Agent-based Modeling with MASS CUDA on Multi-GPUs over Multi-Hosts

Haobo Peng

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

Master of Science in Computer Science & Software Engineering

University of Washington

2026

Reading Committee:
Professor Munehiro Fukuda, Chair
Professor Wooyoung Kim
Professor Clark Olson

Program Authorized to Offer Degree:
Computer Science & Software Engineering

University of Washington

Abstract

Agent-based Modeling with MASS CUDA on Multi-GPUs over Multi-Hosts

Haobo Peng

Chair of the Supervisory Committee:
Professor Munehiro Fukuda

Agent-based modeling (ABM) is an approach for simulating the behaviour of a system through simple interation between individual agents and the environment, which is widely adopted in various fields. For better simulation performance, attempts have been made to utilize the GPU to parallelize the simulation tasks, and the CUDA version of the Multi-Agent Spatial Simulation library (MASS CUDA) is one of the general purpose ABM libraries aims at providing high-level APIs for users to implement and execute their simulations on a single NVIDIA GPU. However, as the scale and complexity of the ABM simulation increase, a single GPU may not be sufficient to handle large-scale simulations that cannot fit into the device memory. To handle such situations, a new version of MASS CUDA is intorduced in this paper that support offloading simulation tasks across multiple GPUs over multiple hosts. With the buffer-based collaborative communication design and Nvidia Collective Communication Library (NCCL) for cross-device data transfer, collaboration among multiple GPUs on the same ABM simulation is realized. Together with the hybrid cross-hosts parallelism pattern, the GPUs can be distributed on different hosts. Moreover, with the refined memory layout for attributes and `Place` and `Agent` classes, the new version of MASS CUDA achieves significantly better performance and spatial scalability for large-scale simulations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

MASS: Multi-Agent Spatial Simulation Library

MASS CUDA: MASS that is implemented on GPU with CUDA

Places: An N-dimensional grid representing the environment in which agents interact.

Place instance: A single instance (cell) of a *Places* grid

Agents: A collection of entities residing on and interacting with each place instances in the *Places* grid.

Agent instance: A single instance of an *Agents* collection

Ghost Places: A set of place instances that act as a copy of another set of place instances on a neighboring GPU

Chapter 1

INTRODUCTION

Agent-Based Modeling (ABM) is a computational modeling approach used to simulate the actions and interactions of autonomous individuals, referred to as agents, to understand the behavior of a system. Each agent follows a set of predefined rules and behaviors, interacting with other agents and their environment over time. ABM has been widely adopted in various fields due to its ability to capture and simulate complex phenomena from simple interactions. Common applications include simulating epidemic spreading in epidemiology, traffic flow and congestion in urban planning, fluid dynamics in physics, and crowd behaviors in sociology.

However, implementation of an ABM simulation from scratch can still be a challenging and time-consuming task. In addition, as the complexity and scale of ABM simulation increases, the performance and scalability of single CPU on single computing node has become a main limitation. To address that, the Multi-Agent Spatial Simulation (MASS) Library was developed to support distributing and parallelizing a simulation task across multiple computing nodes while abstracting the complexity of execution, multithreading, communication and synchronization. It introduces the concept of *Places* to represent the environment and *Agents* to represent a collection of agents, allowing users to focus on defining the behavior of each individual agent while the library takes care of the rest.

Besides parallelization using multiple CPUs over multiple nodes, another strategy is to utilize the GPU, which contains a large number of cores and supports executing thousands of threads concurrently. Although a single GPU core is typically slower than a CPU core, the higher level of parallelism allows GPUs to achieve better performance for large-scale simulations. For that reason, a dedicated version of MASS, named MASS CUDA, was developed to support offloading the simulation task to Nvidia GPUs.

Currently, the MASS CUDA library only supports executing simulations on a single GPU. Although there is an initial multi-GPU implementation, it only supports *Places* and only support GPUs connected over NVLink on the same hosts. In addition, the performance of the current implementation is not ideal due to the inefficient memory layout design. This paper focus on resolving these problems. For the former, a buffer-based collaborative communication design and the Nvidia Collective Communication Library (NCCL) are adopted to support both *Places* and *Agents* on multiple GPUs. Together with a hybrid cross-host parallelism pattern, the participating GPUs can be distributed across different hosts. For the latter, the memory layout of the data maintained on GPUs, including the attributes and the instances of `Place` and `Agent` classes, are redesigned to better utilize GPU memory bandwidth and improve memory efficiency.

Chapter 2
BACKGROUND

2.1 MASS

The MASS Library, short for Multi-Agent Spatial Simulation Library, is a general purpose ABM library available on Java and C++ that provides a high-level abstraction over high-performance and distributed simulation. It supports executing simulations over multiple threads and multiple computing nodes while abstracting away the complexity of cross-node data transfer and synchronization from the users [1]. To implement an ABM simulation using MASS, users only need to define the behavior of the agents using the provided high-level APIs and the library will automatically handle the distribution of agents on multiple computing nodes and the necessary communication between them.

The MASS Library generalize ABM problems with two main concepts: *Places* and *Agents*. A *Places* is an N-dimensional grid representing some environment for simulation and each cell of the grid is referred to as a place instance. Each place instance maintains its own data attributes, performs computations and exchanges data with neighboring place instances. An *Agents* is a collection of entities residing on place instances and interacting with them. Each of them is referred to as an agent instance, which aligns with the common concept of an Agent in ABM. An agent instance also manages its own data attributes and can access the data attributes of the place instance it is currently residing on. Besides, it can also move to another place instance (Migration), spawn child agent instances on another place instance, or terminate itself [1].

To define application specific behavior during simulation, users can inherit the `Place` and `Agent` classes and provide custom data attributes and methods. In order to allow the library to correctly invoke the custom methods, users also need to provide an id for each method

and override the `callMethod` method to route each invocation request to the correct method based on the id. With these custom behaviors of each place instances and agent instances defined, users can then coordinate the overall simulation process in the main function using the `callAll`, `exchangeAll` and `manageAll` APIs. The `callAll` function invokes a method on all place instances or agent instances, the `exchangeAll` function transfer data between neighboring place instances and the `manageAll` function handles the migration, spawning and termination process of agent instances [1].

2.2 MASS CUDA

The MASS CUDA Library is a new version of MASS that utilizes Nvidia's GPU to reach even higher simulation performance. Compared to CPU, GPU has much more cores, allowing higher level of parallelism by spawning thousands of threads. In MASS CUDA, *Places* and *Agents* are allocated on GPU memory and operations on each place instance and agent instance are delegated to GPU threads using Kernel functions. Similar to MASS on Java and C++, the complexity of GPU programming is abstracted away from the users, allowing them to use similar APIs to implement their simulations [2].

The MASS CUDA Library has undergone one major update since its initial design and implementation. Initially, data attributes of each place instance and agent instance were stored in a subclass of `PlaceState` and `AgentState` respectively, while each place instance and agent instance only maintain a pointer to its state. These states are allocated in a contiguous array in GPU global memory [2]. However, this design encounters performance issue due to uncoalesced memory access every time a place instance or agent instance access its attributes in its state. To fix this issue, a new design was proposed and implemented, which abandons the state classes and adopts a Structure of Arrays (SoA) memory layout for attributes of place instances and agent instances. In this design, values of each attribute are stored in separated arrays in GPU global memory. The start address of these attribute arrays are maintained by another array, whose address is maintained by each place instance and agent instance. To identify the attributes for accessing, a new concept named attribute

tag is introduced, which is a unique integer identifier for each attribute. This new design significantly improves the performance of MASS CUDA by enabling coalesced memory access patten for attribute access. In the GameOfLife benchmark, the new design uses 60.13% less time in each simulation iteration at 2500×2500 problem size compared to the old design [3].

After this major update, the architecture of MASS CUDA, as presented in Figure 1, mainly contains three components: MASS API, Dispatcher and DeviceConfig. The MASS API, consisting of the `Places`, `Agents` and `Mass` classes, is the top-level interface for the users to interact with the library. The Dispatcher is a singleton class containing the GPU kernel functions, the host-side logics for invoking these kernels and data transfer between CPU and GPU. The DeviceConfig is a class responsible for allocating and managing device resources such as the pointers to the attribute arrays, place instances and agent instances on GPU memory [3].

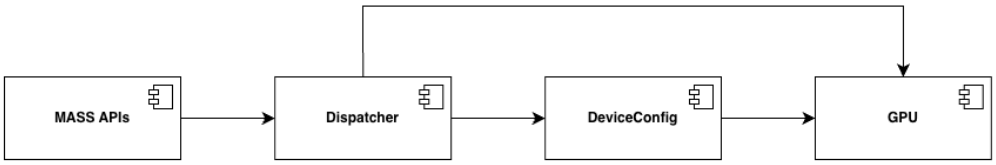

Figure 1: MASS CUDA Architecture

2.3 Initial Multi-GPU Implementation

Although the SoA design has significantly improved the performance of MASS CUDA, the execution is still limited to single GPU, blocking it from handling larger scale simulation that cannot fit in the memory of a single GPU. To address this issue, an initial multi-GPU implementation for *Places* was developed. In this design, the *Places* grid is split into multiple segments linearly along the first dimension and each GPU is responsible for processing one segment [4]. As a result, each GPU only maintains a portion of the place instances in the complete grid, allowing MASS CUDA to process larger simulations using more GPUs.

In a simulation using MASS CUDA, a place instance can access the data of its neighbouring place instances. However, with the multi-GPU design, the neighbor of some place instances may be on another GPU and not directly accessible. For this difficulty, the concept of Ghost Places is adopted. The Ghost Places is a collection of additional place instances at the boundary of each segment of the grid representing a copy of the place instances on the neighboring GPU. Figure 2 shows an example of a 6×4 *Places* split into two segments with 2 GPUs. Theoretically, each GPU should contain a 3×4 segment. However, with the Ghost Places design, each GPU actually contains a 4×4 segment, where the last row of the first segment are the Ghost Places holding copies of the second row of the second segment and the first row of the second segment are the Ghost Places holding copies of the third row of the first segment. When the attribute values of the actual place instances on the neighboring GPU are updated, the library will copy the new values to the Ghost Places to maintain the consistency. These Ghost Places are read-only instances only used for supporting accessing the neighboring place instances on the neighboring GPU, as a result, the user-defined methods will never be invoked on them during the `callAll` API call [4].

With this initial multi-GPU implementation, the MASS CUDA library is able to handle larger simulation with better performance using multiple GPUs. In the Heat2D benchmark, the multi-GPU implementation with 2 GPUs reduce 32% of the total runtime at 3000×3000 problem size and increase the maxinum feasible problem size by 77% compared to the previous single GPU implementation [4].

2.4 Current Limitations

Although the initial multi-GPU implementation has presented promising results, there are still several limiations. First, the implementation is not complete since only *Places* is supported to be distributed to multiple GPUs while *Agents* is still limited to single GPU execution. Second, only GPUs on the same host connected over NVLink is supported. If NVLink is not available or the simulation scale exceeds the maximum capability of multiple GPUs on a single host, the library will not be able to handle that. Third, even with the new

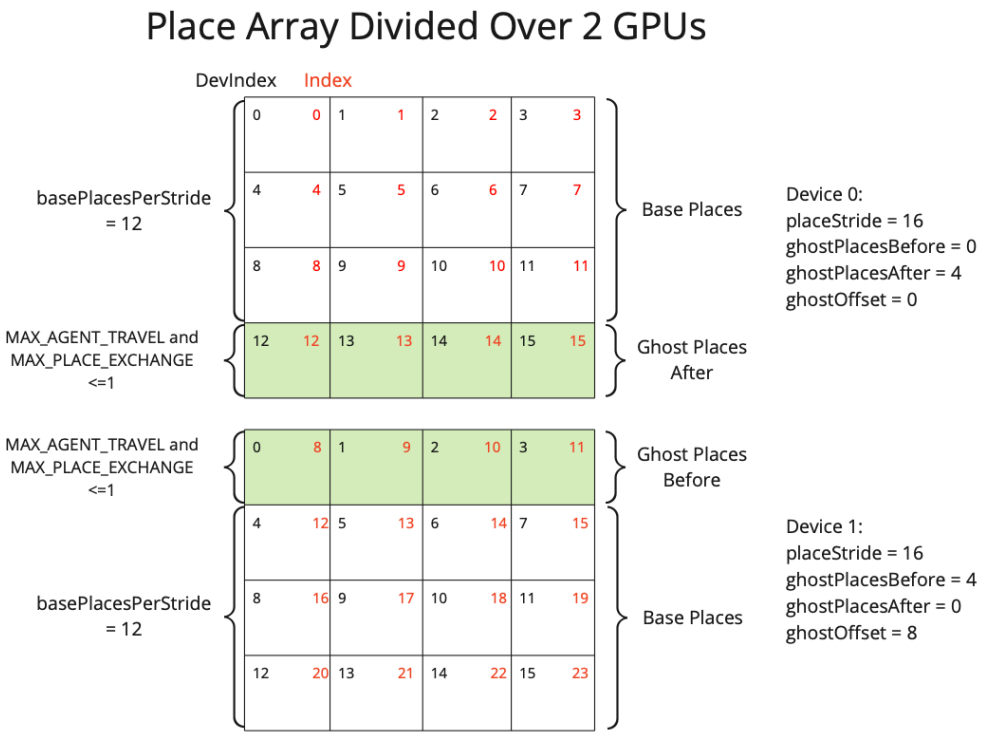

Figure 2: Ghost places [4]

SOA design being adopted, the current memory layout design is still not ideal, especially for 2D attributes. As a result, coalesced memory access is still not perfectly achieved and the GPU memory bankwidth is still not fully utilized. Finally, the current architecture design of MASS CUDA is not ideal for further extension to support multi-GPU execution for *Agents* due to the bulky `Dispatcher` design. Multi-GPU implementation for *Agents* requires more complex logics being added into the `Dispatcher` class, which can make it hard to maintain.

This paper focus on addressing these limitations within the new version of MASS CUDA, involving the following aspects. First, the initial multi-GPUs implementation is extended and generalized to support both *Places* and *Agents*. Second, add support for GPUs connected over different protocols other than NVLink, such as PCIe. Third, add support for GPUs distributed across different hosts. Fourth, improve the performance and scalability of MASS

CUDA by refining the memory layout design. Finally, the architecture is redesigned to separate different responsibility into different components for better maintainability and extensibility.

Chapter 3

RELATED WORK

3.1 Similar ABM Libraries

Besides MASS CUDA, there are other ABM libraries that support GPU parallelization. The FLAME GPU 2 is one of them, which is also the main competitor of MASS CUDA. It is an agent-only general purpose ABM library that also provides high-level APIs for defining agent behavior and supports executing the simulation on GPU. Unlike MASS CUDA, it does not include the concept of *Places* and usually need to simulate the environment using separated agents. In addition, it uses messages based communication between agents instead of direct memory access, which increases concurrency safty but can introduce higher overhead. Another important feature of FLAME GPU 2 is its ability to execute multiple agent functions concurrently with custom dependency definitions, which may further improve the performance in some complex simuations. Although the perforamnce of FLAME GPU 2 has been highly optimized on single GPU, it does not support executing simulation over multiple GPUs [5].

MCMAS is another ABM framework that provides high-level Java APIs over OpenCL for executing simulations on both many-core CPUs and GPUs. It features a plugin system for dynamically adding functionalities and supports process parallelization by using optimized primitives to accelerate certain steps of simulations, or model parallelization by implementing new plugins. This allows it to integrate with existing ABM frameworks to accelerate simulations without changing existing architectures. However, it requires falling back to writing OpenCL code if the required primitives are not provided, and it does not support multi-GPU execution [6].

These existing ABM libraries with GPU support are well optimized for single GPU

execution and provide high-level APIs for users to hide the complexity of GPU programming. However, currently none of them include complete support for multi-GPU execution, limiting their performance and scalability for larger simulations. This is exactly one of the main goals of this work, which is completing the multi-GPU support for MASS CUDA.

3.2 ABM Applications with Dedicated GPU Parallelization

Besides the high level general purpose ABM library, there are also researches solving specific ABM problems using multiple GPUs with dedicated codes. One research focusing on SIM-CoV modeling the spread of virus and immune response in lungs has sucessfully executed the simulation on arbitary number of GPUs on multiple hosts, allowing it to solve larger problem instances that cannot fit into a single A100 GPU while still outperforming the CPU version. It splits the voxels representing the lung space into smaller tiles where each tile is assigned to a GPU. Communications between GPUs are done with the UPC++ library and the concept of Ghost Voxels, which is similar to the Ghost Places in the multi-GPU MASS CUDA. The research also introduces a random number based conflict resolution algorithms to avoid T cells from moving into the same voxel during the migration process [7]. This can be a good mechanism for simulations that allow only one agent per environment position, but may not be a good general solution for a general purpose ABM library like MASS CUDA.

Another research implements a traffic simulation using multiple GPUs. It introduces a very compact data structure to model the road network and the vehicles. Specifically, it uses one char array to represents a lane, where each char represents one meter and the value shows the speed of the car there. All the lanes are stored together in one large array. It also introduces a sophisticated graph partitioning method to divide the road network and distribute them among different GPUs, which helps to minimize communication overhead among GPUs [8].

These application specific implementations include sophisticated optimization and multi-GPU parallelization designs, and achieves good performance for their specific problems. However, most of these designs and optimizations are not suitable for being generalized for

different ABM simulations, which limits their use cases to very specific domains. This is exactly what a general purpose ABM library such as MASS CUDA aims to address.

Chapter 4

DESIGN

4.1 Architecture and Components

In the previous architecture, the `Dispatcher` was already a highly complex class, and this complexity was further exacerbated by the introduction of multi-GPU support since it is responsible for kernel implementation, host side execution logic and communication logic. To improve the maintainability and extensibility of MASS CUDA, a new architecture is proposed here. The foundamental idea of the new design is to strictly separate the communication logic, kernel / host execution logic and resource menagement into different components. Specifically, as presented in Figure 3, the new architecture now includes four main components: the MASS API, the `RootDispatcher`, the `DeviceDispatcher` and the `DeviceDriver`.

The `DeviceDriver` is a simplified version of the previous `DeviceConfig` class responsible for GPU related resource management. Unlike the `DeviceConfig`, it has no domain knowledge about the MASS APIs and only focuses on proving primitives for managing and interacting with GPU resources such as memory, streams and events.

The `DeviceDispatcher` handles the host and kernel execution logic that supports the MASS CUDA APIs on a single GPU. Its role is very similar to the `Dispatcher` class in the previous single-GPU architecture. Each GPU is assigned exactly one `DeviceDispatcher` instance, which maintains a `DeviceDriver` instance to manage the resources for that specific GPU. Because it does not handle data transfers between GPUs, each `DeviceDispatcher` works independently and has no knowledge of other GPUs or `DeviceDispatcher` instances.

The `RootDispatcher` is the coordinator of all the `DeviceDispatcher` instances, responsible for dispatching MASS API requests to them and handling the cross-device data

transfer. Each host has one `RootDispatcher` instance assigned, which maintains a set of `DeviceDispatcher` instances together with the neighboring information for each them to support cross-device data transfer.

Figure 3: Redesigned Architecture

4.2 Grid Topology

In the initial multi-GPU implementation, a *Places* is split linearly along the first dimension into multiple segments. This strategy is referred to as Linear Topology. Another potential strategy for spliting the *Places* grid is to split it into smaller blocks or tiles, which is referred to as Grid Topology.

The linear topology is very strageforward to understand and implement. No matter what dimension the *Places* is, each segment only has two neighbors at most. In addition, the place instances that need to be transferred to the neighboring GPU are always contiguous in memory, which further simplifies the implementation for communication. On the other

hand, the number of neighbours for each block in the grid topology increases with the count of dimensions. When dimension is 2, each block can have up to 8 neighbors (top, down, left, right, top-left, top-right, down-left, down-right) and when dimension is 3, each block can have up to 26 neighbors. In addition, the place instances that need to be transferred are typically not contiguous in memory. As a result, the implementation of cross-device communication for the grid topology is way more complicated than that of the linear topology.

However, the grid topology can significantly reduce the amount of data being transferred. Consider an $N \times N$ *Places* split by 4 GPUs. Figure 4 and Figure 5 present the split results of the linear topology and the grid topology respectively, where the colored lines represent the boundaries where data exchange occurs between neighboring GPUs. With the linear topology, there are 3 internal boundaries, each of length N. Since data must be sent in both directions across each boundary, the total amount of data is $3 \times N \times 2 = 6N$. In contrast, the grid topology divides the *Places* into a 2×2 grid, creating 4 internal boundaries of length $N/2$. Thus, the total amount of data for the grid topology is $4 \times N/2 \times 2 = 4N$, which is only 2/3 of the linear topology. This demonstrates that the grid topology can achieve better communication efficiency by reducing the amount of data being transferred. Although it is more complicated to implement, the performance benefit is more important for MASS CUDA. As a result, it is chosen for the new multi-GPU design of MASS CUDA.

4.3 Cross-Device Communication Design

4.3.1 Communication for Places

Similar to the initial multi-GPU implementation, the idea of Ghost Places is adopted for *Places* to support cross-device communication. But due to the grid topology design, the place instances involved are no longer just the two contiguous regions at both ends, but several non-contiguous regions along the boundaries of the blocks. To better illustrate the design, two key concepts need to be introduced: the **Boundary Region** and the **Ghost Region**. A Boundary Region consists of normal place instances within a block that need

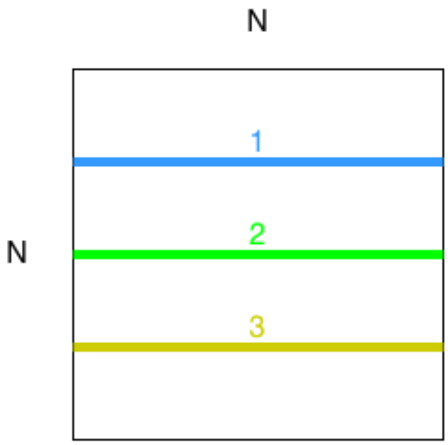

Figure 4: Linear topology with 4 GPUs

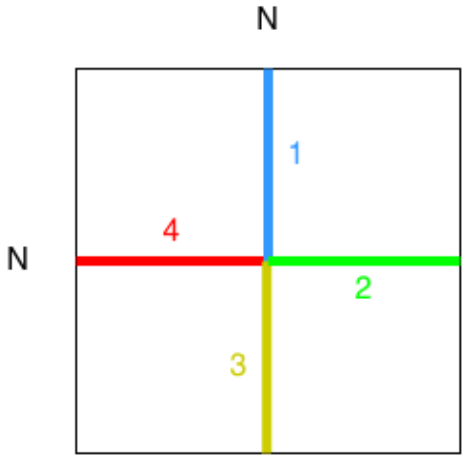

Figure 5: Grid topology with 4 GPUs

to be transferred to the corresponding Ghost Region on a neighboring GPU, while a Ghost Region is a region of place instances that act as a local copy of the corresponding Boundary Region on a neighboring GPU. Boundary Regions and Ghost Regions always come in pairs with each pair taking care of one neighbor of a block.

Figure 6 shows an example of the Boundary Regions and the Ghost Regions for a 2D *Places*, where the cells with dashed lines are place instances within the Ghost Regions and the cells with solid lines are normal place instances of a block. The matching colored region pairs (e.g.: regions with dark blue and light blue) represent corresponding Boundary Region and Ghost Region pairs. For 2D *Places*, there are 4 different types of these pairs. (1) neighbors on the left and right, (2) neighbors on the top and bottom, (3) neighbors on the top-right and bottom-left, (4) neighbors on the top-left and bottom-right.

For *Places*, the data that actually being transferred during cross-device communication are the place instances within all the Boundary Regions of each block. For each of these regions, they will be copied to the corresponding Ghost Region on the neighboring GPU.

In the design of the initial multi-GPUs support, data transfer is triggered automatically after every execution of the `callAll` function, and all attributes are transferred. However, in

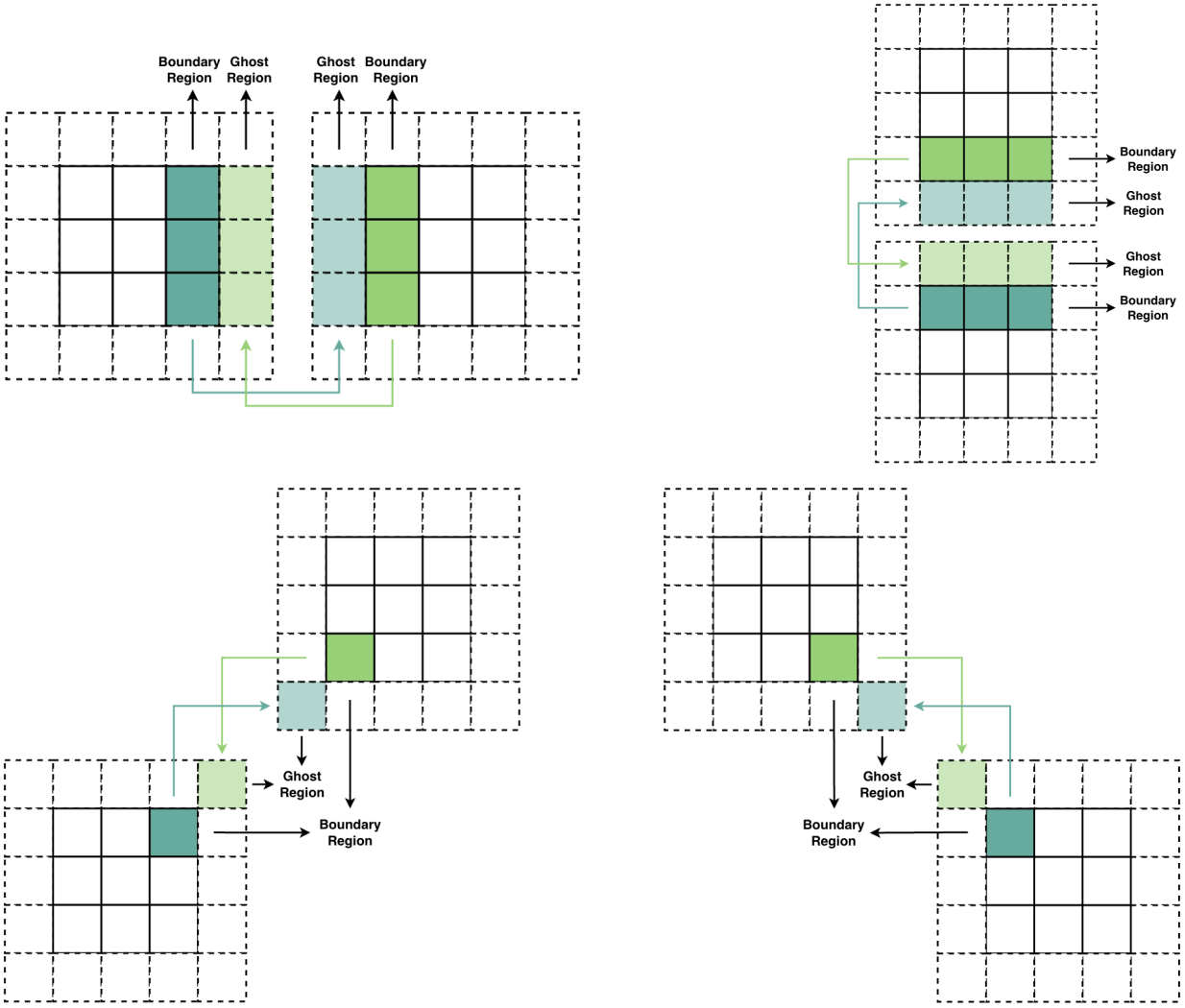

Figure 6: Boundary Regions and Ghost Regions with Grid Topology for 2D Blocks

certain simulations, not all `callAll` invocations require data from neighbors. For instance, in a simulation with three `callAll` invocations per step where only the final one access data from neighboring place instances, data transfer is only needed before the last `callAll`. Furthermore, unchanged attributes do not require being transferred. Consequently, the previous design includes unnecessary communication overhead.

To address this issue, a new function named `syncPlaceBorder` is introduced for users

to manually control the cross-device communication process. It provides two overloads: one without arguments that transfers all attributes, and another that accepts an array of attribute tags to transfer only the specified attributes. This new design enables users to fine tune the communication process and reduce overhead. Although such manual control may slightly increase the programming complexity due to additional API calls, the performance improvement is worth the cost. Besides, similar design has been adopted in MASS Java, which includes a `exchangeBoundary` function to exchange data at the boundary of a grid. As a result, it should not introduce significant burden, especially for users who have experience with MASS Java.

4.3.2 Communication for Agents

Unlike *Places*, *Agents* are not organized in a grid structure and each agent instance only have access to the place instance it currently resides on, making the concept of Ghost Regions and Boundary Regions unnecessary. Cross-device communication for *Agents* occurs during the migration and spawning processes. During migration, agent instances moving to the place instances located on a neighboring GPU are transferred to that GPU. During spawning, an agent instance may request to create children on a place instance located on a neighboring GPU. Such a request, consisting of the destination place instance and the number of children to generate, are referred to as the **Agent Spawning Request**. These requests will be transferred to the target GPU, where the new agent instances are subsequently created.

Since the cross-device communication for *Agents* is inherently related to the migration and spawning processes without requirements for manual control, it is designed to be automatically triggered during each `manageAll` API invocation. Thus, no additional functions are introduced for customization.

4.3.3 Collaborated Communication

In the new architecture, the `RootDispatcher` is responsible for handling the cross-device data transfer, but it is not able to do that directly. The main difficulty is that it has no

knowledge about how the memory is managed on each GPU by each `DeviceDispatcher`, and those data to be transferred may not natually be contiguous in memory. On the other hand, the `DeviceDispatcher` has direct access to the data on the GPU assigned to it through the `DeviceDriver`. So, the `RootDispatcher` requires some help from the `DeviceDispatcher` to collect the data to be transferred and transform it into some contiguous format. This is achieved by introducing the concept of **Communication Buffers**, which are a set of buffers temporarily holding the data to be transferred and the data received from the neighboring GPUs. Each `DeviceDispatcher` will prepare one set of Communication Buffers for each of its neighbor.

Figure 7 presents an example of the cross-device communication process with Communication Buffers, where some data on GPU 0 needs to be transferred to GPU 1. In this example, `DeviceDispatcher` 0 is assigned to GPU 0 and Communication Buffers 0 is the one prepared for GPU 1. Similarly, `DeviceDispatcher` 1 handles GPU 1 and Communication Buffers 1 is prepared for GPU 0. First, the `RootDispatcher` requests `DeviceDispatcher` 0 to collect data on GPU 0 to be transferred, apply necessary format transformation and store them into Communication Buffers 0. Then, the `RootDispatcher` access Communication Buffers 0 and transfer the data to Communication Buffers 1. Finally, the `RootDispatcher` requests `DeviceDispatcher` 1 to access Communication Buffers 1 for the received data and apply necessary format transformation before storing them to target locations on GPU 1.

4.3.4 Technology for Communication

To transfer data between GPUs, there are several technologies available. The simpliest one is the `cudaMemcpyPeer` function provided in CUDA toolchain, which allows transferring contiguous buffer of data between two GPUs on the same host with peer-to-peer connection (e.g.: PCIe or NVLink). However, it does not support GPUs on different hosts.

Another option is to use the Remote Direct Memory Access (RDMA) technology, which enable direct memory access between two GPUs on different hosts without going through the CPU, which can significantly reduce the communication overhead [9]. However, the API

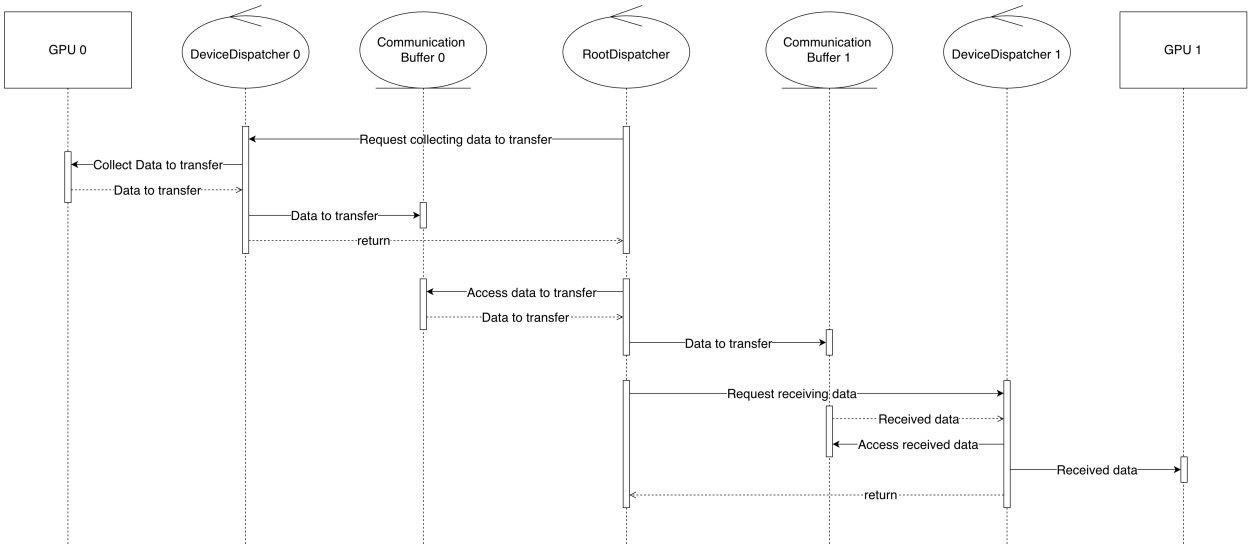

Figure 7: Collaborative Communication Process

of RDMA is very low level, making the implementaition extremely complex. In addition, this technology requires special hardware support and may not be available in all GPUs. For GPUs without RDMA support, the only option is falling back to normal TCP communication on host side.

For better programmability and compatibility, the CUDA-aware MPI and NVIDIA Collective Communication Library (NCCL) are two available options, where both of them provide high level APIs for data transfer between GPUs on either the same host or different hosts. They can both automatically detect the hardware capabilities and choose the best communication method between each pair of GPUs [10] [11]. For this new design, NCCL is chosen for the implementation since MPI is relatively too heavy with lots of features that are not required for the cross-device communication in MASS CDUA.

4.4 Cross-Host Parallelism

In order to support GPUs over different hosts, an execution and parallelism model for multi-hosts needs to be included in the new design. In MASS Java and C++, the Master-Slave

pattern is used for that, where a master node is responsible for runing the main function while the other slave nodes run a loop waiting for instructions from the master node. When the main function on the master node invokes any MASS API, it will send a request and necessary data to all the slave nodes, which will process the request on their local data segments concurrently. The master node will always wait until all the slave nodes finish the task before contiuing to the next step [1]. This design is strateforward but introduces high communication overhead since all the nodes have to synchronize with the master node every time a MASS API is invoked.

In the new design, a hybrid pattern that combines the Master-Slave pattern and the Peer-to-Peer (P2P) pattern is adopted. During initialization of the library, the master node will be launched first and be responsible for launching processes on all the other slave nodes and coordinate the initialization process. Once the initialzation is finished, the concept of Master node is no longer needed and all the nodes will execut the same main function concurrently and independently. As a result, when a MASS API is invoked, all the nodes will call this API locally without being coordinated by the master node and communication will only happen between neighboring nodes when necessary. This design can significantly reduce the overhead of communication since communication and synchronization only occur between neighboring nodes instead of between all the nodes and the master node.

Besides the improvement in communication efficiency, such P2P design does introduce concern related to branch divergence of different nodes, where one node executes the “if” branch while another one executes the “else” branch containing different MASS API invocations. However, such situation only happens when the branching condition in the main function depends on node-specific data, such as hostname of the node or the current system time. This is usually not a valid programming pattern for ABM simulations since the behavior of a simulation should not vary when executed on different hosts. As a result, such situation is treated as a programming error.

4.5 Refined Memory Layout

After the SoA design being implemented, the performance of MASS CUDA has been improved significantly for attribute access. However, the current memory layout still fails to adopt SoA correctly and the memory bandwidth is still not perfectly utilized. To address this issue, a refined memory layout design is proposed and implemented in this work to further improve the performance of MASS CUDA.

4.5.1 Refined Memory Layout for 2D Attributes

In MASS CUDA, 2D attributes are attributes whose values are arrays. In current design, the values of a 2D attribute for all the place instances or agent instances are flattened and stored as a 1D array in GPU memory. Figure 8 illustrates the memory layout for a 2D attribute of length 4 for a *Places*, where each group of 4 contiguous cells represents the array for a single place instance with padding inserted between the arrays for different place instances. When a function is invoked on each place instance through the `callAll` API and access the first element of the attribute, the threads will access the memory with a stride equals to the attribute length plus the padding size, resulting in uncoalesced memory access and low bandwidth utilization.

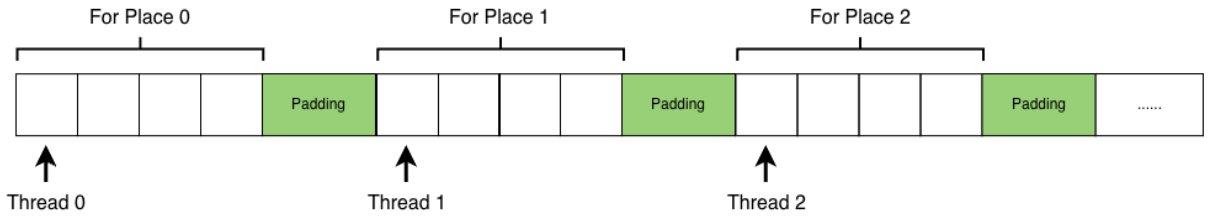

Figure 8: 2D attribute on Device (Old Design)

To fix this issue, the new design transposes the memory layout for 2D attributes. Instead of grouping all array elements of the same place instance or agent instance together, the new

design groups all the elements sharing the same internal index in the arrays across all the place instances or agent instances together. Figure 9 presents the new memory layout for the same 2D attribute with length of 4 as in Figure 8. Here, each group of contiguous cells are the elements at the same index for all the place instances, with padding inserted between each group. In this case, when a function access the first element of the attribute across all place instances, the threads will access strictly contiguous memory, resulting in a perfectly coalesced memory access.

Figure 9: 2D attribute on Device (New Design)

4.5.2 Shared Attribute

In ABM simulations implemented with MASS CUDA, one common pattern is to store some runtime constant values on GPU memory shared by all place instances or agent instances. Currently, users can achieve that by storing them as normal attributes or using compile time constants. The former method support runtime configuration of the values but have to repeatedly store the same value across all the place instances or agent instances, which can significantly increase the GPU memory usage. The latter method does not require extra GPU memory, but the values have to be determined at compile time and cannot be configured at runtime.

To fill this gap, a new type of attributes named **Shared Attributes** is introduced in the new design. A Shared Attribute stores only one instance of the attribute value on GPU

memory and all the place instances or agent instances share this single instance. Access to a Shared Attribute is designed to be read-only since modifying the same instance from multiple threads can cause race condition. This design can significantly reduce the GPU memory usage while supporting runtime configuration of the values. In addition, since there is only a single instance, the access to a Shared Attribute can be very efficient. The GPU only needs to load the value from global memory once, cache it, and broadcast it to all the threads accessing it, which reduce the bandwidth usage and improve the access performance.

The APIs added for creating and accessing Shared Attributes are very similar to the existing ones for normal attributes. Listing 1 presents a simplified example. The `MyPlace` is an example user-defined Place class, which defines a method named `accessSharedAttr` that will access a Shared Attribute named `SHARED_ATTR1`. This is achieved in line 5 by calling the `getSharedAttribute` function with the tag and type of the requested Shared Attribute, which returns a pointer to the attribute value. To create a Shared Attribute, users can call the `setSharedAttribute` function as presented in line 9 with the tag, type and attribute value.

Listing 1: Example of creating and accessing a Shared Attribute

```
1 class MyPlace final : public mass::Place {
2 public:
3     enum SharedAttribute { SHARED_ATTR1 };
4     __device__ void accessSharedAttr() {
5         const auto ptr = getSharedAttribute<int>(SHARED_ATTR1);
6     }
7 };
8
9 places->setSharedAttribute<int>(MyPlace::SHARED_ATTR1, 10);
```

4.5.3 Refined Memory Layout for Place / Agent class

To support SoA in the class-style API for defining behavior of *Places* or *Agents*, the `Place` and `Agent` classes still need to store certain metadata in AoS style, such as pointers to the attribute arrays and the indices of each instance. In current single GPU design, each `Place` and `Agent` class includes 5 members: the index of the instance (`index`), the number

of attributes (`nAttributes`), pointer to the attribute tag array (`attributeTags`), pointer to the base address array of the attribute arrays (`attributeDevPtrs`) and pointer to the attribute pitch array (`attributePitch`). These members, together with the `super` pointer for supporting inheritance, are stored in AoS style as presented in Figure 10, consuming 48 bytes per instance.

When an attribute is accessed, all these members of `Place` or `Agent` classes are needed to resolve the pointer to the attribute value. And every time a member is accessed, the threads access the memory with 48 bytes stride, resulting in only 8/48 of the fetched memory being effectively utilized. This poor bandwidth utilization can significantly degrade the performance. In addition, most of these members share exactly the same values across all instances, which wastes a significant amount of GPU memory. The situation becomes even worse in the initial multi-GPU design, which introduces 3 additional members, increasing the size to 72 bytes per instance and reducing bandwidth utilization to 8/72.

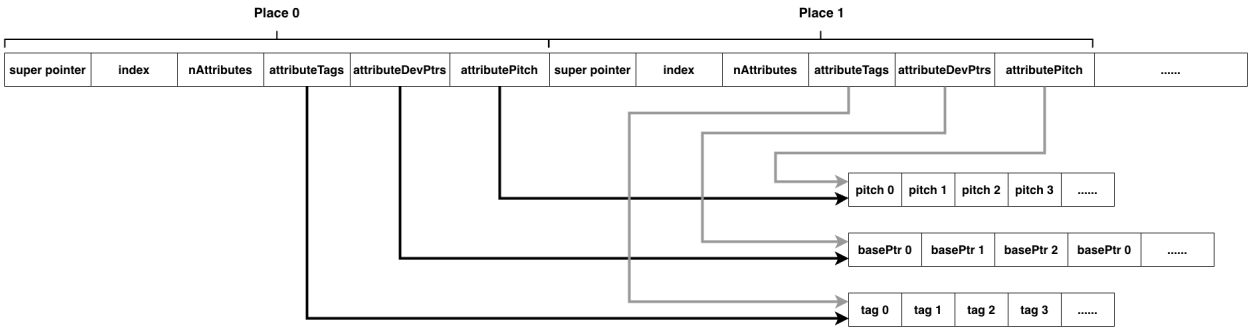

Figure 10: Array of places / agents on Device (Old Design)

To address this issue, the redundant members are extracted into a separated metadata structure as shown in Figure 11. In the new design, each place instance and agent instance only stores a super pointer, its index (`localIndex`) and a pointer to the shared metadata structure (`attrMetaArray`), which reduces the size to only 24 bytes per instance. This shared metadata structure maintains pointers to the attribute tag array, base address ar-

ray and the attribute pitch array for both the normal attributes and the newly introduced Shared Attributes. Notably, the base address array and the attribute pitch array for normal attributes are combined into a single attribute metadata array, where each element is a pair of the base address and the pitch for one attribute. And for the Shared Attributes, the concept of pitch is not applicable, so they only have a base address array.

Although the new design still technically use an AoS layout, the stride between threads is reduced to 24 bytes, increasing the bandwidth utilization to 8/24 when accessing each member. Beside the performance improvement, the GPU memory usage can also be significantly reduced with the shared metadata structure.

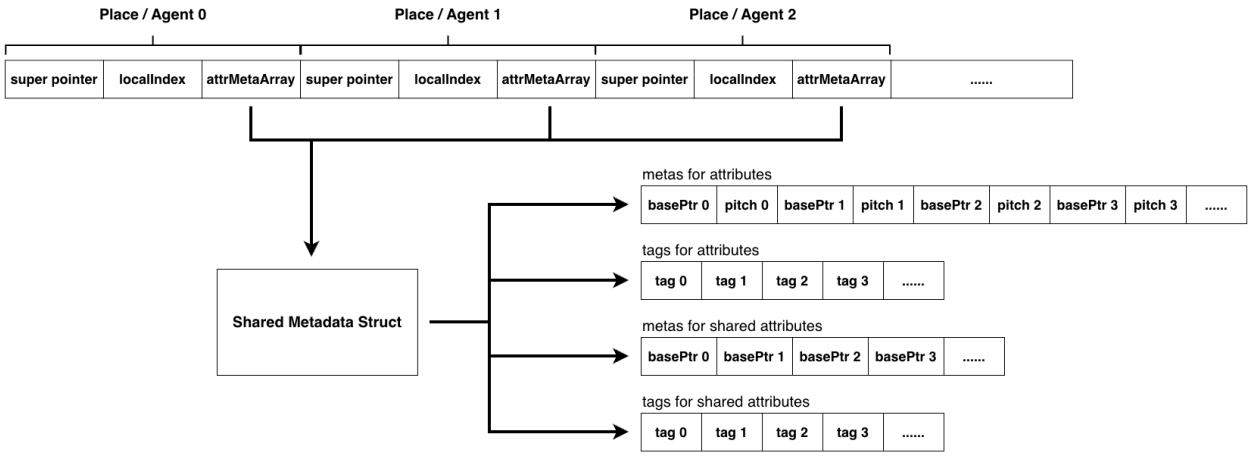

Figure 11: Array of places / agents on Device (New Design)

Chapter 5

IMPLEMENTATION DETAILS

5.1 Architecture Details

The `RootDispatcher` and the `DeviceDispatcher` are the two new key components introduced in the new architecture design to separate the communication logic and the execution logic of single GPU. However, they are not directly implemented as normal C++ classes. Instead, they are both separated into an abstract class as interface and a concrete class as actual implementation. The purpose of this implementation strategy is to leave the flexibility for any potential extensions in the future. If a new `RootDispatcher` or `DeviceDispatcher` is designed in the future that needs to co-exist with the current implementation to handle different use cases, it can simply inherit from the corresponding abstract class with different implementation without having to merge the new implementation into the current one using complex conditioning logic. In addition, this strategy can also help reduce header pollution by hiding some implementation related headers used by the private members of the concrete classes.

All the components in the new architecture maintain lots of resources that need to be properly released at the end of the simulation, such as GPU pointers, streams, events and communication session handles. To ensure proper resource management, the C++ Resource Acquisition Is Initialization (RAII) pattern is adopted. With this pattern, all the components are implemented as move-only types with destructors that automatically release all the acquired resources at the end of their lifetime. This strategy can reduce the risk of resource leaks and increase the maintainability of the code.

As discribed in the new architecture design, the `RootDispatcher` is responsible for dispatching MASS API requests to all the `DeviceDispatcher` instances it manages. Although

this can be done using a simple for loop, some APIs may involve heave CPU side workload in each `DeviceDispatcher`, which may limit the parallelism in multi-GPUs settings if executed sequentially. To maximize the parallelism level, OpenMP is used to enable concurrent execution of the `DeviceDispatcher` instances by assigning one CPU thread to each of them. Listing 2 presents an example of such concurrent dispatch of API requests. Here, the `#pragma omp parallel` in line 5 creates a multithreaded region with the number of threads equals to the count of `DeviceDispatcher` instances. Each thread will execute the code block below concurrently, whose thread id is obtained in line 6 using the `omp_get_thread_num()` function. The thread id is then used to retrieve the assigned `DeviceDispatcher` instance in line 7 and execute the requested API call in line 8 and 9.

Listing 2: Example of concurrent execution of multiple DeviceDispatcher with OpenMP

```
1  void RootDispatcher::callAllPlaces(
2      int32_t handle, int32_t functionId, const void* arg, int64_t argSize
3  ) {
4      #pragma omp parallel num_threads(localDeviceCount())
5      {
6          const auto threadId = omp_get_thread_num();
7          auto& deviceDispatcher = deviceDispatcherForThread(threadId);
8          deviceDispatcher.use();
9          deviceDispatcher.callAllPlaces(handle, functionId, arg, argSize);
10     }
11 }
```

5.2 Implementation Supporting the Grid Design

Unlike the linear topology, the grid topology design introduces more complexity in implementation and requires more concepts and data types to support it. This section will discuss these implementation details for the grid topology design.

5.2.1 Block Size and Index

In multi-GPUs settings, each block of *Places* assigned to each GPU has its own size and index. The block size represents the amount of place instances of this block while the block

index identify the position of this block in the complete *Places* grid. In grid topology, these two values for an N-dimensional *Places* are stored in an N-dimensional format instead of a single integer as in the linear topology. To be more specific, both the block size and index are stored as an integer array of length N, where the value for each dimension is stored at the corresponding position in the array. For example, a 100 row by 200 column block from row 2 column 3 of a 2D *Places* will have its block size stored as array [100, 200] and its block index stored as array [2, 3]. In the implementaion, these arrays are wrapped into two custom classes instead of using raw arrays for better readability.

Ideally, the size of all the blocks from the same *Places* should be identical. However, this may not be possible when the size of the *Places* is not divisible by the number of GPUs at certain dimension. In such cases, some blocks at the boundaries will have smaller size than expected. To distinguish these two sizes, the expected size for all the blocks is referred to as the **nominal size** while the real size of each block is referred to as the **actual size**. Both of these sizes are required in the implementation. For example, the nominal size is needed when converting the local index of a place instance within a block to the global index in the complete *Places* grid, while the actual size is required when calculating the total amount of place instances within a block.

5.2.2 Neighbour Identity of a Block

In grid topology, each block of a *Places* can have more than 2 neighbors. To identify each of them, an array of size N is used for N-dimensional *Places*, where each element of the array is an enum that can be **Front**, **Middle** or **Back** for each dimension. If a neighboring block has smaller index than the current block at a certain dimension, then the value for that dimension is **Front**. Similarly, **Middle** is used for neighbors with the same index at a dimension and **Back** is for neighbors with larger index. Figure 12 illustrates an example with a 2D block, which has up to 8 neighbors. Here, neighbors at the first row have **Front** at the first index because they have smaller index at the first dimension. Similarly, neighbors at the last column have **Back** at the second index since they have larger index at the second

dimension.

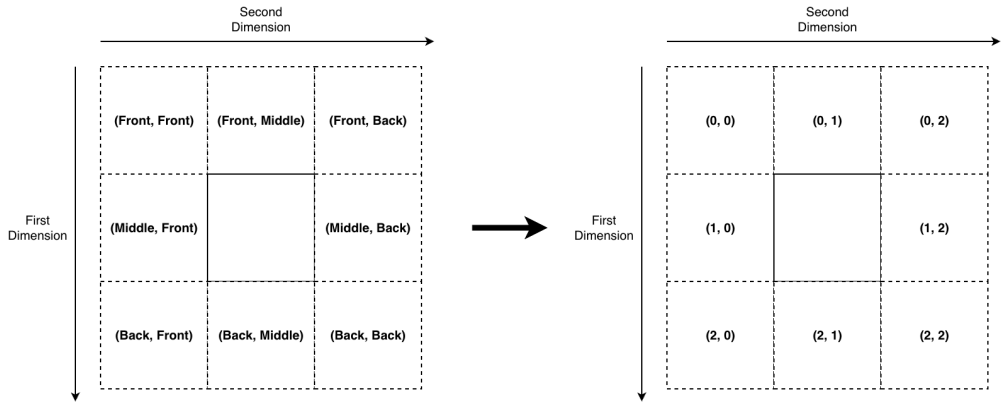

Figure 12: NeighbourId for each neighbour of a Block

Since the value at each dimension has only 3 options, this array for identifying neighbors is essentially a number in based-3 encoding. Figure 12 presents this idea by mapping **Front** to 0, **Middle** to 1 and **Back** to 2. Numbers in based-3 encoding can be easily converted to and from decimal numbers, which can then be stored as an integer. Thus, the actual implementation of the neighbor identity is a class named `NeighbourId`, which contains two integers, one for the identity in decimal (i) and the other for the dimension (N). To retrieve the based-3 value at a specific dimension d, the formula is $((i/3^{N-d-1}) \bmod 3)$.

5.3 Collaborative Communication Implementation

5.3.1 Communication Buffers, Pins and Neighbor Connectors

In the new design, the Communication Buffers is a set of contiguous buffers acting as the contract between the `RootDispatcher` and the `DeviceDispatcher` to support the collaborative cross-device communication process by temporarily holding the data to be transferred and the data received. In the implementation, these buffers are always allocated in pairs, where one is used for sending data to the neighbor (`outBuffer`) and the other one is used for receiving data from the neighbor (`inBuffer`). Each of such pair of buffers, together

with their sizes, are managed in a type named `Pin`. A set of `Pin` instances prepared for one neighbor are grouped to form a **Neighbor Connector**. The usage of the `Pin` instances for *Agents* and *Places* are different and therefore separated classes of Neighbor Connector is implemented for each of them. The `DeviceDispatcher` manages one Neighbor Connector for each neighbor of a block in the form of a `unordered_map` from `NeighbourId` to Neighbor Connector, which is exposed to the `RootDispatcher` for actual data transfer.

For *Places*, the actual class of Neighbor Connector is `PlaceNeighborConnector`, where each instance is associated with one pair of Ghost Region and Boundary Region of a neighbor. Each `Pin` instance inside a `PlaceNeighborConnector` is responsible for one attribute of the *Places*. Figure 13 illustrates a `PlaceNeighborConnector` for a neighbor to the right of a block from a 2D *Places* with two attributes, where the two grids on the left are the attribute arrays and the structure on the right is the `PlaceNeighborConnector`. During the communication process, attribute values of the place instances within the Boundary Regions (blue regions in the grids) will be copied to the `outBuffer` of the `Pin` instances in corresponding `PlaceNeighborConnector` for sending while the received data in the `inBuffer` of the `Pin` instances in each `PlaceNeighborConnector` will be copied to the attribute arrays of the place instances within the corresponding Ghost Regions (green regions in the grids).

In grid topology, the place instances within the Boundary Regions and the Ghost Regions may not be contiguous in memory, which is the same for their attribute arrays. This is illustrated in Figure 14 where the non-contiguous attribute values within the Boundary Region for the neighbor to the top of the block (the colored cells) needs to be copied to the contiguous `outBuffer`. The numbers on the colored cells in the grid is the index of each attribute value within the attribute array, which is referred to as the absolute index. On the other hand, the numbers on the cells in the `outBuffer` is the index within the buffer, which can also be treated as the local index within the Boundary Region and is referred to as the region index. These two indices are needed by the dedicated kernel function responsible for the copy operation to locate the copying source and destination.

For each cell, the region index can be trivially mapped by the thread index, but the corre-

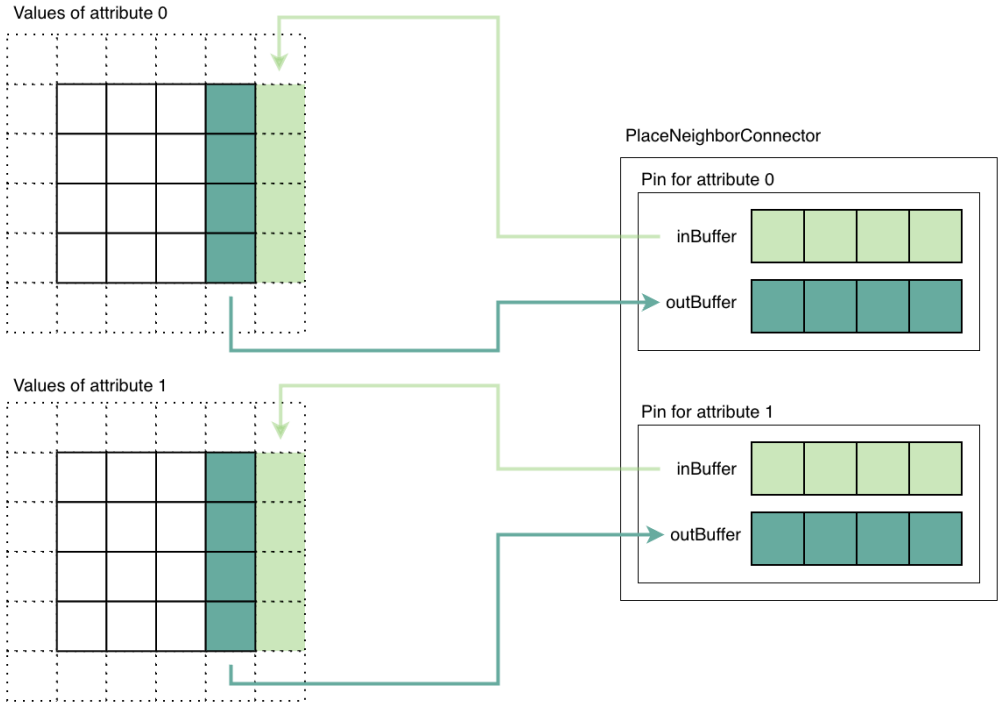

Figure 13: The Neighbor Connector for Places

sponding absolute index reuqires some transformation to be resolved. This is done using a device function presented in Listing 3. It takes in the region index of a cell (`regionIndex`), the size of the attribute array (`blockSize`), the size of the Boundary Region (`regionSize`), the index of the first cell in the Boundary Region within the attribute array (`regionBaseIndex`) and the dimension of the block (`dimension`). The function iterates from the last to the first dimension and maintains the strides of both the Boundary Region and the attribute array for each dimension, which are updated in line 13 and 14. In each iteration, the coordinate of the cell within the Boundary Region at current dimension (`regionCoord`) is calculated from the region index using the region stride and size in line 9, which is then used to retrieve the coordinate within the attribute array at currrent dimension (`blockCoord`) by adding the base index in line 10. Then the absolute index is updated by aggregating the `blockCoord` using the current stride of the attribute array in line 11. After the loop, the maintained

stride of the Boundary Region equals to the total count of cells inside and if the input region index exceeds that, it is out of bound and the function should return -1. Otherwise, the calculated absolute index is returned.

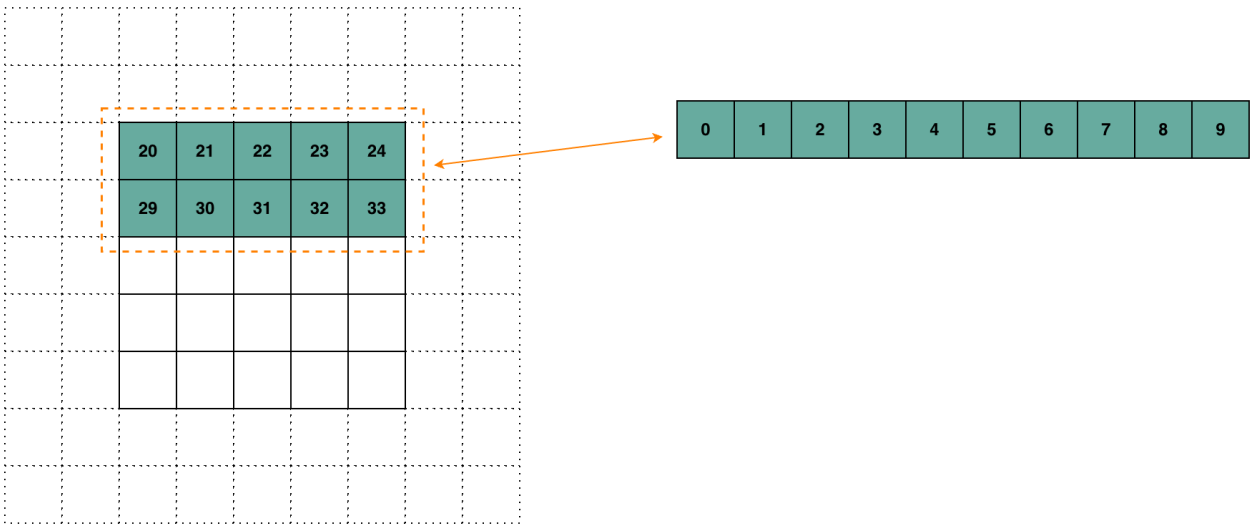

Figure 14: Copy non-contiguous attribute values to Communication Buffer

Listing 3: Map region indices within a Boundary Region to absolute indices in the grid

```
__device__ int64_t mapRegionIndexToAbsoluteIndex(
    int64_t regionIndex,
    int64_t* blockSize, int64_t* regionSize, int64_t* regionBaseIndex,
    int32_t dimension
) {
    int64_t absoluteIndex = 0;
    int64_t regionStride = 1;
    int64_t blockStride = 1;
    for (int32_t dim = dimension - 1; dim >= 0; dim--) {
        int64_t regionCoord = regionIndex / regionStride % regionSize[dim];
        int64_t blockCoord = regionCoord + regionBaseIndex[dim];
        absoluteIndex += blockCoord * blockStride;
        regionStride *= regionSize[dim];
        blockStride *= blockSize[dim];
    }
    return regionIndex < regionStride ? absoluteIndex : -1;
}
```

The actual class of Neighbor Connector for *Agents* is `AgentNeighborConnector`, which includes separated `Pin` instances for migration and spawning. For migration, each `Pin` is responsible for one attribute of the agent instances that are migrating to / from the neighboring device. Similar to the difficulty of *Places*, the agent instances migrating to another deivce and their attribute values may not be contiguous in memory. Therefore, before copying the attribute values, the indices of the agent instances migrating to the neighboring device are first collected in a separated buffer, which is then used to locate the attribute values to be copied within each attribuite arrays. As for handling the received data, the first step is to collect the indices of the agent instances that are not "Alive" to reside the incoming agent instances. Then when copying the received attribute values to the attribute arrays, these indices are used to locate the copy destination.

For agent spawning, there is only one `Pin` in each `AgentNeighborConnector`, which stores the Agent Spawning Requests sending to / received from the neighboring device. In this case, there is no attribute to be copied and the Agent Spawning Requests from the agent instances trying to spawn children to a place instance on a neighboring device are directly collected into the `outBuffer` of the `Pin`. For handling incoming data, the first step is still collecting the indices of the un-alive agent instances in a separated buffer for spawning the new instances. Then, each of the incoming Agent Spawning Request is assigned with a unique subarray of indices from that indices buffer base on the number of children to spawn specified in the request. Finally, the agent instances indicated by the indices in the subarray assigned to each Agent Spawning Request are set to "Alive" and their residing place instance are set to the target one specified in the request.

5.3.2 Cross-Device Data Transfer

In the new design, the `RootDispatcher` is responsible for performing the actual data transfer between devices using the Neighbour Connectors exposed by each `DeviceDispatcher`. The API used for the general data transfer directly between devices is the `ncclSend` and `ncclRecv` functions provided by the NCCL library, which are asynchronous and perform

as submitting a sending / receiving task for concurrent execution. The pseudocode for the cross-device data transfer process is presented in Listing 4, where each `RootDispatcher` traverses through each Neighbor Connector (line 3) within all the `DeviceDispatcher` instances it manages (line 1). For each Neighbor Connector, the rank of the peer device is queried using the rank of the current device and the `NeighborId` associated with this Neighbor Connector (line 4). Then one receiving task and one sending task are registered for the `inBuffer` and `outBuffer` of each `Pin` instance in that Neighbor Connector using NCCL APIs (line 5 to 7).

Listing 4: Pseudo code for the cross-device data transfer

```
1 for deviceDispatcher in localDeviceDispatcherList:
2     rankOfCurrDevice <- deviceDispatcher.rank()
3     for (neighbourId, connector) in deviceDispatcher.neighbourConnectors:
4         rankOfPeerDevice <- queryRankOfPeer(rankOfCurrDevice, neighbourId)
5         for pin in connector.pins:
6             registerNCCLReceiveTask(pin.inBuffer, rankOfPeerDevice)
7             registerNCCLSendTask(pin.outBuffer, rankOfPeerDevice)
```

The `ncclRecv` functions used for receiving data from neighboring devices require the caller to specify the amount of data expected to receive. This is not a problem for *Places* since the amount of data being received can be calculated from the size of the Ghost Region, which has a fixed size during the simulation. However, for the migration and spawning process of *Agents*, the count of incoming migrating agent instances and the count of incoming Agent Spawning Requests may vary and cannot be determined at the receiver side before the transfer. To address this issue, an additional step is added before the actual data transfer where the amount of data to be transferred is shared between the neighboring devices by the `RootDispatcher`. If the two devices are on the same host, this can be done directly in memory. Otherwise, the amount of data to transfer is sent to the other host where the neighboring device resides using TCP connection.

5.4 Process of Library Initialization and Cross-Host Parallelism

A new hybrid cross-host parallelism pattern is introduced in the new design that adopts Master-Slave pattern for library initialization and switches to P2P pattern for the actual

simulation process afterward. In the library initialization process, all the participating hosts are prepared by instantiating all the necessary components and communication channels. These are done within the `Mass::init` function, which takes in the command line arguments for basic configuration, including the rank of the current host and path to the host configuration file. The host configuration file is an XML format file specifying the array of hosts with their hostname, port, username and ssh key. The index of each host in the array must be the same as its rank and the first host in the array is the Master Node.

Before launching the simulation application, the compiled binary and necessary resources should be copied to all the paticipating hosts by the user. Then, the simulation process is first launched on the first host in the configuration file by the user, which should be the Master Node with rank 0. The Master Node first reads the configuration file and use ssh to launch the simulation process on all the other hosts with the same binary while passing the corresponding rank of each host retrieved from the configuration file. The Slave Nodes, once launched, also read the configuration file and establish a TCP connection with the Master Node. Then, they immediately send the number of local GPUs they have to the Master Node. With the number of GPUs on all the hosts, the Master Node assigns a device rank to each GPU and sends that back to the corresponding Slave Nodes. After that, the Master Node generates a unique NCCL ID and broadcast it to all the Slave Nodes. Once all the nodes have the id locally, they register all their local GPUs to the NCCL communication group by passing the NCCL ID and the assigned device ranks to the `ncclCommInitRank` function and get the communication handles for each GPU. Finally, all the nodes creates their own `RootDispatcher`, `DeviceDispatcher` and `DeviceDriver` instances. During this process, although all the hosts execute the same binary, their execution logic diverges based on their ranks so that the Master Node and the Slave Nodes perform different operations.

After the initialization process, the processes on all the host exit the `Mass::init` function and the parallelism pattern is switched to P2P. Starting from this point, the execution path is irrelevant to the rank and all the hosts start executing the exact same codes. As a result, each host is able to independently execute the simulation process and communicate with

other hosts when necessary without relying on the coordination from a Master Node.

5.5 Detail of Refined Memory Layout

In the new design, the memory layout for 2D attributes is re-designed to group the values at the same internal index across all place instances or agent instances together. For an *Places* or *Agents* with N instances on a GPU, the memory layout of a 2D attribute with M elements is essentially an $M \times N$ 2D array. During the simulation, when this attribute is accessed in a `callAll` invocation, it is highly possible that all the GPU threads are accessing values at the same internal index across all instances, which are essentially a single row in this 2D array. Such access pattern benefits from the `cudaMallocPitch` function due to the inserted padding between each row that ensure the starting address of each row aligned with the GPU memory line. In the implementation, if the attribute to be created is a 2D attribute, the `cudaMallocPitch` function is used to allocate the GPU memory by setting the width to the number of instances times the size of each attribute element in bytes and the height to the number of elements in the attribute. A pitch value is then provided by this function to indicate the actual stride in bytes between the starting address of each row.

The `cudaMallocPitch` function is also used in the previous implementation for 2D attributes allocation. However, the layout in the previous implementation is essentially an $N \times M$ 2D array where the length of each row M (the number of elements in the attribute) is usually small. One behavior of the `cudaMallocPitch` function is that the pitch it allocates for each row is no less than 512 bytes, which is usually much larger than M. As a result, the padding inserted between rows takes up a larger portion of the allocated memory than the actual data, wasting a significant amount of memory. This is not an issue in the new implementation since the length of each row N (the number of instances) is usually much larger than 512 bytes and the inserted padding for each row takes only a small portion.

Although the new memory layout for 2D attributes can significantly improve the performance, it also introduces a difficulty when accessing the values. In the previous design, since the values for each place instances or agent instances are stored contiguously, the

`getAttribute` function will simply return a raw pointer to the first element for each instance. However, this is not possible in the new design since values at adjacent internal indicies have a stride equals to the pitch value provided by the `cudaMallocPitch` function.

To hide this complexity and keep the APIs similar to the previous design, a new C++ wrapper pointer type presented in Listing 5 is implemented. This wrapper type, named `ArrayAttrPtr`, overloads the dereference operator, the arrow operator and the subscript operator to simulate the behavior of a raw C++ pointer. It stores the address of the element at index 0 for an instance in the member `ptr` (line 3) and the pitch value in the member `pitch` representing the stride between values at adjacent internal indices (line 4). For the dereference and the arrow operator, they both aims at accessing the value at index 0 and therefore the member `ptr` is directly cast and returned (line 5 and 6). As for the subscript operator, it takes in the internal index of the value to access, which will be multiplied by the `pitch` to get the real byte offset in memory. This offset is then added to `ptr` to get the actual address of the value at that internal index, which is then cast and returned (line 7 to 9).

Listing 5: Definition of ArrAttrPtr

```
1  template <typename E>
2  class ArrayAttrPtr final {
3      void* ptr;
4      int64_t pitch;
5  public:
6      E* operator->() const { return static_cast<E*>(ptr); }
7      E& operator*() const { return *static_cast<E*>(ptr); }
8      E& operator[](int64_t i) const {
9          return *reinterpret_cast<E*>(static_cast<char*>(ptr) + i * pitch);
10     }
11 }
```

In the new design, the `Place` and `Agent` classes made more compact by extracting common members into a shared metadata structure. In the implementation, the metadata for normal attributes and Shared Attributes are stored in separate classes. For normal attributes, the attribute count, pointer to tag array and pointer to attribute metadata array

are encapsulated within the `EntityAttrDeviceMetaArray` class. Similarly, for Shared Attributes, the attribute count, pointer to tag array and pointer to base address array are grouped into the `EntitySharedAttrDeviceMetaArray` class. These two classes are then combined into a single `EntityDeviceMeta` class, which is the actual type for the shared metadata structure. Since the memory layout for both `Place` and `Agent` is identical, the `EntityDeviceHandleStorage` class is implemented to hold the instance index and the pointer to the `EntityDeviceMeta`.

This implementation strategy with multiple nested classes enables fine-grained control over loading required metadata into GPU registers. For example, accessing a normal attribute requires the instance index and the `EntityAttrDeviceMetaArray`. In this case, the device function can first load the entire `EntityDeviceHandleStorage` instance into registers and then fetch the `EntityAttrDeviceMetaArray` instance through the retrieved pointer to the `EntityDeviceMeta`. This process requires only two memory access operations to load all necessary metadata into registers while no unneeded data is accessed.

Chapter 6

EVALUATION

6.1 Evaluation Design and Environment

The evaluation of the new version of MASS CUDA is executed on the UW Bothell's Juno server, which is equipped with an AMD EPYC 7232P 8-Core Processor, 2 NVIDIA A5000 GPUs (24GB device memory each) connected by NVLink, and 128GB host memory. Comparison is made among the new version, the previous single GPU version and the FLAME GPU 2. Although there are 2 GPUs available on the server, only the new MASS CUDA is able to utilize both of them while all the other participating libraries use one of the GPUs. In addition, due to current limitation in the implementation, the new MASS CUDA does not support being configured to use only one GPU. Therefore, benchmark results of running the new MASS CUDA on a single GPU are not obtained.

Although such configurations may seem unfair due to different amount of GPUs being used, multi-GPUs support is one of the most important improvement of the new version of MASS CUDA that makes it stand out from the other libraries. So, it is reasonable to include this "advantage" in the comparison. In addition, even when different amount of GPUs are used, there are still lots of valuable results and insights drawn from the comparison, which will be discussed later for each benchmark. In the future, when the capability of configuring the GPUs to be used is implemented for MASS CUDA, more detailed comparison can be conducted to include the single-GPU performance of the new version of MASS CUDA.

The benchmark applications chosen for the evaluation includes Heat2D, GameOfLife and SugarScape. Heat2D is a physics simulation modeling the heat diffusion process in a 2D plate. In each step, the temperature of each cell is updated based on the temperature of its surrounding cells using Forward Euler method [12]. GameOfLife is a cellular automation

simulation where the state of each cell (live or dead) is updated based on the number of live and dead cells around it [13]. SugarScape is a simulation that models the behavior of an artificial society. In this model, agents autonomously navigate a 2D grid environment to search for, consume, and accumulate a distributed resource known as sugar [14].

The evaluation focus on the performance and scalability of the new MASS CUDA when compared to the previous version and FLAME GPU 2. The performance is measured by 3 different metrics: (1) The total execution time of the program; (2) The initialization time for setting up the library and preparing the data for the simulation; (3) The per-step execution time, which is the average time taken for executing a single iteration of the simulation loop. The scalability is evaluated by running the benchmarks with gradually increasing problem sizes to find the maximum size that each library can handle before crashing due to out-or-memory error.

Note that due to the much higher scalability of the new design and FLAME GPU 2, all the figures illustrating the benchmark results are presented in pair where the one on the left inlcudes all the participating libraries while the one on the right includes only the new version of MASS CUDA and the FLAME GPU 2 for comparison at larger problem sizes.

6.2 Fixes and Optimization Made for Benchmark Applications

The benchmark applications are already implemented with the previous version of MASS CUDA and FLAME GPU 2, which are then modified to use the new MASS CUDA. During this process, some fixes are made for correct and consistent behavior while some optimizations are adopted for better performance without affecting the behavior.

For Heat2D, the original implementation using previous MASS CUDA does not behave correctly due to incorrect simulation steps. In each iteration, its steps are 1) update the temperature of the cells at the boundary; 2) apply heat at the heater cells; 3) update the temperature of interior cells. However, the correct order should be $3 \rightarrow 1 \rightarrow 2$, which is the order used in the FLAME GPU 2 implementation. In addition, cells at the 4 corners of the plate should be updated based on the temperature of their diagonal neighbors close to the

center of the plate instead of the vertical ones. Together with these fixes, it is also possible to combine all these steps into one single `callAll` invocation, which help reduce the overhead of kernel launches. For the implementation using FLAME GPU 2, the execution order of these steps is already correct, and no fix is needed, but its formula for calculating the new temperature for the interior cells is slightly updated to be consistent with the MASS CUDA implementations and avoid the problem of floating point accuracy. Besides, one of the agent function named `start` in the implementation is redundant and can be combined with the `update` agent function for better performance.

For GameOfLife, a phase design was previously introduced in order to solve a race condition issue. In MASS CUDA, if all the place instances try to read an attribute value from their neighbors and update their own value at the same time, some of them may get the old values while the others get the updated values, which is a race condition. To solve this issue, this type of attributes that needs to be updated based on the values from the neighbors are created as a 2D attribute with length of 2, where one value represent the current value and the other one is the updated value. In addition, another attribute named `phase`, which can be either 0 or 1, is created to indicate which value in the 2D attribute is the current value and which one is the updated value. At the end of each step, the `phase` attribute is updated to `1-phase` to switch the role of the two values in the 2D attribute [15]. Although this design solves the race condition, it intorduces overheads by requiring one addtional `callAll` invocation to update the `phase` attribute. Besides, the value of `phase` is always the same across all the place instances and storing it as an attribute with independent value for each place instance is redundant. To improve the performance of this fix, the `phase` value is removed from the attributes and provided as an argument of the `callAll` function. In each iteration, the phase value is calculated by the CPU and passed to all `callAll` invocations that need this value. With this optimization, the additional `callAll` invocation and the storage in memory for the `phase` attribute can be removed to improve the performance and memory efficiency.

For SugarScape, the original implementation includes a redundant `exchangeAll` invo-

cation in each iteration that will update the array of neighbors for each place instances to `[top, bottom, left, right]`. This neighbor setting is never accessed during the simulation and is always immediately overwritten by the next `exchangeAll` invocation using another array of neighbors. This redundant invocation is removed for better performance. Furthermore, the original implementation shuffles the array of neighbors on CPU and upload that to all place instances in each iteration to randomize the agent migration. This can be inefficient since the process of updating the array of neighbors for each place instance is relatively expensive. A better approach is fixing the array of neighbors and shuffle an additional array of indices. This array of indices has the same length as the array of neighbors where each integer inside is a unique index within the array of neighbors. In each iteration, this array of indices is shuffled on CPU and passed as an argument to the required `callAll` invocations, which will access the array of neighbors based on the order indicated by the shuffled array of indices. With this optimization, the original `exchangeAll` invocation can be replaced by a normal `callAll` with arguments, which is more efficient.

To present the effect of these optimizations, the performance of both the unoptimized version and the optimized version using the previous MASS CUDA are included in the evaluation results. Note that the Heat2D benchmark does not have an unoptimized version since the original implementation does not behave correctly and comparing the perforamnce with an incorrect implementation is not meaningful.

6.3 Heat2D Results

The benchmark results for Heat2D are presented in Figures 15, 16 and 17 while the data points are shown in Tables 1, 2 and 3. The previous MASS CUDA is the fastest at size 100×100, which can be caused by its simplicity of the host-side implementation. However, as the problem size increases, its performance quickly degrades and becomes the slowest amoung all the implementations for sizes larger than 500×500. When the problem size reaches 3000×3000, the new MASS CUDA is around 6.7 times faster than the previous version in terms of per-step execution performance. Here although the new version uses 2

GPUs, the performance improvement should not go beyond 2x, especially when the communication overhead is considered. As a result, this 6.7x improvement presents better single-GPU execution performance, which show the effectiveness of the refined memory layout.

As for initialization, the new version is slightly slower than the previous version due to the NCCL initialization and additional cross-device communication when creating the *Places* instance. However, the difference is not significant and also not significantly affected by the problem size. As a result, even with slower initialization, the new MASS CUDA is still faster than the previous version in terms of total execution time, which is over 3.4 times faster at size 3000×3000.

When compared to the FLAME GPU 2, the new MASS CUDA is around 99% faster at size 9000×9000 to execute a single simulation step with 2 GPUs. As mentioned before, even with 2 GPUs, the performance improvement should notably be less than 2x due to communication overhead. Although the 1.99x improvement here is indeed less than 2x, the difference is not significant, and it is reasonable to claim that the single-GPU performance of MASS CUDA should at least match that of FLAME GPU 2. Together with 228% better initialization performance, the new MASS CUDA presents 118% improvement in the total execution time compared to FLAME GPU 2 at this problem size. In FLAME GPU 2, the process of agent creation and initialization is fully done on CPU, making it sensitive to the problem size. As a result, the initialization time of FLAME GPU 2 increases quickly as the problem size grows.

When it comes to scalability, the new MASS CUDA can handle problem sizes up to 14150×14150 with 2 GPUs. This is over 17 times larger than that of the previous version, which can only handle up to 3428×3428. Considering the fact that the new version has doubled device memory, the theoretical maximum problem size it can handle on a single GPU should be approximately $14150 \times 14150/2 = 100111250$, which is still around 8.5 times larger. Such improvement in scalability can be attributed to the more efficient design of the `Place` and `Agent` classes and the correct usage of `cudaMallocPitch` function for 2D attributes. However, when compared to FLAME GPU 2, which can handle problems with size up to

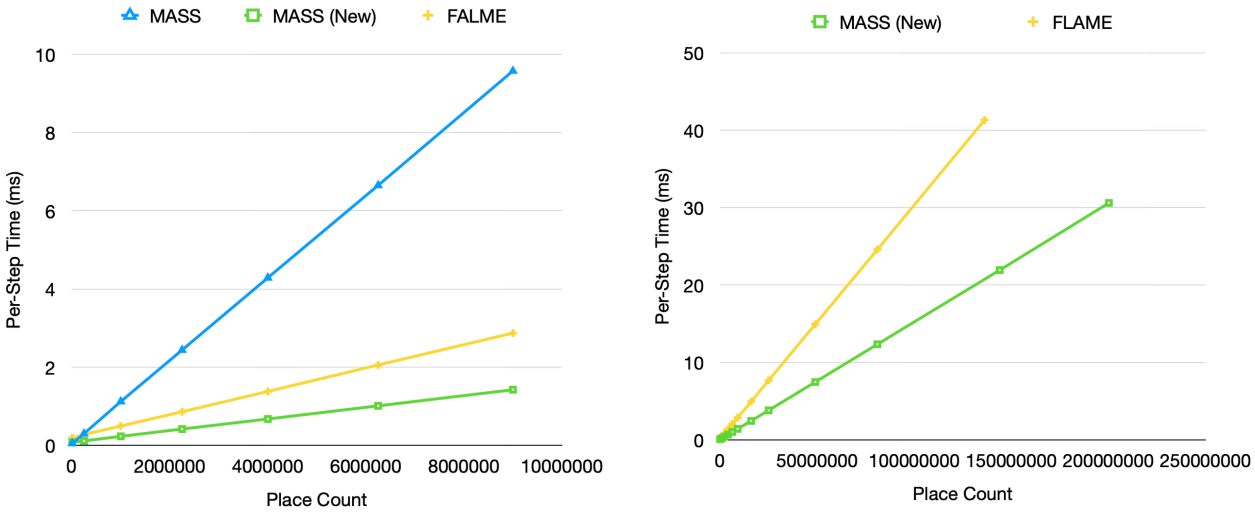

Figure 15: Per-Step Runtime of Heat2D

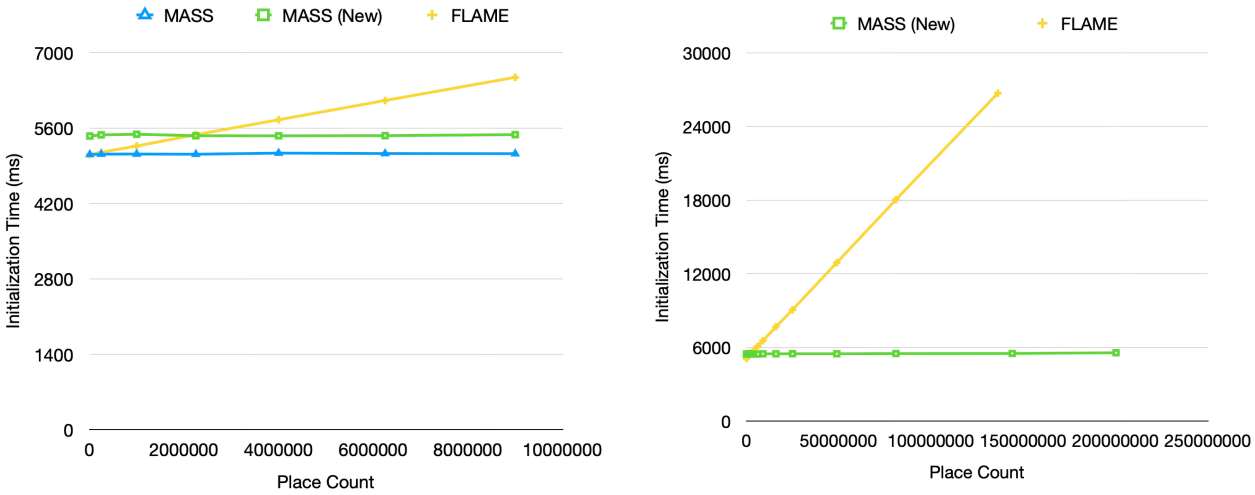

Figure 16: Initialization Time of Heat2D

11671 × 11671, the theoretical scalability of the new MASS CUDA on single GPU is only 73.5% of that of FLAME GPU 2. This is likely caused by the large number of unused pre-defined attributes in the new MASS CUDA. For example, the `AGENTS`, `POTENTIAL_AGENTS` and `AGENT_POPS` attributes on *Places* are designed for supporting *Agents*, which are not useful for the Heat2D benchmark since it does not require *Agents*. Therefore, these attributes

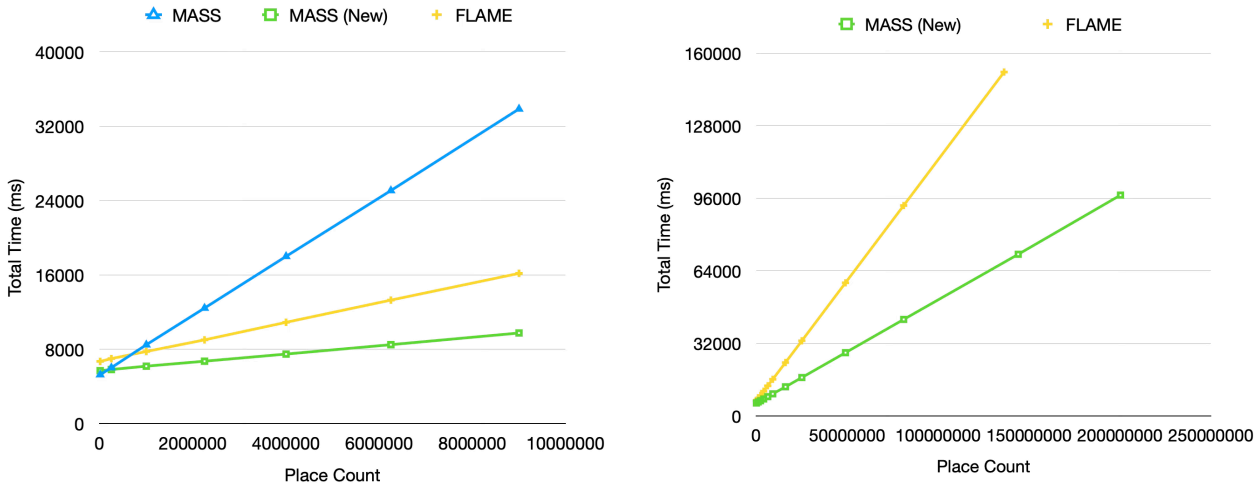

Figure 17: Total Runtime of Heat2D

are never accessed but still consume a significant amount of device memory, limiting the scalability of the new MASS CUDA.

6.4 GameOfLife Results

The benchmark results for GameOfLife are presented in Figures 18, 19 and 20 while the data points are shown in Tables 4, 5, 6 and 7. Here, the previous MASS CUDA without application level optimization is the fastest for small sizes while the new MASS CUDA and FLAME GPU 2 are significantly faster for larger sizes. The application level optimization improves the per-step performance by over 11% for sizes larger than 1500×1500, but still lags behind the implementation using the new MASS CUDA. The pattern of performance difference between the new MASS CUDA and the previous version with optimization is similar to that in Heat2D, where the new version is approximately 6.2 times faster at size 3000×3000 for single step execution, 6% slower for initialization and 2 times faster for total execution time. The less significant improvement in total execution time here compared to that in Heat2D can be caused by fewer iterations in the GameOfLife benchmark, which is set to 1000 while the value for Heat2D is 3000. With fewer iterations, the time spent on initialization takes a larger portion of the total execution time and limits the performance improvement brought

by the better per-step performance. Again, the more-than-2x performance improvement for executing a single simulation step with 2 GPUs shows better single-GPU performance and demonstrates the effectiveness of the refined memory layout.

Compared to FLAME GPU 2, the per-step performance of the new MASS CUDA with 2 GPUs is around 99% faster at size 3000×3000 and 109% faster at size 9000×9000. The improvement for large simulation sizes here is even higher than that in Heat2D, making it also reasonable to claim that the performance of the new MASS CUDA on single-GPU outperforms FLAME GPU 2. As for the total execution time, the new MASS CUDA is only about 42.7% faster at size 3000×3000, but it increases to around 2.2 times faster at size 9000×9000. This can again be caused by the fewer iterations of the GameOfLife benchmark that increases the portion of the total time spent on initialization. As the problem size increases, the time spent on actual simulation process becomes more dominant and the performance difference in total execution time becomes larger.

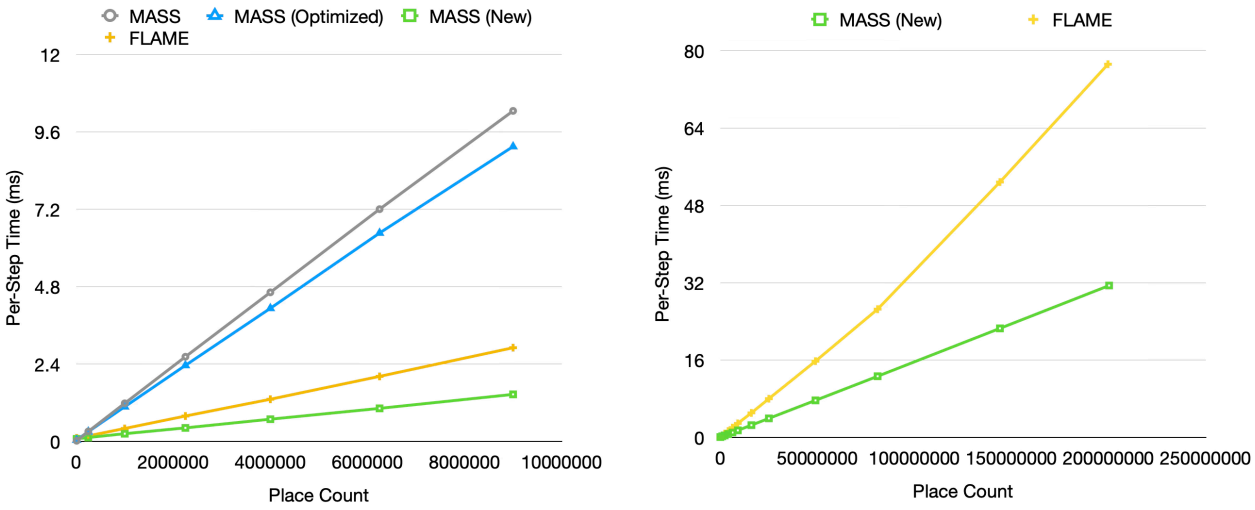

Figure 18: Per-Step Runtime of GameOfLife

For scalability, the new MASS CUDA with 2 GPUs improves the maximum problem size from 3428×3428 to 14150×14150 compared to the previous version with application level optimization, which is approximately a 17 times improvement. However, even with doubled

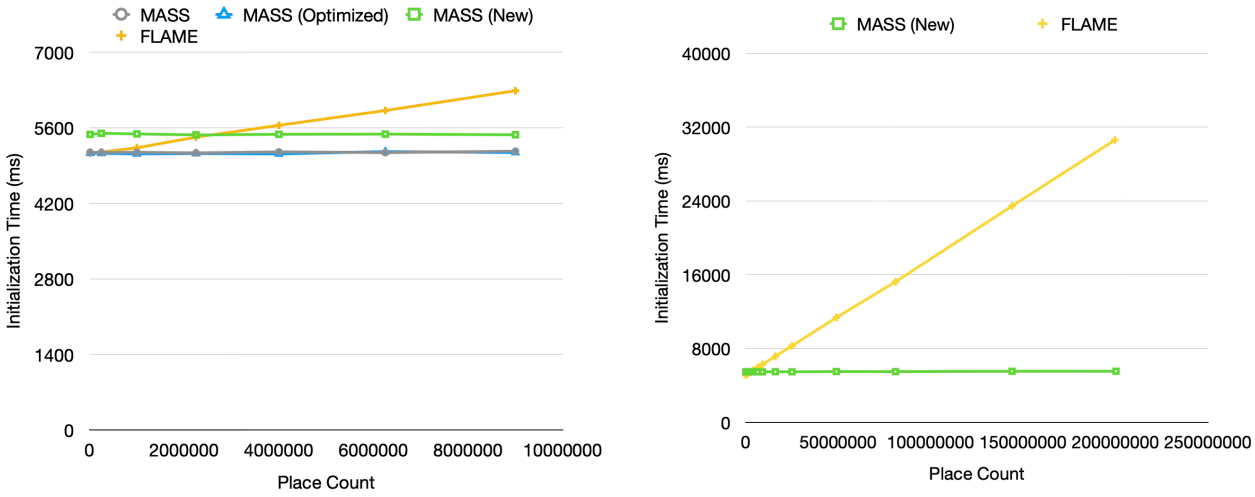

Figure 19: Initialization Time of GameOfLife

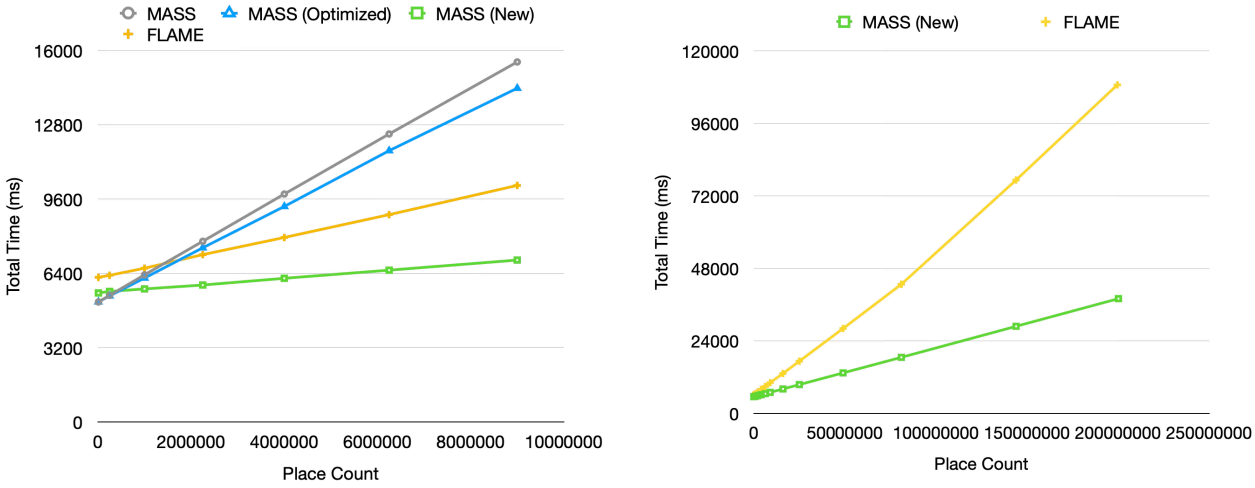

Figure 20: Total Runtime of GameOfLife

device memory, this is less than 1% larger than the scalability of FLAME GPU 2, which can handle problems with size up to 14129×14129 on a single GPU. This result is far less than the ideal 2x scaling with doubled device memory available, showing that the scalability of the new MASS CUDA on single GPU is not comparable to that of FLAME GPU 2. Besides the issue of having numerous unused pre-defined attributes, another possible reason that enlarge the gap is the fact that FLAME GPU 2 does not use additional attributes to store the

coordinates of each agent in GameOfLife. Such coordinates are needed in Heat2D to access the 4 orthogonal neighbors, but in GameOfLife, a build-in function is used to directly access the 8 surrounding neighbors. This further reduces the memory consumption of the FLAME GPU 2 implementation and widens the gap in scalability compared to MASS CUDA.

6.5 SugarScape Results

The benchmark results for SugarScape are presented in Figures 21, 22 and 23 while the data points are shown in Tables 8, 9, 10 and 11. This is the benchmark application where the most significant performance improvement is observed. Although the new MASS CUDA is still slower than the previous version with or without application level optimization at size 100×100, it quickly outperforms all the other implementations as the problem size increases. When the size reaches 3000×3000, the per-step execution time of the new MASS CUDA with 2 GPUs is around 14.7 times faster than the previous version with application level optimization and almost 9 times faster than FLAME GPU 2. For larger problem size 7000×7000, the improvement compared to FLAME GPU 2 goes further to about 14.2 times faster. All the per-step performance improvement here are significantly higher than 2x, so even with the fact that the new MASS CUDA uses 2 GPUs, it is enough to demenstrate better performance on single-GPU compared to the previous version and FLAME GPU 2.

The reason for such significant improvement can be related to the migration conflict resolution process required by SugarScape, which is used for forbidding multiple agents from migrating to the same location. In both the new and the previous versions of MASS CUDA, this operation is build-in with heavy access to 2D attributes, which is significantly improved by the refined memory layout for 2D attributes introduced in the new version. As for FLAME GPU 2, due to its agent-only design, it has to simulate the behavior of *Places* provided in MASS CUDA using agents and manually implement complex logic for resolving migration conflicts, which is relatively inefficient.

The advantage of faster initialization of MASS CUDA can also be observed in the results of SugarScape, where both the new version and the previous version are significantly faster

than FLAME GPU 2 while not being too sensitive to the problem size. Here, the initialization time of FLAME GPU 2 grows quadratically as the problem size increases, reaching 3.7 minutes at size 3000×3000 and over 105 minutes at size 7000×7000. As a result, even the previous MASS CUDA with application level optimization is able to surpass FLAME GPU 2 in total execution time.

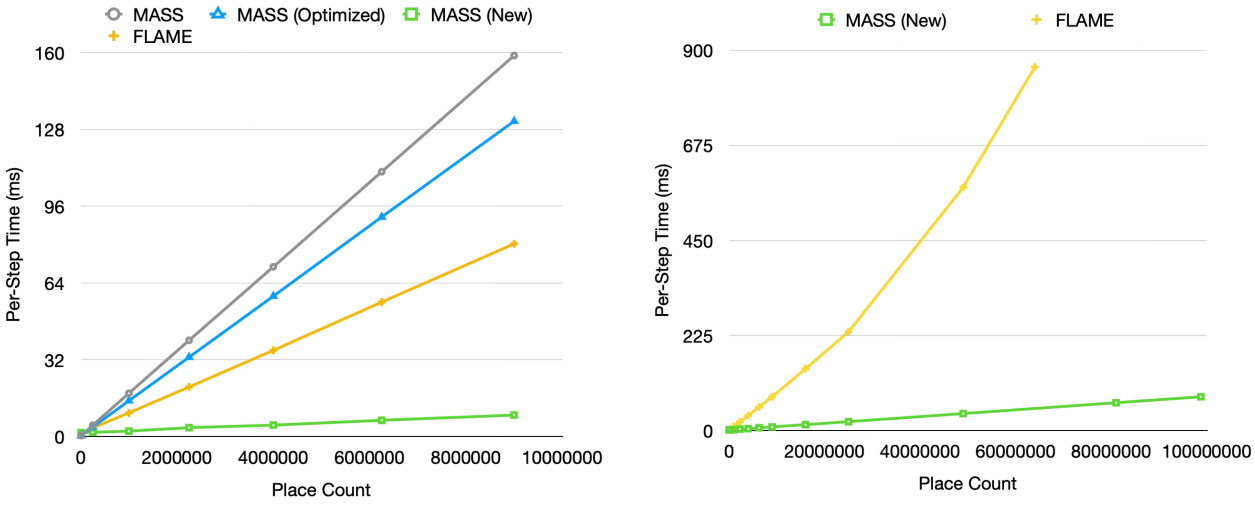

Figure 21: Per-Step Runtime of SugarScape

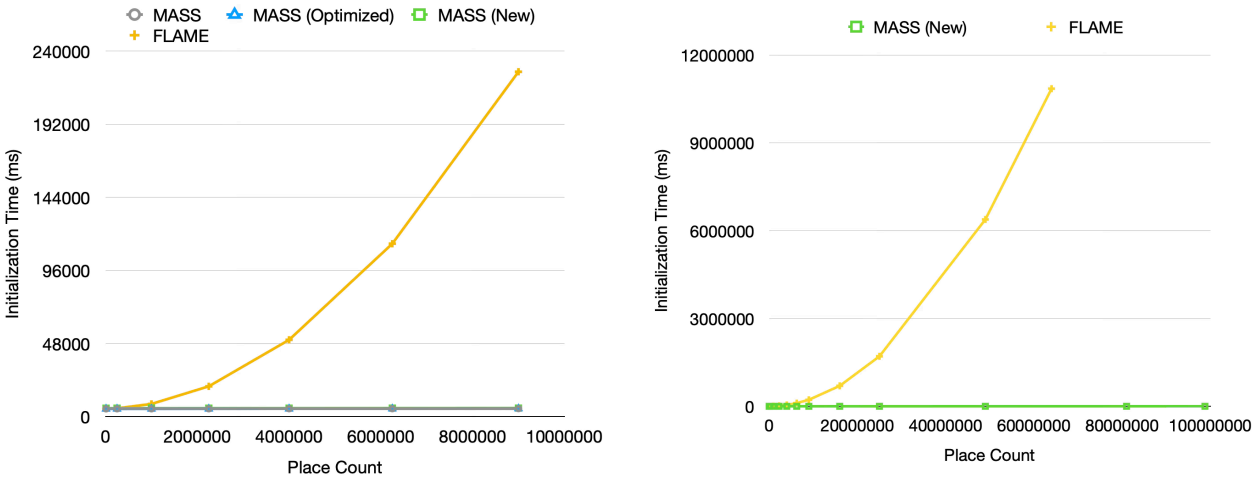

Figure 22: Initialization Time of SugarScape

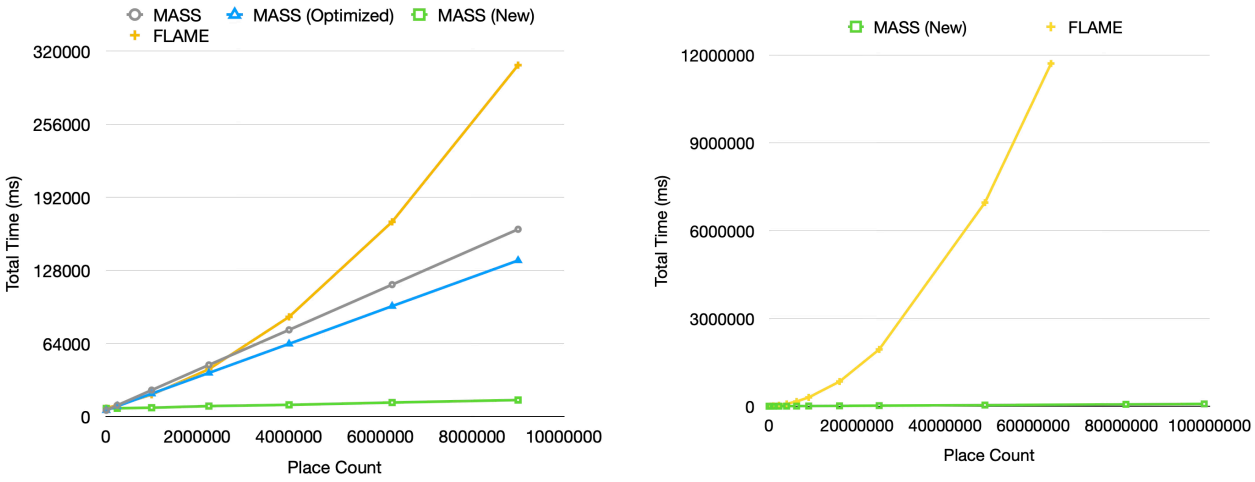

Figure 23: Total Runtime of SugarScape

As for the scalability, the previous version of MASS CUDA can handle problem sizes up to 3363×3363 while the new version with 2 GPUs can handle up to 9938×9938, presenting approximately an 8.7 times improvement. For FLAME GPU 2, the accurate maximum problem size is not obtained due to the extremely long execution time. However, the value is confirmed to be between 8000×8000 and 9000×9000, meaning that the improvement of the new MASS CDUA with 2 GPUs should be between 22% and 54%. Again, since the improvement in max problem size is significantly less than 2x, the scalability of the new MASS CUDA on single GPU is expected to be worse than that of FLAME GPU 2. However, this time all the pre-defined attributes are useful since *Agents* is being used in SugarScape. As a result, the problem here is more likely to be related to the inefficient design of those pre-defined attributes. For example, the `AGENTS` attribute has length of 1 for this benchmark since each place instance can only have at most 1 agent instance residing on it, but the length of the `POTENTIAL_AGENTS` attribute is still set to 12 to support the migration conflict resolution process, which consumes a large amount of memory and limits the scalability.

6.6 Summary of the Evaluation Results

Overall, the new version of MASS CUDA shows significant improvement in both performance and scalability over the previous version. When compared to FLAME GPU 2, the performance of the new MASS CUDA with 2 GPUs is also better for both initialization and single-step execution. It is true that the comparison may seem unfair due to different number of GPUs being used. However, the fact that the new MASS CUDA with 2 GPUs can achieve approximately 2x or even higher single-step speedup in all benchmarks despite the existance of inter-GPU communication overhead still indicates its high efficiency.

As for scalability, although the new MASS CUDA is now able to handle much larger simulation sizes, the gap with FLAME GPU 2 remains. In all the benchmarks, the new MASS CUDA with doubled device memory is never able to reach 1.6x maximum problem sizes than FLAME GPU 2, which is far less than the ideal 2x improvement. The GameOfLife benchmark shows the worst result, where the maximum problem size that the new MASS CUDA can handle is very close to that of the FLAME GPU 2 even with doubled device memory available. Possible reason for this includes the large number of unused pre-defined attributes for *Places*-only simulations and the inefficient design of those pre-defined attributes for simulations that use *Agents*.

Currently, the evaluation design is still relatively simple and naive. It does not include strict simulation execution result of the new MASS CUDA on a single GPU and does not support evaluating the contribution of each optimization to the overall performance improvement. For the former, it requires the library to support configuring the GPUs to be used for a simulation, which is not currently supported. For the latter, since most of the optimization are tightly coupled with the architecture and implementation of many components, making it very difficult to selectively disable each of them. Consequently, this will require implementing separate versions where each of them has one optimization enabled, which is a large amount of work and is excluded from this thesis. However, such kind of benchmark results help understand the effectiveness of different optimization strategies and can be used

to find other performance bottlenecks and inspire further optimization. So, this can still be valuable future direction.

Chapter 7

CONCLUSION AND FUTURE DIRECTION

This thesis presents a new version of MASS CUDA that support offloading simulation tasks to multiple GPUs distributed across multiple hosts. The architecture of the library is redesigned using components with clear separation of duties to improve the maintainability even after addting all the complex implementation for supporting the multi-GPU features. For the multi-GPUs support, several technologies are adopted. First, the newly introduced Collaborative Communication design with Communication Buffers and the NCCL works together to handle communication between GPUs. The hybrid cross-hosts parallelism pattern that combines Master-Slaves and P2P further extends the supports to GPUs on multiple hosts. Finally, the Grid Topology is adopted for spliting the *Places* into multiple blocks in replacement of the Linear Topology used in the initial multi-GPU implementation, which help reduce the communication overhead between devices. In addition to these enhancements for multi-GPUs support, several optimizations are made to the memory layout, including tranposed storage for 2D attributes, Shared Attributes for faster access to shared constants and more compact design for the `Place` and `Agent` classes. These optimizations have significantly improved the performance and scalability of MASS CUDA.

Although the new version of MASS CUDA has shown promissing improvements in performance, scalability and complete multi-GPU support, there are still four limitations that should be addressed in future work. First, the library is only evaluated on a single host with 2 GPUs connected over NVLink due to the current limitation of the available hardware resources in the lab. In other words, although the features of running simulation on multiple hosts is implemented, the correctness and performance are not yet evaluated in such environment. This should be of the highest priority for future work once the neces-

sary hardware resources are available. Second, as presented in the evaluation results, the scalability of the new MASS CUDA is still not ideal when compared to FLAME GPU 2. For simulations without *Agents*, the large number of unused per-defined attributes used to support *Agents* is likely to be the main cause of this issue. And for simulations that requires *Agents*, the inefficient design of those pre-defined attributes may be the main limitation. Therefore, a better design for the pre-defined attributes should be considered in future work to further reduce unnecessary memory consumption and improve the scalability of MASS CUDA. Third, most of the CUDA APIs used in the implementation are the synchronous versions, which simplify the implementation but may introduce synchronization overhead between the CPU and the GPU. Future work can consider adopting async versions of the CUDA APIs to overlap the execution of CPU and GPU for better performance. Finally, the evaluation in this thesis is relatively simple without benchmark results of single-GPU execution and the contribution of each optimization. Future work can consider adding more detailed benchmarks and implementing necessary support in the library for better insights to the performance improvement and potential bottlenecks.

BIBLIOGRAPHY

[1] T. Chuang and M. Fukuda, "A Parallel Multi-agent Spatial Simulation Environment for Cluster Systems," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, Dec. 2013, pp. 143–150. DOI: 10.1109/CSE.2013.32 Accessed: Apr. 17, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6755210

[2] L. Kosiachenko, N. Hart, and M. Fukuda, "MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models," en, 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., Cham: Springer International Publishing, 2019, pp. 139–152, ISBN: 978-3-030-24209-1. DOI: 10.1007/978-3-030-24209-1_12

[3] W. Liu, "Programmability and performance enhancement of MASS CUDA," *University of Washington, Bothell, Capstone Report, Jun*, 2024. Accessed: Apr. 14, 2026. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/WarrenLiu_whitepaper.pdf

[4] H. Ogden, "Multi-GPU Parallelized Agent-Based Modeling," en, Mar. 2025. Accessed: Jul. 13, 2025. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/HoltOgden_wi25.pdf

[5] P. Richmond, R. Chisholm, P. Heywood, M. Leach, and M. Kabiri Chimeh, "FLAME GPU," Nov. 2024. DOI: 10.5281/ZENODO.5428984 Accessed: Jul. 12, 2025. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.5428984

[6] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, B. Herrmann, and L. Philippe, "MC-MAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation," en, in *Euro-Par 2013: Parallel Processing Workshops*, D. an Mey et al., Eds., Berlin, Heidelberg: Springer, 2014, pp. 544–554, ISBN: 978-3-642-54420-0. DOI: 10.1007/978-3-642-54420-0_53

[7] K. Leyba, S. Hofmeyr, S. Forrest, J. Cannon, and M. Moses, "SIMCoV-GPU: Accelerating an Agent-Based Model for Exascale," in *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '24, New York, NY, USA: Association for Computing Machinery, Aug. 2024, pp. 322–333, ISBN: 979-8-4007-0413-0. DOI: 10.1145/3625549.3658692 Accessed: Nov. 14, 2025. [Online]. Available: https://doi.org/10.1145/3625549.3658692

[8] X. Jiang, R. Sengupta, J. Demmel, and S. Williams, "Large scale multi-GPU based parallel traffic simulation for accelerated traffic assignment and propagation," *Transportation Research Part C: Emerging Technologies*, vol. 169, p. 104 873, Dec. 2024, ISSN: 0968-090X. DOI: 10.1016/j.trc.2024.104873 Accessed: Nov. 14, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0968090X24003942

[9] *RDMA Core*. Accessed: Dec. 5, 2025. [Online]. Available: https://github.com/linux-rdma/rdma-core

[10] *Cuda-aware MPI*. Accessed: Dec. 5, 2025. [Online]. Available: https://docs.open-mpi.org/en/main/tuning-apps/networking/cuda.html

[11] *NVIDIA/nccl*, Mar. 2026. Accessed: Mar. 16, 2026. [Online]. Available: https://github.com/NVIDIA/nccl

[12] J. Nguyen, "Development of FLAMEGPU2 and MASS CUDA Benchmark Programs (Heat2D)," en, Tech. Rep., 2024. Accessed: May 1, 2026. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/JohnNguyen_wi24.docx

[13] M. Gardner, “Mathematical games : The fantastic combinations of John Conway’s new solitaire game ”life”,” *Sci. Am.*, vol. 223, no. 4, pp. 120–123, 1970. Accessed: May 19, 2026. [Online]. Available: https://cds.cern.ch/record/431586

[14] J. Kehoe, *The Specification of Sugarscape*, arXiv:1505.06012 [cs.MA], Nov. 2016. DOI: 10.48550/arXiv.1505.06012 Accessed: May 21, 2026. [Online]. Available: http://arxiv.org/abs/1505.06012

[15] Zad Castaneda, “Debugging and Improving MASS CUDA Benchmark Programs,” Term Report, 2025. Accessed: May 21, 2026. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/ZadCastaneda_au25.pdf

Appendix A

EVALUATION DATA

Table 1: Heat2D Evaluation Results - MASS CUDA

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5109.000	0.050	5260.667
250000	5115.333	0.299	6014.000
1000000	5116.667	1.121	8479.000
2250000	5113.000	2.438	12427.000
4000000	5134.333	4.286	17992.000
6250000	5126.333	6.650	25078.000
9000000	5123.333	9.578	33858.667
11751184	5136.333	12.499	42635.000

Table 2: Heat2D Evaluation Results - MASS CUDA (New)

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5450.667	0.079	5688.000
250000	5472.667	0.114	5814.667
1000000	5484.000	0.231	6176.333
2250000	5456.667	0.418	6709.333
4000000	5455.000	0.675	7479.667
6250000	5458.333	1.011	8490.000
9000000	5476.667	1.423	9745.000
16000000	5473.333	2.452	12829.000
25000000	5483.000	3.817	16935.000
49000000	5478.667	7.475	27904.333
81000000	5495.333	12.358	42568.000
144000000	5504.333	21.938	71318.333
200222500	5561.333	30.620	97423.000

Table 3: Heat2D Evaluation Results - FLAME GPU 2

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5106.223	0.189	6682.667
250000	5148.220	0.277	6987.333
1000000	5266.240	0.494	7754.667
2250000	5474.950	0.860	9005.667
4000000	5753.580	1.378	10895.333
6250000	6110.920	2.056	13286.000
9000000	6541.313	2.871	16164.000
16000000	7674.340	4.952	23540.667
25000000	9060.263	7.690	33140.333
49000000	12907.733	14.939	58736.333
81000000	18037.000	24.597	92841.000
136212241	26715.567	41.332	151725.667

Table 4: GameOfLife Evaluation Results - MASS CUDA

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5146.333	0.022	5168.333
250000	5140.667	0.310	5453.000
1000000	5145.000	1.182	6337.000
2250000	5133.667	2.626	7781.000
4000000	5153.333	4.623	9815.333
6250000	5139.667	7.201	12402.000
9000000	5165.667	10.248	15503.333
9480241	5158.000	10.818	16070.000

Table 5: GameOfLife Evaluation Results - MASS CUDA (Optimized)

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5128.333	0.045	5174.000
250000	5124.667	0.304	5430.667
1000000	5114.333	1.076	6200.000
2250000	5121.333	2.359	7501.000
4000000	5114.667	4.128	9281.333
6250000	5159.667	6.463	11682.667
9000000	5139.333	9.143	14371.667
11751184	5147.667	11.903	17168.333

Table 6: GameOfLife Evaluation Results - MASS CUDA (New)

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5475.333	0.081	5556.333
250000	5496.667	0.123	5621.000
1000000	5485.667	0.241	5730.667
2250000	5467.333	0.421	5898.333
4000000	5478.667	0.691	6187.667
6250000	5482.000	1.026	6538.000
9000000	5470.000	1.460	6972.667
16000000	5486.667	2.539	8103.667
25000000	5477.667	3.940	9543.667
49000000	5506.000	7.663	13415.333
81000000	5500.667	12.650	18566.667
144000000	5534.000	22.553	28843.000
200222500	5538.000	31.407	38001.667

Table 7: GameOfLife Evaluation Results - FLAME GPU 2

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5127.843	0.110	6229.113
250000	5151.017	0.174	6316.337
1000000	5228.013	0.402	6622.050
2250000	5425.933	0.789	7206.230
4000000	5645.027	1.309	7945.210
6250000	5918.913	2.018	8927.663
9000000	6286.580	2.907	10186.067
16000000	7164.707	5.070	13229.767
25000000	8296.327	8.010	17298.433
49000000	11364.667	15.764	28121.500
81000000	15244.533	26.517	42755.233
144000000	23460.567	52.806	77261.733
199628641	30617.367	77.165	108778.333

Table 8: SugarScape Evaluation Results - MASS CUDA

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5131.667	0.385	5526.000
250000	5135.667	4.712	9857.000
1000000	5134.667	18.024	23168.333
2250000	5141.333	40.071	45222.000
4000000	5189.333	70.723	75922.333
6250000	5212.667	110.315	115537.667
9000000	5260.333	158.739	164010.000
11309769	5302.333	198.677	203990.000

Table 9: SugarScape Evaluation Results - MASS CUDA (Optimized)

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5115.667	0.370	5495.333
250000	5113.000	3.913	9036.667
1000000	5128.000	14.921	20060.000
2250000	5158.667	33.054	38224.667
4000000	5181.667	58.497	63690.667
6250000	5220.000	91.479	96711.667
9000000	5298.000	131.406	136717.333
11309769	5310.000	165.301	170623.333

Table 10: SugarScape Evaluation Results - MASS CUDA (New)

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5476.333	1.640	7125.667
250000	5498.333	1.783	7290.333
1000000	5502.000	2.249	7761.000
2250000	5503.667	3.688	9202.333
4000000	5522.667	4.753	10285.333
6250000	5541.333	6.761	12311.667
9000000	5557.333	8.937	14504.333
16000000	5614.000	14.244	19868.000
25000000	5683.000	21.324	27017.333
49000000	5886.000	40.479	46376.333
81000000	6145.333	66.061	72217.333
98763844	6289.333	80.207	86507.333

Table 11: SugarScape Evaluation Results - FLAME GPU 2

Place Count	Initialization Time (ms)	Per-Step Time (ms)	Total Time (ms)
10000	5118.347	1.244	7354.223
250000	5417.353	3.606	10016.730
1000000	8290.543	9.855	19137.100
2250000	19874.667	20.657	41523.200
4000000	50393.133	35.918	87304.433
6250000	113527.667	56.014	170530.667
9000000	226428.667	80.294	307718.000
16000000	702885.333	146.329	850209.667
25000000	1711313.333	234.074	1946390.000
49000000	6381926.667	575.444	6958370.000
64000000	10847333.333	859.874	11707266.667