© Copyright 2021

Benjamin D. Pittman

Multi Agent Spatial Simulation (MASS) in multiple GPU Environment with NVIDIA CUDA

Benjamin D. Pittman

A white paper

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2021

Committee:

Munehiro Fukuda, Chair

Erika Parsons

Michael Stiber

Program Authorized to Offer Degree:

Computing and Software Systems

University of Washington

Abstract

Multi Agent Spatial Simulation (MASS) in multiple GPU Environment with NVIDIA CUDA

Benjamin D. Pittman

Chair of the Supervisory Committee:

Professor Munehiro Fukuda, Ph.D.

Computing and Software Systems

Agent-based models are used to simulate real-world random and pseudo-random systems. These models leverage many autonomous agents and require high computation capabilities. The Multi-Agent Spatial Simulation (MASS) grew from this need and is now implemented independently in Java, C++, and CUDA. MASS implements Agents and Places to both act amongst and with each other where Places remain resident and Agents may migrate to other Places.

The cloud computing infrastructure, its ever expanding compute capabilities, and the increasing demands for these resources to solve challenging problems, demand MASS CUDA grow to leverage multiple graphical processing units (GPU). This research accomplishes this through refactorization of the existing code base to initialize models on multiple GPU devices using parallel computing algorithms and strategies. These improvements include (1) ghost-spacing, copies of Places of each bordering device's border Places, (2) GPU direct communication, and an (3) application specific garbage collection, array compaction, and provisioning scheme for Agents.

This MASS CUDA implementation allows larger model sizes and faster processing time for simulations larger than 10,000 Place objects and shows an increase for size of models developed in MASS CUDA of more than double the previous implementation. This research begins an implementation of a neural signal simulation using the MASS CUDA library. The simplified BrainGrid implementation requires Agents to be added to the simulation throughout processing and this research implemented Agent termination and refactored Agent spawning to facilitate. The dynamic Agents' implementation allows for a greater number of simulations using the MASS CUDA library.

MASS CUDA Multiple GPU (MGPU) is an efficient choice for large simulation sizes. Future research should test different data apportioning schemes over devices; test sharing and splitting computations amongst host and devices; and test related strategies for coalescing memory in the system and with like computations.

Chapter 1. INTRODUCTION 1

1.1 Agent-Based Modeling 1

1.2 NVIDIA Compute Unified Device Architecture 2

1.3 Multi-Agent Spatial Simulation (MASS) 3

1.4 MASS CUDA 5

1.5 MASS CUDA MGPU Project Goals 6

1.6 MASS CUDA MGPU Achievements 6

Chapter 2. MASS CUDA BACKGROUND 7

2.1 MASS CUDA Beginnings 7

2.2 MASS CUDA Improved 8

Chapter 3. RELATED WORKS 9

3.1 FLAME GPU 9

3.2 MCMAS 10

3.3 OpenMP 10

Chapter 4. Development of MASS CUDA MGPU 10

4.1 MASS CUDA Library Architecture and Device Management 11

4.2 Multiple Device Improvements to MASS Places 12

4.3 Multiple Device Improvements to MASS Agents 16

4.3.1 Agent Migration 17

4.3.2 Agent Spawn 20

4.3.3 Agent Termination 21

4.4 Neural Simulation: A simplified implementation of BrainGrid 23

Chapter 5. EVALUATION OF RESULTS 26

5.1 Sugarscape Simulation 26

5.2 BrainGrid Simulation 31

Chapter 6. CONCLUSION 32

Chapter 7. REFERENCES 35

Chapter 8. APPENDIX A 39

TABLE OF CONTENTS

LIST OF FIGURES

Figure 1 - MASS CUDA Design Pattern 5
Figure 2 - MASS CUDA MGPU Design Pattern 11
Figure 3 - MASS CUDA MGPU Device Memory Layout 13
Figure 4 - Agent migration across devices 18
Figure 5 - Array Compaction Algorithm 22
Figure 6 - BrainGrid 25
Figure 7 - MASS CUDA MGPU vs. MASS CUDA 27
Figure 8 - Original MASS CUDA Sugarscape Results 30
Figure 9 - MASS CUDA MGPU Sugarscape Results 31

LIST OF LISTINGS

Listing 1: Copy Ghost Places Method 14
Listing 2: Move Agents kernel function 18
Listing 3: Update globally migrated Agents 19

Chapter 1. INTRODUCTION

Multi-Agent Spatial Simulation (MASS) is an Agent-Based Modeling (ABM) library that stores data in a distributed array (called Places) and distributes processes (called Agents) to interact with each other and the data. ABM libraries are efficient for simulations where not all data is needed for each cycle of operations. In these situations, it can be much more efficient to move computation to the distributed data instead of the data to the computation, as classically done.

1.1 AGENT-BASED MODELING

Agent-based models are behavior modeling systems defined by the actors in the system. This method of system design allows for the observation of emergent behavior to explain the environment rather than the enforcement of rules on the actors explaining the environment. Conway's Game of Life is an early and renowned implementation of an ABM that shows simple rules may result in unexpected emergent behavior due to the interaction of neighboring entities. This seminal work is a primary example of what factors lead to ABM as a solution provider. According to Weimar, et al, there exist five primary considerations for choosing ABM [1]:

1. The system or process is representable by distributed, interacting agents
2. The decisions required and the rules by which an entity is to make these decisions are well defined
3. The agent behavior is a focus of the study and how those behaviors might lead to the system-level emergent behavior
4. Adaptation within the system by entities within that system are a focus of the work

5. When adaptations by entities might affect other entities thereby changing the nature of the system under study is an aspect of the study of interest.

These considerations align with numerous areas where modeling environments and their actors have far reaching implications. Public health [4], behavioral sciences [5], transportation [6], and the natural world [7] are some arenas where ABMs are deployed to simulate outcomes.
A primary goal of any model is to explain its environment and ABM systems have evolved to model larger and more complex environments as the computational resources that process them have also progressed [2]. Initially these computational resources were dominated by cluster-based computing resources leveraging multiprocessor CPU's and have since become dominated by these same cluster-based systems adding accelerators to each node [8]. NVIDIA GPUs are a current common hardware accelerator used in high performance computing nodes, and a focus of this research. This is due to their high availability and low cost; consistently updated hardware, software, drivers, and toolkits; and wide adoption in business, academia, and consumer channels.

1.2 NVIDIA Compute Unified Device Architecture

Use of GPUs for general purpose computing (GPGPU) grew from the breakdown of Dennard scaling [9]. Due to increased heat from current leakage at very small transistor sizes. improvements in single processor architecture that allowed a greater number of transistors per unit of area ceased to provide anticipated gains in processor frequency. From this limitation, multi-core processors and hardware accelerators gained wide adoption to allow more processing per computing resource [10]. CPUs designed for multi-purpose computing now leverage multiple processors and related threads of execution. GPUs are specially engineered compute devices designed to process homogenous data in parallel. While these devices were initially designed for manipulating and displaying graphics to a screen, the compute model of single-instruction multiple data (SIMD) is analogous to ABM systems where many agents execute the same instructions in parallel.

NVIDIA’s Compute Unified Device Architecture (CUDA), first released in 2007, extends the C programming language to enable use of GPUs for general purpose computing. This library provides its own memory and compute model and has grown to allow development by novice and experienced programmers through the library’s language hierarchy and is aligned with devices designed, and often manufactured, by NVIDIA. The NVIDIA GPU hardware and CUDA software are dependent on each other and the CUDA library grows with hardware improvements. Further, CUDA provides a unified memory address (UMA) space that combines CPU (host) and GPU (device) memory addresses into a single virtual address space that enables each memory component to deference memory from the others. CUDA UVA also enables device-to-device links across PCIe and directly via NVLINK on some devices. Direct link via NVLINK can decrease memory transfer time by as much as four-times compared to PCIe [11].

1.3 Multi-Agent Spatial Simulation (MASS)

There are two primary goals for the MASS framework – performance and programmability. While we endeavor to process large amounts of data accurately and quickly, it is also a primary aim to furnish the framework for use by non-computing researchers. To accomplish this, MASS provides an application programming interface (API) for researchers to fit their models and leverage the computing power of the framework without knowledge of distributed computing or GPU technologies.

MASS Places are stationary. In many ABM frameworks or libraries, the analogue of Places is merely an array location that can be interpreted as a location in a two or three dimensional plane. MASS implements Places to hold this location data, but also data members that help define them more expressively. Further, Place functions define their behavior and allow them to interact with other Places and Agents.

MASS Agents may migrate to any Place an application programmers' model is implemented to allow. They may migrate across computers in a MASS distributed environment, or across GPUs in MASS CUDA environment. Agents also contain state and behavior to allow them to act with other Agents and with Places. Further, they may terminate or spawn at a time other than the beginning of an implementation.

An application programmer extends the state and behavior of MASS Places and Agents to fit their model. To call the functions that define this behavior, the library provides a base function that is passed the identity of the base or extended function and any parameters. There are two base behavior functions for the MASS Places API:

1. callAll() performs an action on itself or on any Agents that may reside on it.
2. exchangeAll() enables interaction with a Place and it's set of model-defined neighbor Places.

Similarly, the MASS Agents API provides two base behavior functions:

1. callAll() is called on each Agent object to perform an action on itself or on the Place that it resides.
2. manageAll() is called to manage the Agents as a unit. This includes three sub-functions, as follows;

a. terminateAgents() checks for Agents that have been marked as not alive and removes them from the simulation.
b. migrateAgents() transfers migrating Agents to a new Place location.
c. spawnAgents() adds Agents to the simulation.

1.4 MASS CUDA

MASS CUDA adheres to the Model-View-Presenter design pattern and is shown in Figure 1. The presenter manages the data and handles all data access and processing. This is accomplished using two data models – one for the GPU device(s) and one for the host. Both models maintain data in structure of arrays format to facilitate the CUDA compute model of Single Instruction Multiple Data (SIMD). Place and Agent objects are instantiated and maintained in both host and device memory and only state is transferred from device to host when output is requested. An application developer accesses the framework through the view by overloading Place and Agent classes and providing a main class to simulate how they interact for the researcher's environment.

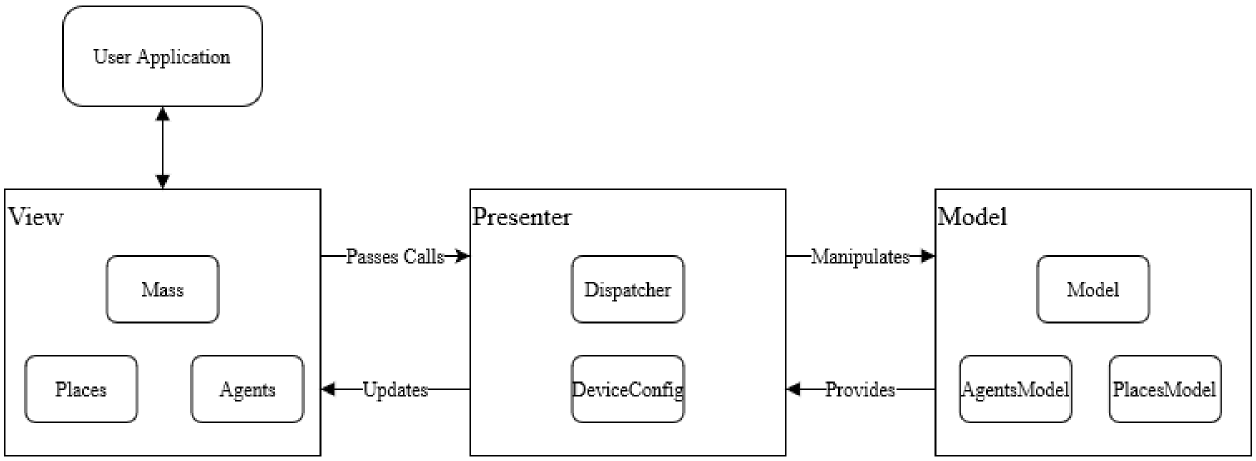


Figure 1 - MASS CUDA Design Pattern

1.5 MASS CUDA MGPU Project Goals

The goals of this research are to (1) Extend MASS CUDA to be able to use multiple GPUs, (2) Build improvements into the library that expand the types of models MASS CUDA can process, and (3) Improve processing time of the previously implemented Sugarscape model.

The sections that follow will expand upon and refine the goals of this research as follows:

1. A brief background on related works involving ABM systems on GPGPU systems will illustrate the broader implications of this work and the knowledge space it resides.
2. The methods used for this research to extend MASS CUDA to leverage multiple GPUs and extend the library for use on a larger number of simulations will be shown.
3. The results of this research will show that MASS CUDA MGPU can process new and larger simulations than previously possible while maintaining the same level of programmability.
4. A brief discussion on where the outcome of this research fits into the field of high-performance computing in general and what future improvements may further the performance and programmability of the MASS CUDA library.

1.6 MASS CUDA MGPU Achievements

This research provides an implementation that allows MASS CUDA to process simulation models on two devices and with a framework that will allow extension to up to sixteen devices with some additional implementation and testing. This implementation processes a basic

Sugarscape simulation model faster than its predecessor single-GPU implementation while allowing a model size 2 ¼ times greater.

Further improvements include extensions to the MASS CUDA library that extend the simulations that may be modeled with it. These improvements include long distance Agent migration, device run-time array compaction of Agents, and refactoring of base functions to allow multiple Agents to remain on a single Place. Finally, a simplified implementation of a neural network growth algorithm is implemented to exhibit these features.

Chapter 2. MASS CUDA BACKGROUND

2.1 MASS CUDA BEGINNINGS

NVIDIA released the Compute Unified Device Architecture (CUDA) platform in 2007 and this enabled developers to use CUDA-enabled graphical processing unit’s (GPU) for general purpose computing. Since this time agent-based modeling (ABM) simulation models previously written for CPU’s were rewritten to run on GPU’s to take advantage of the faster SIMD processing model. However, there remain few general modeling frameworks or libraries that enable researchers to develop ABM simulations that utilize GPU’s and do not require writing CUDA code .

MASS CUDA development began in 2012 with Tosa Ojirua’s and Robert Jordan’s work to take advantage of massive parallelization provided by GPU devices [15, 16]. Rob Jordan’s work resulted in a two-dimensional wave simulation that processes faster in MASS CUDA than MASS C++ when using two GPU’s. This implementation used ghost cell pattern [17] to share border data between devices. However, these MASS CUDA implementations require application modelers to understand and implement the CUDA programming model and many of their strategies were not carried into development that followed..

2.2 MASS CUDA IMPROVED

In 2015, Nathan Hart completed research that placed MASS CUDA in a model-view-presenter design pattern that hid most of CUDA implementation details from application developers [18]. It only requires simulation programmers use of the CUDA compiler (NVCC), .cpp files saved with .cu extension, and a macro placed as a function modifier for any user-defined functions that flag each function for compilation on host and device. However, these factors do not require knowledge of CUDA code or the programming model and increases the ease of programmability for MASS application programmers.

Hart further improved MASS CUDA by splitting the state and behavior of Places and Agents. This change, combined with CUDA improvements that allow function templating, allows Places and Agents and their extensions to be instantiated on host and device, and the transfer of their states between host and device. Other changes include Place and Agent partitioning that enables use of multiple GPU's, and of different specifications, to be leveraged for simulations. Unfortunately, this implementation proved slower than MASS C++ due to the overhead of transferring all states between host and devices at each simulation step to facilitate multiple GPUs.

Lisa Kosiachenko improved MASS CUDA through four main implementations [19]. First, simplification of the implementation to run on one GPU and maintain state on GPU for successive model iterations; only copying from device to host when output is requested. A second memory improvement includes providing a Place->exchangeAll() method that contains a related Place->callAll() method. This allows CUDA to leave the data resident for both calls as it is the same for each call. This greatly reduces the latency of the combined function calls. A third memory latency improvement was to place homogenous and oft-requested data in constant memory. CUDA constant memory resides closer to the processor than global memory and as it is read-only, allows all threads parallel access. This greatly reduces latency. The fourth MASS CUDA improvement was optimization of threads per thread block. This also proved to have a large impact on model simulation runtime. Lower threads per block proved to benefit the cache hit rate due to high memory demands of each Place object.

Chapter 3. RELATED WORKS

3.1 FLAME GPU

FLAME GPU is the most known ABM GPU simulation framework under current development [12]. FLAME GPU seeks to simplify building and executing simulation models for researchers by providing an API that hides complex GPU implementation details from application developers. FLAME GPU application developers use XML schemas to define agents, messages, and movement, and scripts to run simulations. All FLAME models run on a single GPU but can leverage multiple GPUs to run the same model with different parameters. Running FLAME on an NVIDIA V100 device allows up to two-hundred million agents in a FLAME simulation. Models implemented using the FLAME framework include Boids [7], a bird flocking algorithm; Conway's Game of Life [2]; and Sugarscape [13] which is also implemented for this research.

3.2 MCMAS

MCMAS is another ABM GPU framework [14]. MCMAS provides an API for researchers to write plug-ins against to define data and run simulations. However, MCMAS diverges from both MASS and FLAME in difficulty. MCMAS provides a much more open programming interface with which to interact allowing the developer to define simulations, but also how these simulations will process on host and device resources. This means that application developers need knowledge of the GPU programming model and program the solutions. MCMAS is written with OpenCL at the lower level and does not support multiple GPU's. Finally, MCMAS was last updated in 2019 and does not appear to be under ongoing development.

3.3 OpenMP

Open Multi-Processing (OpenMP) is a shared-memory multithreading library for the C, C++, and Fortran programming languages [15]. OpenMP uses compiler directives and a runtime library to expose the power of multithreading in a high-level manner that enables wide adoption as it can be quickly added to parallelize existing applications.

Chapter 4. DEVELOPMENT OF MASS CUDA MGPU

This section outlines the approaches and algorithms used to design and implement MASS CUDA for multiple GPUs and extend it for more simulations. First, there is an overview of changes to MASS CUDA architecture and device management. Second, changes to the distributed array – Places – and how it is apportioned over multiple GPU devices is shown. Next, improvements to how Agents are spread over the solution surface, migrate between devices, spawn, and terminate are shown. Finally, there is a walkthrough of a test implementation to demonstrate these new features.

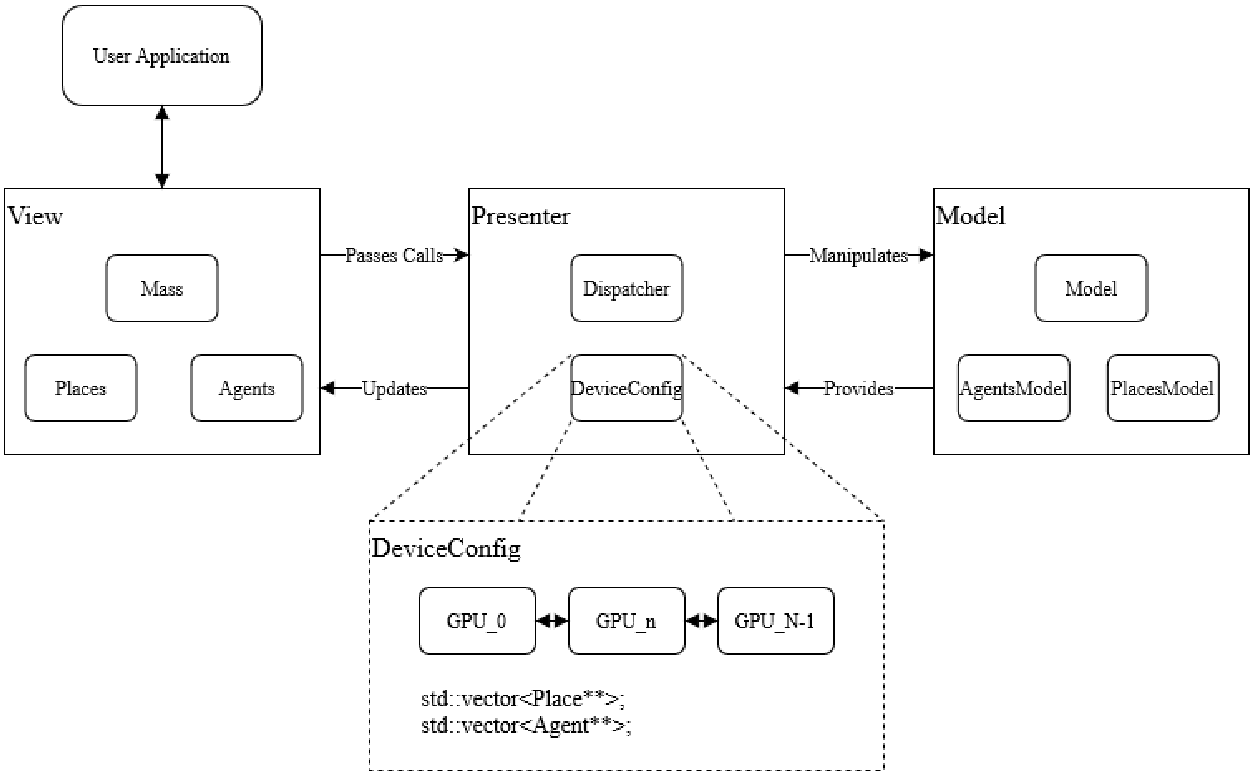


Figure 2 - MASS CUDA MGPU Design Pattern

4.1 MASS CUDA LIBRARY ARCHITECTURE AND DEVICE MANAGEMENT

In MASS CUDA, the presenter controls the simulation after being initialized. A user application instantiates the MASS object and passes data through the view into the Dispatcher object. As Figure 2 shows, the Dispatcher object controls access to the data store objects, presenting them to the view and calling operations on the data when called. At instantiation, the Dispatcher first takes control of the GPU device of the highest CUDA Compute Capability [21] and finds any additional identical devices on the host computer and also takes control of them. To complete initialization, device-to-device direct communication is established using CUDA peering functions between each controlled device after which each device is assigned to an OpenMP thread.

The DeviceConfig object owns the data that resides on the device(s) and the Model object the data that resides on the host. This improved library extends this object rather than implement a DeviceConfig object for each device. To accomplish this, each devices Place, Agent, and associated state objects are allocated to device memory and instantiated with device kernel function calls, these are then stored in vectors on the host whose index in the vector is mapped to the device that owns it. The simulation is then run by calling kernel functions from the host through the MASS API on data stored on the devices. Because Place and Agent objects are stored on the device with their state, the only host-to-device transfers occur as simple parameters for calling functions and the limited data transfers need to ensure each device is synchronized with the other. This latter factor is further minimized as it is accomplished by direct device-to-device memcpy's that are called from within kernel functions.

Further improvements for MASS CUDA MGPU occur primarily in the DeviceConfig and Dispatcher objects and the methods that interact with the simulation data residing on each device. Finally, minor changes to base Place and Agent objects, the View, and Model facilitate hiding multiple device implementation details from the application developer.

4.2 Multiple Device Improvements to MASS Places

Places are the distributed arrays of data that serve as the surface for Agents to travel across. Places may also communicate with neighbor Places and act on resident Agents. To extend MASS CUDA to multiple devices they are first distributed across the multiple devices. CUDA UMA and direct device-to-device communication require multiple device numbers of 2, 4, 8 or 16 devices be deployed in parallel [21].

Place and Agent memory resides in the global memory of its assigned device as shown in Figure 3. The rows of green cells correspond to Places that reside on the neighboring device, while the blue are those Places assigned in its memory. To compute simulations this data is launched a series of kernel functions in parallel on each device. These functions are assigned to do work on individual Place objects and their state, Agent objects and their state, or some combination. All data is stored and accessed in device global memory prior to being loaded into registers for the kernel functions.

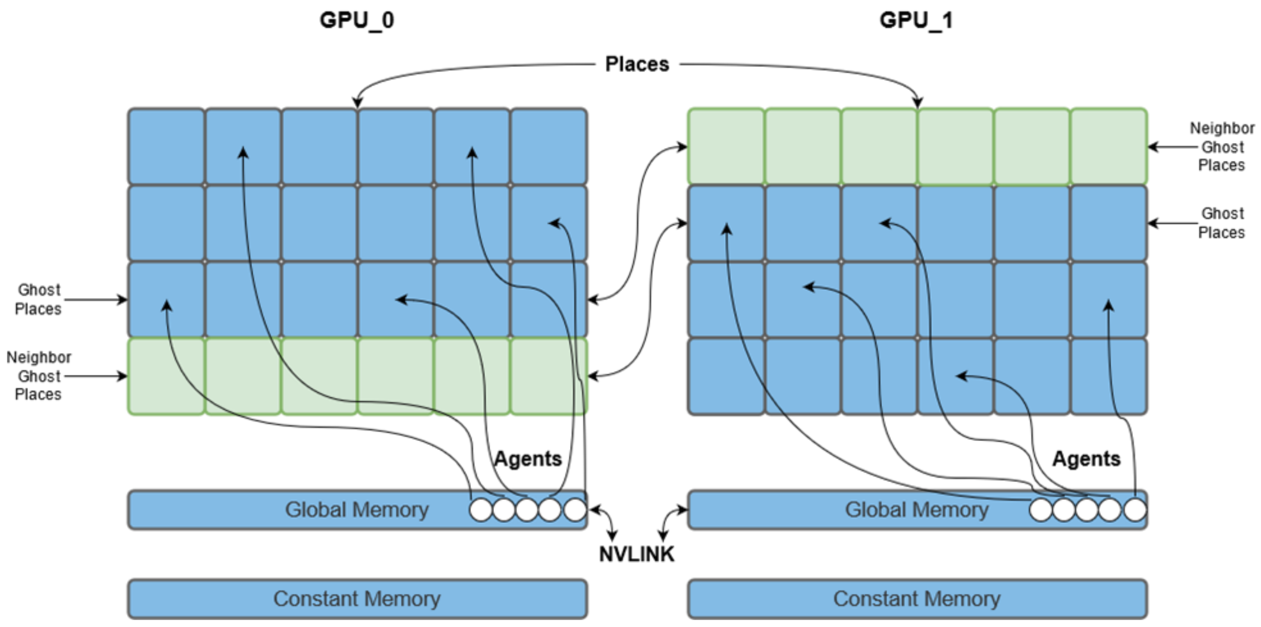


Figure 3 - MASS CUDA MGPU Device Memory Layout

Places must be initialized to an evenly divisible number and are then split amongst the devices according to row major ordering. Places remain in the assigned device’s memory throughout a simulation. Finally, Places are uniquely identified through their index in a one dimensional array, and each has a local index within its device and a global index based on its device’s placement. The one-dimensional array is processed as though it is a two-dimensional space using row-major ordering.

In addition to splitting Places evenly amongst devices, this research expands the size of each devices Place data by adding copies of neighboring border Places to each device. These additional Places, called ghost Places, allow each set of distributed Places to complete operations in parallel on its device [18]. The number of ghost Places on each device is dependent on how far an Agent may travel during one time step multiplied by the dimension of the global Place's space in the row dimension. For example, if an Agent may travel up to three spaces away then the ghost space is equal to three multiplied by the row dimension of the two dimensional space. Ghost Places are along either the top, bottom, or both for devices that occupy the middle ranks of a multi-GPU configuration. Ghost Places do not have methods called on their data members. They exist to allow border Places on each device access to data from Place's on the neighboring device. These ghost Places are updated after each Place method call by copying the state data from the corresponding neighbor Places.

Each ghost Places memory state is copied from the neighbor device after each call to a Place method. These copies are kept on the host in tuples of <Place**, State*>. The device-to-device copy method is shown in Listing 1. Place states from even devices are copied to the next higher ranked device (lines 3-26) and then lower ranked device (lines 27-50). After each copy of Place states a kernel function is called to clear invalid pointers. First for the higher ranked devices (lines 22-24) and then lower ranked devices (lines 46-48).

Listing 1: Copy Ghost Places Method

```
1.  void DeviceConfig::copyGhostPlaces(int handle, int stateSize) {
2.    dim3* pDims = getPlacesThreadBlockDims(handle);
3.   for (int i = 0; i < activeDevices.size(); i+=2) {
4.      // copy right
5.      cudaSetDevice(activeDevices.at(i + 1));
```

```
    CATCH(cudaMemcpyAsync(
      devPlacesMap[handle].topNeighborGhosts.at(i + 1).second,
      devPlacesMap[handle].bottomGhosts.at(i).second,
      MAX_AGENT_TRAVEL * getDimSize()[0] * stateSize,
      cudaMemcpyDefault));
    cleanGhostPointers<<<pDims[0], pDims[1]>>>(
      devPlacesMap[handle].topNeighborGhosts.at(i + 1).first,
      MAX_AGENT_TRAVEL * getDimSize()[0]);
    if (i != 0) {
      // copy left
      cudaSetDevice(activeDevices.at(i - 1));
      CATCH(cudaMemcpyAsync(
        devPlacesMap[handle].bottomNeighborGhosts.at(i - 1).second,
        devPlacesMap[handle].topGhosts.at(i).second,
        MAX_AGENT_TRAVEL * getDimSize()[0] * stateSize,
        cudaMemcpyDefault));
      cleanGhostPointers<<<pDims[0], pDims[1]>>>(
        devPlacesMap[handle].bottomNeighborGhosts.at(i - 1).first,
        MAX_AGENT_TRAVEL * getDimSize()[0]);
    }
  }
  for (int i = 1; i < activeDevices.size(); i+=2) {
    // copy left
    cudaSetDevice(activeDevices.at(i - 1));
     CATCH(cudaMemcpyAsync(
      devPlacesMap[handle].bottomNeighborGhosts.at(i - 1).second,
      devPlacesMap[handle].topGhosts.at(i).second,
      MAX_AGENT_TRAVEL * getDimSize()[0] * stateSize,
            cudaMemcpyDefault));
        cleanGhostPointers<<<pDims[0], pDims[1]>>>(
            devPlacesMap[handle].bottomNeighborGhosts.at(i - 1).first,
            MAX_AGENT_TRAVEL * getDimSize()[0]);
        if (i != activeDevices.size() - 1) {
            // copy right
            cudaSetDevice(activeDevices.at(i + 1));
                CATCH(cudaMemcpyAsync(
                devPlacesMap[handle].topNeighborGhosts.at(i + 1).second,
                devPlacesMap[handle].topGhosts.at(i).second,
                MAX_AGENT_TRAVEL * getDimSize()[0] * stateSize,
                cudaMemcpyDefault));
            cleanGhostPointers<<<pDims[0], pDims[1]>>>(
                devPlacesMap[handle].topNeighborGhosts.at(i + 1).first,
                MAX_AGENT_TRAVEL * getDimSize()[0]);
        }
    }
}
```

Changes to Place->callAll() and Place->exchangeAll() methods that allow for multiple devices finish improvements for MASS Places. These functions are implemented to loop over each device and call the respective kernel function on its Places. After each device has completed its kernel function ghost Place states are copied to neighboring devices.

4.3 Multiple Device Improvements to MASS Agents

This research's Agent improvement begins with refactoring for multiple devices of Agent initialization, method calls, migration, and spawn. Next, Agent termination is implemented to compact alive Agents in device allocated memory. Finally, long distance migration is implemented to allow Agents to migrate between Places that are not immediate neighbors. Agents are initialized on each device either randomly or by developer provided specification. At instantiation, the application developer may decide how to allocate Agents across Places and if not this implementation first randomly generates indices over the entire Place space, then sorts them, assigns them to the corresponding device, and allocates Agents at them on the respective device.

Agents interact with Places and other Agents through a callAll() method that takes an identifier to a simulation defined kernel function. These are performed in parallel on all Agents across all devices. As with the Place->callAll( ) function(s), Agent->callAll( ) was refactored to have the kernel function called on each device's share of the simulation's memory in parallel.

4.3.1 *Agent Migration*

This research expands Agent->manageAll() beginning with Agent migration. This algorithm is implemented by first looping over the devices to resolve conflicts between Agents wanting to migrate to the same Place and then updates their locations. At this step Agents may travel onto ghost Places. After Agents on all devices have migrated locally we begin the steps of the algorithm to move any Agents on ghost Places to their neighbor device as illustrated in Figure 4. This step has two kernel functions – one to move Agents up and another to move them down. As this implementation uses row-major ordering the Place objects are split amongst devices by row.

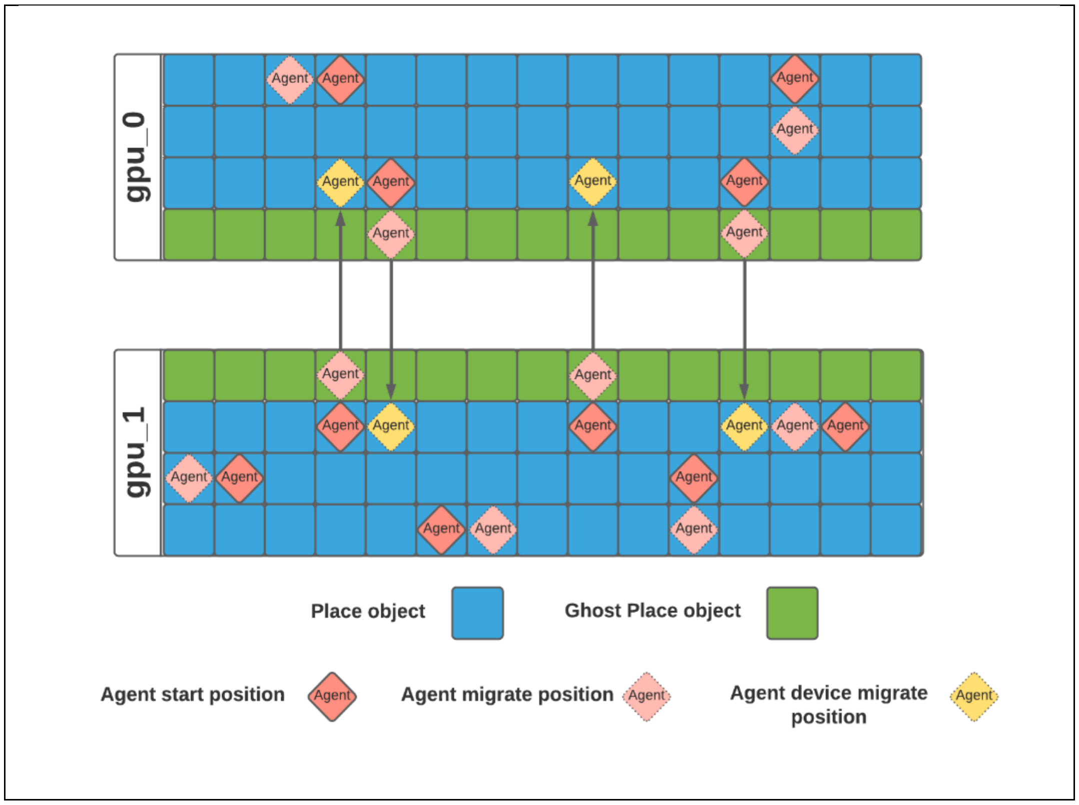


Figure 4 - Agent migration across devices

The up migration is shown in Listing 2. This kernel function uses a CUDA atomicAdd() function to ensure there are no race conditions between Agents when they are being added to the neighbor devices Agent array. After getting a legal array location to move the Agent into (lines 10-12), the Agent is then marked as having traveled (line 13), the Agent state is copied to that location (lines 14-15), and the Agent on the source device is terminated (line 17).

Listing 2: Move Agents kernel function

```
1.  __global__ void moveAgentsUpKernel(Agent **src_agent_ptrs,
2.          Agent **dest_agent_ptrs, AgentState *src_agent_state,
3.          AgentState *dest_agent_state, Place **src_place_ptrs,
4.          Place **dest_place_ptrs, int device, int placesStride,
5.          int ghostPlaces, int ghostPlaceMult, int nAgentsDevSrc,
6.          int *nAgentsDevDest, int stateSize) {
```

```
7.
8.      int idx = getGlobalIdx_1D_1D();
9.      if (idx < nAgentsDevSrc) {
10.         int place_index = src_agent_ptrs[idx]->getPlaceIndex();
11.         if (place_index < device * placesStride) {
12.             int neighborIdx = atomicAdd(nAgentsDevDest, 1);
13.             src_agent_ptrs[idx]->setTraveled(true);
14.             memcpy(&(dest_agent_state[neighborIdx]),
15.                 (&(src_agent_state[idx])), stateSize);
16.
17.             src_agent_ptrs[idx]->terminateAgent();
18.           }
19.     }
20. }
```

After Agents are migrated globally the pointers of each Agent are updated to their migrated Place location with a separate kernel function as shown in Listing 3. This function works on Agents that have just traveled to a new device by first attempting to add the Agent to that Places Agent array and if it can the Place is also set on the Agent (lines 6-14). If the Agent cannot be added to the Place it is terminated (line 18). Finally, the agent migration algorithm finishes by updating the total number of alive Agents on each device and globally.

Listing 3: Update globally migrated Agents

```
1.   __global__ void updateAgentPointersMovingUp(Place** placePtrs,
2.          Agent** agentPtrs, int qty, int placesStride, int ghostPlaces,
3.          int ghostSpaceMult, int device) {
4.      int idx = getGlobalIdx_1D_1D();
```

```
    if (idx < qty) {
        if (agentPtrs[idx]->isAlive() && agentPtrs[idx]->isTraveled()) {
            agentPtrs[idx]->setTraveled(false);
            int placePtrIdx = agentPtrs[idx]->getPlaceIndex() -
                (device * placesStride) +
                (ghostPlaces + ghostPlaces * ghostSpaceMult);

            if (placePtrs[placePtrIdx]->addAgent(agentPtrs[idx])) {
                agentPtrs[idx]->setPlace(placePtrs[placePtrIdx]);
                return;
            }
            // No home found
            agentPtrs[idx]->terminateGhostAgent();
        }
    }
}
```

4.3.2 *Agent Spawn*

The Agents spawn algorithm is implemented to process each devices Agent array to first find if any have new child Agents to spawn for the simulation. If they do, a CUDA provided atomicAdd() function is called to get the starting index for these Agents and provide a starting index for the next Agent spawning in parallel.

Agent termination is initiated in the MASS application developer’s code by calling the MASS library provided terminate Agent function that is applied to all Agents in a callAll() function. This function changes an individual Agents state to not alive and removes it from the Place it resides. At the next call to Agents->manageAll(), the application-wide terminate Agents function is called on all Agents and this begins the algorithm to remove the Agents from the active memory space.

4.3.3 *Agent Termination*

This research provides a termination algorithm that leverages ideas from classic mark and sweep algorithms to compact the active memory space without using additional memory for Agent objects [22]. A key decision marker for this algorithm is its applicability for all models that could be developed in MASS CUDA. For example, if this research used only marked Agents for reuse this would inhibit processing gains of compressed data if many Agents are terminated and not many are spawned. Further, to limit the use of synchronization variables this research breaks operations into separate kernel functions when possible.

Library users set when the termination algorithm will run by first setting a global parameter for the percentage of Agent memory at which to run. Figure 5 provides an illustration of the compaction algorithm. The first step of the algorithm is to set an array of flags indicating the positions of alive Agents. Next, this flag array is used to get a complete count of the alive Agents in the entire Agent memory space using CUDA Unbounded (CUB) device reduce functions [23]. This count is then set as the pivot point where all alive Agents will be located at indices less than it. Next, the count of dead Agents at indices less than the pivot point is found using the same CUB functions and is used to set an array for the alive Agents at indices greater than the pivot. Following this, the CUDA provided atomicAdd() function is used to set these indices into the array for Agent indices greater than the pivot. Finally, the array is accessed to copy Agents into the dead Agent spaces at indices less than the pivot. To complete the termination and Agent array compaction algorithms the pivot point is set as the current count of alive Agents on each device in the simulation.

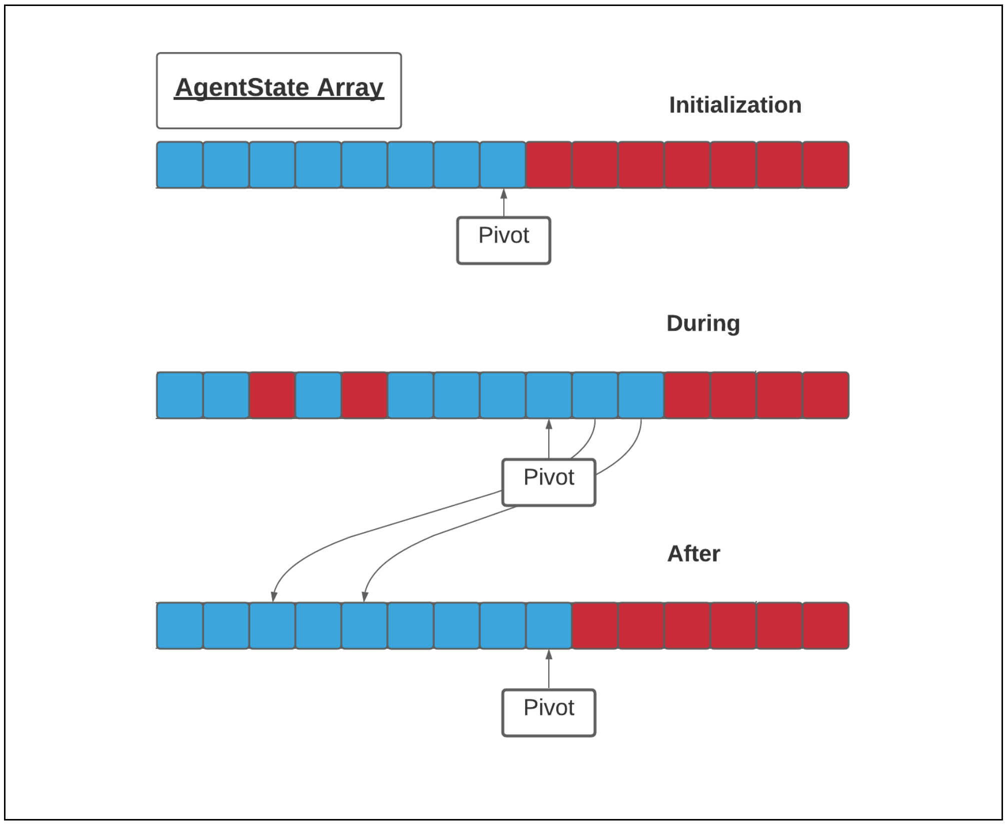


Figure 5 - Array Compaction Algorithm

To facilitate long distance communication between Places, current MASS implementations use the same data structure and process used for neighbors. This is problematic in MASS CUDA, because MASS C++ and Java can use basic data structures that change size during a simulation whereas MASS CUDA allocates this size at compile time. Therefore, to facilitate Place to Place long distance communication either the neighbor array needs to be large, or a different algorithm needs to be implemented for addressing each Place in the neighbor array. This research instead uses Agents to communicate over long distances as the implementation that addresses Agents onto Places is not as strict as the Place neighbor array.

This research adds a Boolean variable to each Agents state that signifies if it is a candidate for long distance migration, and a function for application developers to call that performs it. The long distance migration algorithm runs at the start of the overall Agent migration function before the Agent migration that proceeds one Place at each time step of a simulation.

To begin long distance migration a set of nested loops is called with a kernel function that copies the long distance migrating Agent's state to the device that its destination Place exists. Following this, the devices are looped over, and each Agent is set to its destination Place. If the Place's Agent array is full the migrating Agent is terminated. This requires MASS library users set the size of each Places Agent array accordingly.

The current MASS CUDA implementation uses one manageAll() function that first calls terminateAgents(), then migrateAgents(), and finally spawnAgents(). This works for previous simulations using MASS CUDA but may not for new implementations. For this reason, this research exposes variations of the Agent->manageAll() function with different orderings of the three base functions as well as allowing the individual terminate, migrate, and spawn functions to be called through the MASS API.

4.4 NEURAL SIMULATION: A SIMPLIFIED IMPLEMENTATION OF BRAINGRID

The MASS CUDA MGPU BrainGrid implementation is a pared down neural simulation using the core principles of the BrainGrid Framework developed by Dr. Michael Stiber and his research group [24]. Dr. Munehiro Fukuda adapted this into an algorithm for development using the MASS framework that is shown in Appendix A. Finally, a previously developed version of BrainGrid with MASS C++ was referenced, and this research contains some algorithmic and naming similarities [25].

To set-up the simulation space Place objects are extended by a Neuron class that can be either a soma (cell body of a neuron) or an empty neural space where neural signal actors and receptors may travel and connect as during the simulation. The Agent class is extended to a Growing End class that are instantiated as sets of Axons (that may change into a Synapse) and Dendrites. One of each Growing End type is assigned to each Neuron that is a soma. Figure 6 shows a simplified illustration of the simulation space.

As the specification outlines, Axons and Dendrites grow from a soma at random times in the simulation. These times are set during simulation set-up using the a pseudo-random number generator. One Axon will grow out from a soma and up to seven Dendrites (one for each neighbor that can be grown into) to its Moore neighbors. When a Dendrite is grown and there is another to be grown later, a new Dendrite Agent is spawned at the soma to wait for its turn. This happens at the top of the simulation loop where first Axons and Dendrites are checked if it is there time to spawn and may be set to growing. Next, Dendrite growth direction is established and Axons then Dendrites are grown. Next, calls to Agents->manageAll() complete the growing Agents migration. Next, each grown from soma Dendrite and Synapse are set as the one of each type that may remain on an empty Neuron and if each is occupied and not of the same soma each Growing End is marked as not growing and the Neuron as occupied.

The next set of steps govern the growth of Axons, Synapses, and Dendrites outside of soma Neurons. First the Axon is grown and then checked if it should change to a Synapse. Next, Dendrites and Synapses are branched. These functions also check for growth and spawn an Agent for each branch. Next, the parent Dendrite/Synapse to grow are migrated followed by the branches. Finally, each empty Neurons potential next Agent array is pared down to at most one Dendrite and Synapse that are set as connected if both are present and not of the same soma.

When a Dendrite and Synapse connection is made, the Neuron upon which they made a connection is marked, the Synapse takes the Soma information of the Dendrite, and the Dendrite is terminated.

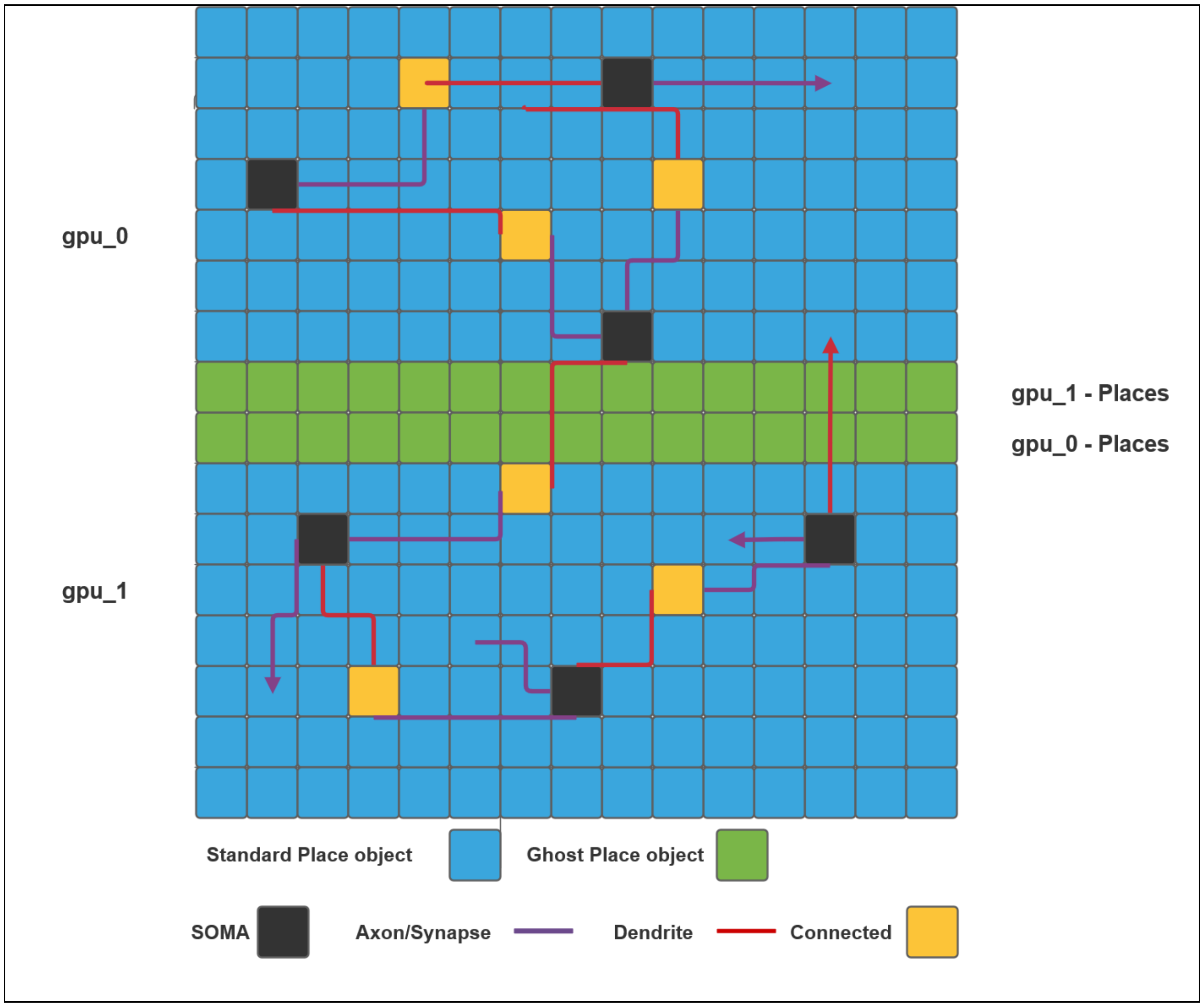


Figure 6 - BrainGrid

The final few steps of the BrainGrid simulation loop collect and transmit signals and set any simulation parameters for the next simulation loop. The collection and communication of signals algorithm uses long distance migration to take a signal from the Synapse Soma and deliver it to the Dendrite Soma. The signal received at the Dendrite Soma is then used in the next simulation step to set its Synapse signals output. The final steps of the simulation reset Agent and Place parameters that maintain synchronization during a single time step.

This research implements three primary factors for the MASS CUDA library. The Agent termination algorithm, ghost space management, and long distance Agent migration allow MASS CUDA to run simulations over multiple devices and expands the types of models it can compute through extension and refactoring of the library to facilitate implementation of the BrainGrid simulation.

Chapter 5. EVALUATION OF RESULTS

This research resulted in improvements to the MASS CUDA framework by extending the size of simulations it can process. Both the MASS CUDA and MASS CUDA MGPU implementation were run on the computing resources outlined in this papers background section.

5.1 SUGARSCAPE SIMULATION

MASS CUDA MGPU was run using a single host thread, with OpenMP host multithreading using one thread per GPU, and with OpenMP host multithreading combined with garbage collection of terminated Agents. Figure 7 shows these performance comparisons. Simulation sizes above 2000 Place's causes MASS CUDA to run out of memory resources while MASS CUDA MGPU successfully processes simulations up to 3000 Place's. The number of Agent objects in the simulation is a factor of the Place objects – one-fifth of the number of Place objects are instantiated as alive and one-fifth as asleep to allow for spawning new Agent objects. Therefore, the MASS CUDA MGPU implementation of sugarscape processes a simulation over

9 million Place objects with 3.6 million Agent objects, a 125-percent increase in simulation size over MASS CUDA single GPU.

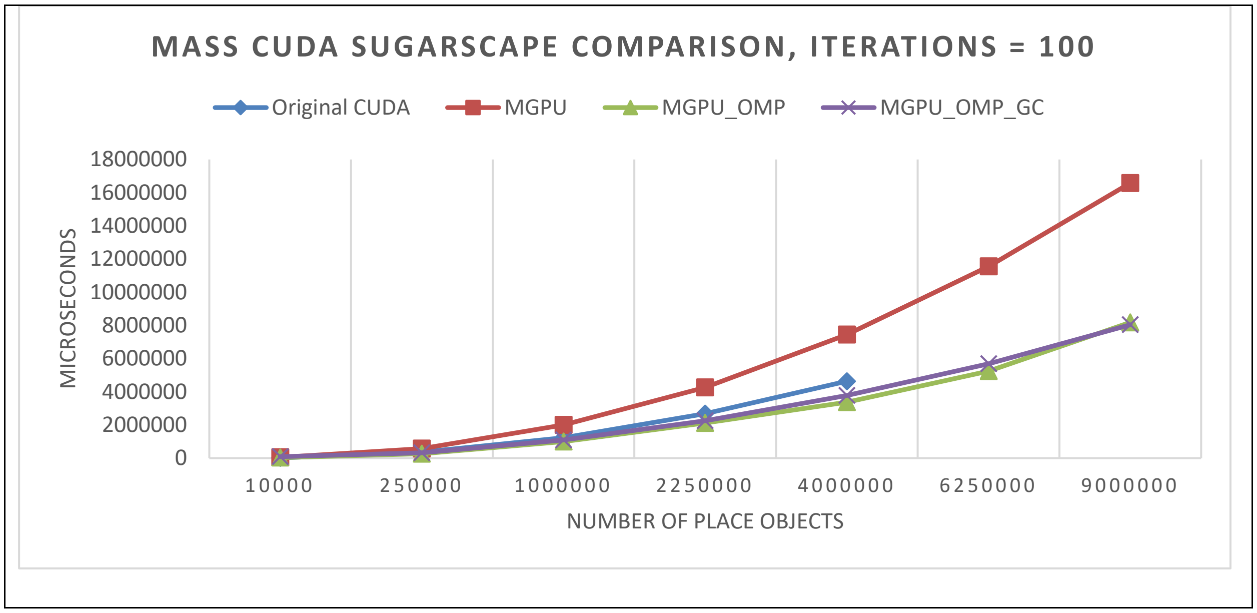


Figure 7 - MASS CUDA MGPU vs. MASS CUDA

The sugarscape simulation processing time comparison is shown in Figure 7. This comparison is between the original MASS CUDA single GPU (shown in blue); MASS CUDA MGPU (shown in red); MASS CUDA MGPU using OpenMP host multi-threading (shown in green); and MASS CUDA MGPU using OpenMP host multi-threading, and this researches Agent array compaction function (shown in gray). This graph was generated using the average of five timings for each comparison computation on the seven progressively larger simulation spaces shown along the x-axis of Figure 7. Each simulation is for 100 iterations and timings are collected from the start of the first iteration to the end of the last iteration. These timings do not include the host-to-device memory transfers that instantiate objects on the device with which to run the simulation.

The original MASS CUDA implementation is faster up to Place object size of 10,000 after which both versions of MASS CUDA MGPU with OpenMP eclipse its performance . MASS CUDA

MGPU single-threaded is slower for all Place object sizes. Finally, at the largest number of Place object size of this comparison, the implementation of MASS CUDA MGPU with OpenMP and Agents array compaction processes fastest, by approximately 14,000 microseconds.

MASS CUDA MGPU is slow when compared to original MASS CUDA due to the extra processing and coordination required to split the simulation space across multiple devices. Additional processing is needed to synchronize and copy ghost place data after any operation on Places or Agents. OpenMP speeds this up considerably because this project is aligned to the API that is exposed to application developers and hides all CUDA programming details. While this allows developers to write simple kernel functions that are passed into the library for execution, it does so by encasing all of the synchronization code within the API and calling it regularly. OpenMP massively improved performance because each thread calls elements of each API method call – callAll(), migrateAll(), exchangeAll() – in parallel from the host to each device rather than each device after the other from a loop. The gains are in part due to synchronization requirements of the library, many of these synchronization needs mirror those of multiple device synchronization resulting in free gains when using host-based multithreading to call kernel functions on each device.

At the at the largest Place object size this research shows best processing time for the sugarscape simulation from MASS CUDA OpenMP with garbage collection. This is occurs because the array compaction algorithm removes Agents that are not going to have work done on them from the set of Agents that will. This results in less threads needing launched per kernel function to process data.

Sugarscape Agent locations after one-hundred iterations in a 100 x 100 Place object simulation space is shown computed by the original MASS CUDA implementation in Figure 8 and as computed by this research in Figure 9. Sugarscape is implemented to first attempt to move an Agent one, then two, then three Places to the right and if those fail, one, then two, then three Places up. Agent’s stop at the edges. In the outputs, this relationship shows at the top right where Agents go right and pile up in the top right corner vertically at an approximate 3-to-1 ratio. These two implementations roughly match with the difference due to how Agents are handled at destination Places when traveling between devices. At the end of the simulations, original MASS CUDA has 531 alive Agents while MASS CUDA MGPU has 492 alive Agents. This difference is because MASS CUDA MGPU device-traveled Agents are deleted at the destination if they migrate to a Place with an Agent on it. In this research’s sugarscape algorithm, Agents try to migrate right or up into Places with more sugar than waste. As part migration, Places choose the lowest ranked Agent that tries to migrate onto it. This step is skipped in MASS CUDA MGPU and Agents that traveled into a neighbors ghost Place are copied to that device and deleted if an Agent is present.

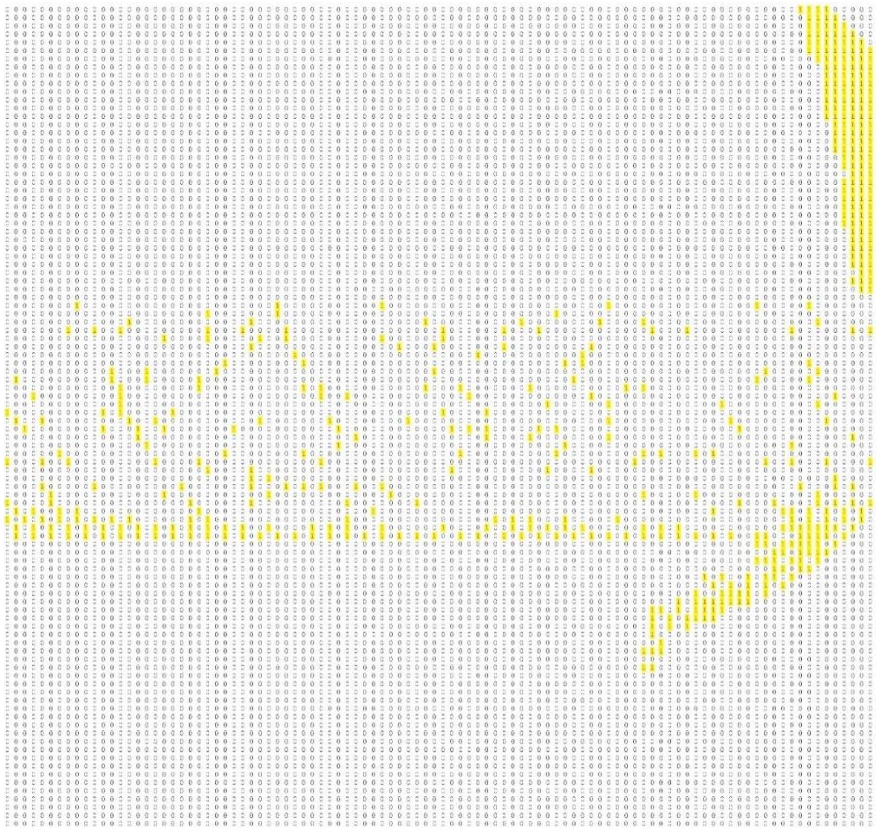


Figure 8 - Original MASS CUDA Sugarscape Results

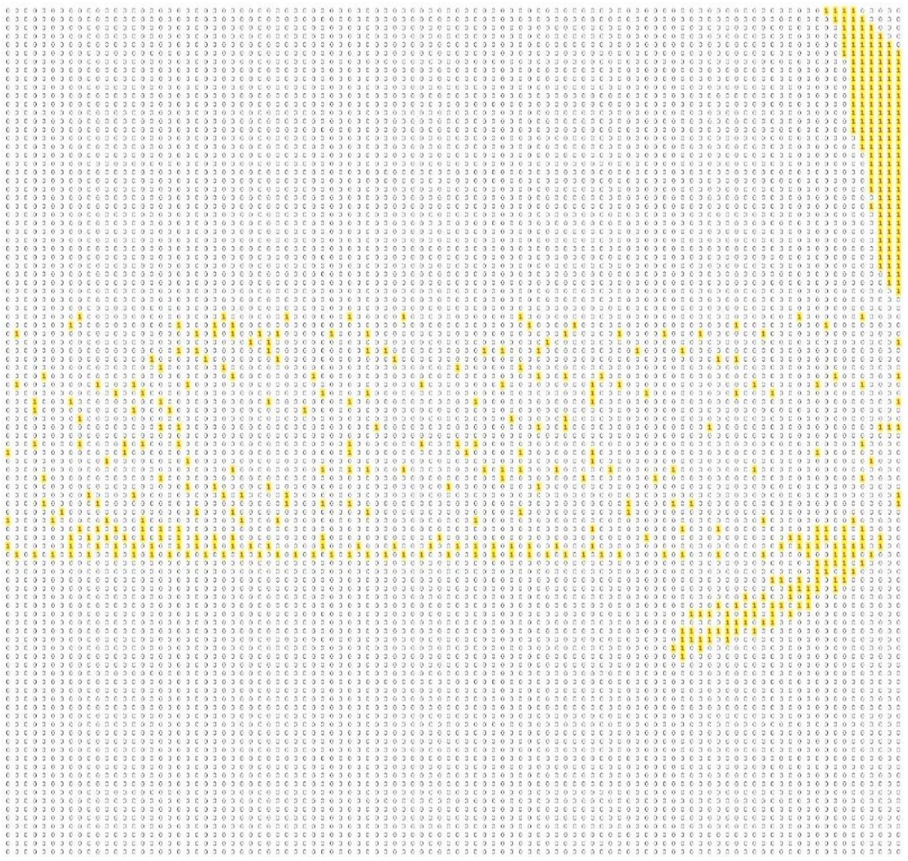


Figure 9 - MASS CUDA MGPU Sugarscape Results

5.2 BrainGrid Simulation

Although the BrainGrid simulation remains incomplete it resulted in several improvements to the MASS CUDA library. First, as the BrainGrid simulation algorithm requires multiple Agents to reside on a Place the library was refactored to improve adding Agents to a Place and removing Agents from a Place by making them thread safe. This was not a concern with original MASS CUDA as the implemented simulations either used no Agents or only allowed one Agent onto a Place. Next, CUDA programs require all device memory for a program to be pre-allocated. One of the challenges presented by this memory model is with Place to Place communication. The number of Place neighbors any Place can communicate with must be set at instantiation. Typically this would be the immediate 4-8 neighbors surrounding a Place, but some implementations require further distance communication. To get around this challenge and allow devices to communicate across long distances this research implements a long distance migration algorithm. This algorithm adds a parameter to each Agent object to tell if it is to be long-distance migrated. If so, a nested set of loops is called on and between each device to copy the Agents state and link to the migrated Place. Finally, this research implemented the beginnings of an improved migration algorithm to migrate Agents before any are accepted for that simulation iteration, run the Agent acceptance algorithm, and if an Agent is accepted it is copied back to its source with a termination message.

Chapter 6. CONCLUSION

This research was completed using two NVIDIA RTX 2080 Super GPUs connected via NVLink on a desktop running Ubuntu 18.04. These two GPUs were launched in parallel using OpenMP multithreading to manage each independently on their respective sets of data.

To allow these devices to coherently manage their respective data various extensions to the MASS CUDA library were developed. First, the distributed array, called Places, was extended to first split them evenly amongst the devices. Next, ghost spacing of some Places located on the neighboring device(s) was added; with these Places updated after each call on them by their respective device.

Improvements to agents, called Agents in MASS, start with extensions to Agent travel that allow them to migrate onto each devices Places in parallel, the ghost Places that correspond to the neighbor device(s), and, to allow them to migrate to neighbor devices. A further extension of the MASS library performs garbage collection of dead Agents and compaction of the Agents memory space. This extension allows simulations where the number of alive Agents is dynamic throughout a model’s simulation runtime and BrainGrid, a simplified neural simulation application, demonstrates these new Agent features.

These improvements to the MASS CUDA library show in the existing Sugarscape test model and in the newly developed BrainGrid simulation. First, the size of model simulations that can be run with MASS CUDA is increased by 125% using MASS CUDA MGPU on two devices. Second, MASS MGPU using OpenMP with no Agent termination processes Sugarscape faster than MASS CUDA by approximately 30%, while MASS CUDA MGPU with OpenMP and Agent termination eclipses its performance with Agent termination at the largest model size by approximately 2%.

While this research enables MASS CUDA over multiple devices and, as a result, increases the size of simulations, opportunities for further improve are aplenty. The Agent garbage collection algorithm should be improved to remain resident on devices at all stages, decrease the amount of additional memory used, and utilize shared memory and warp level instructions to speed-up its operation.

Currently, MASS CUDA is managed with a single host thread and uses only the default CUDA stream of each device. While it is prudent to use only one CUDA stream per device due to the architecture of MASS CUDA, its role as a library for implementations, and potential issues of data synchronization, host multi-threading will improve memory allocation time. CUDA memory operations block on the host and this prevents memory allocations from happening in parallel on each device.

A final consideration for further improvement considers development of a library-wide device memory management framework. MASS CUDA allocates device memory only at instantiation. Memory allocation is slow and MASS CUDA simulations are not indeterminate – we know a simulation’s need at instantiation – therefore there is little use for memory allocation after initialization. However, many simulations do require dynamic memory during a simulation to compute efficiently. For example, without memory compaction the library may call functions on objects in memory that are not alive which results in threads being assigned to objects doing no work, wasting compute resources. Similarly, memory allocations at instantiation may result in dead blocks due to how CUDA handles sizing and assigns chunks of memory. Library-wide memory management can manage both these concerns without blocking operations, meaning that a simulation will continue to process while memory management is occurring. This is not possible using the CUDA library to manage memory during device runtime.

This research shows that MASS CUDA can run simulations of greater size and complexity. The further features outlined may enable MASS CUDA to process even larger simulations and be considered by non-computing researchers seeking a library to implement their ABM simulations.

Chapter 7. REFERENCES

[1] C. W. Weimer, J. O. Miller, and R. R. Hill, “Agent-based modeling: An introduction and primer,” in *2016 Winter Simulation Conference (WSC)*, Washington, DC, USA, Dec. 2016, pp. 65–79. doi: 10.1109/WSC.2016.7822080.

[2] L. S. Schulman and P. E. Seiden, “Statistical mechanics of a dynamical system based on Conway’s game of Life,” *J Stat Phys*, vol. 19, no. 3, pp. 293–314, Sep. 1978, doi: 10.1007/BF01011727.

[3] E. Bonabeau, “Agent-based modeling: Methods and techniques for simulating human systems,” *Proceedings of the National Academy of Sciences*, vol. 99, no. Supplement 3, pp. 7280–7287, May 2002, doi: 10.1073/pnas.082080899.

[4] M. Tracy, M. Cerdá, and K. M. Keyes, “Agent-Based Modeling in Public Health: Current Applications and Future Directions,” *Annu. Rev. Public Health*, vol. 39, no. 1, pp. 77–94, Apr. 2018, doi: 10.1146/annurev-publhealth-040617-014317.

[5] E. R. Smith and F. R. Conrey, “Agent-Based Modeling: A New Approach for Theory Building in Social Psychology,” *Pers Soc Psychol Rev*, vol. 11, no. 1, pp. 87–104, Feb. 2007, doi: 10.1177/1088868306294789.

[6] G. O. Kagho, M. Balac, and K. W. Axhausen, “Agent-Based Models in Transport Planning: Current State, Issues, and Expectations,” *Procedia Computer Science*, vol. 170, pp. 726–732, 2020, doi: 10.1016/j.procs.2020.03.164.

[7] C. M. Macal and M. J. North, “Agent-based modeling and simulation,” in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, Austin, TX, USA, Dec. 2009, pp. 86–98. doi: 10.1109/WSC.2009.5429318.

[8] A. Geist and D. A. Reed, “A survey of high-performance computing scaling challenges,” *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 104–113, Jan. 2017, doi: 10.1177/1094342015597083.

[9] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc, “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions,” *PROCEEDINGS OF THE IEEE*, vol. 87, no. 4, p. 11, 1999.

[10] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll, “A Survey on Agent-based Simulation Using Hardware Accelerators,” *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–35, Feb. 2019, doi: 10.1145/3291048.

[11] Kinghorn, D. (n.d.). P2P peer-to-peer on NVIDIA RTX 2080Ti vs GTX 1080Ti GPUs. Retrieved July 20, 2020, from https://www.pugetsystems.com/labs/hpc/P2P-peer-to-peer-on-NVIDIA-RTX-2080Ti-vs-GTX-1080Ti-GPUs-1331/

[12] P. Richmond, D. Walker, S. Coakley, and D. Romano, “High performance cellular level agent-based simulation with FLAME for the GPU,” *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, May 2010, doi: 10.1093/bib/bbp073.

[13] R. Axtell, R. Axelrod, J. M. Epstein, and M. D. Cohen, “Aligning simulation models: A case study and results,” *Comput Math Organiz Theor*, vol. 1, no. 2, pp. 123–141, Feb. 1996, doi: 10.1007/BF01299065.

[14] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, B. Herrmann, and L. Philippe, “MCMAS: A Toolkit to Benefit from Many-Core Architecure in Agent-Based Simulation,” in *Euro-Par 2013: Parallel Processing Workshops*, vol. 8374, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 544–554. doi: 10.1007/978-3-642-54420-0_53.

[15] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Mar. 1998, doi: 10.1109/99.660313.

[16] T. Ojiru, “Implementing the Multi-agent spatial simulation (MASS) library on the Graphics Processor Unit,” University of Washington, Seattle, WA, 2012.

[17] R. Jordan, “Multi-GPU MASS Library,” University of Washington, Seattle, WA, 2012.

[18] F. B. Kjolstad and M. Snir, “Ghost Cell Pattern,” in *Proceedings of the 2010 Workshop on Parallel Programming Patterns - ParaPLoP ’10*, Carefree, Arizona, 2010, pp. 1–9. doi: 10.1145/1953611.1953615.

[19] Nathaniel B. Hart, “MASS CUDA: Abstracting Many Core Programming From Agent Based Modeling Frameworks,” University of Washington, Seattle, WA, 2015.

[20] L. Kosiachenko, N. Hart, and M. Fukuda, “MASS CUDA: A General GPU Parallelization Framework for Agent-Based Models,” in *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems: The PAAMS Collection*, vol. 11523, Y. Demazeau, E. Matson, J. M. Corchado, and F. De la Prieta, Eds. Cham: Springer International Publishing, 2019, pp. 139–152. doi: 10.1007/978-3-030-24209-1_12.

[21] "CUDA C++ Programming Guide." NVIDIA, Apr. 20, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[22] S. Stanchina and M. Meyer, "Mark-sweep or copying?: a 'best of both worlds' algorithm and a hardware-supported real-time implementation," in *Proceedings of the 6th international symposium on Memory management - ISMM '07*, Montreal, Quebec, Canada, 2007, p. 173. doi: 10.1145/1296907.1296928.

[23] Duane Merrill, NVIDIA Research, "CUDA Unbounded," *CUB Documentation*, 2011-2016. https://nvlabs.github.io/cub/ (accessed May 01, 2021).

[24] M. Stiber, F. Kawasaki, D. B. Davis, H. U. Asuncion, J. Y.-H. Lee, and D. Boyer, "BrainGrid+Workbench: High-performance/high-quality neural simulation," in *2017 International Joint Conference on Neural Networks (IJCNN)*, Anchorage, AK, USA, May 2017, pp. 2469–2476. doi: 10.1109/IJCNN.2017.7966156.

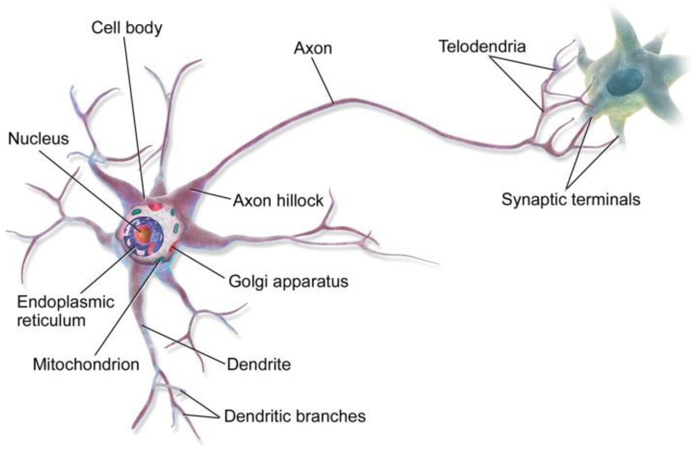
[25] Panther, S. (2020, June 13). BrainGrid. Retrieved February 10, 2021, from https://bitbucket.org/mass_application_developers/mass_cpp_appl/src/SARAH-MASS-BRAIN-GRID/Benchmarks/MASS/MASS_BrainGrid/

Chapter 8. APPENDIX A

Brain Grid : Self-Organizing Neural Network

by Dr. Fukuda

https://www.youtube.com/watch?v=il2uc-ZUZQ4 Neuron:



(1) Parts

a. Soma: a cell body

b. Dendrite: signal receptors

c. Axon: a signal cable. Just one cable extended from each neuron

d. Synaptic terminals: signal actors

(2) Type

a. Excitatory neuron: amplify a signal

b. Inhibitory neuron: reduce a signal

c. Neutral neuron: convey a signal

Basic Simulation:

(1) An S x S simulation space

a. Each cell can become:

i Excitatory neuron: E%

ii Inhibitory neuron: I%

iii Neutral neuron: N%

iv Space: 100 – (E + I + N) %

(2) For each cell

a. Neuron:

i. Create an axon in a random direction once at a random time unit.

ii. Create up to 7 dendrites, each in a different direction at a random time unit.

iii. If it is excitatory, a signal (whose value is 1.0) is activated with A%

iv. If a signal arrives

1. Excitatory neuron amplifies it to M% and forwards it to the axon.
2. Inhibitory neuron reduces it to M% and forwards it to the axon.
3. Neutral neuron forwards it to the axon.

b. Axon:

i. Continuous growing mode with G%

1. Grow the same direction or +/- 45 degrees at each time unit.

ii. Switching to generation mode of synaptic terminals with 100 – G%

1. Repeat branching up to R times.
    a. Branch may happen B%
    b. Each branch goes to +/- 45 degrees
2. Keep growing (even upon a branch) at each time unit
3. Stop
    a. When encountering a dendrite
    b. Keep growing K times after the last branch
    c. With S%

iii. Forwarding a signal toward synaptic terminals if $\sum$ signals received by this neuron’s dendrites during this time step is over or equal to T

c. Dendrite:

i. Repeat branching up to R times.
1. Branch may happen B% but may not happen when starting from a neuron
2. Each branch goes to +/- 45 degrees

ii. Keep growing (even upon a branch) at each time unit

iii. Stop
1. When encountering a synaptic terminals
2. Keep growing K times after the last branch
3. With S%

iv. Receive a signal from the corresponding synaptic terminal and forward it to the neuron.

Default Values:

S (size)	100	G (growing mode)	90%
E (Excitatory)	10%	R (Repetitive branches)	3
I (Inhibitory)	10%	B (Branch possibility)	33%
N (Neutral)	10%	K (branch growth)	5 times
A (Activating signal)	10%	S (stop of branch growth)	10%
M (Signal modulation)	10%	T (threshold signal value)	0.4