Multi-GPU Parallelized Agent-Based Modeling

Holt Ogden

Winter 2025 Term Report

University of Washington

Mar 25, 2025

Project Committee:

Munehiro Fukuda, Committee Chair

Michael Stiber, Committee Member

Robert Dimpsey, Committee Member

1. Introduction

1.1 Overview

This document presents the work performed for CSS 595 for the Winter quarter of the 2024-2025 school year. This project will extend the features of the Multi-Agent Spatial Simulation (MASS) library to include support for using multiple Graphics Processing Units (GPU) within the Compute Unified Device Architecture (CUDA) framework. Specifically, the goal of this project is to use NVIDIA's NVLink hardware and associated software to connect two GPUs, improving the performance and spatial scalability of programs running the MASS CUDA library.

The MASS library is designed to allow for the parallelization of multi-entity interactions over multiple computing nodes and multiple threads within those nodes. It is based on two key components: places and agents. A places element, or place, is a location in memory that holds data or attributes and can also host Agent instances, or agents. Places can exchange data between each other or allow residing agents to access its data. Agents are execution instances that are located within a place. They can 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. This allows data to move between places while also allowing agents to access data within the new Place.

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 device memory and are executed in parallel, taking advantage of the thousands of cores within modern GPUs. The library handles copying of data and execution of functions between the system memory controlled by the CPU (called the host) and the GPU memory used by the GPU (called the device). This capstone project will further extend the MASS CUDA library to function on multiple GPUs connected by an NVLink bridge, with the goal of allowing greater spatial scalability for programs running on the library. Due to time constraints, the scope of this project will now be to establish a working multi-gpu places implementation, with multi-gpu agents to comprise a separate, future project.

1.2 Background

The University of Washington, Bothell MASS lab has been developing a CUDA-based implementation of the MASS library since 2012 [6]. Since then, members of the lab have been continually improving this

library, with two main types of projects taken on by members of the lab: incremental improvements to the MASS CUDA library and implementations of multi-GPU MASS CUDA. A brief overview of each of these efforts is described in this section.

1.2.1 Incremental Improvements to the MASS CUDA Library

The most recent incremental improvement to the MASS CUDA library was completed by Warren Liu in the Spring of 2024. While he originally intended to implement a version of multi-GPU MASS CUDA, he found that the existing single-GPU implementation of the library required additional debugging and improvement [2]. Most critically, the single-GPU MASS CUDA implementation required performance enhancements to compete in benchmark tests with FLAME GPU2, another Agent-Based Modeling library running on CUDA. Because multi-GPU MASS CUDA is anticipated to improve spatial scalability moreso than runtime, performance improvements would first have to be made to the single-GPU library. His project therefore pivoted to working on improved benchmarking, performance, and documentation of single-GPU MASS CUDA [3].

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 MASS place data members are created, stored, and accessed by both the host and the device. Previously, a MASS place was stored as two separate objects: Place and PlaceState. Place held all the functionality of a MASS place, mainly its accessor functions for data, functions to add and remove agents, and the callMethod 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, including its neighbors, 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 functions.

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 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 eliminate the PlaceState object and change all data members to *attributes* of the Place object. This means that each data member is stored in a contiguous array of attributes for all places, with attribute index *i* corresponding to the data member for Place *i*. This user uses setAttribute and getAttribute functions to set and access the data within their program. This new implementation of place data member storage improved the performance of the MASS CUDA library [4].

1.2.2 Previous Implementations of 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, Brian Luger, and Ben Pittman. The most recent and complete implementation was done by Ben Pittman, 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. The implementation of multi-GPU MASS CUDA described in this paper uses some elements of Ben Pittman's 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 [5]. However, ultimately I decided to start my own multi-GPU implementation based on Warren Liu's single-GPU library updates. This is mostly due to the fact that the removal of PlaceState and the new attributes system made much of the previous work no longer applicable to the current MASS CUDA library.

1.3 Related Works

There are other existing libraries that have implemented agent-based modeling (ABM) for simulations, such as Multi-Agent Simulator Of Neighborhoods (MASON) from George Mason University, NetLogo from Northwestern University, and Recursive Porous Agent Simulation Toolkit (Repast) from the University of Chicago. Currently, however, none of these libraries provide direct support for single or multi-GPU modeling.

While these libraries do not provide built-in multi-GPU support, 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 [7]. 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 [1]. However, this simulation used ABM and multi-GPU modeling but did not combine them together.

The primary competitor for multi-GPU MASS CUDA will be FLAME GPU2. This program is another GPU-enabled ABM developed by the University of Sheffield. It supports ABM modeling as part of the core library, including abstractions to allow inexperienced programmers to utilize the multi-GPU support without understanding parallel programming. However, at this time FLAME GPU2 does not support multi-GPU ABM [8]. Previous capstone projects within the MASS lab have used FLAME GPU2 as a comparison to test MASS CUDA against another real-world project.

2. Implementation

2.1 Data Structure Organization

Figure 1. Data structure of Mass, Dispatcher, and DeviceConfig objects.

The MASS CUDA library is designed so Mass, Places, and Agents objects contain a pointer to the Dispatcher object, which serves as a central manager for MASS function calls. The Dispatcher object in turn contains a pointer to the DeviceConfig object, which manages all functionality and stores data related to the GPU, such as the number of active devices, arrays of PlaceArray structs, and arrays of AgentArray structs. Each of these PlaceArray structs contain pointers to the Place array on the device, pointers to the place attributes on the device, and any additional information needed to manage Places on the GPU. This structure is shown in Figure 1.

In the current implementation of single-GPU MASS CUDA, CUDA functionality such as allocating memory, copying memory, and calling kernels is split between both the Dispatcher and the DeviceConfig objects. Sometimes, function calls are passed from the Mass or Places objects to the Dispatcher and the Dispatcher communicates with the GPU, while other times the Dispatcher passes this function call further down to the DeviceConfig. This arrangement presented the first issue when implementing multi-GPU, as *only the DeviceConfig* knows that there are multiple GPUs. In addition, my goal was that all calls to CUDA and any pointers to memory should be stored centrally in one object. Therefore, the first

step of my implementation was to move all CUDA function calls and all device pointers to the DeviceConfig object. Future reorganization of this structure to reduce redundancy is discussed in the Future Work section.

2.2 OpenMP

My multi-GPU MASS CUDA implementation uses OpenMP to make CUDA function calls on multiple devices, as seen in previous multi-GPU MASS CUDA implementations. This creates an OpenMP thread for each device and then assigns that thread to manage one specific device. This can be seen in the following code segments:

```
omp_set_dynamic(0);
omp_set_num_threads(devices.size());
#pragma omp parallel {
     int gpu_id = -1;
     const int thread_id = omp_get_thread_num();
     CATCH(cudaSetDevice(thread_id));
}
```

With this code, each thread is now separately managing a single device for the duration of the program. The cudaSetDevice call means that anytime that thread makes a CUDA function call it will run on the device assigned to that thread. Therefore, any segments that need to call functions on multiple GPUs can now create a parallel section in OpenMP and any CUDA functions called within that section will be called in parallel on each device.

2.3 Mass::init()

MASS init contains numerous changes to initialize multiple GPUs instead of just one. First, the function determines the number of available devices and checks if each device is able to access the other device by calling `cudaDeviceCanAccessPeer()`. This confirms that the compute capability is sufficient, NVLink is set up properly, and the devices are compatible with the current CUDA version. Once each device is checked, it is added to the active device list, which tracks the integer ID of each device. Next, each device enables memory exchange with each other device using nested `for` loops calling `cudaDeviceEnablePeerAccess()`. Finally, init performs the OpenMP setup described above.

This setup ensures that the MASS CUDA library now has multiple GPUs setup, that they can communicate with each other using NVLink, and that they each have a thread assigned to run CUDA functions in parallel. This process is shown in the code segment below.

```
for (int d = 1; d < gpuCount; d++) {
    int canAccessPeer = 0;
    cudaDeviceCanAccessPeer(&canAccessPeer, 0, d);
    if (canAccessPeer) {
       devices.push_back(d);
    }
}

for (std::size_t i = 0; i < devices.size(); i++) {
    cudaSetDevice(devices.at(i));
    for (std::size_t j = 0; j < devices.size(); j++) {
        if (i != j) {
           cudaDeviceEnablePeerAccess(devices.at(j), 0);
        }
    }
}
```

2.4 createPlaces()

The Mass::createPlaces() function runs the following steps within the DeviceConfig object to create places over multiple GPUs. First, the DeviceConfig creates a new PlaceArray struct to store all data and pointers for places with this handle. Once this is setup, the function will determine the number of base places (places that were part of the original place array and were assigned to this device) and ghost places (copies of base places from adjacent devices to facilitate data transfer) for each device. This process is shown in Figure 2 and explained in the following paragraphs.

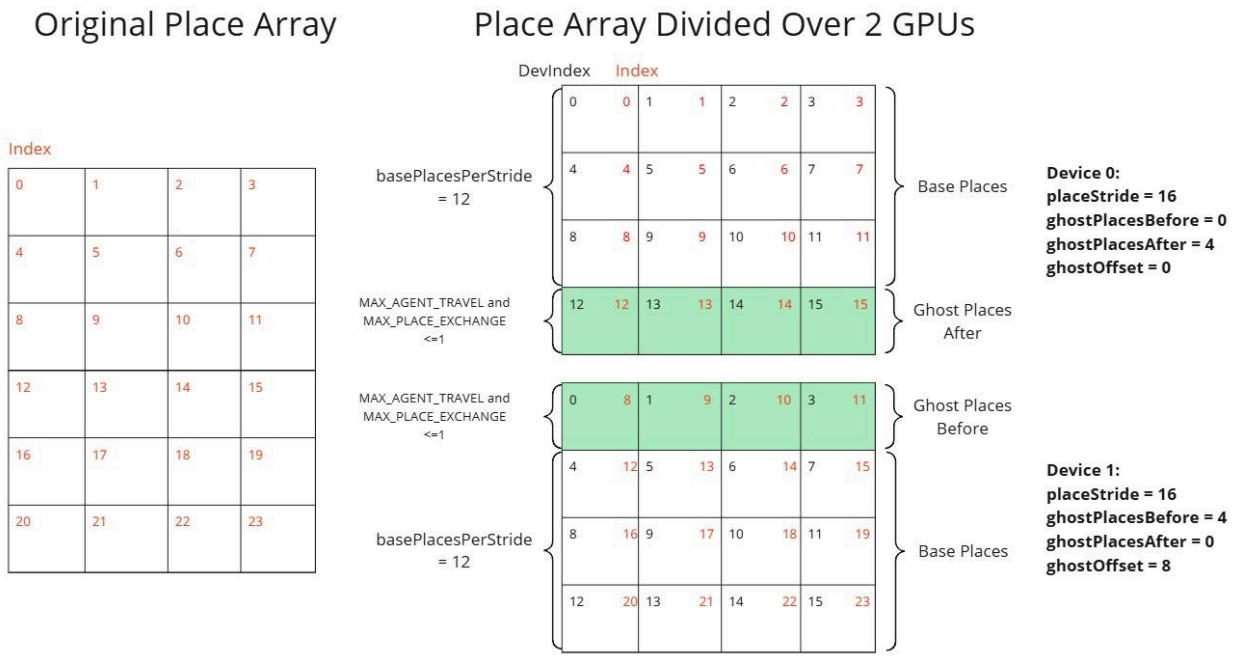

Figure 2. Place Array division between devices, including descriptions of data stored to manage the number of places and ghost places.

The number of base places is determined by dividing the overall places grid into base place strides based on the number of devices and number of rows in the original grid, such that basePlacesPerStride = (rows / numDevices) * columns. 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.

For each device in parallel, the function 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). 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 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 quantity 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.

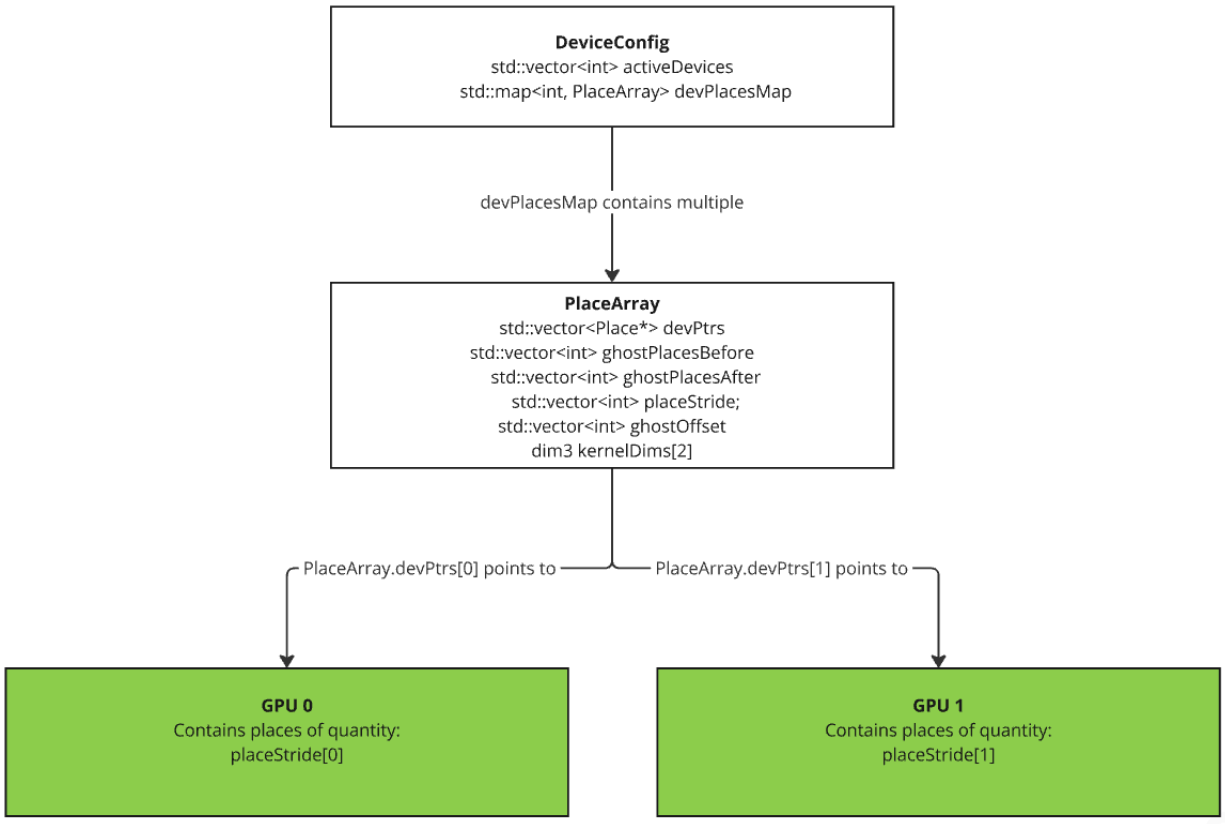

Figure 3. Data structure of PlaceArray for each Places object and pointers to places stored on each device.

Once the numbers of places are determined, the createPlaces function allocates memory on each device and initializes Place objects on them. Figure 3 shows how I modified PlaceArray 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 the code below, 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.

```
	template <typename PlaceType>
__global__ void instantiatePlaceArrayKernel(Place *places, int
ghostPlaceBefore, int ghostPlaceAfter, int placeStride, int offset)
{
	unsigned idx = getGlobalIdx_1D_1D();

	if (idx < placeStride) {
		if (idx < ghostPlaceBefore ||
		idx >= placeStride - ghostPlaceAfter) {
			new (&places[idx]) PlaceType(idx + offset, idx, false);
		}
		else {
			new (&places[idx]) PlaceType(idx + offset, idx, true);
		}
	}
}
```

Once this code is run, each device will have an array of Place objects initialized where each place knows if it is a base place or a ghost place and all base and ghost places have saved their original indices. In addition, DeviceConfig now has pointers to each array of places stored on each device.

2.5 setAttribute()

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 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.

Data for attributes is stored within the corresponding Places object as three vectors: std::vector<int> attributeTags, std::vector<void**> attributeDevPtrs, and std::vector<size_t> attributePitch. Each index in these vectors represent a different attribute, with the Tag storing a user-enumerated int label for the attribute, DevPtrs storing an array of void* to where the attribute is stored on each device, and Pitch storing the byte offset for storing 2-D attributes on the device. I modified the attributeDevPtrs to store a void** instead of void* so that it can 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 until finalizeAttribute is called.

2.6 finalizeAttribute()

While each attribute has its memory allocated when calling setAttribute, a Place object which is initiated 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, finalizeAttribute must be called to transfer the data needed to access attributes (namely, the vectors described in the previous section) to the device.

The finalizeAttribute function allocates memory for each attribute vector and stores pointers to that memory within this Places object's PlaceArray struct within the DeviceConfig object. Because CUDA does not support vectors, the DeviceConfig instead stores these as C arrays of the following form: int **d_attributeTags, void ***d_attributeDevPtrs, and size_t **d_attributePitch. For each of these arrays, the finalizeAttributes copies these values to each device. One thing to note is that each device only needs access to its *own* array of devPtrs, since trying to access the devPtrs to the other device would simply return an error. Therefore, finalizeAttributes first copies each device's DevPtrs into a new, contiguous void** array for each device, and then copies only that device's DevPtrs array to the device.

The final step in finalizeAttribute is to ensure that each Place on the device has a pointer to its attribute arrays. To do this, finalizeAttribute launches a device kernel on each device that saves pointers to each Place object's attribute arrays as Place object data members.

2.7 getAttribute()

When a Place object running a callAll function needs to access one of its attributes, it calls getAttribute(int tag, int length). This function uses the pointer to the attributeTag array stored on this Place to search the attributeTag array for the index corresponding to the given tag. Then, getAttribute simply accesses the attributeDevPtr array at that index to get a pointer to where that attribute is stored in device memory. Finally, the function accesses the attribute data at the devIndex of this Place to find the attribute value that corresponds to this particular place. It is important to note that this function now accesses the attribute at devIndex and not Index, as the original Index of a place no longer corresponds to the idx of a Place when it is split over two devices. The code for this function is shown below:

```
template <typename T>
__device__ T *Place::getAttribute(int tag, int length) const
{
    int i = find(attributeTags, nAttributes, tag);
    if (i == -1)
    {
        return NULL;
    }

    // If the length is greater than 0, then it is a 2D array
    // Otherwise, it is a 1D array

    // If it is a 2D array, we need to get the row first
    if (attributePitch[i] > 0)
    {
        // Get the row
        T *row = (T *)((char *)attributeDevPtrs[i] + devIndex *
attributePitch[i]);
```

```
        return row;
    }
    // If it is a 1D array, we can just return array[index]
    else
    {
        T *array = static_cast<T *>(attributeDevPtrs[i]);
        return &array[devIndex];
    }
}
```

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 getAttribute function with the following parameters allows for this access:

```
template <typename T>
__device__ T* getAttribute(size_t desIndex, int tag, int length) const;
```

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, this attempt would result in a segmentation fault. To address this issue in a simple way, I used the MAX_PLACE_EXCHANGE setting within the settings.h file (as described in the createPlaces section) as the limit for getAttribute from other places. This checks that the maximum number of rows away a place can access is equal or less than the rows of ghost places, otherwise the program will throw a descriptive MASS error rather than a segmentation fault.

2.8 downloadAttribute()

When a Place object running a callAll function needs to access one of its attributes, it calls getAttribute(int tag, int length). This function uses the pointer to the attributeTag array stored on this Place to search the attributeTag array for the index corresponding to the given tag. Then, getAttribute simply accesses the attributeDevPtr array at that index to get a pointer to where that attribute is stored in device memory. Finally, the function accesses the attribute data at the devIndex of this Place to find the attribute value that corresponds to this particular place. It is important to note that this function now accesses the attribute at devIndex and not Index, as the original Index of a place no longer

corresponds to the idx of a Place when it is split over two devices. The code for this function is shown below.

When a user of the MASS CUDA library sets attributes and then calls finalizeAttribute(), all data for place objects is now stored on the device. Any user-defined callAll functions can access data using the getAttribute() function described in the previous section. However, callAll functions all have void return types, so data remains stored on the device during program execution. While this configuration improves performance by minimizing data transfers between host and device, users may need to transfer data back from the device to the host in order to review that data, or to save the final results. To do this, they call the downloadAttributes function. This function copies data from the devices to the host using the cudaMemCpy with the cudaMemcpyDeviceToHost parameter.

The downloadAttributes function is a template function that returns a pointer to an array containing the given attribute for every Place. 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 the control of the memory is transferred to the user's function on completion and must therefore be freed by the user.

There are two complications regarding downloadAttributes that arise from using multiple GPUs. 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 it 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 downloads its own separate attribute memory, but the downloadAttributes function should return a single contiguous array in host memory. Therefore the downloaded memory must be combined to be accessed by a single pointer. The following code shows how this is accomplished:

```
template <typename T>
T* DeviceConfig::downloadAttributes(int handle, void** d_attributeDevPtrs,
size_t pitch, unsigned int length, unsigned int qty) {

    PlaceArray* p = &devPlacesMap.at(handle);

    // Allocate host memory to hold attribute data
```

```
T* h_attribute = new T[qty * length];

#pragma omp parallel {
        int gpu_id = -1;
        cudaGetDevice(&gpu_id);

        // Copy attributes to temporary h_temp array on host (also
        copies ghost place attributes)
        T* h_temp = new T[p->placeStride[gpu_id] * length];
        if (length <= 1) { // If the attribute is a 1D array
              (cudaMemcpy(h_temp, d_attributeDevPtrs[gpu_id], sizeof(T)
              p->placeStride[gpu_id], D2H);
        } else { // If the attribute is a 2D array
              cudaMemcpy2D(h_temp, length * sizeof(T),
              d_attributeDevPtrs[gpu_id],   pitch, length * sizeof(T),
              p->placeStride[gpu_id], D2H);
        }

        // Determine how many non-ghost Places are on this device
        int numPlaces = qty / p->placeStride.size();
        // Determine the starting position of this device's attributes
        within  the combined array
        int startPos = 0;
        for (int i = 0; i < gpu_id; i++) {
              startPos += numPlaces;
        }

        // Copy from h_temp (excluding ghost places) to h_attribute,
        starting at the determined startPos
        std::copy(h_temp + (p->ghostPlacesBefore[gpu_id] * length),
        h_temp +((p->ghostPlacesBefore[gpu_id] + numPlaces) * length),
        h_attribute + (startPos * length));

        delete[] h_temp;
  }
  return h_attribute;
}
```

This code first allocates host memory for the entire Place array (combined devices) with h_attribute.

Next, within a parallel section on each device the function creates temporary host memory with h_temp

and copies the attribute memory (including ghost places) from the device to this memory location. Next, the function determines the number of base places on each device to determine how much memory to copy when ghost places are excluded. In addition, the function offsets each subsequent device by this number, so that each device's base places are slotted correctly into the contiguous array. Finally, the device offsets from the start of each h_temp attribute memory to the start of the base places, then copies memory from each device to h_attribute. The steps of this operation are shown in Figure 4 below.

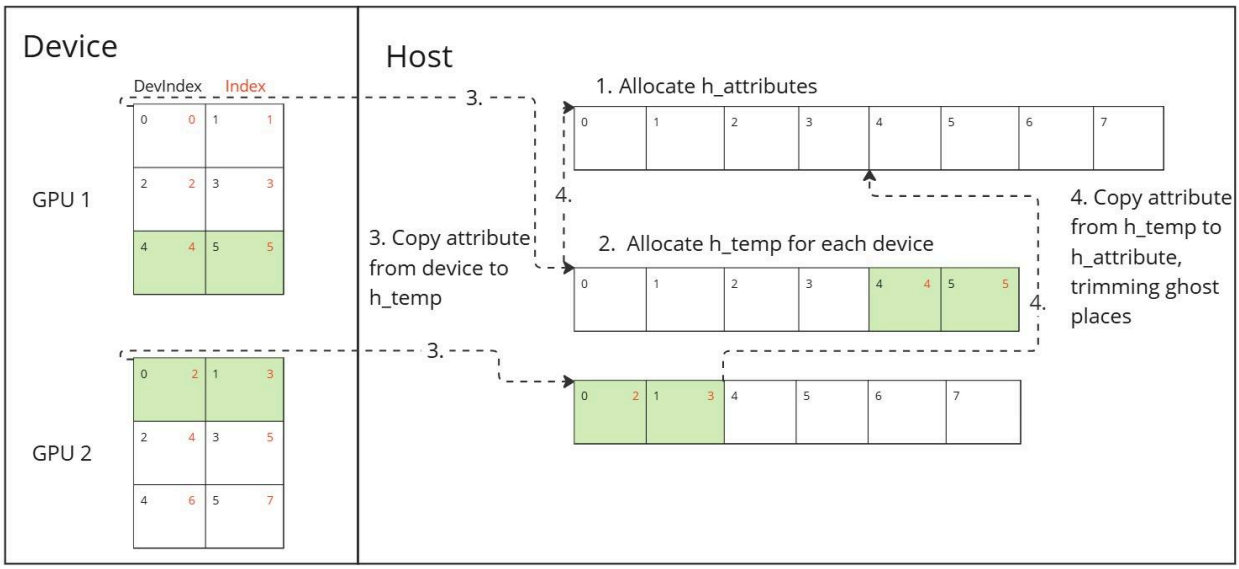

Figure 4. Steps to perform downloadAttributes() by copying attribute data from device to temporary host locations h_temp, then to a final contiguous h_attribute location.

2.9 exchangeAll()

The Places object's exchangeAll function determines and saves the neighbors of each Place. Effectively, this gives each Place object an array storing the indices of its neighboring Place objects, and also ensures that each Place has a pointer to those neighboring Place objects so that it can access their data or call functions on them. A Place object's neighbors are defined by the user by calling the exchangeAll function. The user creates and provides a vector containing relative coordinates for each of a Place object's neighbors. An example of this vector and the corresponding neighbors is shown in Figure 5 below.

```
vector<int*> destinations;
int north[2] = {-1, 0};
destinations.push_back(north);
int east[2] = {0, 1};
destinations.push_back(east);
int south[2] = {1, 0};
destinations.push_back(south);
int west[2] = {0, -1};
destinations.push_back(west);
places->exchangeAll(destinations);
```

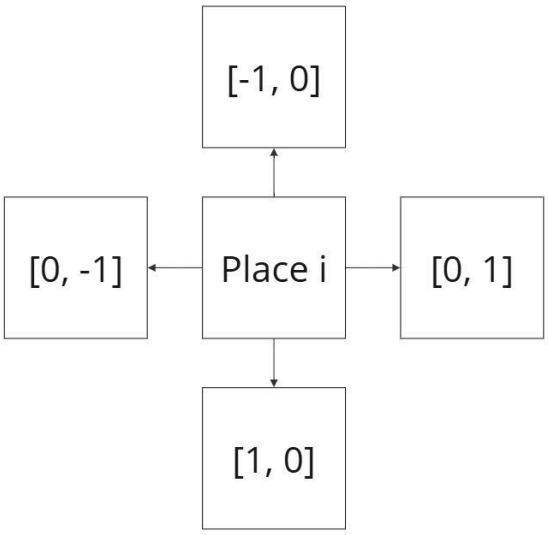

Figure 5. Code run by user to create the destinations vector, and corresponding visualization of the neighbors that will be set for each Place by the exchangeAll(std::vector<int*>* destinations) function.

In this implementation for a two-dimensional array of Place objects, the user is setting 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, I 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 I adjusted the kernel to perform the calculation in terms of both the device idx (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. I 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.

2.10 exchangeGhostPlaces()

As described in the createPlaces section, 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 6 below.

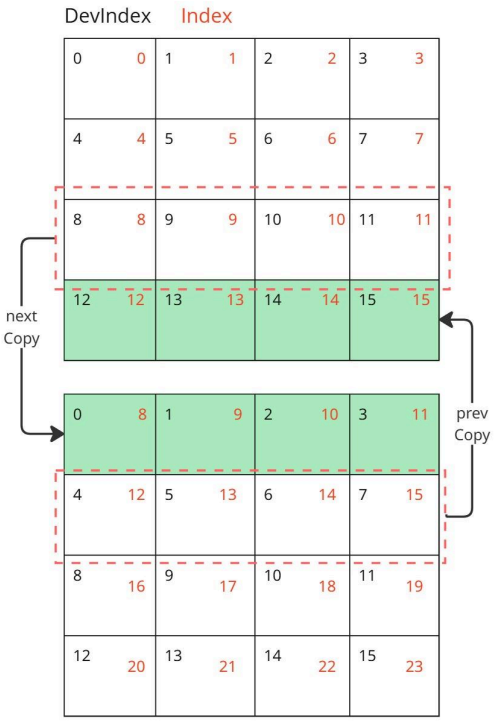

Figure 6. exchangeGhostPlaces copying data from base places to the next and previous device's ghost places. This example shows one row of ghost places, so only one row of base places needs to be copied.

The exchangeGhostPlaces function runs the following code to copy data between devices:

```
pragma omp parallel {
      int gpu_id = -1;
      CATCH(cudaGetDevice(&gpu_id));

      if (activeDevices.size() > 1 && gpu_id > 0) {
            int prev_gpu = activeDevices[gpu_id - 1];
            // Find the offset of the Ghost Places on the previous device
```

```
            int dstOffset = p->placeStride[prev_gpu] -
            p->ghostPlacesAfter[prev_gpu];
            // Find the offset of the Base Places on this device
            int srcOffset = p->ghostPlacesBefore[gpu_id];

            for (int i = 0; i < p->nAttributes; i++) {
                  int count = p->ghostPlacesAfter[prev_gpu]; // Number of
                  bytes to copy
                  int offsetSize; // Size in bytes that each Place stores
                  for this attribute
                  if (attributePitch->at(i) > 0) { // If attribute is 2D
                        count *= attributePitch->at(i);
                        offsetSize = attributePitch->at(i);
                  } else { // If attribute is 1D
                        count *= attributeSize->at(i);
                        offsetSize = attributeSize->at(i);
                  }
                  // Determine the start ptr of the Ghost Places in the
                  previous device
                  void* dst = (char*)(attributeDevPtrs->at(i)[prev_gpu]) +
                  (offsetSize * dstOffset);
                  // Determine the start ptr of the Base Places in the
                  current device
                  const void* src= (char*)(attributeDevPtrs->at(i)[gpu_id])
                  + (offsetSize * srcOffset);

                  CATCH(cudaMemcpyPeerAsync(dst, prev_gpu, src, gpu_id,
                  count));
            }
      }
```

The code above shows only the copying from a device's base places to the previous device's ghost places, but copying in the opposite direction (current device to next device) 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 how the offsets (in number of places) to where the source and destination places begin. This is shown in the calculation of dstOffset and srcOffset in the code. For example, in the example shown in Figure 6 the srcOffset would be four places on the second device, to point to the start

of the place with DevIndex four, while the dstOffset 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). If it is one-dimensional, the number of bytes to copy is equal to number of ghost places * sizeOf(AttributeObjectType). If it is two-dimensional, the number of bytes is equal to number of ghostPlaces * attributePitch. Similarly, the
2. Use pointer arithmetic to determine the destination pointer, dst, to the start of the ghost places on the previous device
3. Use pointer arithmetic to determine the source pointer, src, to the start of the corresponding base places on this device
4. Copy data between the devices using cudaMemcpyPeerAsync. 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.

3. Evaluation

I ran single-GPU MASS CUDA and multi-GPU MASS CUDA on a series of test functions that created places, set attributes, called callAll functions, and downloaded the results. I performed these tests for a range of place array sizes, from 100 to 1000000 Place objects. The results of this benchmarking are shown in Table 1.

Table 1: Runtimes of benchmark testing programs in single-GPU and multi-GPU MASS CUDA.

Number of Places	Single-GPU Runtime (ms)	Multi-GPU Runtime (ms)	% Difference
100	5400	6724	24.53%
1000	5407	6637	22.75%
10000	5429	6657	22.62%
100000	5424	6733	24.14%
1000000	5583	7140	27.89%

This code shows a clear performance hit from splitting place data over multiple GPUs. This is likely due to several factors. First, all place and attribute creation must be performed twice for each GPU. While this is run in parallel within OpenMP parallel sections, there still may be performance impacts from running multi-threaded operations rather than a single operation. The second and primary reason for the performance hit is likely due to the exchangeGhostPlaces function. This function runs every time a callAll function runs, and must copy data between the two GPUs. This means that the next callAll function cannot run until the copying is complete.

4. Future Work

4.1 Testing with Heat2D

As discussed in the Evaluation section, for each of the core functions listed in the Implementation section I ran a series of tests to confirm that the programs were working correctly. The next step in the testing process is to confirm that multi-GPU MASS CUDA can function correctly with one of the benchmark programs used by the MASS lab to test program performance. Because my project will only complete the place implementation and not agents, I selected the Heat2D benchmark program, which only uses places. The Heat2D program was updated to run on the newest version of MASS (the version that includes attributes) so it should be able to run on my multi-GPU implementation. I began setting up Heat2D for testing towards the end of Winter quarter, and I anticipate fixing any errors and benchmarking my multi-GPU MASS CUDA using the program in the beginning of Spring quarter.

4.2 Reorganization of Dispatcher and DeviceConfig

Section 2.1 and Figure 1 of this report discuss the organization of the MASS CUDA library, with a hierarchical object-oriented structure of Mass.h, Dispatcher.h, and DeviceConfig.h. However, as I have finished the code for the places implementation I have found the separation of the Dispatcher and DeviceConfig to be unnecessary. As currently written, the Dispatcher's main function is to pass function calls from Place, Places, and Mass objects to the DeviceConfig. The only data that the Dispatcher stores itself is information on the neighbor settings of a place array, as described in section 2.9 on exchangeAll. However, in other implementations of MASS, such as Java and C++ the Dispatcher object is the main distribution object for MASS functionality. Therefore, even though in MASS CUDA the Dispatcher is

largely redundant, I plan to reorganize the MASS CUDA library to eliminate the DeviceConfig object and move all its functionality to the Dispatcher object in order to maintain continuity with other forms of MASS. I plan to do this as a final cleanup step after I finish testing my implementation.

4.3 Agents

In discussions with Professor Fukuda, we decided that it would be preferable for me to complete a fully-functional implementation of multi-GPU MASS places than to rush to complete a barebones implementation of agent functionality. Therefore, updating the MASS CUDA library to use multi-GPU agents will be taken on by another student as a future project. However, I intend in my final white paper to include discussion and analysis of my suggested approach for this project and any challenges I foresee with the implementation as a guide for future work.

5. How to Run Code

The code written this quarter is on a new branch of the MASS CUDA core library called holt/multi-gpu-cuda. To run the test code, open a new terminal in the mass_cuda_core folder and run the following commands:

- make develop
- make build
- make test

To change which tests to run, edit the main method in the test/main.cu file and change the testing::FLAGS_gtest_filter = "MASS_Misc.*" to match the tests you want to run.

References

[1] 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, Aug. 2011, pp. 145–148. doi: 10.1109/IEMBS.2011.6089915.

[2] W. Liu, “Agent-Based Models Library Over Multiple GPUs: Term Report,” University of Washington, Bothell, Term Report, Dec. 2023. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/WarrenLiu_au23.pdf

[3] W. Liu, “Agent-Based Models Library Over Multiple GPUs: Term Report,” University of Washington, Bothell, Term Report, Mar. 2024. Accessed: Jul. 06, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/WarrenLiu_wi24.pdf

[4] W. Liu, “Programmability and Performance Enhancement of MASS CUDA” University of Washington, Bothell, Capstone Report, Jun. 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/WarrenLiu_whitepaper.pdf

[5] B. Pittman, “Multi Agent Spatial Simulation (MASS) in multiple GPU Environment with NVIDIA CUDA” University of Washington, Bothell, Capstone Report, August. 2021. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BenPittman_whitepaper.pdf

[6] M. Fukuda, “MASS: A Parallelizing Library for Multi-Agent Spatial Simulation.” Accessed: Jul. 07, 2024. [Online]. Available: https://depts.washington.edu/dslab/MASS/index.html

[7] N. M. Ho, N. Thoai, and W. F. Wong, “Multi-agent simulation on multiple GPUs,” Simulation Modelling Practice and Theory, vol. 57, pp. 118–132, Sep. 2015, doi: 10.1016/j.simpat.2015.06.008.

[8] P. Richmond, R. Chisholm, P. Heywood, M. K. Chimeh, and M. Leach, “FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs,” *Software: Practice and Experience*, vol. 53, no. 8, pp. 1659–1680, 2023, doi: 10.1002/spe.3207.