Agent-Based Models Library Over Multiple GPUs: Term Report

Warren Liu

School of STEM, Computer Science & Software Engineering

University of Washington

CSS 595: Master's Project, Winter 2024

3/12/2024

Project Committee:

Professor Munehiro Fukuda, Committee Chair

Professor Kelvin Sung, Committee Member

Professor Clark Olson, Committee Member

Table of Contents

1 Introduction

This term report highlights the advancements in my capstone project throughout the Winter 2024 quarter, entering the second phase of implementing the MASS CUDA library. Unlike the initial proposal aimed at enabling multi-GPU support, my focus shifted towards diagnosing performance bottlenecks and conducting a comprehensive overhaul of the library. This strategic redirection not only resulted in a substantial performance boost but also laid a solid foundation for future enhancements, including the integration of multi-GPU functionality.

2 Background

In parallel and distributed computing, the strategy of utilizing multiple machines, or computing nodes, to tackle a problem is common. The essence of this approach is to divide a problem into smaller parts, each of which is handled by a different computing node simultaneously. Theoretically, if a problem is divided among four nodes, it should take roughly a quarter of the time to solve, assuming perfect efficiency. The expectation is that by increasing the number of computing nodes, the time to solve the problem decreases, improving performance.

However, the improvement in performance isn't always proportional to the number of added computing nodes. One primary reason is the communication overhead among the nodes. As the number of nodes increases, so does the requirement for communication between them, leading to significant overhead. Two main issues contribute to this overhead:

1. Data Distribution: Initially, data must be distributed to each computing node, a process that inherently lacks parallelism. With more nodes, the data distribution phase becomes more cumbersome and time-consuming.
2. Inter-node Communication: Each node may require data or state information from its neighbors to proceed with its computation. This necessitates constant communication between nodes, increasing linearly with the number of nodes and adding to the overhead.

Agent-based modeling (ABM) offers an innovative approach to addressing the scalability and communication challenges inherent in traditional parallel and distributed computing models. ABM focuses on defining autonomous agents with their own behaviors and rules, which interact within a defined environment. Here’s why and how ABM is effective:

- Data Locality: In ABM, data is distributed to nodes but remains stationary. Computation, in the form of agents, moves to where the data resides. This significantly reduces the need for data transfer, as agents process data locally where possible.

- Reduced Communication Overhead: Since agents operate on local data and only interact with their immediate environment, the volume of data that needs to be transferred between nodes is minimized. Agents only communicate when necessary, based on their interactions within the simulation environment, which is often less data-intensive than constantly syncing state information across nodes.
- Scalability: By minimizing communication overhead and keeping data localized, ABM can scale more effectively than traditional parallel computing approaches. The addition of more computing nodes does not exponentially increase the communication burden, allowing for more linear scalability under the right conditions.

The Multi-Agent Spatial Simulation (MASS) library is a specialized framework designed for the development of agent-based models within the realms of parallel and distributed computing. It serves as a tool to efficiently create, manage, and simulate agents and their interactions within a defined environment. MASS is structured around two fundamental concepts: Agent and Place.

- Place: This component acts as the spatial container or environment where agents operate. It holds data or attributes relevant to the simulation and defines the spatial relationships, including the identification of neighboring places. This structural design facilitates the exchange of information among places, making it convenient for an agent to access and retrieve data about its immediate surroundings or neighbors.
- Agent: This is the dynamic, computational unit within MASS. Agents contain user-defined logic or functions that dictate their behavior within the simulation. They are mobile and can move from one place to another, executing computations that can influence both their state and the state of the places they interact with. The mobility of agents allows for a dynamic simulation environment where interactions and behaviors evolve over time.

To streamline the management of agents and places, MASS introduces two classes: Agents and Places. These serve as containers or managers for agent and place objects, respectively. They offer a high-level API through which users can interact with and manipulate the agent-based model. This design not only simplifies the development process but also enhances the scalability and flexibility of simulations, allowing researchers and developers to focus on the domain-specific aspects of their models without getting bogged down by the underlying computational complexities.

The MASS library has been equipped with functions like *Places.exchangeAll()*, *Places.callAll()*, and *Agents.manageAll()*, designed to operate across all Place or Agent objects concurrently. These functions are inherently parallel, making them ideal candidates for acceleration using General-Purpose Graphics Processing Unit (GPGPU) technology. This is the premise behind the development of MASS CUDA.

MASS CUDA enhances the library by enabling it not just to run on distributed memory systems but also to exploit the computational prowess of GPUs, particularly Nvidia GPUs with CUDA technology. Unlike a conventional computing mode where a single CPU handles computations, a GPU boasts thousands of cores capable of executing many threads simultaneously. This architecture allows for the massively parallel processing of Place and Agent computations, significantly accelerating the execution of agent-based models.

3 Goals

In my last report, I reviewed our code extensively, slowed by limited documentation. Unlike C++ and Java, the GPU version needed a new approach due to unique algorithm challenges. I created a stable single-GPU version, benchmarked it, identified issues, and looked into performance improvements.

This quarter's goal was to start on the multi-GPU version. However, benchmarks showed our library lagged behind FLAME GPU 2, which is our GPU-based ABM competitor, significantly. This led to prioritizing performance enhancements before adding multi-GPU support. My focus was on:

1. Detailed benchmarking against MASS CUDA and FLAME GPU 2 to spot performance differences in initialization, simulation, and shutdown.
2. Analyzing performance, identifying bottlenecks, and implementing solutions for better efficiency.
3. Creating clearer documentation for easier use and understanding of the library.

4 Achievements This Quarter

Initiating with Nvidia's performance guidelines, I utilized Nvidia Nsight Compute for profiling, pinpointing the performance bottleneck. Through a comprehensive library overhaul, focusing on data structures and algorithms, I achieved a remarkable 258-fold increase in performance for initialization and 10-fold for execution phases. The following details the approaches and methodologies employed.

4.1 Profiling with Nsight Compute

To tackle the performance bottleneck, I commenced by delving into Nvidia's documentation and aligning it with our implementation, applying recommended optimizations like employing faster memory (constant or shared), minimizing conditional statements in kernel functions, and employing loop unrolling. Despite these efforts, the improvements were marginal. Progressing to a more detailed analysis, I utilized Nsight Compute to profile each kernel function. One representative outcome, which was commonly observed, is presented below.

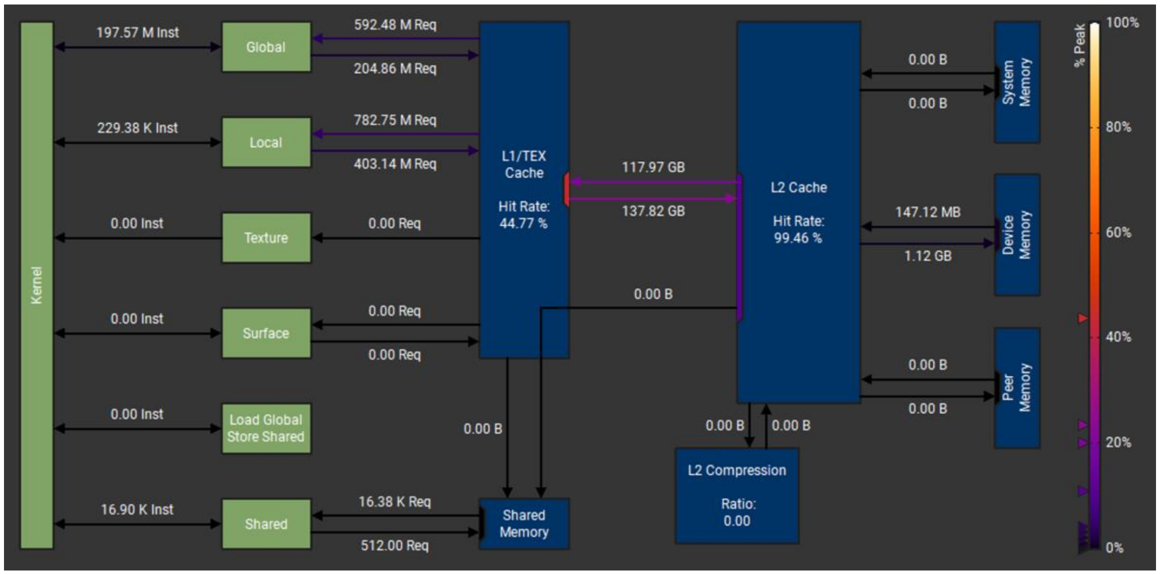

Figure 1 Nsight Compute Profiling Result

This profiling revealed a critical insight: the L1 cache hit rate was below 50%. This implies that for every data retrieval attempt by a register from the L1 cache, there's a 50% likelihood the data isn't there, necessitating a search in the slower L2 cache. Given the L1 cache's proximity to each Streaming Multiprocessor (SM) in contrast to the L2 cache's external placement, access times to L2 are notably longer, thus hindering overall performance [1], as shown in Figure 2. Elevating the L1 cache hit rate could therefore markedly enhance performance. Achieving this requires a strategic accumulation of necessary data from L2 to L1 in one operation, underlining the importance of comprehending memory access patterns and the data transfer mechanisms between caches.

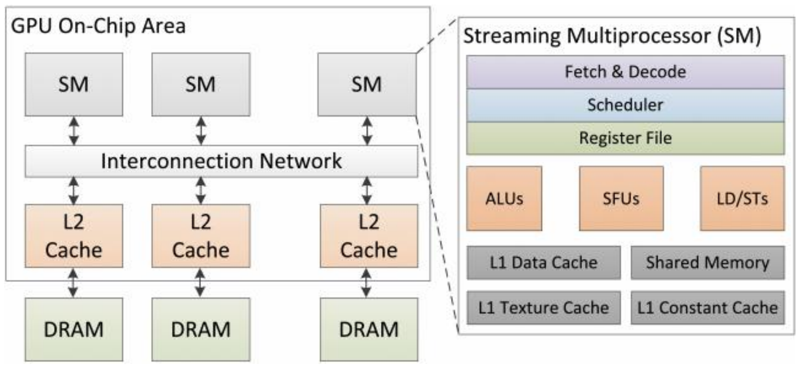

Figure 2 Baseline GPU architecture [2]

4.2 Understand Spatial Locality and Prefetching

Spatial locality and prefetching principles play critical roles in how data is accessed from memory. Rather than retrieving solely the needed data, CPU and GPU architectures preemptively fetch a contiguous block of data around the requested index. This approach, predicated on the principle of spatial locality, operates under the assumption that data near a recently accessed item will soon be needed.

When a thread accesses data within an array, the memory system fetches and stores a larger data block encompassing the requested index into a nearer, quicker memory cache. This block, known as a cache line in CPUs or a memory transaction in GPUs, contains the sought data alongside neighboring elements.

By anticipating future data requests, this strategy significantly reduces the need for costly global memory accesses. It is a cornerstone of memory system optimization across CPUs and GPUs, aimed at minimizing computational delays caused by data retrieval.

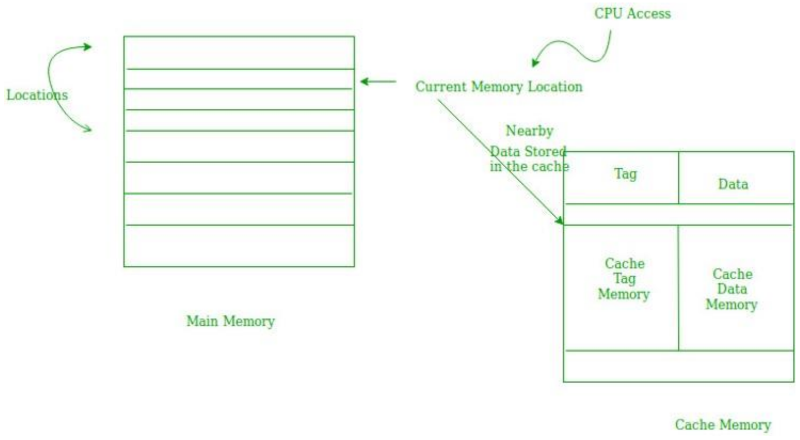

Figure 3 Spatial Locality [3]

4.3 GPU Coalesced Memory Access

Building on the concept of prefetching to leverage spatial locality, we now explore a unique optimization within GPUs: 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, 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.

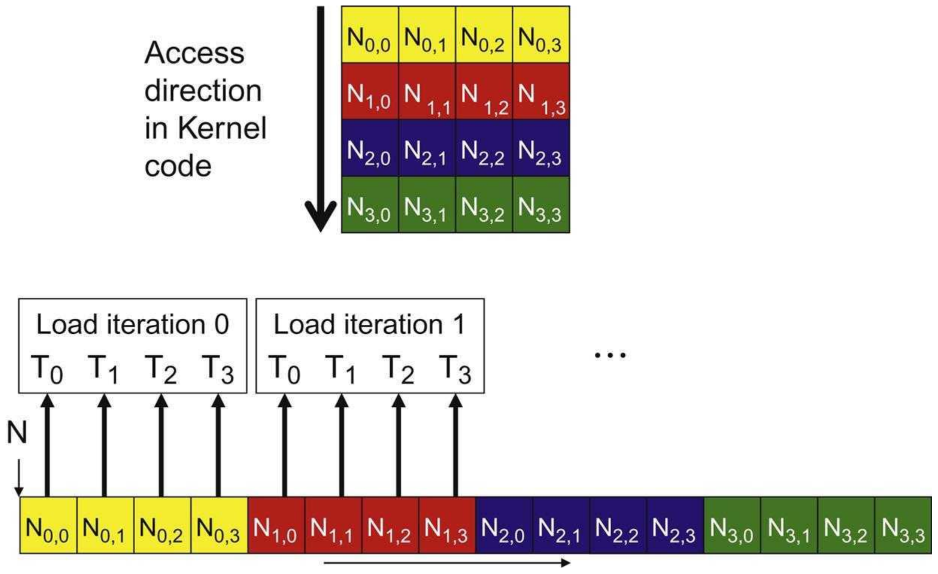
Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
...
N

Figure 4 Coalesced Memory Access [1]

4.4 Original Data Structure of MASS CUDA

Central to the MASS CUDA library are the Place and Agent objects. The former encapsulates complex data alongside some straightforward functions, whereas the latter houses simpler data but is tasked with executing more complex computational functions. In the initial design, these elements were distinct, segregating functions from data within separate classes. For instance, the Agent and its corresponding AgentState class were designed such that an Agent object would house functional logic, linked to an AgentState object containing relevant data through pointers, same to Place and PlaceState.

Within the AgentState or PlaceState objects, several predefined variables were established to facilitate the library's operations, such as

```
1. unsigned int index, the index of the Place/Agent
2. Place *neighbors[], the pointers to neighbor Place of current Place
3. Agent *potentialAgents[], the pointers to the Agent that are going to migrate to current
Place
```

Upon initializing MASS, four arrays are allocated on the device, and each object is interconnected with its state counterpart via pointers, as shown in Figure 5.

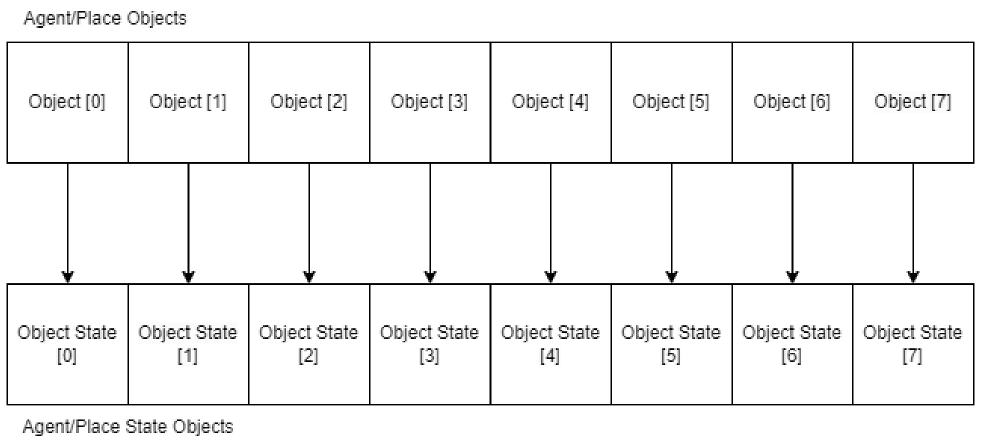

Figure 5 Agent/Place Objects and State Objects

4.5 Root Issue: Uncoalesced Memory Access

In the MASS framework, kernel functions are executed across all Place or Agent objects concurrently to facilitate parallel computation. These functions often require reading data from the AgentState or PlaceState objects. Despite arranging these objects in arrays for what seems like an optimal setup for coalesced memory access, issues persist, as highlighted by Nsight Compute: the presence of uncoalesced global memory access across nearly all kernel functions. This inefficiency leads to unnecessary data being pulled into the L1 cache, a problem stemming from the spatial locality principles discussed earlier.

The crux of the issue lies within the PlaceState and AgentState objects' structure and size. A closer examination reveals that each object is at least 280 bytes, rendering them overly complex for efficient memory access within kernel functions. For instance, accessing the “index” variable in a PlaceState object, given its 280-byte size, means a significant distance in memory between indices of consecutive objects. This arrangement disrupts the continuity in memory access, rendering spatial locality and prefetching techniques ineffective. Consequently, the excessive size of AgentState and PlaceState objects surpasses the hardware's capability for transferring data from global to local memory in a single operation. This inefficiency not only hampers the ability to achieve coalesced memory access but also necessitates individual data fetching by each thread, significantly slowing down computational performance.

4.6 Solution: SOA Instead of AOS

The initial setup of the MASS library utilized an Array of Structures (AOS), complicating the data architecture for CUDA processing and leading to uncoalesced memory access. To address this inefficiency, a transition to a Structure of Arrays (SOA) is proposed. This restructuring involves segregating the data contained within PlaceState objects, for instance, which house variables like *int index* and *int numAgents*. Instead of a collective array of PlaceState objects, the revised approach mandates separate int arrays for *indexes* and *numAgents*, corresponding to each Place.

This reorganization ensures that when threads access a specific variable, they uniformly target the same array, thereby simplifying the data type involved. Such a streamlined process ensures coalesced access to data, significantly enhancing the efficiency of memory transactions.

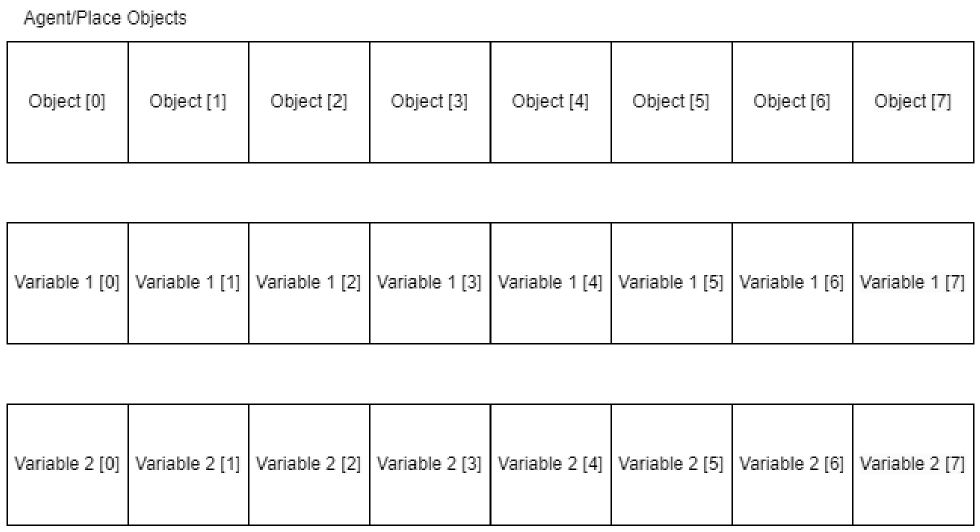

Figure 6 AOS Data Structure

4.7 Dynamic Attribute Setting Solution

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 customizing their own PlaceState or AgentState by inheriting base classes and adding variables as needed. To maintain this flexibility while adapting to the new data structure, a solution was devised to allow users to dynamically add and access attributes for Place/Agent objects within kernel functions.

The solution involves the initialization of three pointer arrays in Agent/Place:

```
1. int *attributeTags: Stores attribute names (tags).
2. void *attributeDevPtrs: Holds pointers to attribute arrays on the device.
3. size_t *attributePitches: Keeps track of the pitch size crucial for CUDA when allocating
attribute arrays on the device.
```

When creating an attribute, users invoke *setAttribute()*, specifying the attribute's tag, length, and an optional default value. MASS then allocates memory on the device for the attribute array and updates attributeDevPtrs, attributePitches, and attributeTags accordingly. Here's an illustrative pseudo-code:

```
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 (want to set default value):
                set default value (defaultValue)
            return true
        return false
```

Before utilizing an attribute, users must execute *finalizeAttributes()*, prompting MASS to distribute these array data across all Agent/Place objects on the device. Subsequently, within kernel functions, retrieving an attribute is simplified through *getAttribute()*, enabling threads to access relevant attributes efficiently. The adapted data structure ensures coalesced memory access, significantly enhancing performance:

```
getAttribute(attributeTag, attributeLength):
    if (attributeTag not in attributeTags):
        return None
    else:
        index = get the index of attributeTag in attributeTags
        attributeDevicePointer = attributeDevPtrs[index]
        attributeValue = attributeDevicePointer[object index]
    return attributeValue
```

This dynamic attribute setting mechanism not only restores the original implementation's flexibility but also aligns with the optimized SOA data structure, ensuring efficient memory access and computation.

5 Results

The optimization efforts significantly improved both the initialization and kernel execution times. A demo application, Game of Life, was developed using the revised MASS framework to benchmark these enhancements against both the original MASS implementation and our key competitor, FLAME GPU 2. The findings are presented as follows.

5.1 Initialization Time Comparison

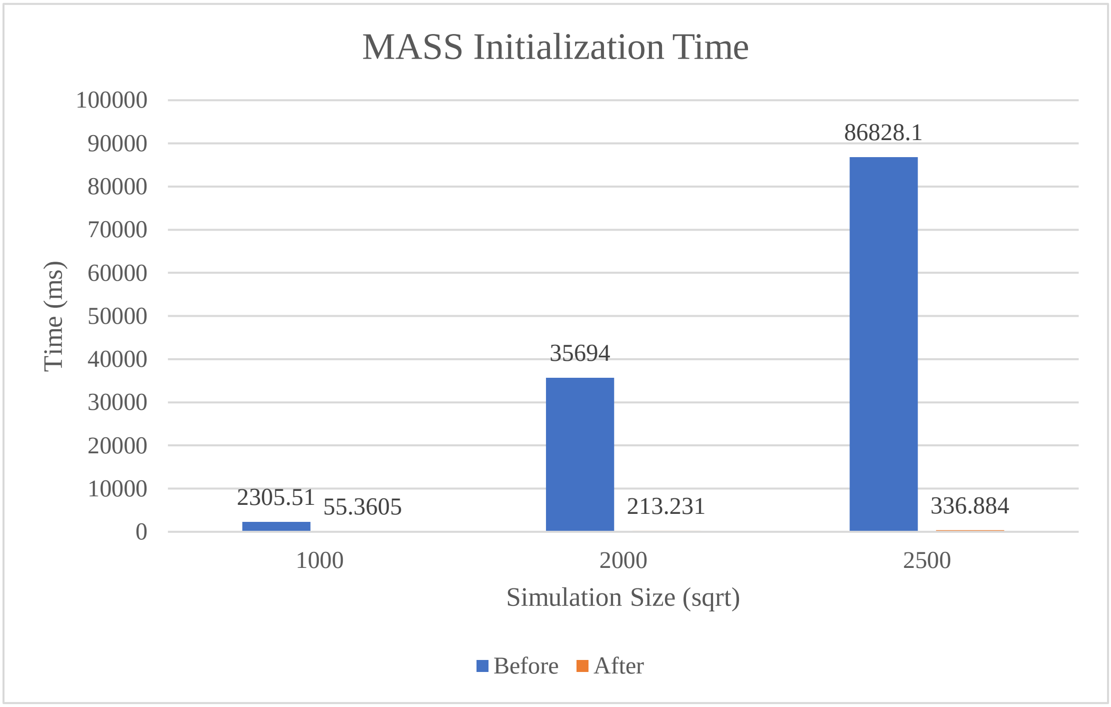

Figure 7 MASS Initialization Time Comparison

The analysis revealed that the original implementation's initialization phase was significantly slower—ranging from 40X to 258X—than the revised MASS framework. Furthermore, while the original setup exhibited exponential slowdowns as the simulation size increased, the new implementation showed only linear growth in initialization time with increasing simulation size. This improvement is attributed to the new data structure's ability to parallelize array initialization on the device and synchronize operations with CPU code, minimizing the impact of additional attributes on initialization time.

5.2 Step Time Comparison

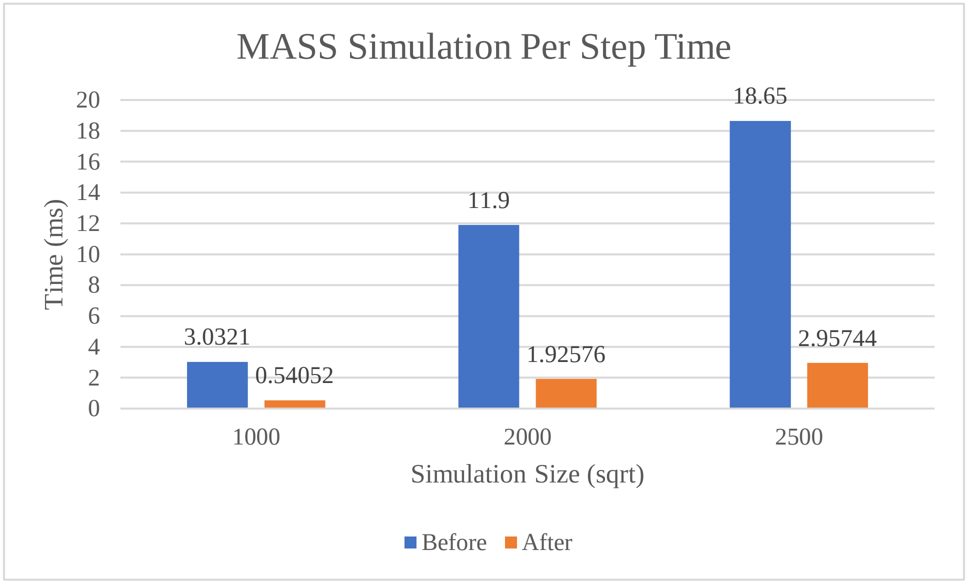

Figure 8 MASS Step Time Comparison

5.3 Total Simulation Time Comparison

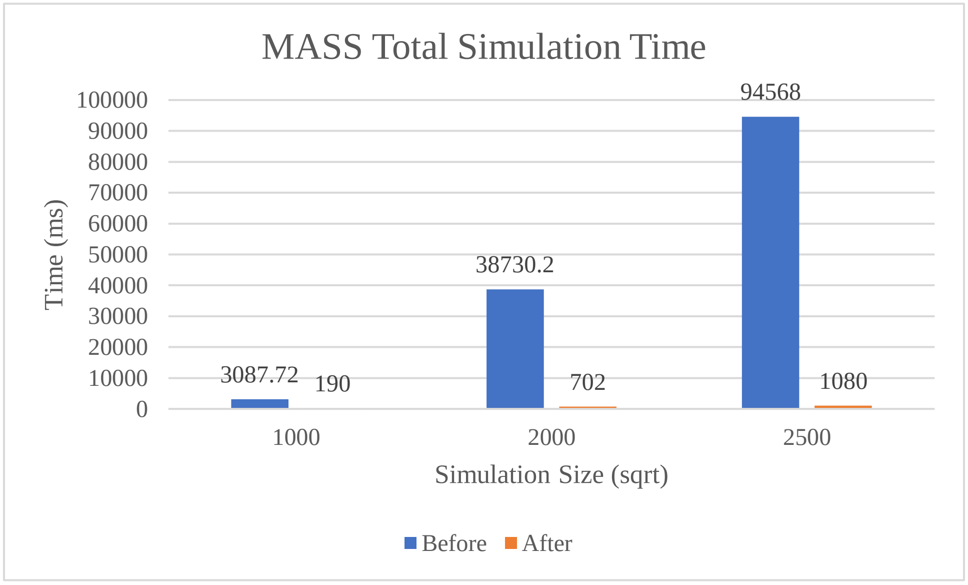

Figure 9 MASS Total Simulation Time Comparison

The overall simulation time for running 250 generations of the Game of Life saw remarkable improvements, showcasing the efficacy of the data structure optimizations.

5.4 FLAME GPU 2 Comparison

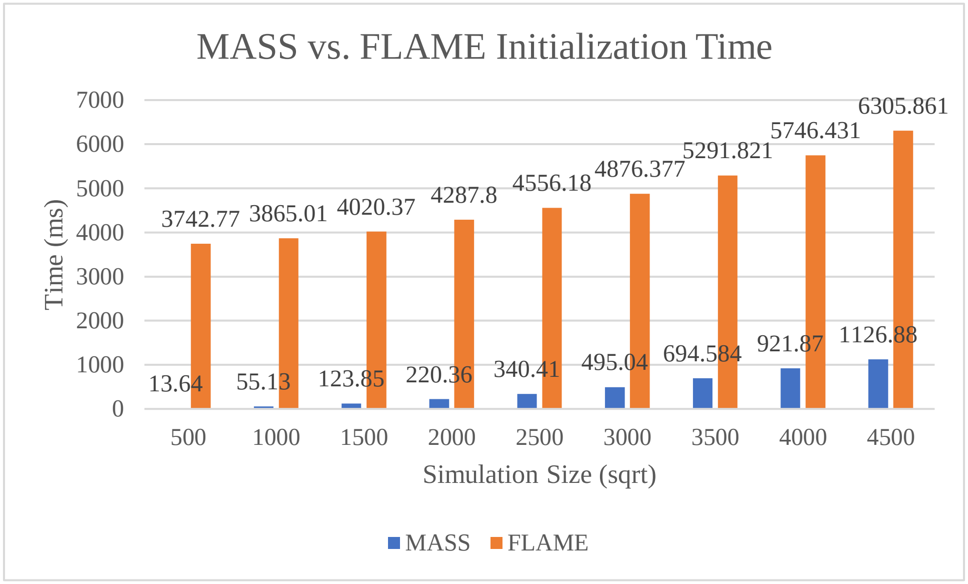

Figure 10 MASS vs. FLAME Initialization Time

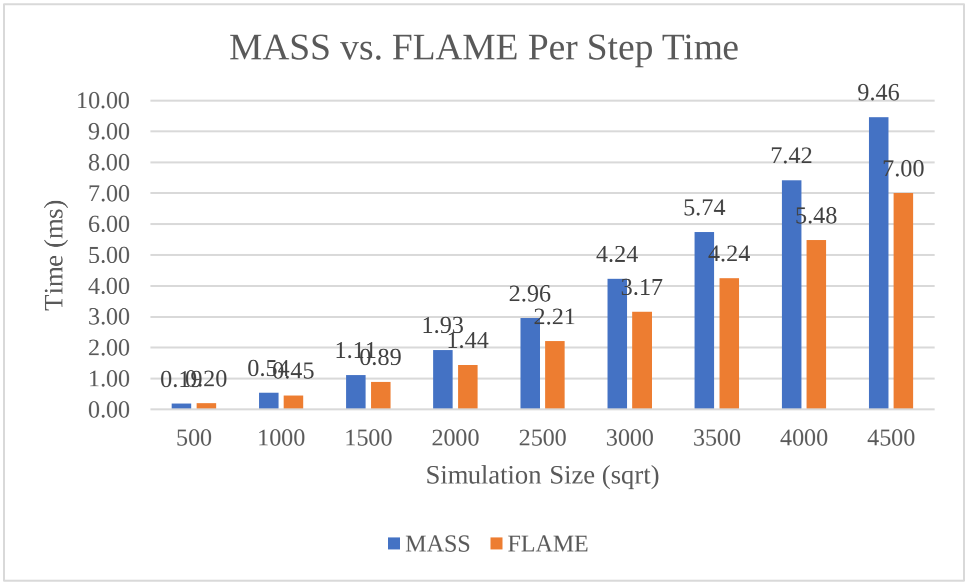

Figure 11 MASS vs. FLAME Simulation Step Time

The introduction of the new data structure significantly enhanced initialization times, outperforming FLAME GPU 2. However, the simulation execution time remains roughly a quarter slower than that of FLAME GPU 2, highlighting an area for ongoing optimization efforts.

6 Next Quarter’s Plan

In line with the outlined proposal and my personal timeline, I aim to conclude the development of the MASS CUDA library by Spring break. I will dedicate the break to refining the library with additional user-friendly helper functions and completing comprehensive documentation to aid both users and future developers in navigating the library effectively.

For the upcoming quarter, my agenda includes dedicating approximately four weeks to the development of another application utilizing the MASS CUDA framework. This collaborative effort, involving other students, is expected to surface further refinements and necessitate minor adjustments to the library. While I intend to address these smaller issues and update the documentation accordingly, more substantial concerns will be earmarked for future development phases.

Concurrently, early in the next quarter, I plan to draft the white paper and start the preparations for my final defense presentation.

7 Summary

The journey of developing the MASS CUDA library over two quarters has been immensely rewarding. Diving deep into CUDA, exploring both its software and hardware aspects to enhance performance, has been a distinct and enriching experience as a software engineer. Despite the challenges and moments of frustration encountered during the library's comprehensive refactoring and the development of dynamic attribute setting functionalities—where giving up seemed like a viable option—the perseverance to continue has been gratifying. Witnessing the library's performance nearly match that of our competitor, exceeding initial expectations, brings a sense of pride and accomplishment. Although numerous improvements and ideas have emerged through this process, time constraints mean leaving these for future developers to explore and implement.

Appendix

Function Implementation

Dispatcher::createAttribute()

This is the fundamental function to be used by MASS to let users create attributes on Place/Agent objects.

Linked Sections: Section 4.7

Usage Example

In the Game of Life, the only attribute we need is "health". In the simulation main program, we add the attribute "health" to all Place objects. Later in the *Places::callAll()* function, we'll use the customized call function to manipulate the "health". See usage example of *getAttribute()*.

```
// Initialize MASS
      Mass::init();

      // Initialize places
      mass::Places *places = mass::Mass::createPlaces<Life>(0, nDims, placesSize,
mass::Place_v2::MemoryOrder::ROW_MAJOR);

      // Initialize neighbors
      vector<int *> neighbors;
      …

      // Set the neighbors for each place
      places->exchangeAll(&neighbors);

      // Add the attribute health to the Place
      places->setAttribute<int>(Life::HEALTH, 1, 0);
      places->finalizeAttributes();

      // Initialize Life cells' health
      places->callAll(Life::INIT_HEALTH);
```

Implementation

```
template <typename T>
__host__ void Dispatcher::createAttribute(unsigned int length, unsigned int qty, void
**attributeDevPtrs)
```

```
{
	if (length <= 1)
	{
		T *d_attr = NULL;
		CATCH(cudaMalloc((void **)&d_attr, sizeof(T) * qty));
		*attributeDevPtrs = (void *)d_attr;
	}
}

template <typename T>
__host__ void Dispatcher::createAttribute(
	DeviceConfig::ObjectType objectType, int handle, unsigned int length, unsigned int
qty,
	T defaultValue, void **attributeDevPtrs)
{
	if (length <= 1)
	{
		T *d_attr;

		CATCH(cudaMalloc((void **)&d_attr, sizeof(T) * qty));

		// Original way to set the default value
		// This will not work because cudaMemset only works for simple types
		// Complex types like class or struct will not work
		// CATCH(cudaMemset(d_attr, defaultValue, sizeof(T) * qty));

		// Get the number of blocks and threads
		dim3 thread, block = 0;
		if (objectType == DeviceConfig::PLACE)
		{
			dim3 *dims = deviceInfo->getPlacesKernelDims(handle);
			thread = dims[0];
			block = dims[1];
		}
		else if (objectType == DeviceConfig::AGENT)
		{
			dim3 *dims = deviceInfo->getAgentsKernelDims(handle);
			thread = dims[0];
			block = dims[1];
```

```
               }
               else
               {
                         throw MassException("Invalid object type");
               }

               // Set the default value
               setDefaultValueKernel<T><<<block, thread>>>(d_attr, defaultValue, qty);

               *attributeDevPtrs = (void *)d_attr;
     }
}

template <typename T>
__host__ void Dispatcher::createAttribute(unsigned int length, unsigned int qty, void **attributeDevPtrs, size_t *pitchIn)
{
     if (length > 1)
     {
               T *d_attr = NULL;
               CATCH(cudaMallocPitch(&d_attr, pitchIn, length * sizeof(T), qty));
               *attributeDevPtrs = (void *)d_attr;
     }
}

template <typename T>
__host__ void Dispatcher::createAttribute(
     DeviceConfig::ObjectType objectType, int handle, unsigned int length, unsigned int qty,
     T defaultValue, void **attributeDevPtrs, size_t *pitchIn)
{
     if (length > 1)
     {
               T *d_attr = NULL;
               CATCH(cudaMallocPitch(&d_attr, pitchIn, length * sizeof(T), qty));

               // Get the number of blocks and threads
               dim3 thread, block = 0;
               if (objectType == DeviceConfig::PLACE)
```

```
            {
                    dim3 *dims = deviceInfo->getPlacesKernelDims(handle);
                    thread = dims[0];
                    block = dims[1];
            }
            else if (objectType == DeviceConfig::AGENT)
            {
                    dim3 *dims = deviceInfo->getAgentsKernelDims(handle);
                    thread = dims[0];
                    block = dims[1];
            }
            else
            {
                    throw MassException("Invalid object type");
            }

            // Set the default value
            setDefaultValue2DKernel<T><<<block, thread>>>(d_attr, defaultValue, qty, *pitchIn, length);

            *attributeDevPtrs = (void *)d_attr;
    }
}
```

Agent::getAttribute() (same as Place::getAttribute())

This is the function to get the attribute of Agent/Place on device.

Linked Sections: Section 4.7

Usage Example

In the Game of Life simulation, we use customized call function, which will be executed by Places::callAll(), which further will be executed by each Place object on the device, to calculate the new "health" attribute.

```
__device__ void Life::computeNextState()
{
      // Get current index
      int index = getIndex();
      // Get attribute NEIGHBORS
```

```
    int *neighbors = getAttribute<int>(index, PlacePreDefinedAttr::NEIGHBORS, MAX_NEIGHBORS);
    // Get the attribute of health
    int *health = getAttribute<int>(index, ATTRIBUTE::HEALTH, 1);

    unsigned int aliveNeighbors = 0;

    // Count alive neighbors
    for (int i = 0; i < MAX_NEIGHBORS; i++)
    {
        if (neighbors[i] != -1)
        {
            // Get the health of the neighbor
            int *health = getAttribute<int>(neighbors[i], ATTRIBUTE::HEALTH, 1);
            // If the neighbor is alive, increment the counter
            if (*health == 1)
            {
                aliveNeighbors++;
            }
        }
    }

    // If current cell is alive
    if (*health >= 1)
    {
        // If alive neighbors are less than 2 or more than 3, then die
        if (aliveNeighbors < 2 || aliveNeighbors > 3)
        {
            *health = 0;
        }
    }
    // If current cell is dead
    else
    {
        // If alive neighbors are exactly 3, then live
        if (aliveNeighbors == 3)
        {
            *health = 1;
```

```
                }
        }
}

```

Implementation

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

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

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

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

How to Run

The code for the MASS CUDA library is available at Bitbucket. Detailed instructions on how to set up and run the code can be found in the README file in the repository. This includes information on installation, dependencies, and execution of the library. A user manual that is currently being developed can be found in the project Wiki page.

References

[1] FANG, "The CUDA Parallel Programming Model - 5. Memory Coalescing," 4 December 2019. [Online]. Available: https://nichijou.co/cuda5-coalesce/. [Accessed 2024].

[2] K. Y. S. L. S. X. F. Jingweijia Tan, "Energy-Efficient GPU L2 Cache Design Using Instruction-Level Data Locality Similarity," *ACM Transactions on Design Automation of Electronic Systems,* vol. 25, no. 6, pp. 1-18, 2020.

[3] "Locality of Reference and Cache Operation in Cache Memory," 24 Feb 2023. [Online]. Available: https://www.geeksforgeeks.org/locality-of-reference-and-cache-operation-in-cache-memory/. [Accessed 2024].