Multi-GPU Agent-Based Modeling
MASS CUDA Agent Implementation

Robert Foskin

A White Paper

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 Clark Olson

Professor Michael Stiber

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Multi-GPU Agent Based Modeling
MASS CUDA Agent Implementation

Robert Foskin

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

Agent-based models (ABM) attempt to simulate complex systems from the ground up using independent agents. ABMs are particularly suited to examining the emergent behavior of large-scale problems such as biological, economic or social systems. Success in modeling these systems depends on the ability to parallelize the execution of these models, reducing run times for very large problem sets.

The Multi-Agent Spatial Simulation (MASS) library has been under development at UW Bothell since 2011. MASS abstracts many of the lower-level implementation details required for parallelizing agent-based modeling problems, providing researchers with a more accessible framework for developing large-scale simulations. MASS CUDA extends this approach to GPU-based execution, allowing agent-based models to take advantage of the massive parallelism offered by modern graphics processors. The library has undergone several iterations and enhancements,

with recent development enabling simulation spaces to be extended across multiple GPUs connected using NVIDIA's NVLink.

This project further extends MASS CUDA by enabling agents to migrate across GPU boundaries, allowing ABM simulations to fully function across multiple GPUs. Experimental results showed increases in both simulation size and execution performance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to thank my family for their unwavering support these last few years. Without their help, I would not have been able to succeed in my studies.

I would also like to thank Professor Fukuda for his tireless efforts in supporting the MASS Lab, including this project, and for creating a platform for the growth and development of many students here at UW Bothell.

Finally, I would like to thank the other members of my advisory committee, Professor Olson and Professor Stiber, for their time and their valuable feedback on this capstone project.

Chapter 1. INTRODUCTION

Agent-based models, or ABMs, model large-scale simulations of complex systems. These models predict the emergent behavior of these systems and can predict the complex interactions that are inherent to these simulations. The Multi-Agent Spatial Simulation, or MASS, library has been in development at UW Bothell since 2011. The MASS library enables the development of custom ABM models that can be applied across wide ranging fields of study. The MASS library is available in Java, C++, and CUDA. This project focuses on high-performance ABM computing with MASS CUDA over two GPUs.

1.1 Agent Based Modeling

An automaton is a mathematical model of a simple machine; it is defined as having states, following a set of rules, taking input and producing output. Cellular Automata (CA) are a grid of many automata and are constructed in one to three dimensions, with each cell having multiple adjacent neighbors. CAs originated in the 1950s and were popularized by Conway's Game of Life in 1970. Conway's Game of Life defined a grid of cells, with each cell having a state, the cells could be alive or dead. Each cell's state would then change based on its neighbors' state, these CAs were a simplified version of ABMs. [1]

What distinguishes ABMs from CAs is that in CAs the grid applies rules mechanically whereas in ABMs, the agents evaluate their situation and act accordingly with the grid being an environment in which the Agents move. A true agent-based model was developed by the economist Thomas Schelling in 1969 that modelled segregation in housing. Schelling modelled housing segregation using two types of agents and used ABMs to demonstrate that even slight individual

preferences will generate segregated patterns [2]. This is an example of the emergent behavior that ABMs can predict, which is generated from inputs, and derived from the bottom-up.

In 1996, the epidemiologist Joshua M. Epstein published his book “Growing Artificial Societies” in which he detailed the “Sugarscape” ABM, this model is also studied as part of this project [3].

ABMs have uses in the field of economics, replacing simplified models and economic equations with bottom-up simulations of interactive agents. Currently ABMs modeling systemic risk are in use by policy institutions such as the European Central Bank and the Bank of England, simulating behavior of European financial institutions. ABMs can also be used to model housing and financial markets [4].

ABMs are an ideal type of problem to be computed using parallel libraries, such as MASS, due to the large number of agents and places that can be updated often and independently, allowing the simulation workload to be divided across many threads or processing units simultaneously.

1.2 MASS

The MASS library has been in development at UW Bothell since 2011. Initial motivation for development of the library was due to the opportunity provided by the growing amount of sensor networks and the possibility of using these nodes to analyze real-time sensor data. Available tools at the time for enabling parallelization of ABMs, such as OpenMP and MPI, were considered too low level. Thus, the MASS library was born, with the aim of reducing the semantic gap for implementation of a multi-agent simulation [5]. The library was initially developed in Java, and in 2013 the MASS library began to be ported to and developed in C++, and CUDA, for use with Nvidia GPUs.

Ultimately at the writing of this paper, the MASS library has three main branches, MASS Java, MASS C++, and MASS CUDA. The MASS lab has four focus areas, Agent-Based Data Sciences, Agent-Based Graph Computing and Database Systems, Agent-Based Modeling and General Cluster Computing, and finally, MASS CUDA, a GPU Platform for Agent-Based Models. The development of the MASS library in CUDA opened the library up to utilizing GPUs to accelerate ABMs.

1.3 Contribution of Work

When starting this project, the main goal was to extend the MASS CUDA library to facilitate agents that can instantiate and migrate across multiple GPUs. This was achieved, enabling agent-based simulations to now run over multiple GPUs. The spatial capacity for benchmarked simulations increased by 68%-86%, while a speedup of between 1.2-1.5 times was achieved for agent-based simulations across multiple-GPUs. An additional 30% increase in the simulation space was also reached by utilizing a direction-ordered collision-free agent migration algorithm. Further testing highlighted the potential redundancy of ghost places, ensuring simplified implementation and a slight performance increase of 1ms per simulation step.

This white paper describes the design and implementation of a multi-GPU ABM simulation library that facilitates agent migration over multiple GPUs. This report details the background of ABMs, the background and key concepts of MASS CUDA, the work carried out as part of this project, and the testing and evaluation of this work.

Chapter 2. BACKGROUND

2.1 ABM PARALLELIZATION

ABMs are highly parallel operations in theory, as many places and agents can be updated simultaneously; however early implementations of MASS CUDA demonstrated slower performance when compared with sequential execution. The main factors when designing a program for execution on GPUs are thread divergence and memory access patterns [6]. The most significant gains in performance for MASS CUDA have come from redesigning the library with more efficient memory access patterns in mind [7].

GPUs execute threads using Streaming Multiprocessors (SMs). Threads are grouped into blocks which are then executed in groups of 32 threads, called warps, on the SM. This follows the Single Instruction Multiple Threads (SIMT) execution model. Efficient warp execution depends on threads following the same execution path. Divergence between threads within a warp reduces parallel efficiency because different execution paths must be serialized [6].

GPU memory is organized hierarchically. On the Dynamic RAM (DRAM) exists global memory, constant memory and texture memory. Constant Memory and Texture Memory are cached on chip during execution, and both are read-only. There are also shared memory and registers. Registers provide the fastest access but are limited in size and private to individual threads. Shared memory is fast on-chip memory shared between threads in the same block and is commonly used to reduce global memory accesses. Global memory provides the largest storage capacity but has significantly higher access latency. Efficient GPU parallelization therefore depends heavily on minimizing global memory accesses and maximizing cache and shared memory utilization [6].

MASS CUDA utilizes the GPU by mapping ABM operations over places and agents onto large numbers of CUDA threads. Since many places and agents can be updated independently during each simulation step, MASS CUDA can launch parallel kernels for operations. This allows simulation workload to be distributed across the GPUs SMs where groups of threads execute in warps.

The MASS CUDA library has been developed utilizing CUDA's unified virtual address space and Nvidia's NVLink. The unified virtual address space contains all memory allocations on the host and the global memory of all connected devices [8]. Nvidia's NVLink is a physical connection between two Nvidia GPUs, it enables direct communication between devices bypassing a typical PCIe connection via the host. This allows peer-to-peer access across multiple devices, allowing one GPU to directly access and manipulate data on a second GPU and vice versa [9].

The devices used as part of this project were 2 Nvidia A5000 GPUs which are based on Nvidia's Ampere architecture. Using the command *nvidia-smi nvlink –status* the bandwidth between the two devices was confirmed to be 56.2 GB/s one way, this contrasts with the theoretical PCIe connection speed of 32 GB/s. The GPUs were connected through PCIe 4.0 ×16 links, as confirmed by the *nvidia-smi* command, giving a theoretical one-way PCIe bandwidth of approximately 32 GB/s per GPU link [10]. This results in 1.8 times faster bandwidth using the NVlink with more recent iterations of the NVLink being able to achieve one-way speeds of 1.8 TB/s, 32-times faster than the NVLink utilized as part of this project [9]. Even though the MASS CUDA library was designed to utilize the NVLink, the library theoretically works without its use. Without the direct physical connection between the devices, they should communicate with each other using the PCIe connection via the host, provided the devices are connected to the same host. However, this hardware configuration remains untested.

2.2 MASS CUDA – Initial Development

2012 signaled the initial development of MASS CUDA and work carried out this year served as the proof of concept for the library. A shortcoming of this early development was that the implementation details of CUDA were not abstracted away from the user, the implementation did not conform to the MASS library, and the library wasn't safely extensible by the user. This prompted a ground-up rebuilding of the MASS CUDA library by former student, Nathaniel Hart in 2015. Harts' implementation succeeded in increasing programmability of the MASS CUDA library, abstracting every CUDA feature away from the user. Performance of the library however, was still slower than the equivalent sequential execution [11].

Former lab member Lisa Kosiachenko made further progress on the library in 2018. She utilized GPU constant memory, minimized kernel context switching and optimized launch configurations to achieve performance improvements of 3.9 times when compared to the sequential CPU MASS version. She also investigated different migration conflict resolution algorithms [6].

The first multi-GPU implementation of the updated MASS CUDA library was completed in 2021 by former student, Ben Pittman, increasing the simulation space by 125%. Ben introduced ghost places, which are dummy places that facilitate border transfer of places and agent migration. Additionally, Ben implemented garbage collection and agent spawning enabling a dynamic amount of agents per simulation [12].

2024 was a milestone in the MASS CUDA library, where the library began to closely resemble the structure present today. Former student Warren Liu, fixed spawning bugs, fixed migration race conditions, and introduced a two-phase execution model for agents which fixed synchronization issues. Also introduced by Warren was an alternate data structure for agent and place attributes

(structure of arrays), ensuring the use of coalesced memory by CUDA threads and increasing performance by 7 times at larger simulation sizes. Warren's work made the CUDA library more usable and much higher performing but necessitated return to a single GPU implementation [7].

The most recent work on the MASS CUDA library was by former lab member, Holt Ogden in 2025. Holt partially implemented Warren's updated library to work across multiple-GPUs, the simulation space was split across multiple GPUs and ghost places were implemented to enable place communication at the border. This allowed place-based simulations, such as Heat2D, a simulation of heat passing through a material, or Game of Life, described in section 1.1, to work across multiple GPUs which are linked together using Nvidia's NVLink. Holt's work stopped short of enabling agents to work across multiple-GPUs [13].

2.3 MASS CUDA – KEY CONCEPTS

To understand the goals, evaluations, and conclusions of this paper, it is necessary to present some of the key concepts of MASS CUDA. This is a mid-level representation of core features that power the MASS CUDA library. The library is designed to be used with Nvidia GPUs that are physically connected using the NVLink and connected to a single host.

2.3.1 *Multi-GPU Simulation Space*

A simulation space consists of a 2-dimensional grid of places. For the multi-GPU implementation of MASS CUDA this grid is split evenly across two devices, additional devices are also supported. For two GPUs, the grid is implemented by instantiating two vectors of place objects, one on either device, shown in Figure 1. Addresses of the vector locations on device are stored on the host, this is enabled by the unified virtual address space, similarly vectors are also set up to instantiate agents on either device.

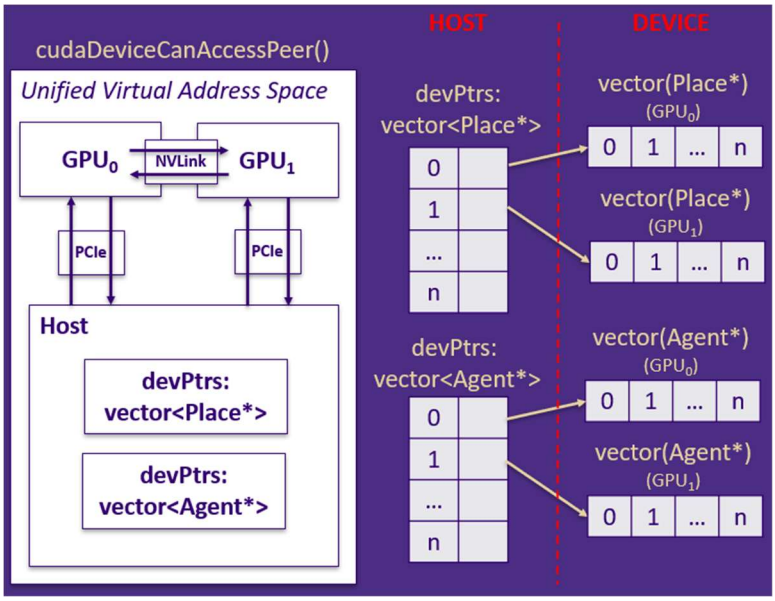

Figure 1. The Unified Virtual Address Space including Place and Agent Data Structure

Direct communication is enabled between these devices, and therefore between places, and agents, on either device. The host and the devices share a unified virtual address space, with each device being able to freely access and manipulate data located on the other device, this is enabled by the library during initialization using the cudaDeviceCanAccessPeer( ) command. The host can store device addresses but cannot manipulate device data. The unified virtual address space is also shown in Figure 1. This is how the host manages the place and agent objects on the device.

2.3.2 *Host Multi-Threading*

During initialization, the host sets up a single thread per device, using OpenMP. When a CUDA kernel call is required, each thread calls the kernel simultaneously for each device. For a call to all the places on each device, the place array pointer on each device, that is stored on the host, is passed into a kernel call. Each kernel call is managed with a different thread. Synchronization between parallel kernel calls to the devices is ensured as OpenMP threads

synchronize at the end of their parallel sections. When there are multiple kernel calls in a parallel section, the kernels calls to each device are synchronized using the *cudaDeviceSynchronize( )* call.

2.3.3 *Neighborhood*

Agents are free to move across the grid of places but can only move to other places that they can 'see', how far an agent can see is defined by the size of a place's neighborhood.

A place's neighborhood is a list of the places which a place can directly access, these are adjacent places in the grid. A place can access all attributes, including resident agents, from any place in its neighborhood. The neighborhood is user defined and is required to be initialized before a simulation can begin; the neighborhood can be set-up by the user with the *exchangeAll* function. Near the border between devices, these neighbor places will include ghost places.

2.3.4 *Ghost Places*

Ghost places are how place data is managed at the border between devices. They represent a copy of their corresponding cells on the neighboring device and ensure data synchronization. The purpose of ghost places is to ensure that information is available for neighboring places at the border to read as input.

In Figure 2, we can see two separate simulation spaces. Both are 7 x 10 grids, giving 70 places total split across two devices, these are represented in purple and are called base places. At the border between devices, we can see the ghost places highlighted in yellow.

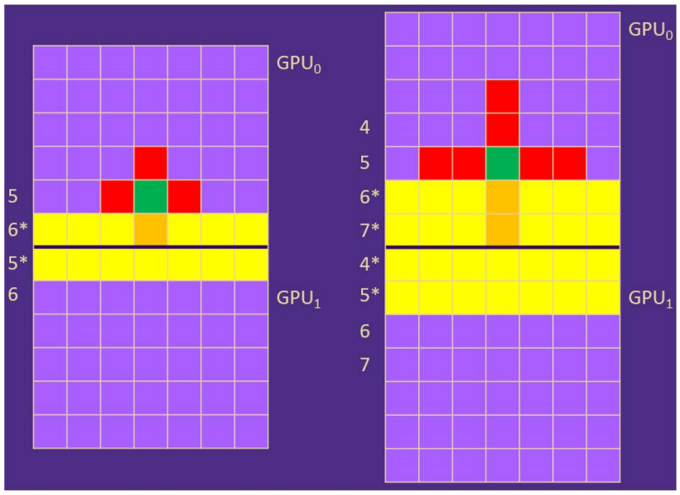

Figure 2. Two Simulation Spaces Examples with Ghost Places and Neighborhoods

The green cell's neighborhood is represented by the red and orange cells around it, these are the cells whose data is accessible from the green cell. The orange cells are the ghost places in the neighborhood. In the simulation space on the left, there only needs to be one row of ghost places as the neighborhood only extends one place in each direction. In contrast, in the simulation space on the right, there needs to be two rows of ghost places as the neighborhood extends two places in each direction.

After every place call by the host, a place function called will be executed across all base places. Assuming the call gets places to read data from their neighborhood as input and store this input in some way, the state of all base places has now changed.

A subsequent call to *exchangeGhostPlaces* copies base place data on one device to their corresponding ghost place on the other device, making the updated data available as input for the next round of place calls. Examining the rightmost simulation space in Figure 2, this means that rows 4 & 5 on GPU_0 get copied to rows 4* & 5* on GPU_1. Similarly, rows 6 & 7 on GPU_1 get copied to rows 6* & 7* on GPU_0. Data from ghost places in rows 4*, 5*, 6*, and 7* can now be used as input data for any cells whose neighborhood they are in.

2.3.5 *Agent Migration & Spawning*

During a simulation loop the user can call an agent function, take for instance a function that results in the agent moving between places (this is usually in response to an input from the surrounding environment). After an agent function such as moving, the user must then call *Agents.manageAll* from within the simulation loop. This in turn calls *migrateAgents* and *spawnAgents*. The result is that the agent functions are managed in that order, all agents that want to migrate are migrated and then all agents that need to spawn new agents, spawn new agents. Previously *terminateAgents* was part of agent management, Warrens implementation for agent spawning removed the necessity for this step [7].

A key complication for agent migration in ABMs is conflict resolution, how to handle multiple agents trying to migrate to the same space. Migration algorithms were previously studied by Lisa Kosiachenko, and in her thesis, she presented a custom migration conflict resolution algorithm as the ideal migration method due to its determinism and potential for customization despite its performance [6].

In the custom migration conflict resolution described in Lisa's work, agents register their interest in a place they want to move and then get added to a potential agents list at the place. During the *manageAll* step, the program calls *resolveMigrationConflicts* across all places, places then prioritize this list of potential agents. It is here that custom agent prioritization rules can be set, and any agents that are allowed to move to this place are moved to a residing agents list. updateAgentLocations is then called on all agents. Any agents that need to migrate check the place that they want to migrate to, if they are included on the places residing agent list, that agents location is now updated to its new location. If the agent is not on the residing agents list, the agent simply stays put [6].

Directional-ordered collision-free agent migration is another migration algorithm examined by previous MASS lab member, Christopher Bowzer. This algorithm is not implemented in MASS CUDA but was considered in this project. This is a simple migration algorithm that avoids agent collisions by moving agents North, East, South and West, in that order. If there is an agent in the way of a migrating agent, that agent doesn't migrate. Directional-ordered collision free agent migration is easy to implement as well as deterministic [14].

2.4 GOALS

Given the most recent developments of the MASS library by Holt Ogden, the next logical extension of MASS CUDA was to implement agents to function across multiple GPUs. The measure of success of this implementation was the successful implementation of several agent-based programs such as Tuberculosis and Sugarscape, described later in section 0. Additional verification of the library changes against Holt's benchmarking of the place based Heat2D simulation was required to ensure functionality and performance remained unchanged.

Chapter 3. RELATED WORK

There are several publicly available libraries that enable researchers to develop their own ABM models. However, few of these enable ABM development using GPUs.

3.1 FLAME

FLAME GPU 2 is a framework for developing agent-based models that can execute efficiently on GPUs while abstracting much of the low-level CUDA implementation from the user. Its purpose is similar to MASS CUDA in that both frameworks aim to make GPU-based ABM development more accessible. However, its implementation differs fundamentally from MASS CUDA. FLAME GPU 2 is primarily agent-centered and does not organize the simulation space as an array of Place objects in the same way as MASS CUDA. Instead, models are defined through agents, states, functions, messages, and environment properties. Agent-function execution is organized through a directed acyclic graph, which allows the framework to determine dependencies and schedule functions efficiently on the GPU [15].

3.2 MULTI-GPU ABMS

SIMCoV-GPU is an ABM that models COVID infections. Initially a CPU-based software, it was later adapted for use over multiple GPU nodes achieving significant performance improvements over the CPU version [16]. It is the clearest example of a multi-node, multi-GPU ABM. However, it is not generalizable in the same way that FLAME or MASS are.

3.3 COMPARISON WITH MASS

Among GPU enabled ABM modelling software libraries, the MASS CUDA library lies in a unique spot of being generalizable while also being able to run over multiple GPUs. So far there doesn't appear to be any other library out there that can be characterized in this manner.

Chapter 4. MULTI-GPU PARALLELIZATION USING MASS

4.1 AGENT MIGRATION

4.1.1 *Direction-Ordered Collision-Free Agent Migration*

Direction-Ordered Collision-Free Agent Migration was previously examined by former student Chris Bowzer [14]. The migration algorithm was included in this research to simplify the migration of agents across the GPU boundary when compared with the existing Custom-Conflict Resolution Agent-Migration Algorithm, described in section 2.3.5. A visualization of the migration algorithm is shown in Figure 3 with the simulation space shown in purple and the agents shown in gold.

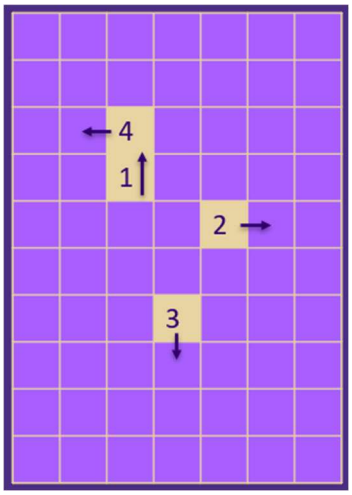

Figure 3. Direction-Ordered Collision-Free Agent Migration

Once all agents that are moving in a simulation step register their migration destination, *manageAll* is called, which triggers *migrateAgents*, the function that resolves all agents that need to migrate. For direction-ordered migration, four iterations over all the agents are required. The initial iteration calculates the direction that every agent is going to move based on their destination. On this first pass, the agents that need to move north are also moved. On the three subsequent passes over the list of agents, agents are moved in corresponding to their remaining directions E, S & W. On this simulation step, the agent marked "1" does not move as the space to the north is occupied.

4.1.2 *Border Migration*

Ghost places were initially intended to aid in agent migration, as ghost places can register migrating agents and resolve migrating agents' collisions in the same manner as base places. However, attempting to migrate agents across the device boundary using ghost places results in undefined behavior. This undefined behavior is due to the exchange of ghost place data (through *exchangeGhostPlaces*) and the functioning of the agent migration (using *manageAll*) being separate from each other. To correctly function together, these two operations would need to be tightly coupled, however, tight coupling of these functions would remove the flexibility currently afforded to the user by the MASS CUDA library.

Since the correct functioning of Agents and Places is already partially decoupled in the MASS CUDA library, it is then necessary to fully decouple their operation to enable agent migration across the boundary. The first step required to do this is to provide a direct reference for agents to the neighboring GPU, currently the neighboring place at the boundary visible by agents is the neighboring ghost place. The second step is to remove any reference of agents in the ghost place to prevent any undefined behavior.

Providing a direct reference for agents to pass across the GPU boundary is relatively simple. The *ACTUAL_ADDRESS* attribute was defined for each place, this is shown in Figure 4. *ACTUAL_ADDRESS* is initialized during instantiation of the neighborhood. For ghost places the value is initialized as the address of the place on the neighboring GPU that the place represents. For base places, the value is initialized simply as the address of that place, this is important to maintain the correct value as ghost places will copy this value each time ghost place exchange is called. During agent migration, the agent then checks if the place it is trying to move to is a ghost place, if so, the agent travels to the place pointed to by the *ACTUAL_ADDRESS* attribute.

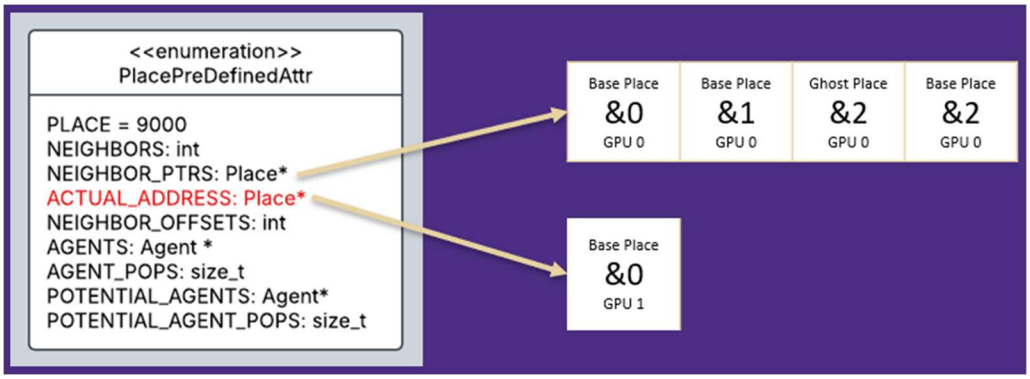

Figure 4. Place Attributes - Neighbor Pointers and ACTUAL_ADDRESS

Ghost places participate in the current migration algorithm, and can potentially cause undefined behavior, as the *resolveMigrationConflictsKernel* is executed on all places not just the base places. Executing this kernel solely on base places prevents ghost places from ever registering an agent to migrate there.

4.2 Host Design Considerations

The addition of a second GPU required the modification of data structures for storing pointers to agents on the device. This was implemented by leveraging the existing implementation for the place object vectors. These updated data structures are shown marked in red in Figure 5. As the

program now had to point to two separate vector arrays of places on two different devices, the existing *d_agent:Agent** attribute was changed to *devPtrs:vector<Agent*>*, a vector with an entry for every device. This matched the existing place data structure. Similarly, the pointers to the agent attribute arrays had to be nested behind an extra pointer so the attributes could now point to multiple devices.

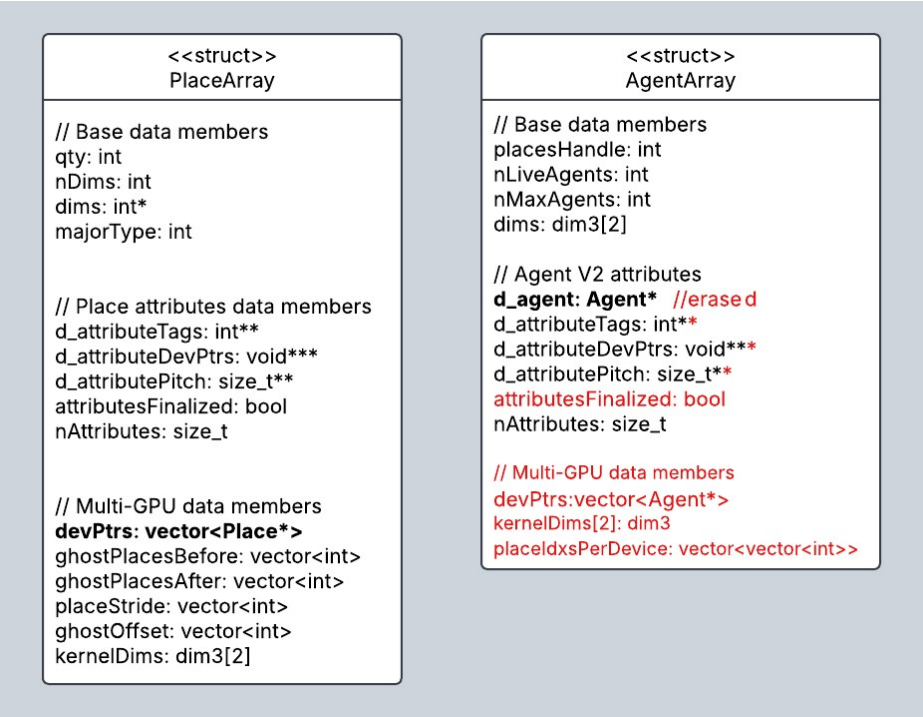

Figure 5. Host Side Place Attributes & Updated Agent Attributes

A consequence of updating these agent data structures meant that any functions involved in initializing agents on the device had to be reviewed and updated to work with the new data structures.

Another host design consideration was the pre-simulation step of agent instantiation across multiple devices. The existing library can handle an even number of places split across multiple devices, but instantiating an uneven number of agents across multiple devices now had to be considered. During simulation set-up an argument linking agents to their residing places is

required, *placeIdxs* is passed in by the user as a vector of integer values representing the place each agent is to be initially instantiated on. To implement this across multiple devices the incoming integers need to be parsed, split up into multiple vectors, and recalculated to give a device specific integer value. These vectors are stored in a vector called *placeIdxsPerDevice* in the *AgentArray* in Figure 5. This process is illustrated in Figure 6 .

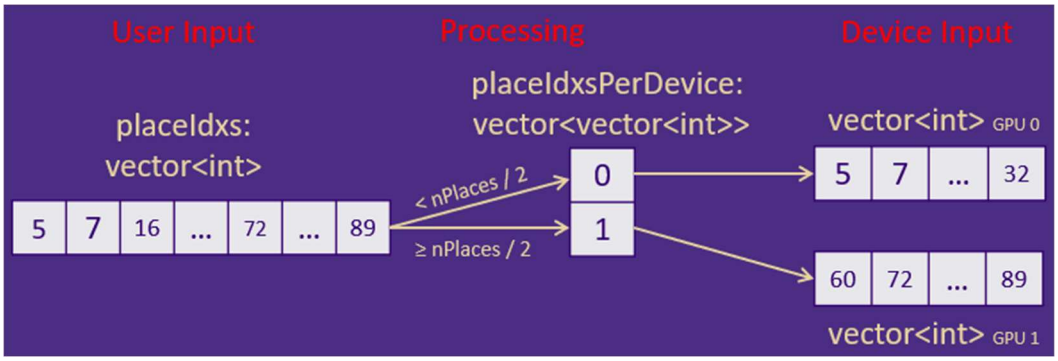

Figure 6. Processing of User Agent Location Index

4.3 GHOST PLACES

Motivated by the need to decouple agent migration from ghost place function, the question of the necessity of ghost places needs to be asked. In pursuit of this question an alternate initialization path was implemented. Using the current library implementation, references to ghost places in any neighborhood were updated to the ghost places *ACTUAL_ADDRESS* attribute. This provided a direct reference between the bordering base places of two GPUs and bypassed the need for ghost places. Additional benchmarking was carried out using this configuration on Heat2D.

Chapter 5. EVALUATION

Heat2D, SugarScape and Tuberculosis are ABM programs commonly used in the MASS lab for benchmarking, all programs were run on one and two devices as part of the evaluation of the implementation of MASS CUDA, giving two separate series of results for each program. Each series of results were based on increasing the simulation space, starting at 100 x 100 places and increasing in size until the program would not run due to lack of memory. This was the maximum simulation size for that configuration. The times for each run were recorded and the timing results for each configuration were then graphed against the size of the simulation space. Presented below are the benchmark applications, the results of benchmarking and the analysis of two migration algorithms.

Benchmarks were run on Heat2D for validation of library function after substantial changes were made, previous benchmarking results from Holt Ogden served as the baseline. Additionally, Heat2D, having significant cross border communication between places, is ideal for testing out the removal of ghost places. SugarScape is a true agent-based simulation and is ideal for benchmarking the library update as it demonstrates agent performance increases across multiple GPUs, additionally SugarScape has previously been benchmarked by Warren Liu and Lisa Kosiachenko with Lisa providing benchmarks on migration algorithms using SugarScape. Finally, Tuberculosis was selected to provide a second agent-based simulation from which agent performance gains could be observed.

These programs were benchmarked using two NVIDIA RTX A5000 graphics cards, each with 24 GB of device memory, connected using an Nvidia NVLink 4.0.

5.1 Benchmark Applications

5.1.1 *Heat2D*

Heat2D models the flow of heat represented in a 2-dimensional space, the simulation output is shown in Figure 7. In our benchmark test heat is applied at one edge of the plane for a sustained period. The heat then passes through the material, from place to place. Agents are not used in this simulation, which makes it more like a CA as described earlier in section 1.1, as opposed to a true ABM.

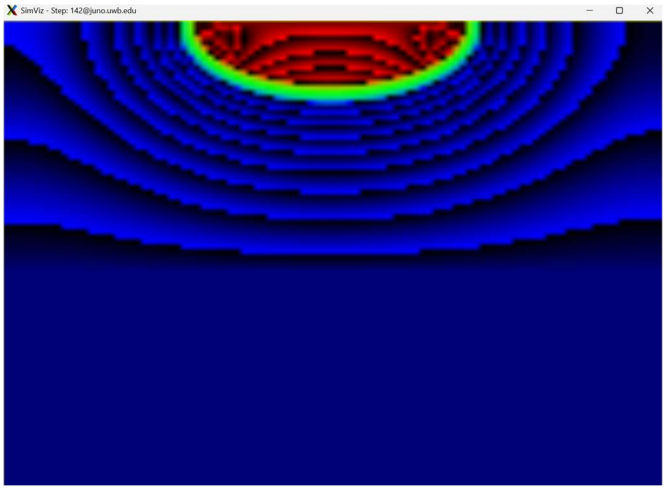

Figure 7. Heat2D Simulation Visualization

Each simulation step involves calling each place, taking in temperature information from the place's neighbors, and calculating the place's current temperature accordingly. This gives us a time complexity of O(n) in terms of the number of places.

The result is a simulation which focuses on the place implementation of MASS CUDA. Holt Ogden's benchmarking of this program in 2025 contrasted the Heat2D program when ran on one and two devices. Holt found a 1.7 times increase in performance when using two GPUs as well as a spatial increase of 77% [13]. Although this benchmark test relates solely to places, benchmark

testing was also carried out during this project to verify correctness and to ensure correct functionality.

5.1.2 *SugarScape*

SugarScape, shown in Figure 8, initializes two piles of sugar in two different locations on a 2-D simulation space. Ants are represented by agents and are spawned across the grid. The ants respond to the level of sugar in the places and will move toward higher sugar levels. SugarScape's time complexity is related to the number of agents and places, given by O(nPlaces + nAgents), effectively O(n). SugarScape does not feature any agent spawning.

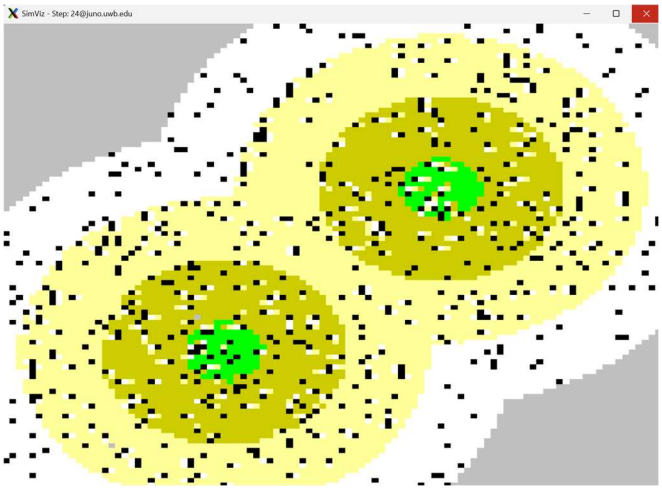

Figure 8. SugarScape Simulation Visualization

This program was chosen as it was an agent-based program that has been benchmarked previously by Warren Liu [7] and Lisa Kosiachenko [6]. Lisa's paper also investigates several migration conflict resolution algorithms, these will serve as a comparison with the new multi-GPU agent implementation. Similarly, comparison of results for the single GPU execution time, with Warren's results, will provide verification of the implementation and will stand as a comparison to the multi-GPU implementation.

5.1.3 *Tuberculosis*

The tuberculosis (TB) simulation, seen in Figure 9, is used to model an infection of TB in human lung cells. The space is created by a grid of places which store attributes of bacteria and chemokine levels, places can pass the bacteria to neighboring places. The body fights the infection using T-Cells and Macrophages represented by agents, these agents follow the chemokine levels in the places and attack the bacteria when located.

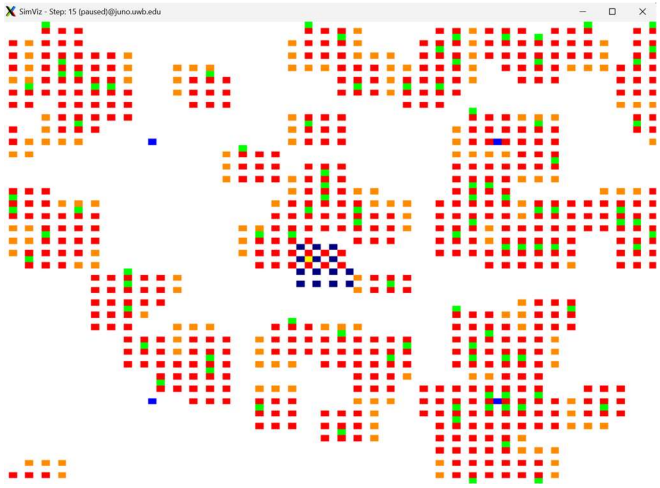

Figure 9. Tuberculosis Simulation Visualization

The complexity of this simulation is given by O(nPlaces + nAgents1 + nAgents2), practically O(n). Therefore, any real performance variation in this simulation should be dictated by data access patterns on the device. This is evidenced in Holts Ogden's Spring 2025 report which benchmarks TB using Brian Lugers MASS CUDA version 0.5.3 against Warren Lius version 0.7.1 [17]. Before Warrens update to MASS CUDAs memory structure, poor memory access patterns effectively made the runtime of the TB an O(nlogn) operation. Warrens updated memory structure drastically improved performance.

The TB program was chosen for benchmarking using the updated library as the results would serve as an alternate to SugarScape. Any potential differences between the two ABMs could

provide useful insight. Additionally, a key feature of the TB program that makes it interesting to benchmark, is that it utilizes agent spawning, a feature not present in SugarScape.

5.2 Single vs Multi-GPU Performance

In all benchmark programs the relationship between simulation space size and simulation time proved to be linear. The slope of this linear line is the increase in simulation time per increase in simulation size. Dividing the slope of the 2 GPU series values into the slope of the 1 GPU series value for each benchmark program provides the *speedup* value, this reflects the performance increase provided by adding a second GPU.

Additionally, the last value in each data series reflects the largest simulation space size that can be run on each configuration. Comparing these values between 1 and 2 GPU configurations provides an increase in simulation size value, given as a percentage.

5.2.1 *Heat2D*

The results of Heat2D benchmarks are shown in Figure 10. These benchmarks compare the total simulation time of the Heat2D program when using one and two GPUs.

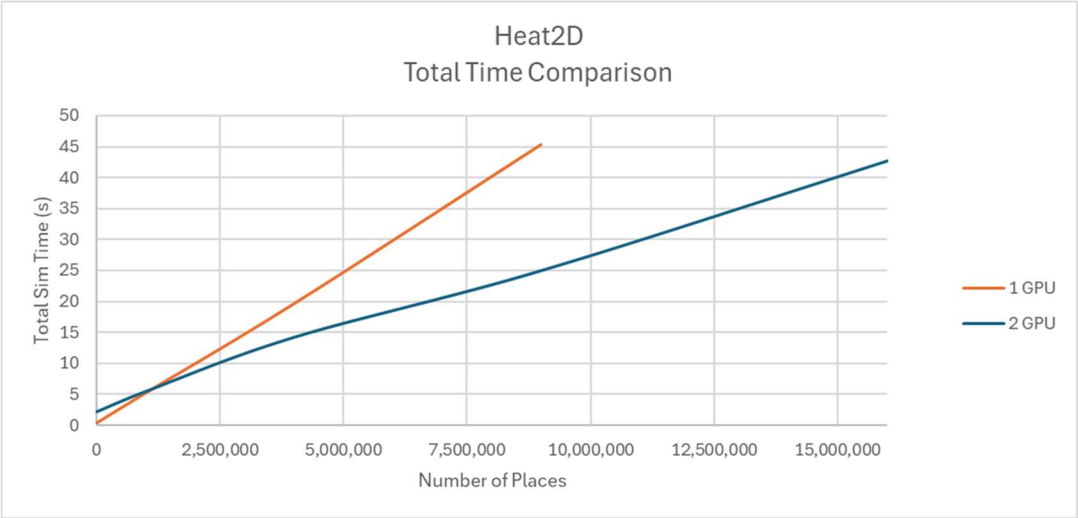

Figure 10. Heat2D Benchmarks using 1 and 2 GPUs.

For Heat2D, the largest simulation size that could run on a single-GPU was one with 9.0 x 10^6 places, this produced a slope of 4.99 x 10^{-6}. In contrast the 2-GPU implementation could run with a maximum of 16.0 x 10^6 places and produced a slope of 2.49 x 10^{-6}.

Using two GPUs provides a 1.95-times speedup when compared with the single GPU implementation for Heat2D. There is also an increase in the simulation space of 78%. These results are broadly in line with those of Holts results of 77% increase in simulation size and speed-up of 1.7 times.

The difference in speedup values between these benchmarks and those of Holt Ogden's was entirely due to a higher slope for the single GPU implementation data series, the performance of the 2-GPU implementation was identical. This is likely due to differences in how the single GPU data was generated. Holts benchmarks involved using Warren Liu's single-GPU implementation of MASS CUDA, whereas the benchmarking for this program involved modifying the current multi-GPU version of the MASS CUDA code to only set up a single GPU during initialization. This likely introduced a higher initialization overhead for the single-GPU benchmark in this project.

5.2.2 *SugarScape*

The results of SugarScape benchmarking are shown in Figure 11. These benchmarks compared two migration algorithms, Custom Conflict Resolution Agent Migration and Direction-Based Collision-Free Agent Migration, ran across one and two GPUs.

Running SugarScape on one GPU produced a maximum simulation space of 7.56 x 10^6 places for custom conflict resolution agent migration and 10.89 x 10^6 places for direction-ordered agent-based migration. When run on 2 GPUs the simulation space increased to 13.69 x 10^6 places and 20.25 x 10^6 respectively for custom migration and direction-ordered migration. This reflects an

increase in the simulation space between 81-84% when 2 GPUs are used and an increase in the simulation space of 31-32% when direction-ordered agent-based migration is used. This indicates that there is a lower memory overhead when using direction-ordered migration, allowing for larger simulations.

Figure 11. SugarScape Benchmarks using 1 and 2 GPUs.

For Custom Migration Conflict Resolution, running SugarScape on a single GPU produced a slope of 7.76×10^{-8}, running on 2 GPUs produced a slope of 6.38×10^{-8}. Running the Direction-Ordered Collision-Free Agent Migration algorithm across one GPU produced a slope of 7.37×10^{8}, and on 2 GPUs the slope was 6.03×10^{-8}.

These results correspond to a 1.1-times speedup between the different migration algorithms whether run on either 1 or 2 GPUs, and a 1.2-times speedup when running either algorithm across 2 GPUs.

5.2.3 *Tuberculosis*

The results of the tuberculosis benchmarking are shown in Figure 12. These benchmarks compare the Tuberculosis program when ran on 1 and 2 GPUs. The maximum simulation size was $7.29x10^6$ when ran on 1 GPU and $12.25x10^6$ when run on 2 GPUs, an increase of 68%.

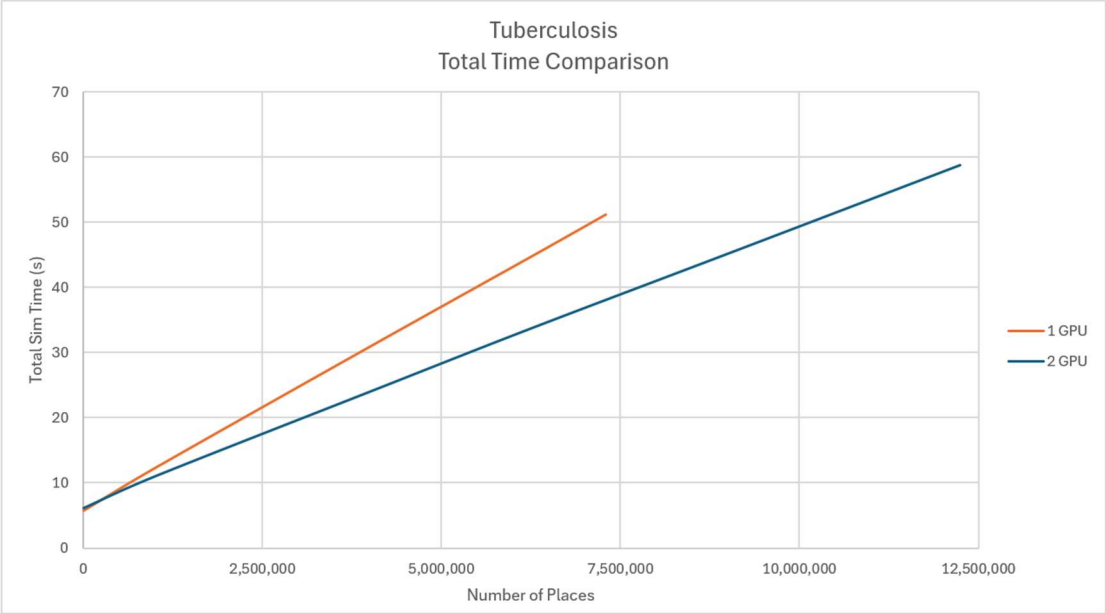

Figure 12. Tuberculosis Benchmarks using 1 and 2 GPUs.

The slope provided by the simulation when run on a single GPU was 6.23×10^{-6} and the slope when using 2 GPUs was 4.29×10^{-6}. Marking a 1.45-times speedup when ran on two GPUs.

5.3 Ghost Place Exchange and Direct Access Comparison

The average simulation step time of Heat2D utilizing ghost place exchange, Heat2D using ghost place exchange without the exchange step, and Heat2D run with direct access of places across the device boundary is shown in Figure 13.

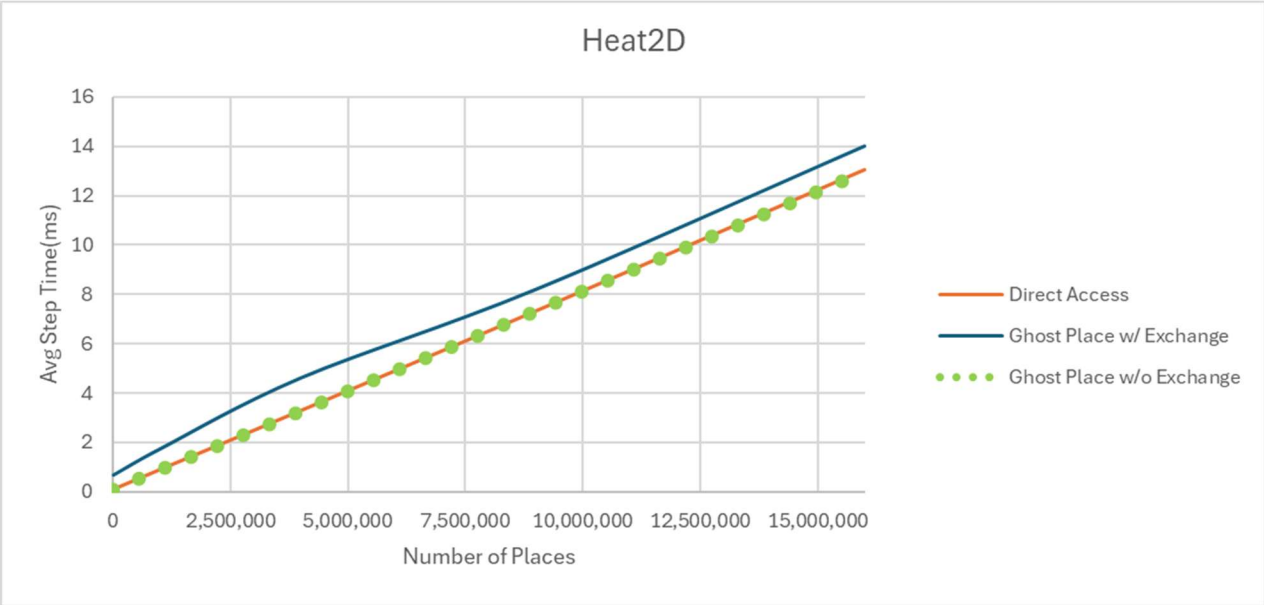

Figure 13. Ghost Places and Direct Access Comparison

These results demonstrate that enabling direct access of places across the GPU boundary does not negatively affect the speed of the Heat2D program run on the MASS CUDA library. There is no degradation in performance as the rate of execution of the kernel functions across all the places at the boundary doesn't exceed the rate at which the data is able to transfer between the two devices.

5.4 Parallel Performance Discussion

A summary of the benchmark results is shown in Table 1, below. There is a varied increase in the simulation space across all three programs, and the performance gains of the agent-based simulations are less than that of the place-based heat2D simulation.

Table 1. Summary of Spatial and Performance Gains

	Heat2D	**SugarScape Custom Conflict**	**SugarScape Direction Based**	**Tuberculosis**
Simulation Space Increase	78%	81%	86%	68%
Speedup	1.95-times	1.22-times	1.22-times	1.45-times

The most notable difference in this data is the varied speedup in performance across the different simulations due to switching to two GPUs. The simulation that benefits the most from extra GPUs is Heat2D, followed by Tuberculosis, and finally SugarScape.

This is likely due to the overhead from fetching of data from global device memory. Heat2D shows nearly a perfect 2-times increase in performance when adding an extra GPU, this result is intuitive. As the GPU processes the whole array of places, the only extra data that is needed by each place is from its neighbors, this data should already be loaded into local memory from the global device memory. This was detailed by both Lisa Kosiachenko and Warren Liu [6, 7].

In contrast, in Tuberculosis and SugarScape, the device iterates through each Agent array, making calls to the place in which it resides and to its neighbors. This requires more memory fetching from global device memory than is required from a purely place based simulation, resulting in a decrease in performance.

There is still a difference between the Tuberculosis and SugarScape speedup that must be explained. Further investigation into the code revealed the likely cause. SugarScape and TB have

fundamentally different methods of enabling agents to view their neighboring cells. SugarScape makes two *exchangeAll* function calls per simulation, the first is to update each places attributes using neighbor information, the second call is to randomize the direction of the agents. *exchangeAll* rebuilds the neighborhood of every place, and in SugarScape it is called twice in a row, creating extreme computational overhead.

TB, in contrast, simply calls every agent to peek into their resident places' neighborhood to decide the next place to migrate to. Data fetching of the places, when agents are already loaded, causes TB to not reach the performance increase achieved by Heat2D, however TBs speed-up is still better than SugarScapes.

5.5 Migration Algorithm Performance

Part of the SugarScape benchmarking involved the investigation of the Direction-Ordered Collision-Free agent migration algorithm. Testing showed this algorithm to have a speedup of 1.1 times compared to the Custom Migration Conflict Resolution migration algorithm. This speedup is due to there being one less kernel call across all the places than in the direction-based algorithm. Lisa Kosiachenko originally tested several migration algorithms and proceeded using the Custom Conflict Resolution algorithm, despite worse performance, due to its potential for customizability of conflict rules [14].

A result that was unexpected was the increase in simulation size that could be run when using the direction-ordered migration algorithm. Increases of 30% and 32% of simulation size were achieved on 1-GPU and 2-GPUs respectively when using the direction ordered algorithm. This is likely due to a reduced device memory overhead when using the more straightforward direction ordered algorithm.

Direction-Ordered migration was originally studied in the hopes its simplicity of implementation would make cross border migration more straightforward. It has been shown to have modest increases in performance over custom-conflict agent migration that may not be worth the tradeoff of not being able to choose conflict resolution rules. However, given that there is nearly a one-third increase in the simulation space when using direction-ordered agent migration, the algorithm should be included as an option for the user to pick when implementing an ABM simulation.

5.6 ATTRIBUTE STORAGE

In addressing limitations of his implementation of the MASS CUDA library, Warren Liu noted that his revised data structure "consumes more memory compared to the previous design" [7]. This has important implications for the performance and spatial scalability of the MASS CUDA library. Research from others in the MASS lab identified inefficient storage of attributes on the device using CUDA functions. Refinement of this has produced significant spatial and performance increases.

Chapter 6. CONCLUSION

6.1 Summary of Work Contribution

The main goal of this research was to extend the agents implementation of the MASS CUDA library across multiple devices. This was achieved, and the implementation of this goal brought increases in simulation space in line with those from the places only implementation of MASS CUDA.

Another contribution from the testing carried out as part of this project was to detail the performance increases achieved in agent-based simulations. The testing demonstrated increases in performance of agent-based simulations but showed that these increases are more modest than purely place based simulations.

While researching this project, the question also began to be asked about the necessity for ghost places. Would removing ghost places hinder the simulation performance due to excessive data travel across the NVLink? Testing showed this is not the case and that removing the ghost places and implementing direct addressing between border places resulted in the same performance.

6.2 Future Work

Due to the potential for significant space and performance increases by implementing an improved data structure for 2-D attributes, work carried out by others in the MASS lab should be verified using the current MASS CUDA library through implementation of the same storage improvements and subsequent comparison of simulation space and performance increase.

Direct addressing between places at the boundary should be considered for any future rewrite of the MASS CUDA library. There are modest performance and spatial increases to be made doing

this, but the many parts of the library become extremely simplified by using this method as opposed to utilizing ghost places.

There is one note of caution with removing ghost places however, due to the potential large increase in simulation size and performance, the simulations may reach a size and performance where utilizing direct addressing between places across the device boundary may cause bottlenecks in the simulation. In addition to implementing the improved data structure, the comparison tests run earlier in section 0 should be rerun to see if this bottleneck occurs.

Significant work has been put into abstracting all the CUDA implementations away from the user to make the library more accessible, but considerable effort is still required for someone not familiar with the MASS CUDA to develop an ABM using the library. It would be helpful to abstract the implementation of an ABM further away from the user. This could be achieved by mapping out ABM development workflows, generating templated agents and places, and having this implementation hidden behind a more user-friendly interface.

Finally, to bring MASS CUDA in line with MASS C++, additional development of the library should be considered to enable development of a 3-dimensional simulation space. The usefulness of a 3-dimensional implementation against the effort of this update should be evaluated, however. This depends on the applications that can be developed for the library and if there is an associated demand for a 3-dimensional simulation space. This library update should not be prioritized over a new user interface that would simplify the development of MASS CUDA simulations.

BIBLIOGRAPHY

[1] A. Adamatzky, Ed., *Game of Life Cellular Automata*. London: Springer London, 2010. doi: 10.1007/978-1-84996-217-9.

[2] T. C. Schelling, "Models of Segregation," *The American Economic Review*, vol. 59, no. 2, pp. 488–493, 1969.

[3] J. M. Epstein and R. L. Axtell, *Growing Artificial Societies: Social Science from the Bottom Up*. The MIT Press, 1996. doi: 10.7551/mitpress/3374.001.0001.

[4] R. L. Axtell and J. D. Farmer, "Agent-Based Modeling in Economics and Finance: Past, Present, and Future," *Journal of Economic Literature*, vol. 63, no. 1, pp. 197–287, Mar. 2025, doi: 10.1257/jel.20221319.

[5] J. Emau, T. Chuang, and M. Fukuda, "A multi-process library for multi-agent and spatial simulation," in *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, BC, Canada: IEEE, Aug. 2011, pp. 369–375. doi: 10.1109/PACRIM.2011.6032921.

[6] E. Kosiachenko, "Efficient GPU Parallelization of the Agent-Based Models Using MASS CUDA Library," 2018.

[7] W. Liu, "Programmability and Performance Enhancement of MASS CUDA," 2024.

[8] "2.6. Unified and System Memory — CUDA Programming Guide." Accessed: Jun. 02, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/understanding-memory.html

[9] "NVLink & NVLink Switch for Advanced Multi-GPU Communication," NVIDIA. Accessed: Jun. 02, 2026. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[10] "PCI Express: Delivering Needed Bandwidth for Open Compute Project | PCI-SIG." Accessed: Jun. 02, 2026. [Online]. Available: https://pcisig.com/blog/pci-express-delivering-needed-bandwidth-open-compute-project

[11] N. B. Hart, "MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks," 2015.

[12] B. D. Pittman, "Multi Agent Spatial Simulation (MASS) in multiple GPU Environment with NVIDIA CUDA," 2021.

[13] H. Ogden, "Multi-GPU Parallelized Agent-Based Modeling," *Capstone Paper*, 2025.

[14] M. Fukuda, C. Bowzer, B. Phan, and K. Cohen, "Collision-Free Agent Migration in Spatial Simulation," presented at the 2017 Federated Conference on Computer Science and Information Systems, Sep. 2017, pp. 65–73. doi: 10.15439/2017F172.

[15] P. Richmond, R. Chisholm, P. Heywood, M. K. Chimeh, and M. Leach, "FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs," *Softw Pract Exp*, vol. 53, no. 8, pp. 1659–1680, Aug. 2023, doi: 10.1002/spe.3207.

[16] 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*, Pisa Italy: ACM, Jun. 2024, pp. 322–333. doi: 10.1145/3625549.3658692.

[17] H. Ogden, "MASS CUDA Tuberculosis," University of Washington Bothell, Term Report, Spring 2025.

APPENDIX A
COMPILATION INSTRUCTIONS

There are separate repositories for the MASS CUDA application source code, and the MULTI-GPU enabled MASS CUDA library source code. Within the application repository there is a make that is used to compile the application. Within the applications make file, the MASS CUDA library version is referenced, this can be changed to compile the library using a specific library branch. The library branch, and the application branches used in this project, are listed in Table 2.

Additionally, within the applications make file the source of the MASS CUDA library can be specified, this can be from bitbucket or can be a repository that is stored locally. For development of this project the library code was cloned locally, and a new branch was set up for development. The applications make file was then set to reference the locally developed branch of the MASS CUDA library.

Table 2. Project Repository Locations

Repository	**Branch**	**Source Code Location**
Heat2D	holto/Heat2D-multi-gpu	Heat2D/Heat2D_MASS/src
Tuberculosis	holto/Tuberculosis	Tuberculosis/src
SugarScape	holto/Heat2D-multi-gpu	SugarScape/SugarScape_MASS_2024/src
MASS CUDA Library	robfosk/multi-gpu-cuda-agents	mass_cuda_core/src

MASS CUDA applications can be cloned from:

https://bitbucket.org/mass_application_developers/mass_cuda_appl

The MASS CUDA library can be cloned from:

https://bitbucket.org/mass_library_developers/mass_cuda_core