` label for the attribute, the device pointers storing an array of size N GPUs, each with a `void*` to where the attribute is stored on that device, and pitch storing the byte offset for storing two-dimensional attributes on the device. This is a CUDA functionality that allows two-dimensional attributes to have their data aligned in memory for faster access. The vector `attributeDevPtrs` was modified from the single-GPU version to store a `void**` instead of `void*` so that it can store an outer array to hold the pointers to the attribute memory on `each` device. Every time `setAttribute()` is called, it uses a parallel section to allocate memory on each device of size `placeStride[gpu_id] * sizeOf(AttributeObject)`, then stores the pointer to that memory for each device within a `void**` array. Finally, the `Places` object pushes the tag, device pointer array, and pitch back onto the respective vector to save those values within the `Places` object. These values are only saved within the `Places` object temporarily until `finalizeAttribute()` is called.

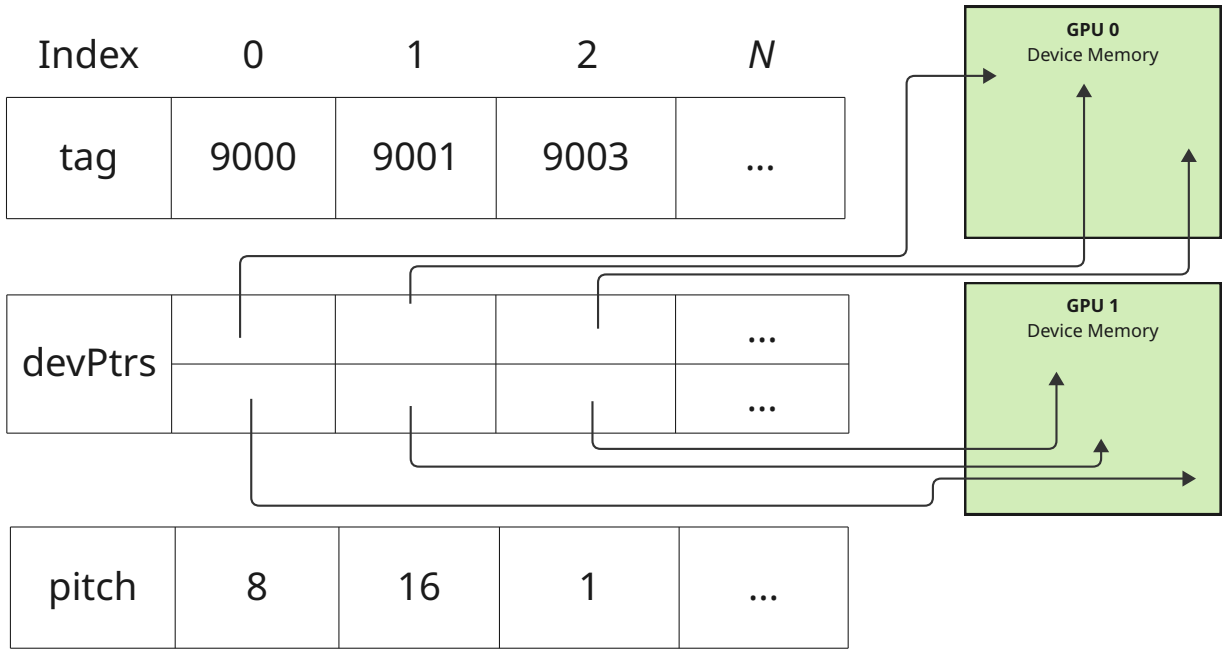

Figure 4.4: Attribute data vectors. The vectors `vector<int> attributeTags`, `vector<void**> attributeDevPtrs`, and `vector<size_t attributePitch` store the data for each attribute on the host upon attribute creation. Tag stores a user-enumerated ID, the device pointers store pointers to the attribute data for each device, and the pitch stores a CUDA value for aligning two-dimensional attributes.

4.5 Attribute Mapping

While each attribute has its memory allocated when calling `setAttributes()`, a `Place` object on the device cannot yet access this memory because the attribute device pointers are stored on the host as a data member of the `Places` object. Therefore, `Places::finalizeAttributes()` must be called to transfer the data needed to access attributes (namely, the vectors described in section 4.4) to the device. Figure 4.5, which is an updated version of Figure 4.14 within Warren Liu's paper, shows the sequence diagram for

mapping attributes to each Place object within the device [4]. The `finalizeAttributes()` function allocates memory for each attribute vector (tag, device pointers, and pitch) and stores pointers to that memory in the `PlaceArray` struct within the `DeviceConfig` object. Because CUDA does not support vectors, the `DeviceConfig` stores these attribute arrays as C arrays of the following form: `int **d_attributeTags`, `void ***d_attributeDevPtrs`, and `size_t* d_attributePitch`. The `finalizeAttributes()` function copies each of these arrays to each device.

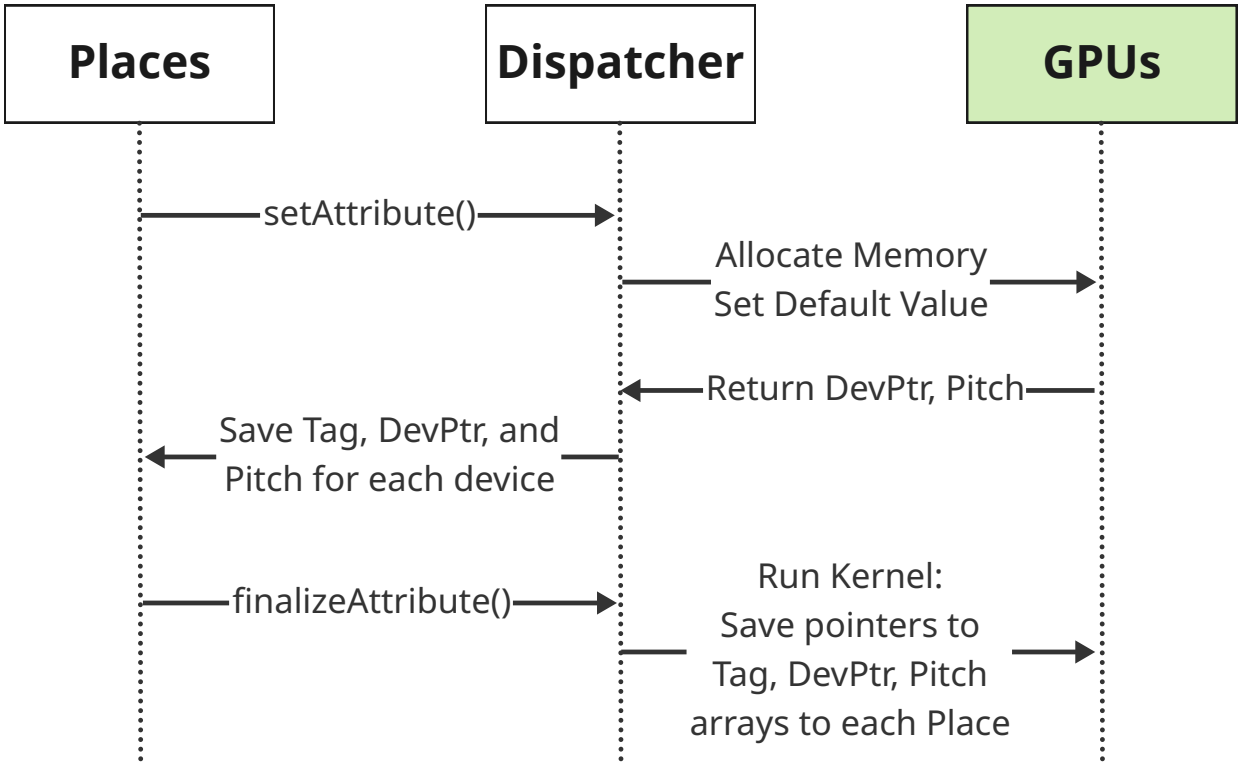

Figure 4.5: Attribute creation and mapping sequence, an updated version of Warren Liu's Figure 4.14. [4]. This shows how attribute data is created using `setAttribute()` on the `Places` object and then allocated on the GPUs by the `Dispatcher`. The three arrays tracking this data are temporarily saved within the `Places` object, and then transferred to the GPU upon calling `finalizeAttribute()`. This function also runs a kernel giving each `Place` object a pointer to these arrays.

One thing to note is that each device only needs access to its own array of device pointers, since trying to access the pointers to the other device would simply return a memory out of bounds error. Therefore, `finalizeAttributes()` first copies each device's device pointers into a new, contiguous void array for each device, and then copies only that device's device pointer array to the device. The final step in `finalizeAttribute()` is to ensure that each Place on the device has a pointer to the attribute arrays. To do this, `finalizeAttribute()`

launches an `updateDevAttributesKernel()` kernel on each device that saves pointers to the attribute arrays as each `Place` object's data members. Effectively, this means that each `Place` can now access attribute arrays allowing them to retrieve pointers to where that place can find the relevant attribute data it needs.

4.6 Attribute Access

When a Place object running a `callAll()` function needs to access one of its attributes, it calls `getAttribute(int tag, int length)` within the `callAll()` function. This function uses the pointer to the `attributeTag` array transferred to this `Place` during `finalizeAttribute()` to search the `attributeTag` array for the index corresponding to the given tag, as seen in line 4 of Listing 4.5. Then, `getAttribute()` simply accesses the `attributeDevPtr` array at that tag index to get a pointer to where that attribute is stored in device memory, shown in line 9 of Listing 4.5. Finally, the function accesses the attribute data at the `devIndex` of this `Place` to find the attribute value that corresponds to this particular place, shown in line 10 of Listing 4.5. This function is largely unchanged from the single-GPU version, except that it now accesses the attribute at `devIndex` and not `Index`, as the original `Index` of a place no longer corresponds to the thread `idx` of a `Place` when it is split over two devices.

In addition to the `getAttribute()` function for accessing a `Place` object's own attributes, a place may also need to access the attributes of adjacent places. To do this, an additional overloaded function with the parameters `getAttribute(size_t des_Index, int tag, int length)` allows for this access. This function accesses the memory of adjacent places by offsetting the accessed attribute memory block by the given `desIndex` value instead of the place's own `Index`. For multi-GPU MASS CUDA, this function presents the problem that a place may attempt to access adjacent places that are stored on another device. While the `desIndex` may be a valid place, if the place is on another device this attempt would result in a segmentation fault. To address this issue in a simple way, we used the `MAX_PLACE_EXCHANGE` setting within the `settings.h` file (as described in section 4.3) as the limit for attribute access

Listing 4.5: Code to access attributes upon a getAttribute() call. First, it searches the attribute tag array to find the index of the given tag. Then, it gets the attribute device pointer at that index. Finally, it returns the specific attribute value at the `devIndex` of the device pointer array.

```
1   template <typename T>
2   __device__ T* Place::getAttribute(int tag, int length) const
3   {
4       int i = find(attributeTags, nAttributes, tag);
5       if (i == -1)
6       {
7           return NULL;
8       }
9       T* array = static_cast<T*>(attributeDevPtrs[i]);
10      return &array[devIndex];
11  }
```

from other places. When a `Place` attempts to access another `Place` object's attributes, the function first checks that the `Place` being accessed is within the ghost place rows, otherwise the program will throw a descriptive MASS error telling the user they need to increase `MAX_PLACE_EXCHANGE`, rather than a segmentation fault.

4.7 Attribute Transfer

When a user of the MASS CUDA library sets attributes and then calls `finalizeAttribute()`, all data for `Place` objects is now stored on the devices. All data is stored on the devices because maintaining a host-side copy of every attribute would likely exceed the memory available on the host, and synchronizing host attribute memory with device attribute memory would significantly reduce performance. Any user-defined `callAll()` functions, which run on the devices, can access data using the `getAttribute()` function described in section 4.6. However, `callAll()` functions all have void return types, so data remains stored on the devices during program execution. While this configuration improves performance by minimizing data transfers between host and devices, users may need to transfer data

back from the devices to the host in order to utilize intermediate simulation data or to save the final results. To do this, they call the `downloadAttributes(int handle, void** d_attributeDevPtrs, size_t pitch, unsigned int length, unsigned int qty)` function, which copies attribute data from the devices to the host.

The `downloadAttributes()` function is a template function that returns a host pointer of the `Template` object type to the user, allowing them to use a copy of the device attribute data. The user specifies which attribute they wish to download using both the attribute tag and by entering the attribute object type as a template. The function allocates host memory to store the attributes, with the understanding that control of the memory is transferred to the user's function on completion and must therefore be freed by the user.

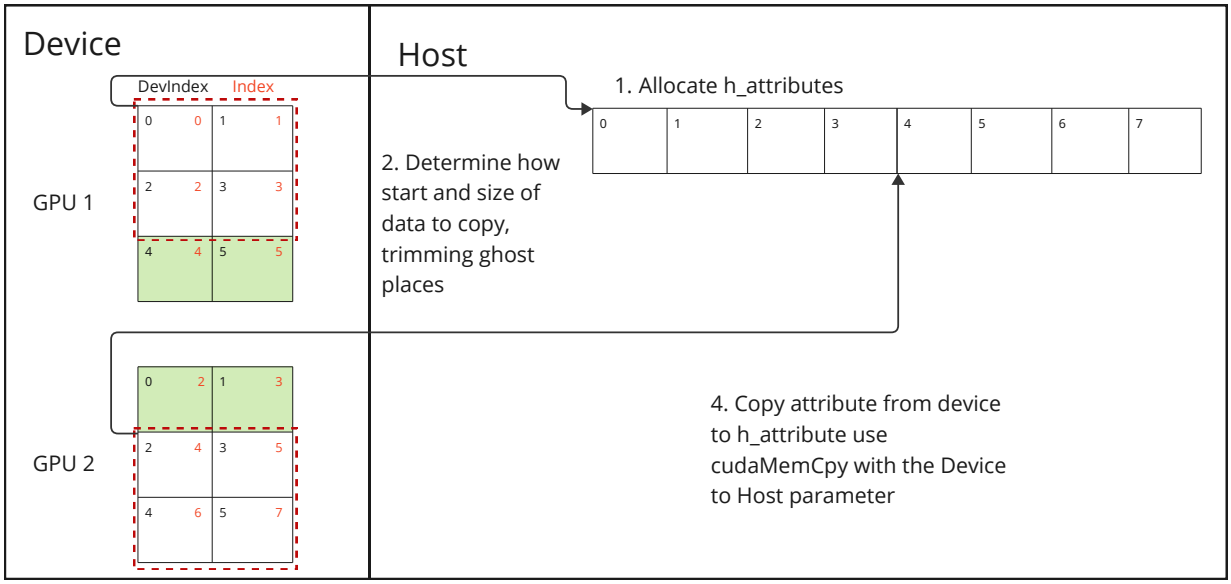

Figure 4.6: Steps in attribute transfer from host to device. First, host memory is allocated for the entire attribute array (from both devices) as `h_attribute`. Next, the function determines the start of the device base places and the destination section of `h_attribute` on the host for each device. Finally, the function copies the memory from device to host.

There are two complications regarding `downloadAttributes()` that arise from using

multiple devices. First, the attribute data stored on each device includes ghost places at the beginning or end (or both) of the places on each device. However, this data is a duplicate of the base places they represent, and so should not be downloaded to the host for the user to access. Therefore, the `downloadAttributes()` function must trim ghost place attribute data. Second, each device must transfer its own separate attribute memory, but the `downloadAttributes()` function should return a single contiguous array in host memory. This means that the downloaded memory must be recombined to be accessed by a single pointer. Figure 4.6 shows the steps involved in this process.

This code first allocates host memory for the entire place array (both devices combined) with `h_attribute`. Next, within a parallel section on each device the function determines the start of the base places on each device, the size in bytes of the base places on the devices (excluding ghost places), and the start position for each device's copy in `h_attribute`. Next, it copies the attribute memory from the device to `h_attribute`. The result is that the user receives a pointer to a single attribute array, while all ghost places are discarded when the data is copied.

4.8 Inter-Place Communication

In order to exchange information between `Place` objects, each `Place` must first determine which neighboring `Place` objects it wants to exchange information with, and then save pointers to each of those objects. To do this, the user calls `Places::exchangeAll()`. Effectively, this gives each `Place` object two arrays: one storing the indices of its neighboring `Place` objects, and one storing a pointer to those neighboring `Place` objects so that it can access their data or call functions on them. The user creates and provides a `destinations` vector containing relative coordinates for each of a `Place` object's neighbors. An example of this vector and the resulting neighbors that will be stored on the device is shown in Figure 4.7.

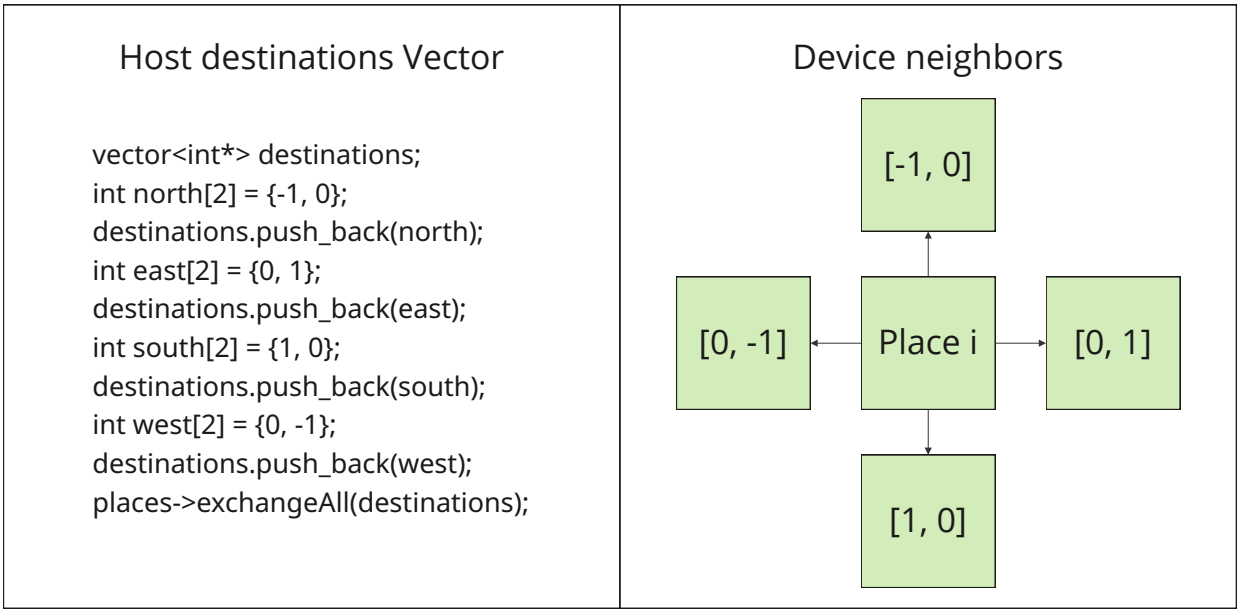

Figure 4.7: Host `destinations` vector and resulting neighbors on device. The user creates a `destinations` vector containing `int` arrays, each of which contain an offset for each dimension of the place array. For instance, the `int` *north* contains the values {-1, 0}, which suggests will add a neighbor one row before this place. Once all `int` arrays have been added, the user provides the `destinations` vector as an argument to the `exchangeAll()` function.

In this implementation for a two-dimensional array of `Place` objects, the user sets the standard neighbors for a `Place` as directly above, to the right, below, and to the left of each `Place`. The `exchangeAll()` function then takes this destinations vector, saves a copy of it as a data member of the `Dispatcher` object, and then calls `exchangeAllPlacesKernel()` to have each `Place` save the adjacent `Place` objects as its data members and save pointers to those objects.

To adapt this implementation to multi-GPU MASS CUDA, we had to make two changes. First, the original algorithm for determining the `Index` values of neighboring `Place` objects, given the destinations vector, relied on the `Index` value of a `Place` being equal to the device thread `idx` within the `exchangeAllPlacesKernel()`. However, with multiple

GPUs this is no longer the case, so we adjusted the kernel to perform the calculation in terms of both the device `devIndex` (in order to get pointers to the neighboring `Place`) and the `Index` (to save the actual `Index` of the neighboring `Place`). In addition, using multiple GPUs for `exchangeAll()` creates similar complications to the `getAttribute()` function, wherein a `Place` may attempt to access data (in this case its neighbor information) from another GPU. We addressed this issue in a similar way by requiring the offsets to be within the `MAX_PLACE_EXCHANGE` range set in the `settings.h` file. If the user attempts to call `exchangeAll()` with destinations that are beyond the number of rows set in `MAX_PLACE_EXCHANGE`, MASS will throw an error telling the user they need to increase `MAX_PLACE_EXCHANGE`, rather than a segmentation fault.

4.9 GPU Boundary Communication

As described in section 4.3, each device has both base places and ghost places stored in its memory. The ghost places are designed to be duplicates of their corresponding base places, including all their attributes. However, as base places run `callAll()` functions and modify their attributes, the attributes in the corresponding ghost places may become out of date. To solve this problem, the `exchangeGhostPlaces()` function runs after every `callAll()` function and copies all attribute data from the base places to the corresponding ghost places on the adjacent devices. The basic functionality of this is shown in Figure 4.8 and Listing 4.6.

The code in Listing 4.6 show only copying base places from the current device to ghost places on the previous device, but copying from current to next is similar. First, because each device only saves a pointer to the start of its attribute arrays (e.g. the first place's attribute in the array), the function needs to determine the offsets (in number of places) to where the source (start of base places) and destination (start of ghost places on previous device) begin. This is shown in the calculation of `dstOffset` and `srcOffset` on lines 9 and 10. For example, in the example shown in Figure 4.8 the `srcOffset` would be four places on the second device (pointing to the start of the place with `DevIndex` four), while the `dstOffset`

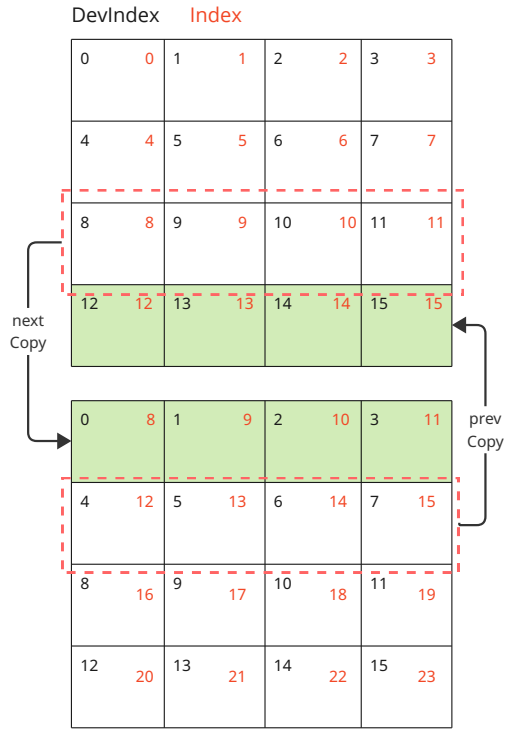

Figure 4.8: Copying of base places to overwrite corresponding ghost places on adjacent devices. This shows how the attributes of base places with `Index` 8 through 11 on the first device are copied to overwrite the corresponding ghost places on the next device. Similarly, the attributes of base places with `Index` 12 through 15 (`DevIndex` 4 through 11)are copied to overwrite the corresponding ghost places on the previous device.

would be twelve places on the first device, to point to the start of the place with `DevIndex` twelve. Once the offsets are established, the code then loops through each attribute and does the following steps:

1. Determine if this attribute is a one-dimensional attribute (only one value) or a two-dimensional attribute (contains arrays of values) on line 17.

2. If the attribute is one-dimensional, the number of bytes to copy is equal to `number of ghost places * sizeOf(AttributeObjectType)`, shown on line 22. If it is two-

Listing 4.6: Ghost place copying to previous device. This determines the previous GPU ID, then gets the `placeStride` and `ghostPlacesAfter` of that device. It uses these to calculate the start position of the ghost places on the previous device, called the `dstOffset`. Then it determines the start of the base places on the current devices, called the `srcOffset`. Next, it calculates the size of the data to copy using either the `attributePitch` (for two-dimensional attributes) or the `attributeSize` (for one-dimensional attributes). Finally, it calculates a `dst` pointer by offsetting this attribute's `attributeDevPtrs` by the `devOffset`, and a `src` pointer by offsetting this attribute's `attributeDevPtrs` by the `srcOffset`. These are the arguments in the `cudaMemcpyPeerAsync()` call.

```
1   #pragma omp parallel
2   {
3       int gpu_id = -1;
4       CATCH(cudaGetDevice(&gpu_id));
5
6       if (activeDevices.size() > 1 && gpu_id > 0) {
7           int prev_gpu = activeDevices[gpu_id - 1];
8
9           int dstOffset = p->placeStride[prev_gpu] - p->ghostPlacesAfter[prev_gpu];
10          int srcOffset = p->ghostPlacesBefore[gpu_id];
11
12          for (int i = 0; i < p->nAttributes; i++) {
13
14
15              int count = p->ghostPlacesAfter[prev_gpu];
16              int offsetSize;
17              if (attributePitch->at(i) > 0) { // If attribute is 2D, count and offsetSize are
                    based on attribute Pitch
18                  count *= attributePitch->at(i);
19                  offsetSize = attributePitch->at(i);
20              }
21              else {
22                  count *= attributeSize->at(i);
23                  offsetSize = attributeSize->at(i);
24              }
25
26              void* dst = (char*)(attributeDevPtrs->at(i)[prev_gpu]) + (offsetSize *
                    dstOffset);
27              const void* src = (char*)(attributeDevPtrs->at(i)[gpu_id]) + (offsetSize *
                    srcOffset);
28
29              CATCH(cudaMemcpyPeerAsync(dst, prev_gpu, src, gpu_id, count));
30  }}}
```

dimensional, the number of bytes is equal to `number of ghostPlaces * attributePitch`, shown on line 18.

3. Use pointer arithmetic to determine the destination pointer, `dst`, to the start of the ghost places on the previous device, on line 26.

4. Use pointer arithmetic to determine the source pointer, `src`, to the start of the corresponding base places on this device, on line 27.

5. Copy data between the devices using `cudaMemcpyPeerAsync()` on line 29. The Async version of `cudaMemcpyPeer()` allows the copy calls to run in the background while the host calculates the `dst` and `src` values for the next attribute, improving performance.

By running `exchangeGhostPlaces()` after every `callAll()` function, the MASS CUDA library ensures that the attributes saved on the ghost places are always up to date with the attributes saved on their corresponding base places. While this data transfer can be computationally intensive, we believe it is more efficient than a piecemeal copying of data between devices by every place. To allow the user to minimize the amount of device to device data transfer, we also provided a `callAll (int functionId, void* argument = nullptr, int argSize = 0) const` version of `callAll()`. This version allows the user to specify that they are not intending to update any attributes with their `callAll()` function, and so `exchangeGhostPlaces()` will not be run. This saves time copying attribute data when no data has been updated.

Chapter 5
EVALUATION

Evaluation of the multi-GPU MASS CUDA implementation consists of a programmability comparison with the single-GPU MASS CUDA library and execution performance tests using the Game of Life and Heat2D benchmark programs. The programmability comparison reviews how a user would convert an ABM program written for single-GPU MASS CUDA to multi-GPU MASS CUDA, and how the libraries compare in a static analysis of the code. The benchmarking assesses the runtimes and spatial scalability of benchmark programs running on the two MASS CUDA versions.

5.1 Programmability

5.1.1 Compatibility with Single-GPU Library

One goal of this multi-GPU MASS CUDA project is to require as few modifications as possible for applications written for the single-GPU MASS CUDA library version. The MASS library maintains a group of common benchmark applications used to compare MASS versions, such as MASS Java to MASS CUDA, as well as to other competitor ABM libraries. Several of these benchmark programs, such as Heat2D, MATSim, and Tuberculosis were already completely rewritten to support the single-GPU MASS CUDA library update to use attributes for data storage [15, 16]. Because rewriting benchmark programs is a time-consuming process, the multi-GPU MASS CUDA library purposely maintains the previous MASS CUDA interface, with some minor exceptions.

Benchmark programs using the multi-GPU MASS CUDA library must make two main changes to their programs: updating the base constructor code and including OpenMP linker flags during compilation. When a user creates a `UserPlace` class that extends the `Place`

class, they must provide a `UserPlace` constructor that overrides the `Place` constructor. Previously, this `Place` constructor only needed a `Place` index to create the base `Place` object. However, now that a `Place` has a separate `index` and `devIndex`, and can be either a base place or a ghost place, the user needs to make the change to their constructor shown in Listing 5.1.

Listing 5.1: Single-GPU and Mult-GPU UserPlace Constructor. This shows the differences in the base constructor the user needs to provide in their `UserPlace` header file. This is the only time where the user needs to use the `devIndex` or `basePlace` data members.

```
1    /* SINGLE-GPU */
2    UserPlace(int index) : Place(index) {}
3
4    /* MULTI-GPU */
5    UserPlace(int index, int devIndex, bool basePlace) : Place(index, devIndex, basePlace) {}
```

The second change the user must make is enabling OpenMP support by adding the following code as flags for the CUDA compiler, nvcc: `-Xcompiler -fopenmp`. The `-Xcompiler` flag tells nvcc to pass any subsequent flags directly to the host compiler instead of the nvcc compiler, while the `-fopenmp` flag tells the host compiler to link the OpenMP library.

In addition to code changes, users must specify the settings values of `MAX_PLACE_EXCHANGE` and `MAX_AGENT_TRAVEL` to match the needs of their benchmark program, as explained in section 4.3. These values should be determined by the user based on the maximum distance (in terms of number of place rows) they expect places to exchange information and the maximum distance they expect agents to migrate, respectively. These can be set in the `settings.h` file within the MASS CUDA source directory, as shown in Listing 5.2.

Listing 5.2: Setting `MAX_PLACE_EXCHANGE` and `MAX_AGENT_TRAVEL` within `settings.h`. This value is determined by the user based on the needs of their benchmark program.

```
1  #ifndef MAX_AGENT_TRAVEL
2  #define MAX_AGENT_TRAVEL 1
3  #endif
4
5  #ifndef MAX_POTENTIAL_AGENTS
6  #define MAX_POTENTIAL_AGENTS 12
7  #endif
```

5.1.2 Static Analysis of Single-GPU and Multi-GPU Libraries

To compare the programmability of the single-GPU and multi-GPU MASS CUDA implementations, we performed a static analysis of the two code bases. This analysis compares the lines of code and the cyclomatic complexity. Cyclomatic complexity is assessed by finding the number of linearly independent paths through the code, with this being increased by conditional statements like `if`, `else`, `for`, `switch`, etc. To perform this analysis, we ran *lizard*, an open-source code complexity tool, on the entire source directory of both library versions. The results of the single-GPU and multi-GPU MASS CUDA static analysis are shown in Table 5.1.

Table 5.1: Comparison of Single-GPU and Multi-GPU static analysis, showing the similarities in LOC and cyclomatic complexity.

Metric	Single-GPU	Multi-GPU
Lines of Code	1106	1316
Cyclomatic Complexity	2.36	2.40

Overall, the modifications to implement multi-GPU MASS CUDA did not drastically increase the lines of code or cyclomatic complexity of the library. The relatively small number of additional lines of code and small increase in cyclomatic complexity is due to the efficiency of using OpenMP to parallelize the same kernel calls over both GPUs. In fact, most of the additional lines of code are from implementing logic to split the places array and calculate ghost place locations.

5.2 Execution Performance

To benchmark the performance of the updated MASS CUDA library, we ran the Game of Life and Heat2D benchmark programs for both single-GPU and multi-GPU MASS CUDA. Both benchmark programs use only `Place` objects, not agents, so they are ideal for our

implementation. In addition, they generally only require minimal data exchange between immediately adjacent place objects, which means they need less ghost places and therefore less data transfer between devices.

Conway's Game of Life is a program that simulates the growth and competitions of living organisms [17]. These organisms are arrayed on a grid of places that represent cells, and each organism lives, reproduces, or dies using a simple set of rules based on how many other organisms are around it. These rules are:

1. A live cell with fewer than two living neighbors dies.

2. A live cell with two or three live neighbors survives to the next generation.

3. A live cell with more than three live neighbors dies.

4. A dead cell with exactly three live neighbors becomes alive.

These simple rules utilize the `exchangeAll()` and `callAll()` functions of the MASS CUDA place implementation within a loop to run the simulation.

Heat2D simulates the dispersion of heat across a metal plate over time. It starts with a single point source of heat applied to the plate, and from there the heat spreads across the plate. In MASS, Heat2D is represented by a two-dimensional grid of places, each representing a discrete point on the metal plate. Each place exchanges information with its neighbors to determine the temperature of those neighbors, then calculates its own temperature based on that using the Euler method. This simulation therefore makes heavy use of inter-place communication, as described in section 4.8. Figure 5.1 shows an example of the spread of heat from a point source at the top middle of the place grid, which was simulated in MASS CUDA and visualized using the Simviz library.

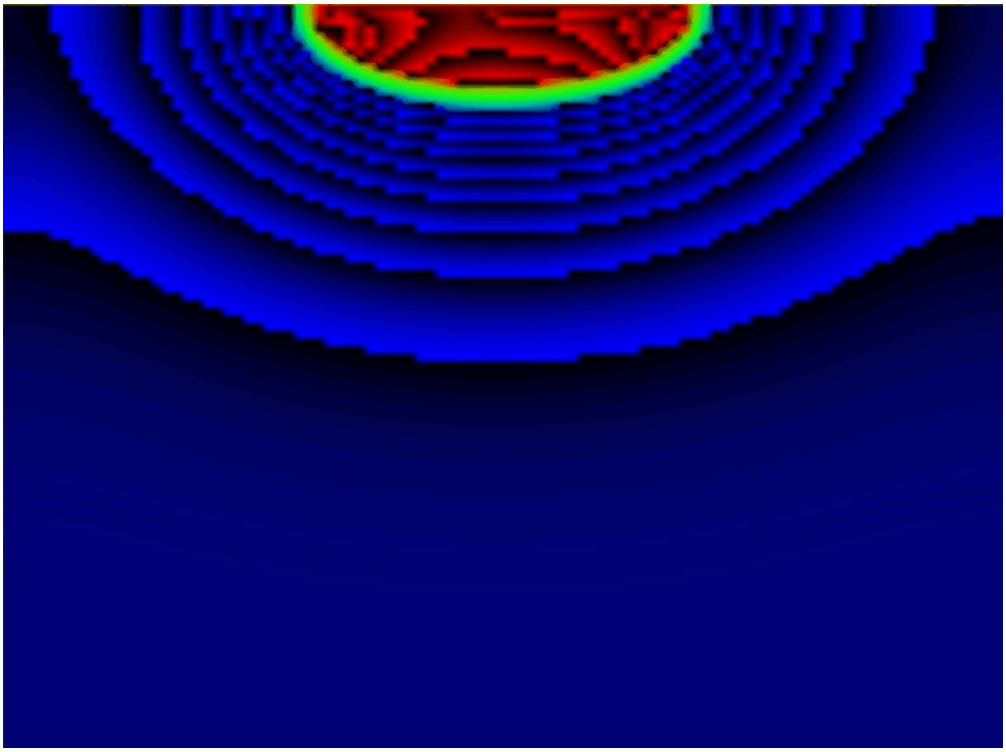

Figure 5.1: Heat2D visualization created using Simviz output tool. Shows a single heat source (in red) being applied at the top center of the plate, with that heat in the process of radiating throughout the rest of the plate.

5.2.1 Benchmark Runtimes

Benchmark program runtimes were tested on the UW Bothell's Juno remote Linux lab computer. This computer has two NVIDIA RTX A5000 graphics cards, each with 24 GB of device memory. The two cards are connected by an NVLink 4.0 bridge; tests for single-GPU runtimes used one of these GPUs, while tests for multi-GPU used both. The simulation used a Timer class within the source code to calculate the runtime from MASS initialization to MASS shutdown, with the simulation in between. The results of the runtimes for various places array sizes are shown in Figures 5.2 and 5.3.

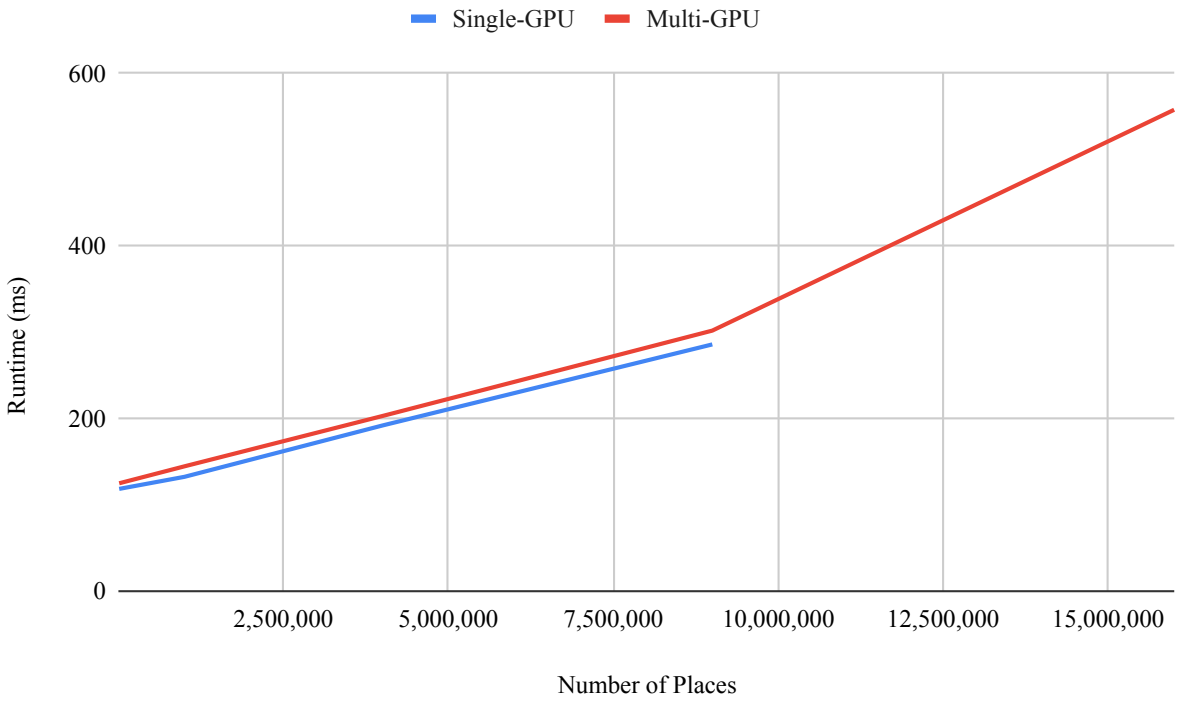

Figure 5.2: Game of Life runtime benchmarks for single and multi-GPU MASS CUDA. The slopes of the graph are 18.9 nanoseconds per `Place` for single-GPU and 26.5 nanoseconds per `Place` for multi-GPU.

Figure 5.2 shows that for this benchmark program, the single-GPU and multi-GPU MASS CUDA implementations had similar runtimes, with the multi-GPU being slightly slower. However, the multi-GPU implementation demonstrated the increased spatial scalability available on two GPUs. The graph for single-GPU Game of Life ends at nine million places, which is the maximum simulation size that could be run before running out of device memory. In contrast, the multi-GPU implementation was able to run Game of Life with a maximum of sixteen million places. This represents a 77% increase in the spatial scalability of the Heat2D simulation using the multi-GPU MASS CUDA library. The Game of Life benchmark suggests that multi-GPU MASS CUDA may be effective for running larger programs even if the runtimes are slower.

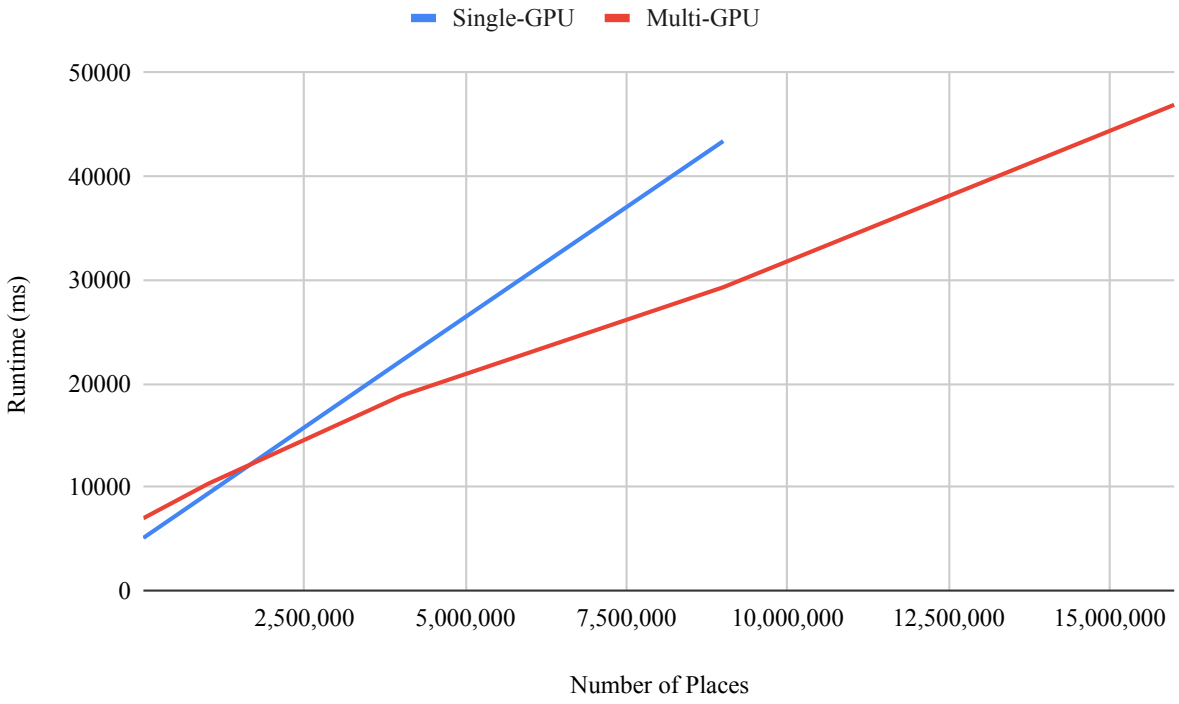

Figure 5.3: Heat2D runtime benchmarks for single and multi-GPU MASS CUDA. The slopes of the graph are 4.26 microseconds per `Place` for single-GPU and 2.47 microseconds per `Place` for multi-GPU.

Figure 5.3 shows multi-GPU MASS CUDA runs significantly faster than single-GPU MASS CUDA as the number of places increases. At nine million places (the maximum size of the single-GPU simulation), the multi-GPU version takes 32% less time to complete the simulation than single-GPU. However, for simulations less than one million places the single-GPU version is faster. This demonstrates the tradeoffs between the additional resources and the higher overhead of a multi-GPU implementation. For smaller numbers of places, the overhead to split the data and copy ghost places between devices outweighs the additional threads available to multiple GPUs. This relationship is reversed with larger simulation sizes, with the two GPUs allowing the simulation of larger data sets with more threads. In

addition to improvements in runtime for large simulations, Figure 5.3 also demonstrated the same improvements in spatial scalability seen in Game of Life.

The improved runtime performance of Heat2D compared to Game of Life is likely due to the increased complexity of the program. While Game of Life simply counts the number of adjacent places that are alive, Heat2D uses the heat equation to calculate the temperature of the current place based on the temperatures of the adjacent places. This more complex calculation can take advantage of the greater number of cores in multi-GPU MASS CUDA, increasing the overall performance. Therefore, in future benchmarks we would expect that more complex programs that require more calculation time would perform better in multi-GPU MASS CUDA as their overhead due to ghost place exchange would be spread over more calculation time.

Chapter 6

CONCLUSION

This project successfully implemented multi-GPU MASS CUDA for place objects, including unit testing and benchmark testing. This includes all aspects of the `Place` object implementation, including initialization, place creation, attribute creation and access, and inter-place communication. In addition, this project implemented synchronization of place data between devices using ghost places and GPU boundary communication. These changes were implemented in a way that largely maintained compatibility with the previous MASS CUDA implementation, which will allow existing benchmark programs to be quickly updated to run on multi-GPU MASS CUDA. The evaluation of this project showed promising results, with the potential for both faster runtimes and increased spatial scalability, depending on the program. This opens the possibility of allowing larger ABM simulations to be run on the MASS CUDA library.

However, there are some limitations to this project and its implementation. First, testing was limited to two GPUs, so the algorithms to split places and attributes over more than two could not be fully tested. However, all algorithms were written with the intent that they can run on N GPUs. Second, there are limited benchmark programs that require only places and not agents, which limited the scope of the runtime evaluation. Lastly, this project focused largely on modifying the existing MASS CUDA implementation for multiple GPUs, and therefore did not focus on any optimization or reorganization of the original MASS CUDA code.

The final limitation to note is that this project is the first half of a larger effort to fully implement MASS CUDA for multiple GPUs. While places have been fully implemented, MASS requires agents to run most benchmark programs and reach full functionality. There-

fore, future work will be required to adapt the single-GPU MASS CUDA code to multiple GPUs. However, we believe that much of the groundwork established in this code can be transferred to agents, such as OpenMP parallelization, algorithms to split data over multiple GPUs, and algorithms to copy attribute data between devices.

Another potential project for future work is to reconcile the differences between the MASS CUDA library and the MASS C++ and MASS Java libraries. The implementation of attributes caused the way users create ABM programs in MASS CUDA and the other MASS versions to diverge. Differences in the codebases, such as MASS CUDA no longer having a `PlaceState` or `AgentState` object and accessing attributes using `getAttribute()`, causes incompatibility between the library versions that make programming more difficult when transferring between libraries. The MASS CUDA implementation of attributes also somewhat undermines the object-oriented nature of the code by requiring all attribute to be declared inline within the main function, rather than being encapsulated within objects. While the decision to change the MASS CUDA library to implement attributes remains sound due to its performance enhancements, some form of interface or setup script for MASS CUDA could restore the `PlaceState` and `AgentState` objects to the user while maintaining the attribute implementation on the background. This future work could greatly improve the compatibility of MASS library versions with each other.

In conclusion, the multi-GPU implementation of the MASS CUDA library greatly enhances both the runtime performance and spatial scalability of the library for large simulations. This allows the library to run larger, more complex simulations that require time and memory usage to obtain results. This significantly improves the capabilities of the MASS CUDA library, making it more competitive with other ABM models in running simulations. Finally, this approach to improving the capabilities of MASS CUDA opens opportunities for further improvements to the MASS CUDA library, such as adding additional GPUs to further increase performance.

BIBLIOGRAPHY

[1] CPU vs. GPU: What's the difference? [Online]. Available: https://www.cdw.com/content/cdw/en/articles/hardware/cpu-vs-gpu.html

[2] N. Hart, "MASS CUDA: Parallel-computing library for multi-agent spatial simulation." [Online]. Available: https://depts.washington.edu/dslab/MASS/docs/MassCuda.pdf

[3] E. Kosiachenko, "Efficient GPU parallelization of the agent-based models using MASS CUDA library," 2018. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/LisaKosiachenko_MasterThesis.pdf

[4] W. Liu, "Programmability and performance enhancement of MASS CUDA," p. 69. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/WarrenLiu_whitepaper.pdf

[5] L. Kosiachenko, N. Hart, and M. Fukuda, "MASS CUDA: A general GPU parallelization framework for agent-based models," in *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, Y. Demazeau, E. Matson, J. M. Corchado, and F. De la Prieta, Eds. Springer International Publishing, pp. 139–152. [Online]. Available: https://doi.org/10.1007/978-3-030-24209-1_12

[6] B. D. Pittman, "Multi agent spatial simulation (MASS) in multiple GPU environment with NVIDIA CUDA." [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BenPittman_whitepaper.pdf

[7] S. Luke, "Multiagent simulation and the MASON library." [Online]. Available: https://cs.gmu.edu/~eclab/projects/mason/manual.22.pdf

[8] U. Wilensky, "NetLogo." [Online]. Available: http://ccl.northwestern.edu/netlogo/

[9] Repast simphony reference manual. [Online]. Available: https://repast.github.io/docs/RepastReference/RepastReference.html

[10] N. Collier, "Repast HPC manual." [Online]. Available: https://repast.github.io/docs/repast_hpc.pdf

[11] D. Perez. NVIDIA® CUDA™ unleashes power of GPU computing. [Online]. Available: https://web.archive.org/web/20070329144655/http://www.nvidia.com/object/IO_39918.html

[12] P. Richmond, R. Chisholm, P. Heywood, M. Leach, and M. Kabiri Chimeh, "FLAME GPU." [Online]. Available: https://zenodo.org/records/5465845

[13] N. M. Ho, N. Thoai, and W. F. Wong, "Multi-agent simulation on multiple GPUs," vol. 57, pp. 118–132. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X15001033

[14] W. Chen, K. Ward, Q. Li, V. Kecman, K. Najarian, and N. Menke, "Agent based modeling of blood coagulation system: Implementation using a GPU based high speed framework," in *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 145–148, ISSN: 1558-4615. [Online]. Available: https://ieeexplore-ieee-org.offcampus.lib.washington.edu/document/6089915

[15] J. Nguyen, "Development of FLAMEGPU2 and MASS CUDA benchmark programs (heat2d)." [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/JohnNguyen_wi24.docx

[16] H. Ogden, "MASS CUDA MATSim." [Online]. Available: https://depts.washington.edu/dslab/MASS/index.html

[17] M. Gardner, "Mathematical games - the fantastic combinations of john conway's new solitaire game "life" - m. gardner - 1970," vol. 223, pp. 120–123. [Online]. Available: https://web.stanford.edu/class/sts145/Library/life.pdf

Appendix A
SOURCE CODE DETAILS

Source code for the Multi-GPU MASS CUDA library Place implementation can be found at `https://bitbucket.org/mass_library_developers/mass_cuda_core/src/0cf061adb6779ca883e2c73b36e6662100b4baf7/?at=holto%2Fmulti-gpu-cuda`

This code has not been merged into the `develop` or `main` branches because it includes only the Place implementation, so it would not be able to run most benchmark programs at this time. We anticipate that the multi-GPU version would use version number 0.8.0 when fully implemented.

To set up the source code on the UW Bothell's Juno computers, follow these steps:

1. Connect to the Juno lab computers by entering `ssh NETID@juno.uwb.edu`
2. Clone the library by running `git clone https://holtogden@bitbucket.org/mass_library_developers/mass_cuda_core.git`
3. Change to the library directory by running `cd mass_cuda_core`
4. Download and install dependencies by running `make develop`
5. Compile the library by running `make build`
6. Run the tests by calling `make test`

An example output for these steps is shown in Listing A.1. Note that other tests can be enabled by modifying the `testing::FLAGS_gtest_filter = "MASS_Places.SetAttribute"` line within the `test/main.cu` file to specify other tests within the file. To run all tests for the place implementation, use `"MASS_Places.*"`.

Listing A.1: Example output of multi-GPU MASS CUDA setup.

```
[holto@juno mass_cuda_core]$ make build
Building MASS library...
MASS library build complete.
[holto@juno mass_cuda_core]$ make test
Building test...
========= COMPUTE-SANITIZER
Note: Google Test filter = MASS_Places.SetAttribute
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from MASS_Places
[ RUN ] MASS_Places.SetAttribute
[ OK ] MASS_Places.SetAttribute (5668 ms)
[----------] 1 test from MASS_Places (5668 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (5668 ms total)
[ PASSED ] 1 test.
========= ERROR SUMMARY: 0 errors
[holto@juno mass_cuda_core]$
```

Appendix B

BENCHMARK CODE DETAILS

Source code for the MASS CUDA library benchmark programs used in this paper can be found at the following locations:

- **Game of Life:** `https://bitbucket.org/mass_application_developers/mass_cuda_appl/src/0cd15bb4b4dd40dc74fa3bc616c0b06e1ac7658d/GameOfLife/GameOfLife_PlaceV2/?at=holto%2FHeat2D-multi-gpu`

- **Heat2D:** `https://bitbucket.org/mass_application_developers/mass_cuda_appl/src/0cd15bb4b4dd40dc74fa3bc616c0b06e1ac7658d/Heat2D/Heat2D_MASS/?at=holto%2FHeat2D-multi-gpu`

To set run these programs on the UW Bothell's Juno computers, follow these steps:

1. Connect to the Juno lab computers by entering `ssh NETID@juno.uwb.edu`

2. Clone the library by running `git clone [url]`

3. Change to the benchmark directory by running `cd [folder]`

4. Download and install dependencies by running `make develop`

5. Compile the library by running `make build`

6. Run the programs by calling `./bin/[program]`

Example output for these programs is shown in Listings B.1 and B.2.

Listing B.1: Example output of multi-GPU MASS CUDA Game of Life benchmark.

```
[holto@juno GameOfLife_PlaceV2]$ make build
Building GameOfLife_PlaceV2...
GameOfLife_PlaceV2 build complete.
[holto@juno GameOfLife_PlaceV2]$ ./bin/GameOfLife_PlaceV2
Game Of Life (MASS Implementation)
Starting MASS CUDA simulation with MAX_NEIGHBORS 8
Running MASS CUDA simulation for 10 generations, with size of 100
CUDA total time 16.2948 ms
CUDA time for initialization 10.294 ms
CUDA time for each step 0.250026 ms
CUDA time after setting up 2.60112 ms
MASS time for size 100: 130 ms
Done with deviceConfig freeDevice().
Ending Simulation
[holto@juno GameOfLife_PlaceV2]$
```

Listing B.2: Example output of multi-GPU MASS CUDA Heat2D benchmark.

```
[holto@juno Heat2D_MASS]$ make build
Building Heat2D_MASS...
Heat2D_MASS build complete.
[holto@juno Heat2D_MASS]$ ./bin/Heat2D_MASS
Running Heat2D with params: size=100, heat_time=2700, max_time=3000, interval=0
CUDA total time 2008.655762 ms
CUDA time for initialization 56.742882 ms
CPU time for initialization 5091 ms
CUDA avg step time 0.605538 ms
MASS time 6933 ms
[holto@juno Heat2D_MASS]$
```

Appendix C

BENCHMARK RESULTS

The full benchmark results for Game of Life and Heat2D are shown in Tables C.1 and C.

Table C.1: Game of Life runtime benchmarks for single and multi-GPU MASS CUDA.

Size	**# Places**	**Single-GPU Runtime (ms)**	**Multi-GPU Runtime (ms)**	**% Difference**
100	10,000	5130	7019	36.82%
1000	1,000,000	9354	10299	10.10%
2000	4,000,000	22183	18814	-15.19%
3000	9,000,000	43412	29294	-32.52%
4000	16,000,000	Out of Memory	46917	N/A
5000	25,000,000	Out of Memory	Out of Memory	N/A

Table C.2: Heat2D runtime benchmarks for single and multi-GPU MASS CUDA

Size	**# Places**	**Single-GPU Runtime (ms)**	**Multi-GPU Runtime (ms)**	**% Difference**
100	10,000	118.5	125	5.49%
1000	1,000,000	132.5	145	9.18%
2000	4,000,000	192	201	5.73%
3000	9,000,000	286	302	5.59%
4000	16,000,000	Out of Memory	558	N/A
5000	25,000,000	Out of Memory	Out of Memory	N/A