Programmability and Performance Enhancement of MASS CUDA

Warren Liu

A capstone report

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2024

Program Authorized to Offer Degree:

Computer Science & Software Engineering

University of Washington

Abstract

Programmability and Performance Enhancement of MASS CUDA

Warren Liu

Chair of the Supervisory Committee:
Professor Munehiro Fukuda
School of STEM Computing and Software Systems Division

Agent-based modeling (ABM) has proven valuable across various fields for capturing the intricacies and heterogeneity of real-world systems. However, as ABM simulations become more sophisticated and larger in scale, the need for efficient parallelization arises. Graphics Processing Units (GPUs) have emerged as a compelling alternative for parallelizing ABM simulations, offering high computational power and parallelism. In this project, we aimed to enhance the MASS CUDA library, a GPU-accelerated ABM framework, by improving its programmability and performance. We implemented essential agent functions, redesigned data structures to enable coalesced memory access, and introduced a dynamic attribute setting mechanism. These enhancements led to significant improvements in programmability and performance, as demonstrated through benchmarking against a previous version of MASS CUDA and a competing library, FLAME GPU 2, using four diverse applications. The evaluation showcased MASS CUDA’s effectiveness in terms of programmability, performance, and scalability. The improved programmability and performance of MASS CUDA enable users to focus on the modeling aspects of their simulations while harnessing the computational capabilities of GPUs. By offering a scalable and accessible framework for GPU-accelerated ABM, MASS CUDA has the potential to accelerate scientific discovery and decision-making processes in numerous fields.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to my advisor and committee chair, Dr. Munehiro Fukuda, for his invaluable guidance, steadfast support, and continuous encouragement throughout this project. I also extend my appreciation to my committee members, Dr. Kelvin Sung and Dr. Clark Olson, for their insightful feedback and valuable suggestions. I am deeply grateful to my friend, Cathie Wang, for her unwavering support and for being a constant source of motivation and inspiration during my graduate studies. Her encouragement and emotional support have been invaluable, especially during times when significant issues arose. Finally, I would like to thank my family for their unconditional love, understanding, and support throughout my academic journey. Their belief in me has been a driving force behind my perseverance and success.

Chapter 1. INTRODUCTION

Agent-based Modeling (ABM) is an approach to simulating complex systems by defining the behavior and interactions of individual agents within a shared environment. Agents are autonomous entities that operate based on their own set of rules and characteristics [1]. As these agents interact with each other and their surroundings, complex behaviors and system-level patterns emerge. ABM has proven valuable across a wide range of fields, from social sciences to biology, due to its ability to capture the intricacies and heterogeneity of real-world systems.

As ABM simulations become more sophisticated and larger in scale, the need for parallelization arises. By distributing the computational workload across multiple computing nodes, parallelization enables faster execution times and the ability to handle more complex simulations. Researchers can explore a broader range of scenarios, run multiple simulations concurrently, and obtain results more quickly by leveraging parallel computing techniques.

Parallelization in ABM has relied on CPU cluster-based approaches, where the simulation is spread across multiple interconnected computers or nodes [2] [3]. Each node typically contains one or more CPUs that work together to process the simulation. While this approach can yield significant performance improvements, it also presents challenges as the number of nodes increases. The overhead associated with distributing data and managing communication between nodes can limit the scalability and efficiency of the simulation.

In contrast, Graphics Processing Units (GPUs) have emerged as a compelling alternative for parallelizing ABM simulations [4]. GPUs are specialized processors designed for highly parallel workloads, boasting a large number of cores that can execute many threads simultaneously. While an individual GPU core may be slower than a modern CPU core, the cumulative processing power of a GPU can vastly outpace that of a modestly-sized CPU cluster. By harnessing the parallelism of GPUs, ABM simulations can achieve significant performance gains without the need for extensive hardware infrastructure.

However, the choice between GPU and CPU-based solutions is not always straightforward and depends on the nature of the ABM tasks. Tasks with high parallelism and low inter-thread communication may benefit more from a GPU's parallelism, such as large-scale simulations where agents operate independently, while others that require complex inter-agent interaction or frequent synchronization might still require the scalability and flexibility of a distributed CPU-based system [4]. It's important to note that CPU clusters can scale by adding more computing nodes to acquire more memory space and computational resources. In contrast, current GPU solutions typically support only a single GPU card with limited memory space (usually 16 or 32 GB), making them more suitable for relatively small simulations that prioritize computational speed over size. As a result, GPU-based solutions are often chosen when the primary concern is the speed of computation rather than the scale of the simulation.

MASS CUDA is an agent-based modeling framework that leverages the power of GPUs to enable massive parallelization while preserving the core principles and ease of use of ABM [5]. Built upon the foundation of the Multi-Agent Spatial Simulation (MASS) library, MASS CUDA offers a flexible and intuitive platform for developing and executing ABM simulations on GPUs. We aimed to maintain a consistent API between the distributed CPU and GPU versions of the library. However, due to the inherent differences in data structures and memory management between CPUs and GPUs, programs need to be modified before they can be moved from one version to another, as the usage becomes completely different.

Our work on the enhancement of MASS CUDA focuses on several key areas: implementing agent functions to ensure that the framework supports the core concepts of ABM, such as agent termination, spawning, and migration; optimizing algorithms and data structures to fully utilize the parallel processing capabilities of GPUs; and creating comprehensive documentation to facilitate the adoption and usability of MASS CUDA.

Chapter 2. BACKGROUND

In this section, we introduce the MASS library, discuss its basic components, functionalities, and how it is parallelized on CPU clusters. We then explore the transition to the GPU platform with the development of MASS CUDA, its current status, and the challenges faced during this process. Finally, we highlight the problems in the previous implementation that are addressed in the project.

2.1 MASS CUDA

MASS is a specialized ABM framework on the CPU platform for developing agent-based models in parallel and distributed computing environments [6]. The MASS library is structured around two fundamental concepts: Place and Agent. Place serves as the spatial container or environment where agents operate, holding relevant data and defining spatial relationships. Agents are the dynamic, computational units that contain user-defined logic and can move from one place to another, executing computations that influence their own state and the state of the places they interact with.

MASS introduces the Agents and Places classes to manage agent and place objects, offering a high-level API for user interaction and manipulation of the agent-based model. These classes provide essential functions such as callAll(), which executes a passed-in function on all place or agent objects, and exchangeAll(), which facilitates information exchange among place objects.

Agents also have a manageAll() function to manage the behavior of all agent objects, including migration, spawning, and termination.

To enhance performance, MASS can be run on a cluster of multiple computing nodes, parallelizing the computation by distributing the workload across nodes. In this setup, place objects are stored in shared memory accessible by all nodes, while agent objects reside in each node's local memory.

MASS CUDA extends the library to leverage the computational power of GPUs, particularly NVIDIA GPUs with CUDA technology. By allocating place and agent objects on the device memory (GPU memory) and assigning each thread to handle one or more objects' computation, MASS CUDA achieves massive parallelization compared to CPU clusters.

2.2 Current Status & Challenges

At the start of this project, the Places-related functions in MASS CUDA were fully functional. However, the Agents-related functions, while structurally in place, had incomplete or non-functional implementations due to algorithmic bugs in agent migration, spawning, and termination.

Additionally, the existing algorithm and data structure, along with previous hardware restrictions, limited the scalability of MASS CUDA to a single GPU. The ultimate goal was to extend the library to run on multiple GPU cards for even greater parallelization.

2.3 Problems In The Previous Work

The primary issues in the previous work included the incomplete implementation of agent-related functions and performance problems caused by uncoalesced memory access. Uncoalesced memory access occurs when threads in a warp (a group of threads executed simultaneously on a GPU) access non-contiguous memory locations, leading to suboptimal memory bandwidth utilization and increased memory latency [7]. Profiling the complete implementation revealed that MASS CUDA was significantly slower than its main competitor, FLAME GPU 2. Moreover, the lack of up-to-date and comprehensive documentation made it challenging to understand, develop, and use the library effectively.

By tackling these problems, we aim to enhance the functionality, performance, and usability of MASS CUDA, making it a more powerful and accessible tool for agent-based modeling on GPU platforms.

Chapter 3. RELATED WORKS

In this section, we explore various ABM platforms and libraries, focusing on their key features, performance, and scalability. We discuss three widely-used CPU-based ABM platforms: MASON, Repast, and NetLogo, and evaluate their strengths and weaknesses based on a comparative study. Additionally, we examine the growing trend of utilizing graphics processing units (GPUs) to accelerate agent-based simulations, introducing three notable GPU-based ABM libraries: FLAME GPU, FLAME GPU 2, and TurtleKit, and assessing their unique features and scalability limitations. Finally, we introduce MASS CUDA, our novel approach to GPU-based ABM that aims to address the limitations of existing libraries, highlighting its key features, intuitive programming model, and the trade-offs between performance and ease of use.

3.1 CPU Based ABMs

MASON (Multi-Agent Simulator of Neighborhoods or Networks) is a Java-based discrete-event multi-agent simulation library core [8]. It is designed to be a fast and lightweight simulation platform, focusing on computational efficiency and extensibility. MASON offers a variety of features, including a discrete-event scheduler, a 2D and 3D visualization suite, and a range of utilities for model development and analysis. It supports both grid-based and continuous space models, as well as network-based models. However, it does not support scaling on a cluster, limiting its ability to handle large-scale simulations.

Repast (Recursive Porous Agent Simulation Toolkit) is another widely-used Java-based ABM platform [9]. It provides a comprehensive set of tools for developing, executing, and analyzing agent-based models. Repast offers three main implementations: Repast Simphony (Java), Repast for Python (Python), and Repast for High-Performance Computing (C++). Repast Simphony is the most widely used implementation and provides a rich set of features for ABM development in Java. It includes a fully concurrent discrete event scheduler, a variety of agent architectures (e.g., grid, continuous space, network), and built-in tools for adaptive behavior and learning [10]. Same as MASON, the Java version is not scalable on a cluster, Repast for High-Performance Computing does support cluster scaling.

NetLogo is a multi-agent programmable modeling environment developed at Northwestern University [11]. It is designed to be an accessible and user-friendly platform for ABM development, particularly well-suited for educational purposes and rapid prototyping. NetLogo uses its own custom programming language, which is based on the Logo language and provides a simple and intuitive syntax for defining agent behaviors and interactions. It is not scalable either.

Railsback et al. [12] conducted a comparative study of these three ABM platforms, along with a Java version of Swarm. They implemented a series of 16 example models using each platform and evaluated their performance, ease of use, and available features. The study found that NetLogo stood out for its ease of use and extensive documentation, making it a good choice for those new to ABM. MASON and Repast offered good performance and a wide range of features, making them suitable for more complex models. The study also highlighted the importance of documentation and user support in the adoption and effective use of ABM platforms. The authors provided recommendations for future development of ABM platforms, emphasizing the need for improved documentation, standardized model description formats, and better tools for model analysis and visualization.

Most papers do not mention Java-based parallel ABM simulators running on cluster systems. Typically, Java-based ABM simulators like Repast Simphony and MASON are designed to run on a single computer, not a cluster.

Repast HPC is a C++ implementation of the Repast Simphony modeling system, designed for large-scale distributed computing platforms. It uses MPI for process communication and synchronization. In Repast HPC, each process is responsible for a set of local agents, and the simulation environment is shared across processes. Agents can be migrated between processes to optimize performance. Repast HPC provides a ReLogo-like language to allow a Logo-like model specification in a parallel environment, aiming to make it easier for modelers to scale up their simulations without dealing with the low-level details of parallel programming [13].

The EURACE project, implemented using the Flexible Large-scale Agent Modeling Environment (FLAME) [14], implemented in C++, aims to construct a large-scale agent-based model of the European economy with up to 107 agents [15]. FLAME leverages the fact that much agent communication is local, and by clustering "neighboring" agents on the same process, costly inter-process communication can be reduced.

3.2 GPU Based ABMs

In addition to the CPU-based ABM platforms, there has been an increasing trend of utilizing graphics processing units (GPUs) to accelerate agent-based simulations. GPUs offer high computational power and parallelism, making them well-suited for large-scale ABMs with many agents and complex interactions. Several GPU-based ABM libraries have been developed to harness the power of GPUs.

FLAME GPU [16] is an extension of the FLAME (Flexible Large-Scale Agent-Based Modeling Environment) framework that allows agent-based models to be executed on GPUs. It provides a high-level abstraction layer for defining agents and their behaviors using an XML-based language called XMML (X-Machine Markup Language). FLAME GPU automatically translates the XMML model specification into optimized CUDA code for execution on NVIDIA GPUs. However, using XMML results in a high learning curve for development, especially for people who are not code-focused, such as scientists.

FLAME GPU 2 [17] is a significant advancement over its predecessor, FLAME GPU. It provides a more flexible and efficient framework for building complex agent-based simulations on GPUs. FLAME GPU 2 introduces a new agent-based modeling approach that allows users to define agent behavior using the Agent API. This API enables the creation of more sophisticated and dynamic agent-based models by providing a set of functions and data structures for defining agent properties, behaviors, and interactions. But the same tradeoff as its predecessor occurs, and the message-based communication within the simulation environment introduces some restrictions.

Another notable platform in the realm of agent-based modeling is TurtleKit [18], a simulation platform that combines a Logo-based simulation model with high-level programming languages. TurtleKit is built on top of the MadKit multi-agent platform and provides a simple yet powerful environment for developing agent-based simulations. While TurtleKit has extended its capabilities to utilize GPUs [19], the lack of clear documentation and guidance on GPU usage has hindered its adoption.

As of the writing of this report, the latest resource available about FLAME GPU 2 from 2021, in a showcase video [20], the FLAME GPU 2 developers clearly stated that the library can only be run on a single GPU due to technical constraints such as data exchange between different GPUs. The same constraints apply to the MASS CUDA library. In the current implementation and design, device memory space for all data needed during the simulation must be allocated when initializing. To achieve data exchange between two GPUs, a larger memory allocation is required on each device, contradicting the goal of scaling to multiple GPUs to utilize more memory space for larger simulations.

3.3 Our Works

In contrast, we present MASS CUDA, which offers a different approach to constructing simulations. Instead of requiring users to describe agent behavior using a specific API, our library allows developers to create classes as they would in traditional object-oriented programming. Users can define agent classes and assign functions to them, which encapsulate the agent's behavior. MASS CUDA then wraps the agent class and automates the parallel execution on the GPU, abstracting away the complexities of GPU programming. We believe this approach is much easier to develop and understand, especially for developers who may not have domain-specific knowledge of GPU programming. By providing a more intuitive and familiar programming model, our library lowers the barrier to entry for researchers and practitioners who want to leverage the power of GPUs for their ABM simulations without the need to become experts in GPU programming.

One of the key features we have introduced into MASS CUDA is the separation of execution instances (Agents) and data storage (Places). In our library, a Place object serves as an attribute storage in the simulation, representing spatial elements in the real world that contain large amounts of data but don't move during the simulation. On the other hand, an Agent object carries minimal data but encapsulates the algorithms and functions necessary for execution. We believe this separation of concerns makes it easier for users to identify, maintain, and implement objects in their simulations.

Moreover, we developed comprehensive and up-to-date documentation to facilitate understanding, development, and usage of MASS CUDA.

Chapter 4. IMPLEMENTATION OF MASS ENHANCED FEATURES

In our work on MASS CUDA, we focused on two main aspects: enhancing the programmability of the library and improving its performance. We began by addressing the functionality and bug fixes related to the Agent class, which is a crucial component of the MASS CUDA library. Our efforts included implementing essential features such as Agent spawning, termination, and migration.

Once we had a fully functional version of MASS CUDA, we conducted performance analysis using the NVIDIA Nsight Compute profiling tool [21]. Through this analysis and comparison with our main competitor, FLAME GPU 2, we identified significant performance gaps between the two libraries. Consequently, the second part of our work involved analyzing the MASS CUDA library, identifying performance bottlenecks in the algorithms and data structures, and rewriting the relevant sections. In this part, we focused on addressing the memory access pattern issues caused by the data structures, which were the root cause of the performance discrepancies.

4.1 PROGRAMMABILITY ENHANCEMENT

Agents in the MASS library serve as the execution instances, carrying minimal data and capable of moving between Places. They are characterized by two essential functions: Agents::callAll()

and Agents::manageAll(). These functions play a vital role in enabling parallel execution and efficient management of Agents within the simulation.

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

Moreover, each Agent in the MASS library is equipped with three primary functions: migrate, spawn, and kill. The Agent::migrate() function enables an Agent to move from its current Place to another, facilitating dynamic interactions within the virtual environment. The Agent::spawn() function allows an Agent to create offspring either at its current location or at a different Place (the first generation of Agent objects must be created by user via our API), contributing to the complexity of the simulation. The Agent::kill() function is used to terminate an Agent, freeing up space for new Agents and excluding the terminated Agents from future parallel operations.

To illustrate these functions more clearly, let's consider a simulation of ants living in a specific environment. The environment would be represented by the Place objects, while the ants would be the Agent objects. In each iteration, ants can move to another location in the environment (using Agent::migrate()), give birth to new ants (using Agent::spawn()), or die in certain cases (using Agent::kill()).

MASS CUDA enables efficient batch operations on Agent objects by leveraging these functions in conjunction with Agents::callAll(). After using any of these functions, it is necessary to call Agents::manageAll(), which instructs the MASS library to execute the actual operations as per the given instructions in parallel. The detailed functionality and application of Agents::manageAll() is explained in Section 4.1.3.

4.1.1 *Agent Spawn*

The Agent::spawn() function enables Agent objects to spawn new Agent instances on any Place objects. When a MASS simulation is created, agents are initialized in GPU memory by allocating an array of Agent objects to hold the specified number of agents. During the simulation, users may terminate or spawn new agents, which involves adding or removing Agent instances from this allocated array. The previous implementation tracked the number of alive agents to determine where to place new agents: when a new agent was spawned, the count of alive agents increased, and the new agent was placed into the array based on this count.

However, this approach had a significant bug, as illustrated in Figure 4.1. Initially, the agent array is populated with 5 live agents and has a maximum capacity of 10. When agent 2 is terminated, reducing the count of alive Agents to 4, the algorithm determines that the next new Agent should be placed at index 4. However, an alive agent already occupies this index, revealing a significant bug (Figure 4.1).

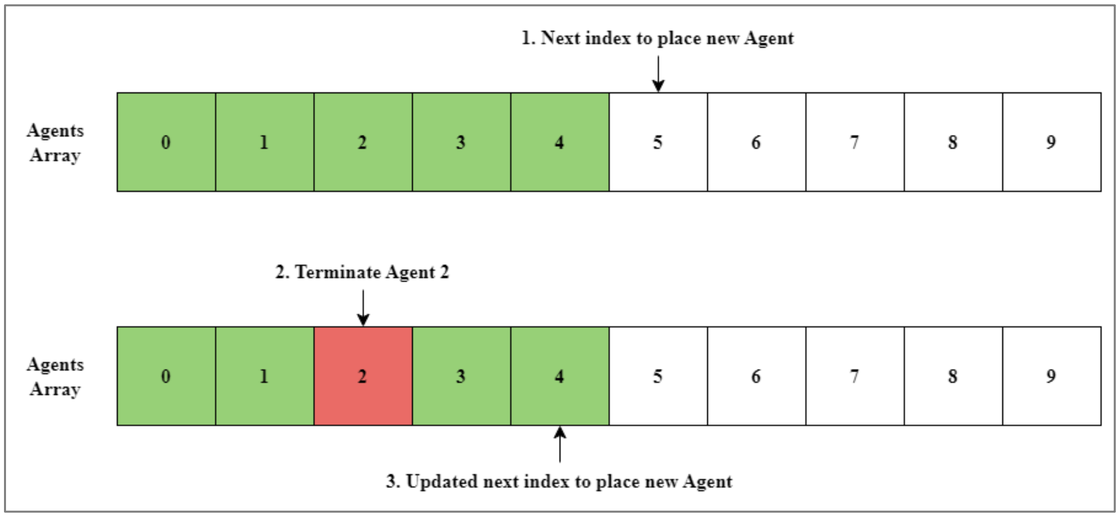
1. Next index to place new Agent
Agents Array
0
1
2
3
4
5
6
7
8
9
2. Terminate Agent 2
Agents Array
0
1
2
3
4
5
6
7
8
9
3. Updated next index to place new Agent

Figure 4.1: Using alive agents count to place new agent, which revealing a significant bug

After discussing this issue with Dr. Fukuda, he pointed out that the original design for managing the agent array involved sorting the array using CUDPP's radix sort after each case of agent termination [22]. This sorting would position alive agents at the beginning and dead agents at the end of the array, while simultaneously counting the number of alive agents (as shown in Figure 4.2). New spawn agents would then be placed at the index indicated by the counter. The existing implementation had not fully realized this original design.

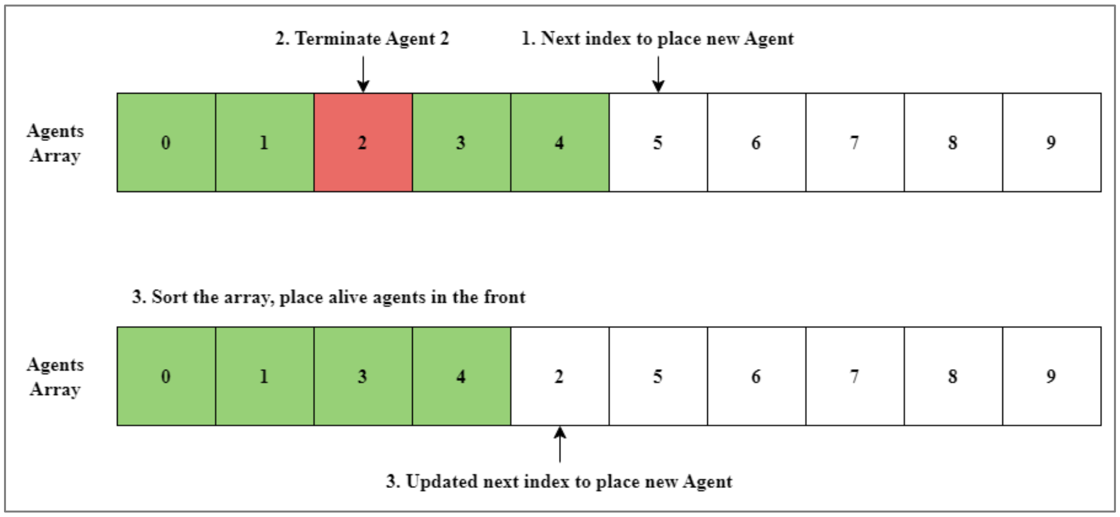
2. Terminate Agent 2
1. Next index to place new Agent
Agents Array
0
1
2
3
4
5
6
7
8
9
3. Sort the array, place alive agents in the front
Agents Array
0
1
3
4
2
5
6
7
8
9
3. Updated next index to place new Agent

Figure 4.2: Original design for managing the agent array: sorting the array using CUDPP's radix sort to place alive agents in the front and dead agents at the end of the array

However, implementing this sorting algorithm within the existing architecture of the MASS CUDA library led to more problems than solutions. To understand these issues, we need to first examine the data structure of the previous MASS CUDA Agent and Place objects.

In the previous MASS CUDA implementation [23], four arrays were used to hold Place and Agent objects (Figure 4.3). The Places array stores Place objects, which contain minimal essential functions. The related data of a Place object is instead held in the PlaceState object, connected to the Place object via a pointer. Another PlaceState array is allocated in GPU memory, having the same length as the Places array due to their one-to-one mapping. Similarly, the agent array consists of Agent objects that encapsulate the necessary functions for the agents' actions within the simulation, while the AgentState array holds the essential data for each Agent object.

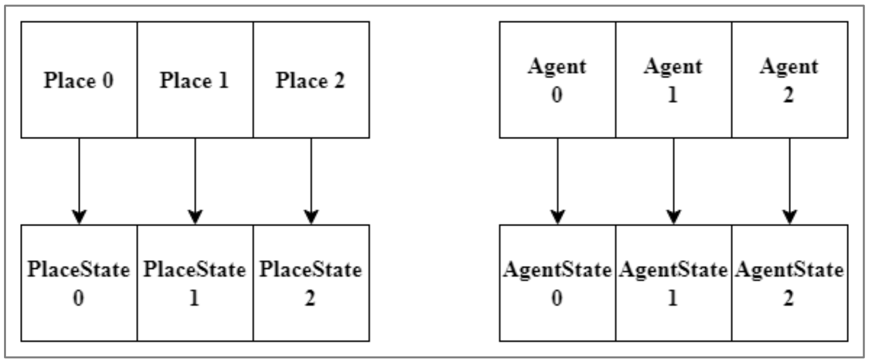

Figure 4.3: Four arrays to hold Place and Agent objects

To illustrate the relationship between agents and places, let's consider an example where we have 3 agents and 3 places, with Agent 0, 1, and 2 assigned to Place 0, 1, and 2, respectively. Since the data of a Place/Agent object is stored in a PlaceState/AgentState object, a pointer in the AgentState points to the Place to indicate that an agent resides on a place. Conversely, a pointer in the PlaceState points to the Agent to indicate that the agent resides on it, forming a double link between them (Figure 4.4).

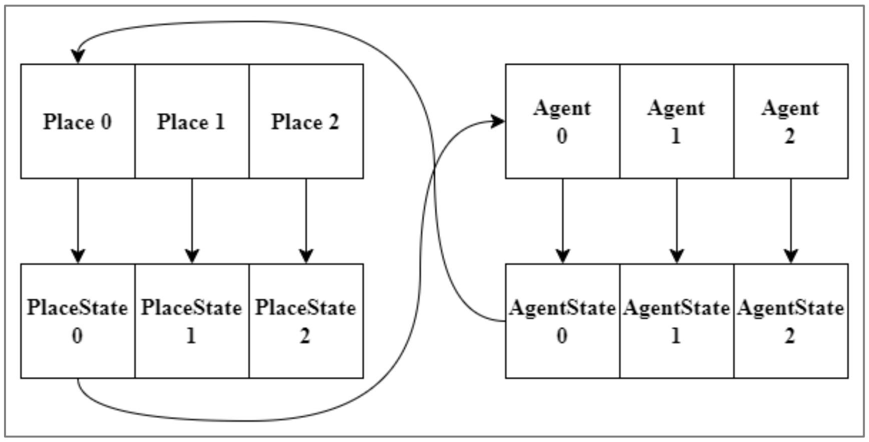

Figure 4.4: Linked Agent and Place objects. Although Place 1 is linked to Agent 1 and Place 2 is linked to Agent 2. For simplicity, only the link between Place 0 and Agent 0 is shown

If we decide to terminate agent 0, according to the previously explained algorithm, we first break the links (pointers) in both the AgentState and PlaceState objects and then sort the Agent array to move Agent 0 to the end. As a result, the order in the agent array would be Agent 1 (alive), Agent 2 (alive), and Agent 0 (dead). However, since pointers reference addresses and not the objects themselves, after the AgentState objects in the AgentState array are switched, the Place 2 still points to its original index (3rd index) in the Agent array, but the data at each index may have been switched (Figure 4.5).

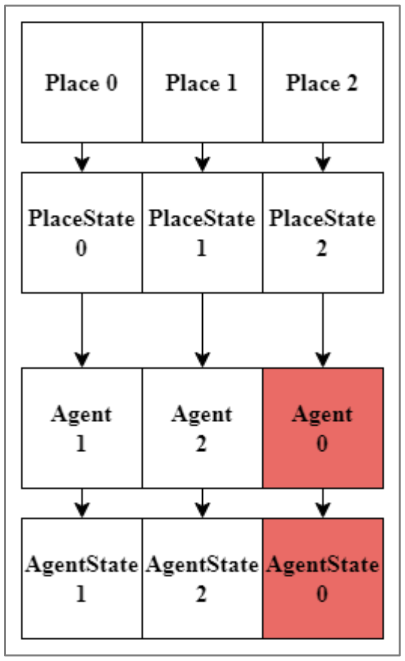

Figure 4.5: Terminate Agent 0, and then move the Agent 0 & AgentState 0 to the end of the array. But Place 2 is still pointing to the 3rd index of the Agent array (it originally pointed to Agent 2)

Several solutions were considered to address this problem, such as sorting all four arrays together or re-linking the places with the correct agents after any movements. However, neither solution was deemed acceptable. Sorting all four arrays would require extra resources and cannot be done in constant time. Additionally, if we change the pointer as soon as an agent is moved, there is an overhead of relinking the Place and Agent objects, which causes a small latency. When moving a large number of agents at a time, the cumulative effect of these small latencies can create a noticeable overall latency. To mitigate this issue, the MASS library employs a two-stage operation, which will be explained in Section 4.1.3. The first stage involves planning the operation, while the second stage commits the plan in parallel, minimizing the latency associated with relinking Place and Agent objects.

Another potential solution involved using a queue to maintain the available indices in the agent array, allowing terminated agents to enqueue their index to show the availability, and newly spawned agents to dequeue values from the queue to find an index to place themselves. However, implementing queue-like data structures and queue operations, such as dequeue and enqueue, on a GPU is extremely difficult and can significantly impact performance. GPUs are not designed like CPUs, which are better suited to these complex data structures and operations.

Therefore, we proposed an alternative solution that eliminates the sorting step altogether. In this new approach, we no longer sort the agent array to position available agents at the beginning and newly spawned agents after them. Instead, we keep all agents in place after some agents are killed and simply find the available index in the array to place newly spawned agents. By leveraging the parallelism capability of the GPU, we can avoid scanning the entire array to find available spots, resulting in an O(N) algorithm as described next.

As illustrated in Figure 4.6, we first create an integer array with the same length as the agent array, initialized with -1s. Then, all inactive agent locations report their availability to this array: if the location is available, the corresponding index in the integer array is updated with its index. After the reporting step, we have an integer array containing either -1 or the available index of the agent array where new agents can be spawned. We then remove all -1s using a reduction operation, leaving only the available indexes.

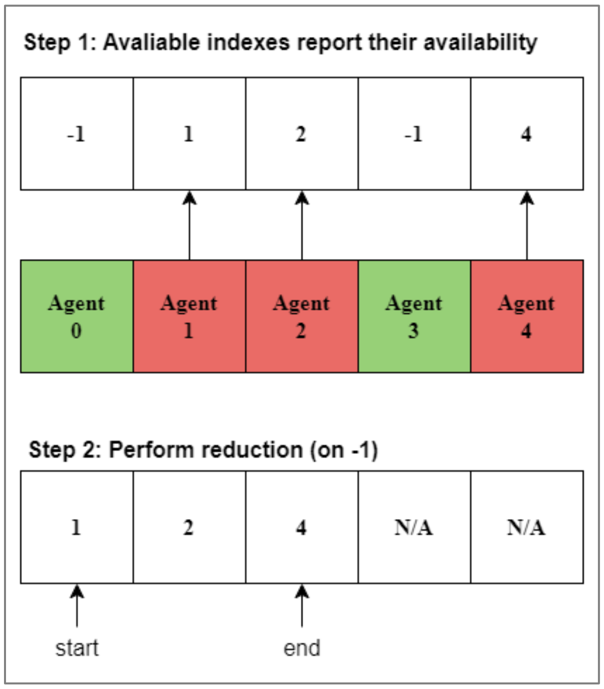
Step 1: Avaliable indexes report their availability
-1
1
2
-1
4
Agent 0
Agent 1
Agent 2
Agent 3
Agent 4
Step 2: Perform reduction (on -1)
1
2
4
N/A
N/A
start
end

Figure 4.6: Agent array reports its available index to the integer array, then use CUDA Thrust Library to perform the reduction, and get the start and end index

This method allows for the direct placement of new Agents in the available indexes. The reporting step is executed as a CUDA kernel function, enabling all agents to perform the reporting concurrently, making it an O(1) operation. Additionally, the CUDA Thrust Library [24], which provides a well-developed reduction function using CUDA, is employed for the reduction process, rendering the removal of all -1s an O(N) operation as well. Moreover, since we directly obtain the available indexes in the agent array and no longer switch any indexes, the pointers in the Agent/Place objects remain pointing to their original locations, maintaining the integrity of the object relationships and avoiding the need for additional pointer updates.

4.1.2 *Agent Migrate*

The Agent::migrate() function enables agents to move from one place to another within the MASS CUDA simulation. When a user intends to migrate an agent, a series of operations are executed to facilitate this movement.

Firstly, in the Agent::migrate() function, the agent does not start the migration immediately. Instead, it is added to the destination place's potentialAgents array, indicating its intention to migrate to that specific place.

Subsequently, in the Agents::manageAll() function, the library resolves any conflicts that may arise when multiple agents express their intention to migrate to the same place. Since each place has a maximum capacity for agents, the current algorithm selects the agents with the lowest N index for migration, where N represents the maximum number of agents a place can accommodate. For example, if agents 1, 2, and 3 all wish to migrate to place 1, but the MASS library is configured to allow a maximum of 2 agents per place, one of the agents is not migrated and remains at its original location. In this scenario, agents 1 and 2 successfully migrate to place 1, while agent 3 stays at its original place. After resolving these conflicts, a kernel function is executed to migrate all eligible agents to their desired destinations in parallel.

However, in the previous implementation, we identified a significant bug when adding agents to the place's potentialAgents array. The addMigrationAgent function, used in Agent::migrate(), was responsible for expressing an agent's interest in migrating to a specific place. This process involved iterating through the potentialAgents array of the place and placing the agent in the first empty location found, as shown in the algorithm (Figure 4.7):

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

```

Figure 4.7: Previous algorithm to add agents to a place's potentialAgents array to express the intention to move to this place

While this function performed well when executed sequentially, it posed a challenge when implemented using CUDA, as it would be executed in parallel by multiple agents (each agent is managed by a thread). Therefore, when several agents were interested in migrating to the same place, they would attempt to add themselves to the potentialAgents array simultaneously, leading to a race condition: they might find the same empty location and update the same index concurrently, causing inconsistencies.

To address this race condition, we introduced an additional local variable to the Place object called numPotentialAgents. This variable keeps track of the number of agents that have been added to the potentialAgents array and serves as the index for adding the next agent. By utilizing the atomicAdd() function provided by CUDA, we ensure that this variable is thread-safe, allowing only one agent to modify it at a time. The updated code is as follows (Figure 4.8):

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

```

Figure 4.8: A new algorithm to add agents to a place's potentialAgents array to express the intention to move to this place, utilizing a new variable with the atomic function

4.1.3 *Agent Kill*

Before explaining the last core function of Agent, Agent::kill(), it is crucial to introduce the Agents::manageAll() function to better understand how these core functions work together.

In sections 4.1.1 and 4.1.2, we illustrated the functionality of Agent::spawn() and Agent::migration(). We also mentioned the Agents::callAll() function, which is similar to Places::callAll(). Functions and arguments can be passed to Agents::callAll(), and MASS CUDA manages the parallel execution on all Agent or Place objects. It is important to distinguish between Agent functions and Agents functions: Agent functions are executed on a single Agent object, while Agents functions serve as the API for users to manipulate all Agent objects in parallel.

To utilize the Agent::spawn() and Agent::migrate() functions, we first need to create an Agent function that contains these functions. Then, we pass this customized function to Agents::callAll() to perform the manipulation on all Agents simultaneously. Let's revisit the example of ants living in an environment. In each iteration, we want the ants to have the possibility of moving to a new location. For instance, in the customized function "Agent::move()", we generate a random integer. If it's odd, the ant moves upwards; if it's even, the ant moves downwards. By passing the "Agent::move()" function to Agents::callAll(), all ants in the simulation call the function concurrently, deciding on a direction to move.

However, it's important to note that the ants do not actually move to a new place immediately after calling "Agent::move()". They have only "decided" on the new place to move to. As mentioned in section 4.1.2, we need to resolve any conflicts that may arise when several ants want to move to the same location. Additionally, we need to perform the actual movement of all ants simultaneously (executed in parallel). To achieve this, we introduce the Agents::manageAll() function, which handles the actual execution of these actions.

To better distinguish between the "expressing intention" functions (Agent::kill(), Agent::spawn(), and Agent::migrate()) and the "actual action" function (Agents::manageAll()), we define that the Agent::kill(), Agent::spawn(), and Agent::migrate() functions set pre-operation instructions for the agents, while Agents::manageAll() reads these instructions and performs the actual operations (Figure 4.9).

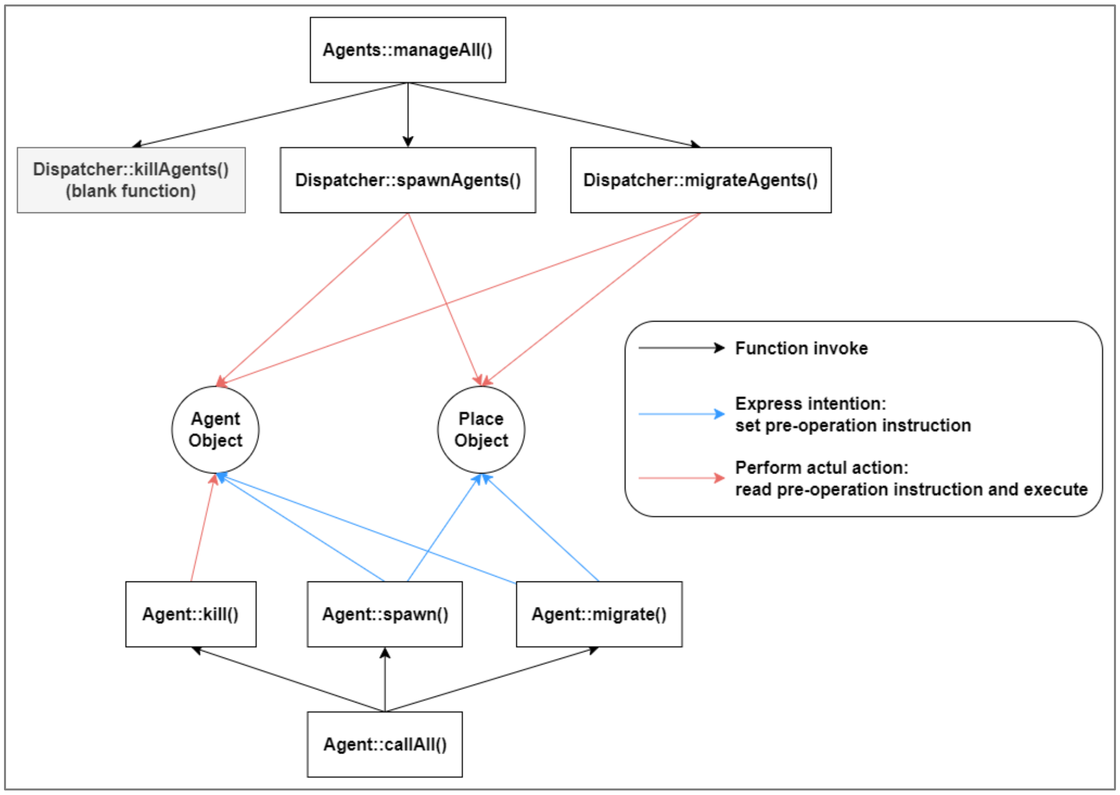
Agents::manageAll()
Dispatcher::killAgents() (blank function)
Dispatcher::spawnAgents()
Dispatcher::migrateAgents()
Function invoke
Express intention: set pre-operation instruction
Perform actul action: read pre-operation instruction and execute
Agent Object
Place Object
Agent::kill()
Agent::spawn()
Agent::migrate()
Agent::callAll()

Figure 4.9: Agent functions set pre-operation instructions and Agents::manageAll() read instructions and perform the actual actions

For example, when Agent::spawn() is used, agents set the pre-operation instruction: birth N agents at place X. Then, in the subsequent Agents::manageAll() function, N agents are generated and placed at place X. Similarly, Agent::migrate() sets the pre-operation instruction: move this agent to place X. In the following Agents::manageAll () function, the agent is moved to place X. Therefore, an Agents::manageAll() function call is necessary after each time any of Agent::kill(), Agent::spawn(), or Agent::migrate() is used (one of these functions per time is allowed, such as one kill, one migrate, and one spawn, followed by an Agents::manageAll()).

However, Agent::kill() is slightly different from the other two functions. While Agent::spawn() and Agent::migrate() set pre-operation instructions because many attributes of the Agent and Place objects need to be changed to execute the operation, terminating an agent is a simpler task. The "alive" attribute of an agent indicates whether it is alive or not. To terminate an agent, we simply need to change the "alive" attribute to False, which is a single step and can be finished in one Agents::callAll() execution. Therefore, the Agent::kill() function is excluded from the Agents::manageAll() function and is intentionally left blank (Figure 4.9).

4.2 Performance Enhancement

After completing the implementation of MASS CUDA with fully functional agents and places, as described in section 4.1, we benchmarked the library using the Game of Life application. The Game of Life is a zero-player game, meaning that its evolution is determined solely by its initial state, requiring no further input [25]. It computes the next state of each simulation instance based on the status of the surrounding instances. Although the Game of Life does not involve sophisticated computations, it consists of a large number of computation instances that can be parallelized, making it an ideal starting point for our benchmark.

We compared the performance of MASS CUDA with FLAME GPU 2, our main competitor and another well-known GPU-based ABM library, which was also used to build the Game of Life. Surprisingly, we found that MASS CUDA's performance was 9 times slower than FLAME GPU 2 for a simulation size of 2000 instances and 20 times slower for a simulation size of 2500 instances. Moreover, the performance gap increased exponentially as the simulation size grew. The detailed comparison can be found in Section 5. This significant performance difference caught our attention, and we further investigated using NVIDIA Nsight Compute, the official benchmark application, to gather more details about the performance bottlenecks. We noticed that the primary issue was related to the memory access pattern, and Nsight Compute suggested that using coalesced memory access could potentially boost the performance by 98.19%.

Next, we discuss how the data structure used for Agent and Place objects in the previous implementation of MASS CUDA contributed to the uncoalesced memory access issue. We also present our approach to address this problem and the updates made to MASS CUDA after this revision.

4.2.1 *Transition from AOS to SOA*

In the previous implementation of MASS CUDA [23], the data for both Agent and Place objects (referred to as "object" in the following context) was stored in separate state objects (referred to as "state" in the following context), named AgentState and PlaceState. The object itself was connected to its corresponding state via a pointer (Figure 4.3). The minimum size of a state was 280 bytes due to the minimum required data to support the MASS CUDA simulation.

This initial data structure setup of the MASS library is known as Array of Structures (AOS) because we have an array of state objects. Due to the nature of MASS CUDA, we utilize the GPU to execute parallel manipulations on the data stored in the state. In this scenario, each time we execute a kernel function, each thread is responsible for manipulating a single state. Before performing the manipulation, each thread needs to read the specific data of its assigned state. In this context, we must consider two important aspects of data access (memory access): spatial locality and prefetching, and coalesced memory access.

When CPUs and GPUs look for data that is not in their immediate cache (such as L1/L2 cache on CPU and local memory on GPU), they search for the data in "further" memory, such as RAM in the context of CPUs and global memory on GPUs. Once the required data is located, it is transferred to the immediate cache.

However, when a single data element is needed and moved, CPU and GPU architectures preemptively fetch a contiguous block of data around the requested index. This approach, based on the principle of spatial locality, assumes that data near a recently accessed item is soon needed. For example, if we have an array of length 10 and the program reads array[0], array[0] is transferred to the cache, and the data of the first 5 indexes. This spatial locality and prefetching strategy of hardware significantly reduces the need for costly "further" memory accesses.

Another unique optimization within GPUs is coalesced memory access. This technique specifically optimizes how threads within a warp access global memory. When threads in a warp simultaneously request data from sequential or suitably aligned memory addresses, as shown in Figure 4.10, the GPU is able to combine these requests into a single memory transaction. This consolidation significantly decreases the total number of memory transactions required, thereby accelerating the process of memory access. Using the previous example, when 10 threads access each index of the array simultaneously, the GPU can combine these 10 requests, and the data of all 10 indexes is transferred at once.

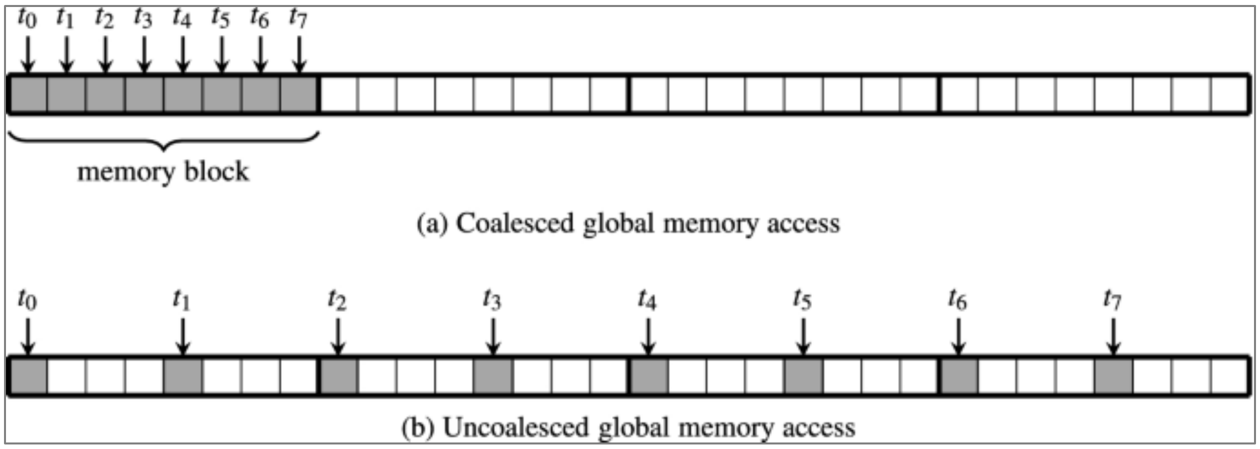
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7
memory block
(a) Coalesced global memory access
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7
(b) Uncoalesced global memory access

Figure 4.10: Coalesced memory access and Uncoalesced memory access [26]

We discovered that the minimum size of a state, which was 280 bytes, significantly hindered the utilization of these two hardware optimizations. When each thread on the GPU accessed the same single piece of data within each state, although a continuous block of data was transferred, due to the large size of a single state, the prefetching strategy could only transfer a few states at once. Moreover, the single piece of data each thread accessed was not sequential in memory. For example, if we were accessing the int index attribute of the state, since the size of the state was 280 bytes, the index attribute of the first state and the second state was 280 bytes apart in memory, which prevented the use of coalesced memory access (Figure 4.11).

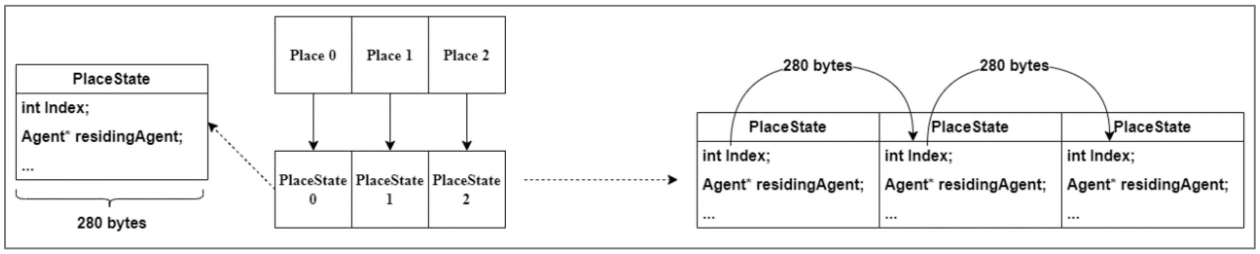

Figure 4.11: An array of PlaceState objects, the data structure of the object, and an attribute of each object is 280 bytes apart in the memory

To solve this problem, we focused on making all attributes of the state contiguous in memory. Therefore, we changed the data structure from Array of Structures (AOS) to Structure of Arrays (SOA). We “flattened” the state such that if the state had three attributes, instead of having an array of the state, we used three arrays, each containing only one attribute of each Agent/Place object (Figure 4.12).

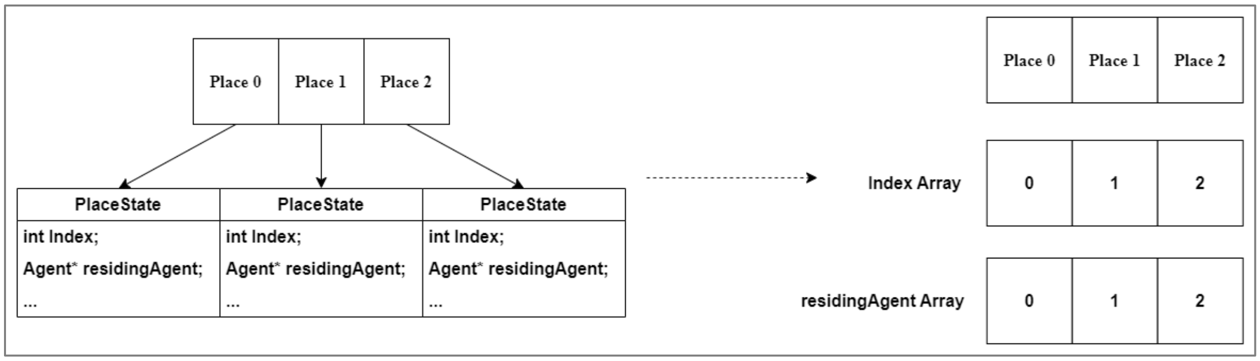

Figure 4.12: “Flatten” the state, so that the data structure is changed from AOS to SOA, and no extra pointer is needed

Due to the nature of MASS CUDA, which involves batch processing the same instructions on all Agent/Place objects (SIMD), if an Agent is manipulating attribute A, all Agents are manipulating attribute A, meaning all objects are accessing the same attribute. After this change, since the same attribute is stored in the same array, we can fully utilize the hardware optimizations mentioned earlier: prefetching and coalesced memory access. Since every object has the same attributes, the length of the array for an attribute is the same as the Agent/Place objects array. Therefore, we can fetch the attribute by the index of the Agent/Place objects in their array without the need for an extra pointer.

4.2.2 *Dynamic Attribute Setting Solution*

The transitioning from the Array of Structures (AOS) to the Structure of Arrays (SOA) in the MASS library eliminated the PlaceState and AgentState classes. This change posed a challenge for users who were accustomed to customizing their own PlaceState or AgentState. Previously, users could inherit the MASS state object base class and add customized attributes. Now, since we no longer have the state objects, we proposed a new way for users to add and use attributes dynamically during the simulation.

The solution involves introducing three arrays in Agent/Place: an attributeTags array for storing attribute names, an attributeDevPtrs array for holding pointers to the attribute arrays on the device, and an attributePitches array for tracking the pitch size when allocating the attribute arrays on the device, used to correctly access the attribute array internally by MASS CUDA.

By utilizing these three new arrays, users can create and use attributes. When creating an attribute, users invoke the setAttribute() function, specifying the attribute's tag, length, and an optional default value. The MASS middleware (MASS.Dispatcher) then allocates memory on the device for the attribute array and updates attributeDevPtrs, attributePitches, and attributeTags accordingly (Figure 4.14). Here's an illustrative pseudo-code (Figure 4.13):

```
setAttribute(attributeTag, attributeLength, defaultValue):
    if (attributeTag exists in attributeTags):
        return false
    else:
        ptr, pitch = allocate memory on the device
        if (allocate success):
            attributeTags.push_back(attributeTag)
            attributeDevPtrs.push_back(ptr)
            attributePitches.push_back(pitch)

            if (defaultValue):
                set default value (defaultValue)
            return true
        return false

```

Figure 4.13: Algorithm of creating attributes on the device for Place/Agent objects

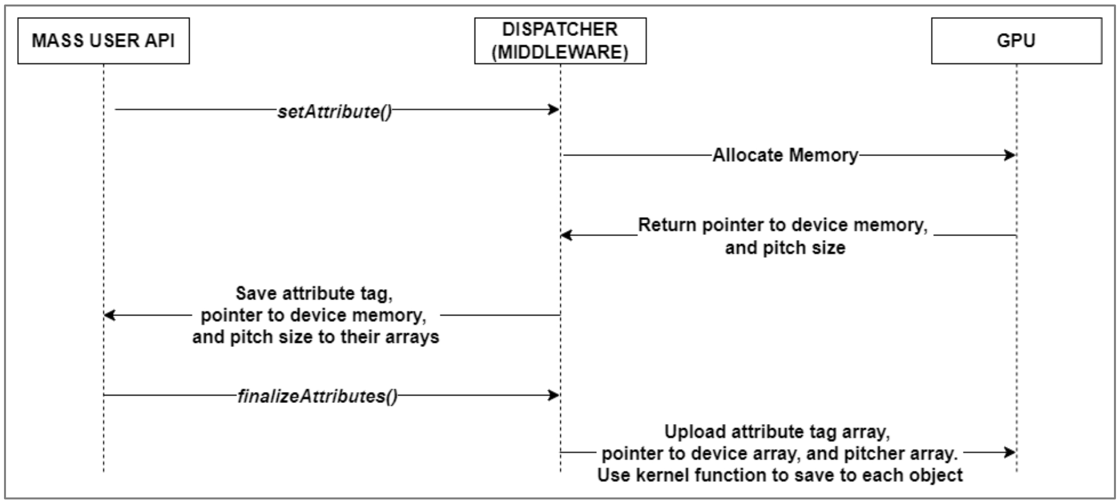
MASS USER API
DISPATCHER (MIDDLEWARE)
GPU
setAttribute()
Allocate Memory
Return pointer to device memory, and pitch size
Save attribute tag, pointer to device memory, and pitch size to their arrays
finalizeAttributes()
Upload attribute tag array, pointer to device array, and pitcher array. Use kernel function to save to each object

Figure 4.14: Algorithm of creating attributes on the device for Place/Agent objects within the MASS CUDA library

However, by using the setAttribute() function, although the memory is allocated on the device, all the information is currently saved in the host API store. We need to update this information to the Agent/Place objects on the device so that they can use the attribute in the kernel function executing on the GPU. Before utilizing an attribute, users must execute finalizeAttributes(), prompting MASS to distribute these array data across all Agent/Place objects on the device (Figure 4.14). Subsequently, within kernel functions, retrieving an attribute is simplified through getAttribute(), enabling threads to access relevant attributes efficiently (Figure 4.15). The adapted data structure ensures coalesced memory access, significantly enhancing overall performance, which is introduced in Section 5.

```
getAttribute(attributeTag, attributeLength):
    if (attributeTag not in attributeTags):
        raise an error
    else:
        index = get the index of attributeTag in attributeTags
        attributeDevicePointer = attributeDevPtrs[index]
        attributeValue = attributeDevicePointer[object index]
    return attributeValue

```

Figure 4.15: Algorithm for getAttribute() function, which is used on the device inside Agent/Place functions to retrieve attributes

This dynamic attribute setting mechanism restores the flexibility of the original implementation and aligns with the optimized SOA data structure, ensuring efficient memory access and computation. From the user's perspective, they are still able to dynamically add customized attributes to objects as before. However, instead of inheriting the base class and adding variables to the class, they can now manage the attributes in the main program during the simulation without the need to inherit the base class. This change simplifies the process of extending object attributes for users, as they no longer need to create and manage separate derived classes.

It's important to note that while the section describing the dynamic attribute setting mechanism may appear relatively short, the actual implementation of this feature was highly complex and required a significant portion of the MASS CUDA library to be rewritten. The transition from AOS to SOA and the introduction of the dynamic attribute setting mechanism necessitated extensive modifications to the library's core architecture, data structures, and memory management systems. This effort involved careful design, implementation, and testing to ensure the correctness and efficiency of the new approach.

From the MASS developer's point of view, this transition increased the need for more complex data and memory management. The dynamic attribute setting mechanism requires careful handling of memory allocation, deallocation, and synchronization between the host and device memory. Developers must ensure that the attributeTags, attributeDevPtrs, and attributePitches arrays are properly maintained and updated to reflect the current state of the attributes. Additionally, the use of separate arrays for each attribute and the distribution of arrays to every Agent/Place object may result in increased memory usage compared to the previous AOS approach, especially when dealing with a large number of attributes or objects.

Chapter 5. EVALUATION

We have benchmarked MASS CUDA on a Linux server equipped with two NVIDIA A5000 GPUs. However, since our library currently supports only one GPU card, one GPU remained idle during the simulation benchmark. The server is powered by an AMD EPYC 7232P 8-Core Processor, 128GB RAM, and runs on Linux version 4.18.0-425.3.1.el8.x86_64.

For each benchmark, we employ two types of timers: one for the CPU clock time and one for the GPU clock time. The CPU clock time reflects the total program execution time, while the GPU clock time represents the time the GPU is actively involved during the simulation, as measured by the timer provided by CUDA for capturing Cuda Event time [27]. As a result, each benchmark program provides three key metrics: (i) The total time, which is the total CPU time of the program from initialization to shut down; (ii) The initialization time, which is the total CPU time taken to initialize the library for starting the simulation; (iii) The simulation per-step time, which is the GPU time taken for calculating each step. The total time and initialization time are measured using the CPU time since it records the total real-world time passed for the simulation and initialization. The per-step time is measured using the GPU clock time, as it focuses on the performance of the parallel execution of the kernel function.

To verify benchmark programs, as in our previous work on the MASS C++ version, we have implemented all these benchmark programs, and they were validated. Now, in MASS CUDA, we compare the results with these programs to ensure correctness and consistency. This approach helps us to confirm that the new implementation in MASS UCDA faithfully replicates the expected behaviors and outputs of the validated benchmarks.

To compare the execution performance of MASS CUDA with that of our main competitor, FLAME GPU 2, we implemented the same benchmark applications using FLAME GPU 2, employing the aligned simulation algorithm as matched as we can. All three times are recorded in the same manner for both libraries.

In this section, we use a benchmark program to compare the MASS CUDA library's performance before and after the optimization mentioned in Section 4.2, showcasing the performance improvements achieved. Subsequently, we compare the programmability and execution performance of MASS CUDA with FLAME GPU 2 using four different benchmark applications.

5.1 BENCHMARK APPLICATIONS

Each of the four benchmark applications we used are introduced, including why we selected them.

(A) Game of Life

Game of Life is a cellular simulation devised by the British mathematician John Horton Conway in 1970. In the simulation, each place represents a live cell, which can be either live or dead. In each iteration, the live cell can become dead or live based on the number of dead and live cells around it. This simulation is a zero-player game, requiring no input, and each iteration represents one evolutionary step [28].

From a computational perspective, in each iteration, we need to compute the next state of each cell. If we are simulating 1000 x 1000 cells, we have to compute 1 million cells in each iteration, making it an ideal application for parallelization and inclusion in our benchmark. This simulation involves only the Place object and neighborhood communication operations.

(B) Heat2D

Heat2D is a simulation of heat transfer in a two-dimensional space. In each iteration, the next state heat of each two-dimensional space is calculated based on the surrounding spaces' heat, simulating how heat is transferred in the two-dimensional space. It is very similar to the Game of Life but employs a different algorithm to calculate the next state. This simulation also involves only the Place object and neighborhood communication operations.

(C) Sugarscape

Sugarscape involves both the Place and Agent objects. It is a model for artificially intelligent agent-based social simulation following some or all rules presented by Joshua M. Epstein & Robert Axtell in their book *Growing Artificial Societies* [29]. In this simulation, we use Agent objects to represent ants and Place objects to represent the places containing sugar. In each iteration, agents move to a place containing sugar, consume sugar, collect sugar, and leave pollution, while places generate sugar.

From a computational perspective, in each iteration, we need to update both the place's next state and the agent's next state. Since agents represent ants and they move continuously, it is an excellent benchmark application. This simulation involves agents' migration and termination, Place object, and neighborhood communication operations.

(D) Social Net

Social Net is an application that simulates the social network among people. In each iteration, we need to find the Nth degree friends of each person in the simulation. To calculate the Nth degree friends of hundreds of thousands of people, we use ABM to perform the parallel calculation. This simulation utilizes graph structure and the place exchangeAll() function.

5.2 MASS CUDA Enhancements After Data Structure Revising

We use the Game of Life to benchmark the performance of MASS CUDA before and after the data structure revision mentioned in section 4.2. The detailed data are listed in the tables in Appendix A.

This simulation only involves places' operations. We run the application in 5 sizes for 250 iterations: 500 x 500 places, 1000 x 1000 places, 1500 x 1500 places, 2000 x 2000 places, and 2500 x 2500 places, and compare the total simulation time (CPU time) in Figure 5.1, simulation initialization time (CPU time) in Figure 5.2, and per-step time (GPU time) in Figure 5.3.

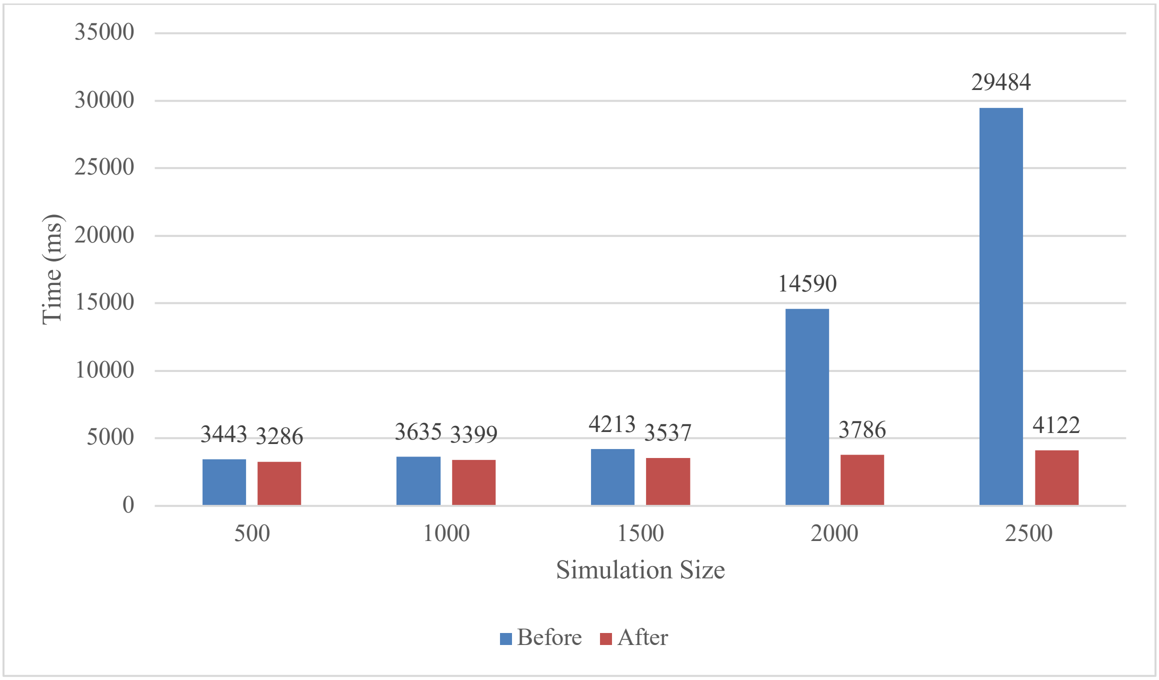

Figure 5.1: Game of Life total simulation time before and after data structure revising

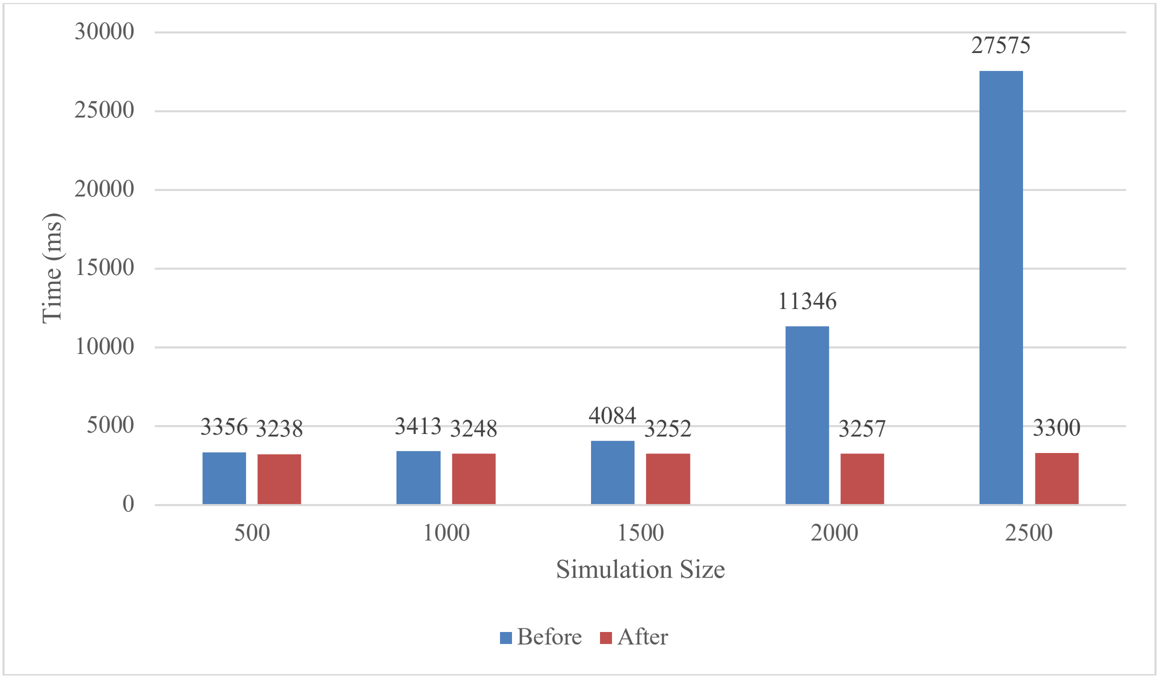

Figure 5.2: Game of Life initialization time before and after data structure revising

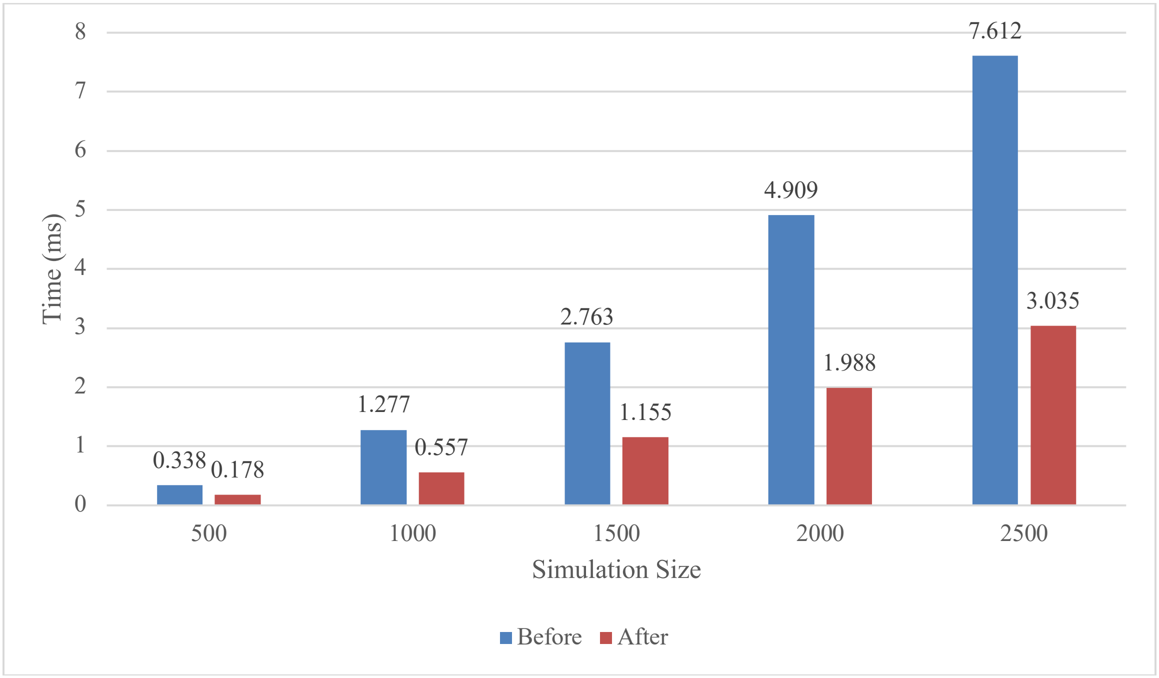

Figure 5.3: Game of Life total per-step time before and after data structure revising

The performance enhancement achieved through the data structure revision is remarkable. The initialization time reveals that the simulation's initialization is no longer influenced by the size of the simulation, a result of the asynchronous execution design of the data allocation on the device. This improvement will be further highlighted in the subsequent experiments comparing MASS CUDA to FLAME GPU 2. Furthermore, the per-step execution time has decreased significantly, as expected, benefiting from the coalesced memory access design. This optimization has led to a substantial reduction in the total simulation time, with the new implementation taking only 1/7 of the time required by the old one when the simulation size reaches 2500. These results clearly demonstrate the effectiveness of our data structure revision in optimizing the performance of MASS CUDA, showcasing its potential for efficiently handling large-scale agent-based simulations on GPUs.

5.3 Programmability Comparisons

We compare the programmability of MASS CUDA and FLAME GPU 2 for each benchmark application using four main metrics: the Line of Code (LoC) needed to implement the application, Cyclomatic Complexity (representing the complexity to develop or maintain a program, measured by a Python package lizard [30]), Boilerplate Code (representing the repeated code needed in the implementation), and Boilerplate Percentage (representing the percentage of repeated code in the total lines of code in the implementation).

(A) Game of Life

Table 5.1: Programmability Comparison of Game of Life

Framework	LoC	Cyclomatic Complexity	Boilerplate Code LoC	Boilerplate Code %
MASS CUDA	171	3.00	8	4.678%
FLAME GPU 2	122	5.67	24	19.672

(B) Heat2D

Table 5.2: Programmability Comparison of Heat2D

Framework	LoC	Cyclomatic Complexity	Boilerplate Code LoC	Boilerplate Code %
MASS CUDA	214	3.64	20	9.346%
FLAME GPU 2	252	8.60	77	30.556%

(C) Sugarscape

Table 5.3: Programmability Comparison of Sugarscape

Framework	LoC	Cyclomatic Complexity	Boilerplate Code LoC	Boilerplate Code %
MASS CUDA	441	3.37	27	6.122%
FLAME GPU 2	353	6.25	103	29.178%

(D) Social Net

Table 5.4: Programmability Comparison of Social Net

Framework	LoC	Cyclomatic Complexity	Boilerplate Code LoC	Boilerplate Code %
MASS CUDA	424	2.47	30	7.075%
FLAME GPU 2	228	8.00	38	16.667%

Overall, the programmability comparisons between both libraries across all applications present similar patterns. The LoC required for both libraries is comparable, with MASS CUDA requiring slightly more code. This is because MASS CUDA is still in its development process, where certain functionalities are not yet implemented (such as updating attributes and generating random numbers on the device), requiring users to manually achieve some features that are built-in in FLAME GPU 2. FLAME GPU 2 consistently exhibits a higher percentage of boilerplate code, primarily due to its built-in functions for setting, reading, and updating simulation variables. As MASS CUDA has not yet implemented these functions, its boilerplate code percentage is comparatively lower. However, it is important to note that a definitive conclusion cannot be drawn until MASS CUDA incorporates these built-in functions, enabling a more equitable comparison.

Despite the absence of certain built-in functions, MASS CUDA demonstrates lower cyclomatic complexity compared to FLAME GPU 2, indicating that our library is more straightforward to develop and maintain. This advantage is expected to persist and potentially improve as additional built-in functions including updating attributes and generating random numbers on the device are implemented in MASS CUDA. The lower cyclomatic complexity can be attributed to our core design philosophy, which prioritizes an object-oriented programming approach to benefit users who are primarily focused on performing simulations rather than coding, such as scientists.

5.4 EXECUTION PERFORMANCE COMPARISONS

To compare the execution performance of MASS CUDA and FLAME GPU 2, we evaluate each benchmark application using three key metrics: total simulation time (CPU time), initialization time (CPU time), and per-step execution time (GPU time). These metrics provide a comprehensive understanding of the performance characteristics of both frameworks. However, for the Social Net benchmark application, we only measure the total simulation time. This decision is based on the specific characteristics and requirements of the Social Net simulation, where the initialization time and per-step execution time are not as relevant nor informative for performance comparison purposes. The detailed data are listed in the tables in Appendix A.

(A) Game of Life

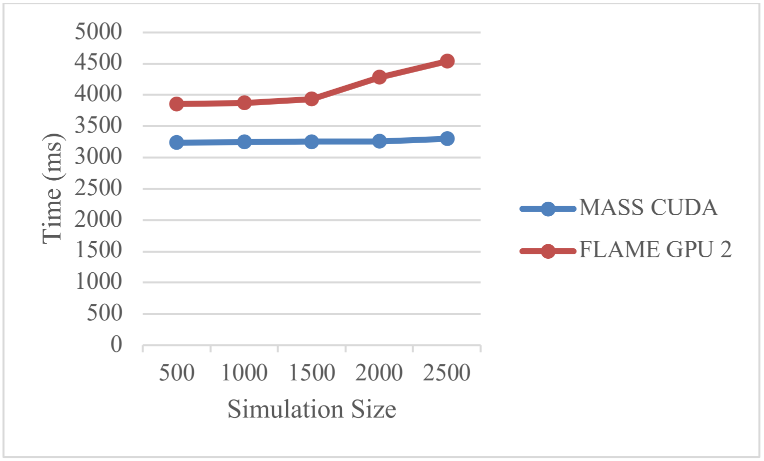

Figure 5.4: Initialization time comparison of Game of Life

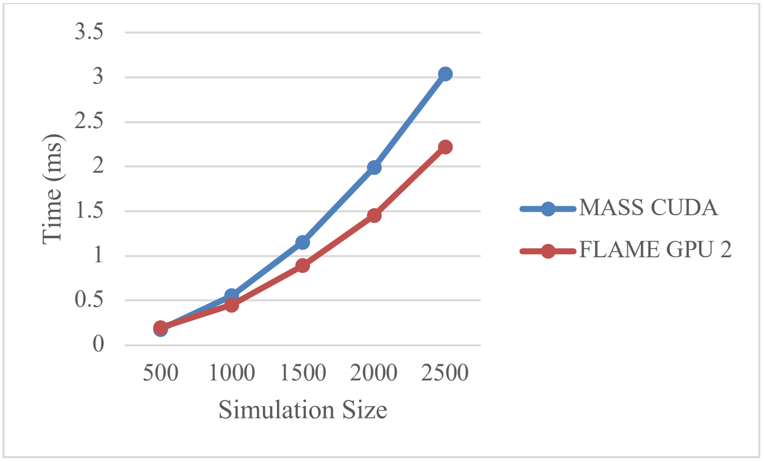

Figure 5.5: Per-step execution time comparison of Game of Life

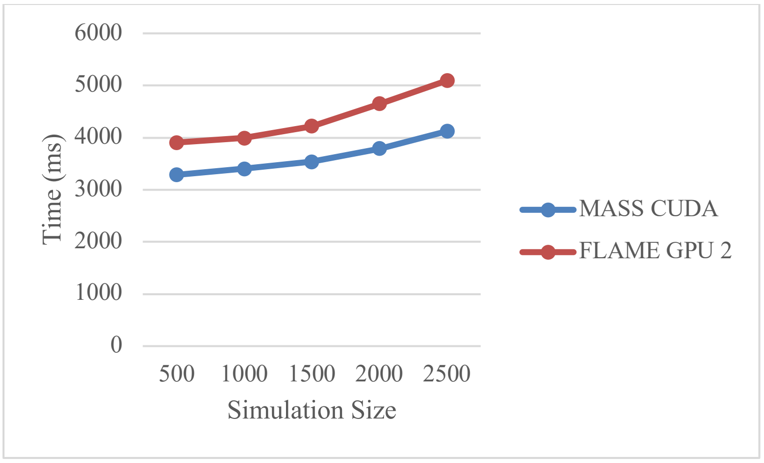

Figure 5.6: Total simulation time comparison of Game of Life

As demonstrated by the improvement in performance after the data structure revision, MASS CUDA achieves a more efficient initialization time that is not significantly influenced by the simulation size. Although MASS CUDA exhibits a better total simulation time compared to FLAME GPU 2, it is crucial to acknowledge that the per-step execution time of MASS CUDA is worse than that of FLAME GPU 2. The advantageous overall simulation time can be attributed to the specific number of iterations used in the simulation. If the number of iterations were to increase, the total simulation time of MASS CUDA could potentially surpass that of FLAME GPU 2.

Considering that the Game of Life program primarily involves place operations, and its algorithm is relatively straightforward, the performance discrepancy can be mainly attributed to the execution time required to launch the kernel function in each iteration. Further analysis reveals that MASS CUDA currently adopts a simple approach of executing the kernel functions sequentially in each iteration, while FLAME GPU 2 utilizes CUDA streams to efficiently schedule and execute tasks in queue. To bridge this performance gap, it is imperative that MASS CUDA incorporates the fundamental technique of leveraging CUDA streams in its future development, as this optimization has the potential to significantly enhance the per-step execution time and overall performance of the library.

(B) Heat2D

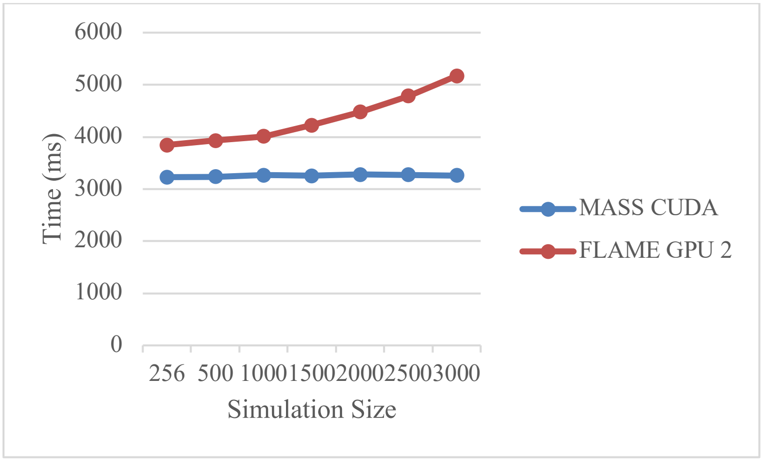

Figure 5.7: Initialization time comparison of Heat2D

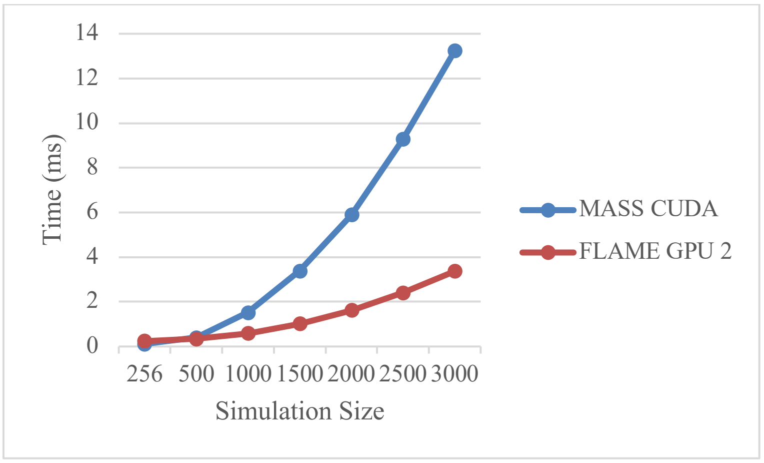

Figure 5.8: Per-step execution time comparison of Heat2D

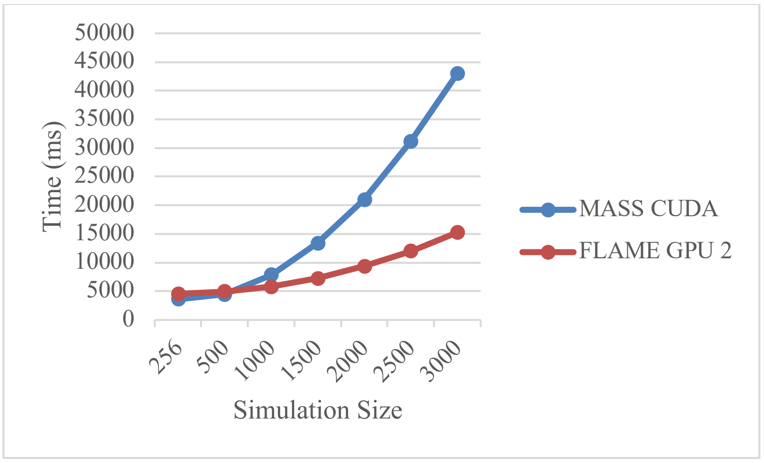

Figure 5.9: Total simulation time comparison of Heat2D

The Heat2D benchmark, while similar to the Game of Life in terms of involving only place operations, presents a more complex algorithm in each iteration. Despite this difference, the same performance issue arises: the overhead of kernel function execution. Additionally, further analysis reveals that although the data structure revision has been successfully implemented, it represents an initial step in the optimization process. To further enhance performance, it is necessary to consider other aspects of data management, such as utilizing faster memory types like constant memory or local memory on the device for more efficient data access. This optimization becomes particularly relevant in the Heat2D simulation, where frequent data reading from surrounding environments (doubling the data access required in the Game of Life, which reads four attributes from four neighbors, whereas Heat2D reads eight attributes from neighbors in each iteration) acts as a performance bottleneck. The significant amount of data access required in Heat2D highlights the need for continued refinement of the data structure and memory management strategies.

(C) Sugarscape

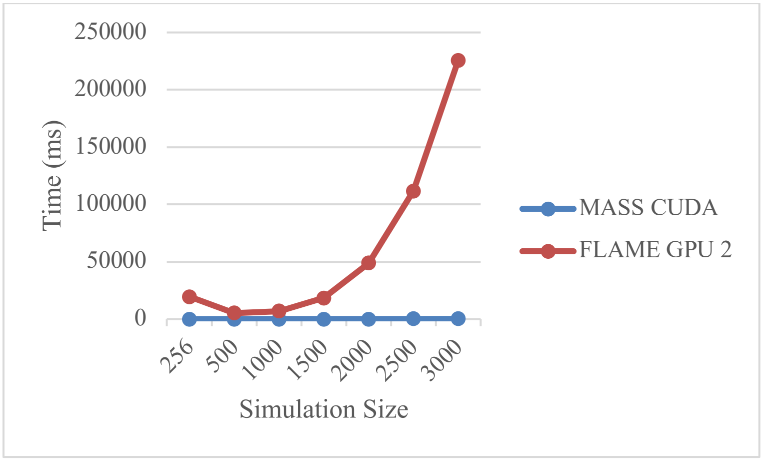

Figure 5.10: Initialization time comparison of Sugarscape

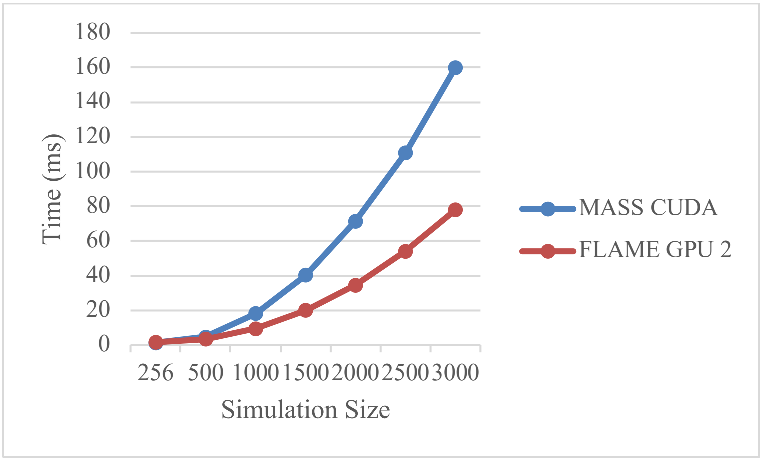

Figure 5.11: Per-step execution time comparison of Sugarscape

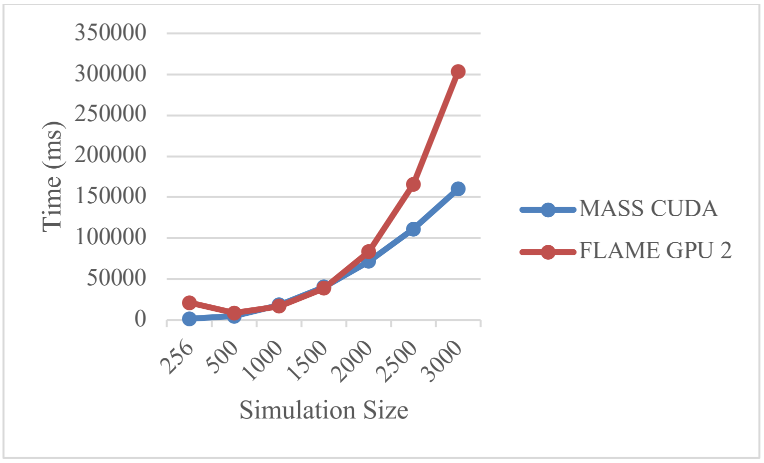

Figure 5.12: Total simulation time comparison of Sugarscape

The Sugarscape simulation stands out as the program with the highest degree of Agent/Place interactions among all the benchmark applications, as it involves both agent and place operations. In each iteration, places need to update the environment, including multiple variables, while agents migrate from one place to another.

It is important to note that the simulation initialization time of FLAME GPU 2 is exceptionally high in this simulation, especially when compared to MASS CUDA. This discrepancy can be attributed to FLAME GPU 2's approach of initializing all functions involved in the simulation at the beginning. As mentioned earlier, FLAME GPU 2 queues all tasks in advance to reduce the overhead of kernel function execution. While this approach significantly benefits its per-step execution performance, it comes at the cost of an extremely higher initialization time compared to MASS CUDA.

Apart from the kernel function execution overhead and data structure issues previously discussed, we have identified another bottleneck in the MASS CUDA simulation execution. Table 5.6 provides a detailed breakdown of the execution time for each function within a single iteration. The functions include updating the environment (Update Env), finding the next destination for agents (Find Next Dest), which are Places::callAll() functions; moving agents to another place (Move), which is an Agents::manageAll() function; and metabolizing sugars (Metabolize), which is an Agents::callAll() function.

Table 5.5: Sugarscape simulation per-step detailed execution time distribution

Size	Update Env	Find Next Dest	Move	Metabolize
256	0.096	0.4	0.907	0.021
500	0.276	1.443	2.971	0.024
1000	0.936	5.419	11.584	0.067
1500	2.042	12.086	28.795	0.168
2000	6.514	21.422	45.985	0.239
2500	5.361	32.872	71.53	0.367
3000	7.757	47.555	103.163	0.521

While the execution time of the Update Env, Find Next Dest, and Metabolize functions increases linearly with respect to the simulation size, we observed that the Move function's execution time increases non-linearly, or nearly quadratically, as the simulation size grows. Further analysis reveals that the Agent::manageAll() function executes all agent spawn, migrate, and kill functions, regardless of whether they are used in the simulation or not. Moreover, the agent spawn function currently takes a considerable amount of time due to its algorithm. Although the improvements made in Section 4.1.1 have enhanced its performance compared to the previous implementation, it is still not optimal. To address this issue, future implementations may need to focus on refining the algorithms of both the Agents::manageAll() function and the agent spawn function to further optimize their performance and scalability.

For example, attributes could be added to MASS to track whether any agents move or spawn during an iteration; if not, the corresponding functions in the Agents::manageAll() function call would not be executed. Furthermore, the current algorithm for finding the available index is executed by two separate kernel functions, which could be combined into one to save the overhead of starting kernel functions. This was not implemented due to time constraints, but doing so could further enhance performance.

(D) Social Net

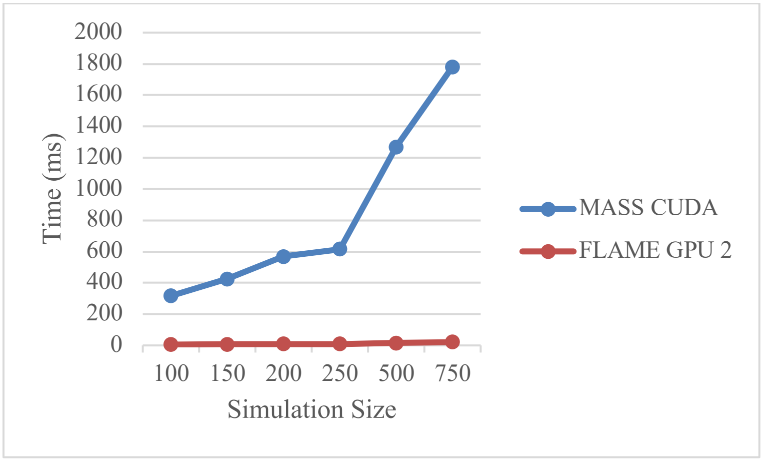

Figure 5.13: Total simulation time comparison of Social Net

Social Net differs from other applications in two key aspects. First, the initialization process, which includes library initialization and the construction of the social network graph, is performed on the host (CPU) and consumes considerable time. As a result, we exclude the initialization time from this benchmark. Second, Social Net does not have an "each iteration" concept because it focuses on finding the Nth degree friends of each person. Therefore, we only measure the total simulation time.

However, our analysis reveals that the execution time of MASS CUDA is significantly slower than FLAME GPU 2. Further investigation uncovers two main issues that contribute to this performance discrepancy, both of which stem from the current functionality and usage restrictions of MASS CUDA. Firstly, since MASS CUDA is still in a development process, users are required to develop extra classes, namely Queue and Set, to support the simulation. The execution of these complex customized classes on the device is expected to negatively impact performance. Secondly, we identified a bug in the current program where the algorithm may unnecessarily search for (N-1)th degree friends each time it looks for an Nth degree friend, leading to increased execution time.

Consequently, given that the benchmark program is not fully functionally correct, we do not consider the execution performance comparison to be valid. However, it is important to address the current functionality restrictions of MASS CUDA.

5.5 Discussions

From the programmability comparison, we see that the programming complexity of MASS CUDA is lower than FLAME GPU 2, matching our original design and target to benefit non-code focused individuals. The usage and understanding of the MASS CUDA library are more straightforward, and easier to maintain. While we understand that there needs to be a balance between ease of use and performance, implying that ease of use may come with some performance trade-offs, we also discovered major issues that impacted the performance.

Further analysis of the benchmark results reveals that most of these performance issues can be attributed to the current functional prematurity of MASS CUDA. While FLAME GPU 2, developed by a team at the University of Sheffield, has undergone extensive development and is now sophisticated and complete, it is obvious that MASS CUDA needs to catch up FLAME GPU 2. By comparing our implementation with FLAME GPU 2, we have identified the following areas for improvement.

First, we need to continue working on data structure revisions to utilize more hardware optimizations and boost data access on the device. Optimizing data structures can significantly impact the performance of memory-bound applications.

Second, we should implement the kernel execution at a lower level to better cooperate with the hardware, such as utilizing CUDA streams and PTX (a low-level parallel thread execution virtual machine and instruction set architecture used for CUDA) [31]. By leveraging these low-level features, we can potentially improve the efficiency of kernel execution and reduce overhead.

Third, as we continue to develop the MASS CUDA library, we need to support more functionalities to ease the usage for users while potentially increasing performance by avoiding the need for users to create complex customized functions. By providing a comprehensive and optimized set of functionalities, we can enable users to focus on their application logic rather than low-level implementation details.

Chapter 6. CONCLUSIONS AND FUTURE WORK

Throughout the course of this project, we have successfully completed all the required functionalities of MASS CUDA, ensuring that all components are working as intended. To further enhance the library's performance and usability, we have made significant revisions to the underlying algorithms and data structures. These optimizations focused on minimizing data transfer, identifying opportunities for parallelization, and leveraging the unique capabilities of GPU hardware. As a result, both the programmability and performance of MASS CUDA have been improved significantly. To facilitate seamless development and integration, we have also produced comprehensive documentation tailored for both developers and users, ensuring that future contributors can work on MASS CUDA with ease and users can quickly integrate the library into their projects.

6.1 LIMITATIONS

Despite the progress made, we have identified several limitations in the current implementation. Firstly, the library is not yet capable of scaling to multiple GPUs. However, given the recent advancements in GPU hardware, this functionality can be achieved in the near future by implementing data exchange mechanisms between GPUs. Secondly, the object-oriented design of MASS CUDA has restricted the ability for users to fine-tune parameters based on different models. While this design choice simplifies integration and is well-suited for our target audience of scientists who prioritize simulation construction and results over performance optimization, exposing certain parameters via API may be considered in future iterations to provide more flexibility. Moreover, the current implementation of Agents::manageAll(), particularly the Agent::spawn() function invoked within it, introduces performance overhead. To address this, we need to develop a more efficient algorithm that minimizes the impact on overall performance. Additionally, the revised data structure currently consumes more memory compared to the previous design, which can be attributed to the early stage of MASS CUDA's development and will be addressed in future optimizations.

6.2 FUTURE WORK

Although the development of MASS CUDA has spanned several years, the recent revision of the data structure marks a significant milestone, establishing a solid foundation for the library to fully leverage the capabilities of GPU hardware through the use of optimized algorithms and data structures. Moving forward, our focus should be on profiling the library using tools like NVIDIA Nsight Compute to identify further opportunities for improvement in areas such as memory access patterns and kernel function execution. Moreover, we should explore dynamic memory allocation on the device based on user requirements to minimize unnecessary memory allocation overhead. Additionally, utilizing faster caches on the device for variables that remain constant throughout the simulation, such as simulation parameters and pointers to attributes, can further boost performance. Besides, none of the benchmark applications currently utilize agent spawn function, we plan to implement Tuberculosis in order to benchmark it. Furthermore, our current implementation requires users to employ functions to set and manually distribute data on the GPU, which exposes unnecessary complexity. We need to continue developing the library to abstract away these low-level details and provide a more user-friendly interface. Most importantly, while we have achieved the expected basic design and functionalities in the current phase, we have yet to fully harness a key component of CUDA: streams. Many operations within the library can benefit from being executed using different streams to reduce the overhead of kernel function execution.

In conclusion, the current version of MASS CUDA represents a significant milestone in its development, with a solid foundation in place for future optimizations and enhancements. By focusing on performance profiling, dynamic resource management, algorithm optimization, and the utilization of CUDA streams, we can unlock the full potential of GPU hardware and provide users with a high-performance, user-friendly library for agent-based modeling. This will allow users to focus on the modeling aspects of their simulations while seamlessly harnessing the computational capabilities of GPUs. As we move forward, it is essential to prioritize scalability, flexibility, and continuous optimization to ensure that MASS CUDA remains at the forefront of GPU-accelerated agent-based modeling solutions.

BIBLIOGRAPHY

[1] A. Antelmi, G. Cordasco, G. D'Ambrosio, D. D. Vinco and C. Spagnuolo, "Experimenting with Agent-Based Model Simulation Tools," *applied sciences,* vol. 13, no. 1, 2022.

[2] M. F. Zhiyuan Ma, "A Multi-Agent Spatial Simulation Library for Parallelizing Transport Simulations," in *Winter Simulation Conference*, 2015.

[3] S. W. Wenwu Tang, "HPABM: A Hierarchical Parallel Simulation Framework for Spatially-explicit Agent-based Models," *Transactions in GIS,* vol. 13, no. 3, pp. 315-333, 2009.

[4] B. G. A. Kalyan S. Perumalla, "Data parallel execution challenges and runtime performance of agent simulations on GPUs," *SpringSim,* vol. 8, pp. 116-123, 2008.

[5] N. H. M. F. Lisa Kosiachenko, "MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models," in *PAAMS 2019*, 2019.

[6] T. C. M. F. John Emau, "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, 2011.

[7] J. D. O. S. N. L. N. S. Rajeev Alur, "Static detection of uncoalesced accesses in GPU programs," *Formal Methods in System Design,* vol. 60, pp. 1-32, 2021.

[8] C. C.-R. L. P. K. S. G. B. Sean Luke, "MASON: A Multiagent Simulation Environment," *SIMULATION,* vol. 81, no. 7, pp. 517-527, July 2005.

[9] "Repast," ARGONNE NATIONAL LABORATORY, [Online]. Available: https://repast.github.io/.

[10] N. T. C. J. R. V. Michael J. North, "Experiences creating three implementations of the repast agent modeling toolkit," *ACM Transactions on Modeling and Computer Simulation,* vol. 16, no. 1, pp. 1-25, 2006.

[11] U. Wilensky, "NetLogo," Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999. [Online]. Available: https://ccl.northwestern.edu/netlogo/.

[12] S. L. L. S. K. J. Steven F. Railsback, "Agent-based Simulation Platforms: Review and Development Recommendations," *SIMULATION,* vol. 82, no. 9, pp. 609-623, 2006.

[13] M. N. Nicholson Collier, "Parallel agent-based simulation with Repast for High Performance Computing," *SIMULATION,* vol. 89, no. 10, pp. 1215-1235, 2012.

[14] S. v. D. H. H. D. Christophe Deissenberg, "EURACE: A Massively Parallel Agent-Based Model of the European Economy," 2008.

[15] "FLAME," Software Engineering Group, [Online]. Available: http://flame.ac.uk/.

[16] P. Richmond and M. K. Chimeh, "FLAME GPU: Complex System Simulation Framework," in *2017 International Conference on High Performance Computing & Simulation (HPCS)*, Genoa, Italy, 2017.

[17] R. C. P. H. M. K. C. M. L. Paul Richmond, "FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs," The Department of Computer Science, University of Sheffield, Sheffield, UK, 2023.

[18] G. B. J. F. Fabien Michel, "The TurtleKit Simulation Platform: Application to Complex Systems," in *First International Conference on Signal-Image Technology and Internet-Based Systems*, 2005.

[19] TurtleKit, "TurtleKit," Université Montpellier 2, [Online]. Available: https://www.madkit.net/turtlekit/index.php. [Accessed 2024].

[20] F. G. 2, "About FLAME GPU," University of Sheffield, [Online]. Available: https://flamegpu.com/about/. [Accessed 2024].

[21] "NVIDIA Nsight Compute," NVIDIA, [Online]. Available: https://developer.nvidia.com/nsight-compute. [Accessed 2024].

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

[23] "mass_cuda_core," [Online]. Available: https://bitbucket.org/mass_library_developers/mass_cuda_core/src/v0.5.3/.

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

[25] "Conway's Game of Life," [Online]. Available: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[26] J. D. O. S. N. L. N. S. Rajeev Alur, "Static detection of uncoalesced accesses in GPU programs," *Formal Methods in System Design,* vol. 60, pp. 1-32, 2021.

[27] NVIDIA, "Event Management," [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.

[28] M. Gardner, "MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life"," 1970.

[29] J. M. Epstein, "Growing artificial societies," in *Brookings Institution Press*, Washington, D.C., 1996.

[30] Terry.Yin, "lizard," [Online]. Available: https://pypi.org/project/lizard/.

[31] NVIDIA, "Parallel Thread Execution ISA Version 8.4," [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

APPENDIX A. DETAILED PERFORMANCE RESULTS

All times are in milliseconds (ms).

Appendix Table A-1: MASS CUDA execution performance of Game of Life before data structure revising

Simulation Size	Initialization Time	Per-step Time	Total Time	Simulation
500	3356	0.338	3443	
1000	3413	1.277	3536	
1500	4084	2.763	4213	
2000	11346	4.909	14590	
2500	27575	7.612	29484	

Appendix Table A-2: MASS CUDA execution performance of Game of Life after data structure revising

Simulation Size	Initialization Time	Per-step Time	Total Time	Simulation
500	3238	0.178	3286	
1000	3248	0.557	3399	
1500	3252	1.155	3537	
2000	3257	1.988	3786	
2500	3300	3.035	4122	

Appendix Table A-3: FLAME GPU 2 execution performance of Game of Life

Simulation Size	Initialization Time	Per-step Time	Total Time	Simulation
500	3856.695	0.197	3906	
1000	3876.693	0.453	3990	
1500	3993.211	0.891	4216	
2000	4283.149	1.451	4646	
2500	4538.715	2.221	5094	

Appendix Table A-4: MASS CUDA execution performance of Heat2D

Simulation Size	Initialization Time	Per-step Time	Total Time	Simulation
256	3228	0.115	3596	
500	3233	0.399	4452	
1000	3263	1.523	7855	

1500	3255	3.379	13417
2000	3281	5.899	21003
2500	3275	9.294	31181
3000	3258	13.238	42999

Appendix Table A-5: FLAME GPU 2 execution performance of Heat2D

Simulation Size	**Initialization Time**	**Per-step Time**	**Total Simulation Time**
256	3845.313	0.236	4555
500	3931.931	0.339	4951
1000	4008.55	0.590	5781
1500	4227.504	1.017	7280
2000	4479.782	1.623	9350
2500	4786.135	2.410	12017
3000	5169.638	3.368	15274

Appendix Table A-6: MASS CUDA execution performance of Sugarscape

Simulation Size	**Initialization Time**	**Per-step Time**	**Total Simulation Time**
256	3.537	1.469	1472.537
500	6.773	4.784	4790.773
1000	17.367	18.156	18173.37
1500	35.465	40.333	40368.47
2000	62.8	71.601	71663.8
2500	93.79	110.826	110919.8
3000	133.383	159.975	160108.4

Appendix Table A-7: FLAME GPU 2 execution performance of Sugarscape

Simulation Size	**Initialization Time**	**Per-step Time**	**Total Simulation Time**
256	19,385.57	1.716	21101.573
500	5,168.11	3.419	8587.11
1000	7,049.09	9.554	16603.094
1500	18,493.21	20.087	38580.206
2000	48,794.21	34.62	83414.211
2500	111,596.62	54.153	165749.615
3000	225,681.51	77.942	303623.51

Appendix Table A-8: MASS CUDA execution performance of Social Net

Simulation Size	Total Simulation Time
100	316.73
150	425.136
200	568.272
250	615.743
500	1268.705
750	1781.945

Appendix Table A-9: FLAME GPU 2 execution performance of Social Net

Simulation Size	Total Simulation Time
100	5.166
150	6.214
200	8.131
250	8.461
500	14.491
750	20.605

APPENDIX B. SOURCE CODE DETAILS

B1. MASS CUDA Library Source Code

The source code for the MASS CUDA Library can be found at https://bitbucket.org/mass_library_developers/mass_cuda_core/src/main/. As of submitting this project report, the latest version is v0.7.2. Specific versions can be found in the Branches and Tags.

B2. Getting Started

Prerequisites: NVCC and C++ 17+ must be set. If running on our school's Juno server, no environment setup is needed.

Building project:

1. In the source code root folder, where the Makefile is located, use `make develop` to install all necessary dependencies for library development and testing.
2. Type `make build` to compile the library.
3. When testing each modification, after executing step 2, type `make test` to build the test program and execute unit tests. The source code for the tests is located in the test folder.

For more information, please visit the source code repository and read the README. The user manual is available on the Wiki page. Additionally, we have developed benchmark applications to benchmark the MASS CUDA Library as described in Section 5.1, which are detailed below.

B3. MASS CUDA Application Source Code

The source code for the MASS CUDA applications can be found at https://bitbucket.org/mass_application_developers/mass_cuda_appl/src/main/. Execution details for each application are provided in their respective folders.

Details About Each Application

1. AppTemplate: A template for developing new applications. Details are in the README and should be followed.
2. Game of Life:
   a. GameOfLife_GPU and GameOfLife_MASS: Not tested, and details are unknown.
   b. GameOfLife_dev: Only compatible with MASS CUDA versions < v0.7.0.
   c. GameOfLife_PlaceV2: The newest implementation, compatible with MASS CUDA versions v0.7.0+.
3. Heat2D:
   a. Heat2D_FLAME: The FLAME GPU 2 implementation of Heat2D.
   b. Heat2D_MASS: Compatible with MASS CUDA versions < v0.7.0.
   c. Heat2D_PlaceV2: The newest implementation, compatible with MASS CUDA versions v0.7.0+.
4. SocialNetwork:
   a. SocialNetwork_FLAME: The FLAME GPU 2 implementation.
   b. SocialNetwork_PlaceV2: The newest implementation, compatible with MASS CUDA v0.7.0+.
5. SugarScape:

a. SugarScape_GPU: Not tested and unknown.
b. SugarScape_MASS: Compatible with MASS CUDA versions < v0.7.0.
c. SugarScape_MASS_2024 and SugarScape_MASS_2024_Aligned_FLAME: The newest implementations, with the latter aligned as closely as possible to the FLAME GPU 2 implementation. Both are compatible with MASS CUDA versions > v0.7.0.

APPENDIX C. LIBRARY CLASSES INVOKE FLOW

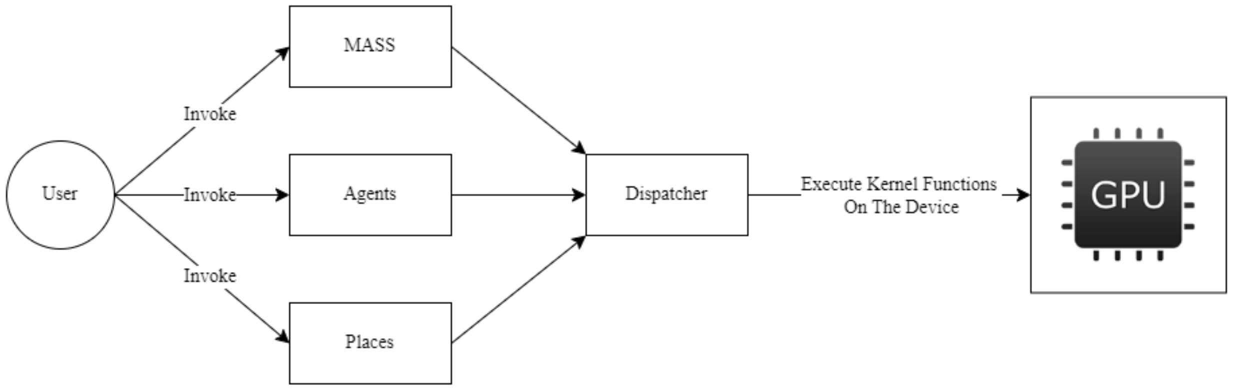

Figure C1.1 Main classes invoke flow of MASS CUDA Library

The MASS CUDA Library exposes three main classes (APIs) to users: MASS, Agents, and Places. When any of these APIs are invoked, they call the Dispatcher class, which then executes the corresponding kernel functions on the device to achieve parallel computation. Therefore, when developing MASS CUDA, it is recommended to start by exploring the MASS class and following the function invocations to the Agents and Places classes to better understand the library's structure.

Below is a brief description of each class in MASS CUDA:

1. Agent: Represents an individual agent object.
2. AgentAttributes: Stores the default attributes of an agent that are used in MASS CUDA.
3. Agents: The API exposed to users for managing all agents on the device.
4. CudaEventTimer: Records the time the GPU is invoked in a specific portion of the program.
5. DeviceConfig: Contains configurations and functions for objects on the device.
6. Dispatcher: Contains all kernel functions that are executed on the device and is invoked by other classes.
7. Mass: The API exposed to users for starting and ending simulations.

8. Place: Represents a single place object.
9. PlaceAttributes: Stores the default attributes of a place that are used in MASS CUDA.
10. Places: The API exposed to users for managing all places on the device.