The University of Washington Bothell
CSS-700: Masters Thesis - Term Report

Parallelization of Agent-based Models over Multi-GPUs

Brian Luger
Professor: Dr. Munehiro Fukuda

16 March 2023

Introduction

The purpose of this paper is to provide a summary of the progress made towards my thesis work during Winter quarter, 2023. The work consisted of finishing the multi-GPU agents implementation and porting SugarScape over to using the new multi-GPU library. In addition to this, some progress was made towards completing my thesis paper and time was devoted towards fixing bugs and supporting the single-GPU version of the MASS CUDA framework.

Background

My thesis work involves scaling the MASS CUDA Agent-based Modeling (ABM) framework from a single GPU to multiple GPUs. The impetus of this work is to enable the execution of larger, more computationally bound, simulations by way of distributing memory and compute across multiple GPU devices. Memory is distributed by partitioning simulation state across the multiple GPU devices. This increases our spatial performance and allows a greater amount of memory to be made available to a simulation. Compute is distributed by breaking work down into task functions that are then allocated to each GPU device for execution. This increases temporal performance by increasing the number of streaming multi-processors available to a simulation.

To assist in analyzing the impact of this work, the SugarScape and BrainGrid ABM applications were selected to measure improvements in the spatial and temporal performance of the multi-GPU library.

Spatial performance will be measured by sampling the memory usage of each application with increasingly larger simulation spaces. Similarly, for temporal performance, we will

measure the execution time of each application. With the exception of simulation size, all other configurable parameters of the applications will be kept constant. This enables consistent results across multiple runs. Additionally, to ensure deterministic results, both BrainGrid and SugarScape have been refactored to support deterministic execution. This was done by means of a configurable seed value, used to initialize the random number generator, that can be passed into the application via a command line option.

Data from both single and multi-GPU versions of the framework will be collected and used to understand the impact of the changes. To help ensure correctness of the benchmark applications, the SimViz GUI will be used to visualize execution results and confirm correct simulation behavior.

SugarScape

SugarScape is a simulation wherein agents are used to model ants attempting to locate and consume sugar that is placed in two distinct mounds within a 2-dimensional simulation space. Each sugar mound contains a single element of sugar on the low end, and grows to 4 elements of sugar towards the peak of the mound. Ants are initialized randomly across the simulation space and by default, make up a fifth of the area.

As soon as an ant is initialized it begins to look for sugar. Up to three cells in the north, east, south, and west direction are visible to the ant. Each tick of the simulation, the ant will evaluate all visible cells and move toward the cell that contains the most sugar. The ant will randomly choose among the favored cells if there are multiple favorites. If no favored cell is identified, or there is simply no sugar near the ant, it will move in a random direction, hoping to locate some. If an ant cannot find sugar within three simulation steps, it ”dies” and the agent is deleted.

When sugar is consumed, waste is left behind by the ant. The total amount of waste, and its impact on the ant, are calculated as an average of the total waste in all immediate neighboring cells. If the waste exceeds a configured threshold, it has the potential to kill off residing ants.

The output of the simulation shows how ants migrate towards areas with higher amounts of sugar. This can be seen in Figure 1 below wherein red dots represent ants and darker shades of yellow indicate more sugar. It can be observed that the majority of ants make their way to the center of the mounds where the most sugar can be found, with fewer and fewer ants at the lower levels.

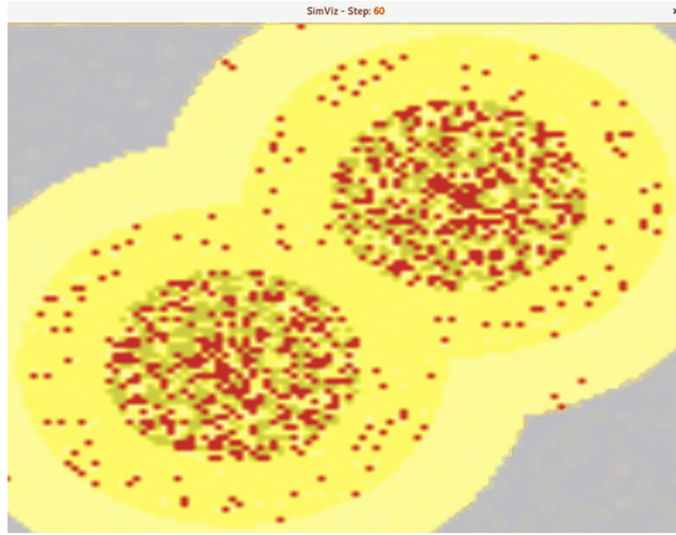

Figure 1: SugarScape simulation output.

BrainGrid

BrainGrid simulates a self-organizing neural network wherein neurons connect through synaptic terminals in axons and dendrites, sending and receiving signals between them. The simulation is initialized with a configured percentage of excitatory, inhibitory, and neutral neurons across a 2-dimensional simulation space. Once initialized, each neuron creates one axon and seven dendrites, each of which grow outwards from the soma in an attempt to locate neighboring neurons. Axons are used to send signals from neurons and dendrites are used to receive them. Only excitatory neurons can create signals, however all neurons can receive and forward them.

Each step of the simulation, an excitatory neuron has a chance to generate a signal. If one is created, it is initialized to a strength of 1.0 and forwarded down the neurons axon. The signal will continue down an axon until it encounters a synaptic terminal, at which point it fans out to all branches. If a neighboring dendrite is encountered, it continues to follow it until it reaches the associated neuron soma.

Different categories of neurons modulate signals in different ways. Excitatory neurons, upon receiving a signal, will amplify it by a pre-configured percentage before propagating it down its axon. Inhibitory neurons will reduce the signal strength by a pre-configured percentage, and neutral neurons will not modulate the signal strength at all.

The output of the simulation is an artificial neural network wherein neurons are connected through their axons and dendrites. Signals are propagated across the network, initiated by excitatory neurons. A visual representation of this can be seen in Figure 2 below. Here the neuron somas are colored a dark, navy blue. Excitatory neurons are purple. Inhibitory neurons are dark gray. Neutral neurons are blue, and signals are green. Axons are shaded slightly darker than dendrites, and signal strength impacts the shade of green used to color them. Inhibited signals being a brighter shade, while excited signals being slightly darker.

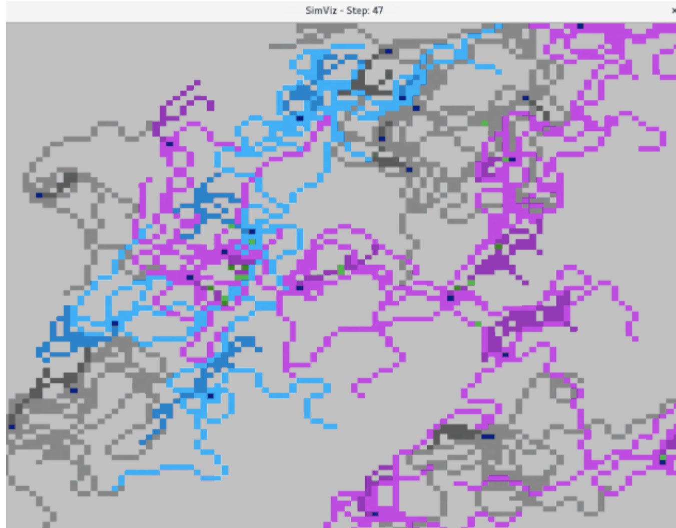

Figure 2: BrainGrid simulation output.

SimViz GUI

SimViz stands for Simulation Visualization and it is a basic GUI tool that makes use of OpenGL[1], GLAD[2], and GLFW[3] to help visualize the output of a simulation. Prior to its creation, visualization of simulations in the CUDA library were relegated to outputting large matrices into logs and attempting to discern correctness from them. I believe this may have led to oversights in simulations such as SugarScape where, prior to being refactored, had ants moving only in the east direction.

SimViz is comprised of a GUI and a basic C++ support library. The support library can be integrated into simulation logic to output simulation steps to a 'viz' file on the local file-system. These files can then be read and stepped through by the GUI application to visualize steps of the simulation. Each step represents a snapshot of the simulation space at a configured output interval.

The GUI application supports the ability for the user to pause the simulation and step through it frame by frame to help glean a deeper understanding of how agent and place objects are behaving. Additionally, simulation space is mapped to a 2-dimensional texture, which is then rendered to the window object. This allows it to render simulations of all sizes without impacting the window size. While not currently supported, it should also make it easier to support zooming in on specific areas of the simulation space, in future iterations of the GUI. Figures 1 and 2 above are both snapshots of the GUI application. GIFs created from real-time playback of simulations can be found in the Heat2D[4], SugarScape[5], and BrainGrid[6] application README files.

The support library makes it easy for developers to integrate it with their simulation logic. It does this by handling the creation of the simulation file and populating it with necessary header data. The only user requirement is that they output RGB information for each cell of the 2-dimensional simulation space. The SimViz app currently only supports 2-dimensions, however support for 3-dimensions may be added in the future should there become a need. Below is a code snippet from the SugarScape application that shows how

a SimViz file is created, and how RGB colors are defined and subsequently used to output simulation steps.

```
1  ...
2  // Excerpt from SugarScapes main function
3  // Create viz file if interval is non-zero.
4  simviz::RGBFile vizFile(size, size);
5  if (interval > 0) {
6    vizFile.open(vm["out_file"].as<std::string>().c_str());
7  }
8
9  ...
```

```
1  ...
2  // Excerpt from SugarScapes displaySugar function.
3  unsigned char* placeColor = new unsigned char[3]{192, 192, 192};
4  unsigned char* sugarColor1 = new unsigned char[3]{255, 255, 153};
5  unsigned char* sugarColor2 = new unsigned char[3]{255, 255, 102};
6  unsigned char* sugarColor3 = new unsigned char[3]{204, 204, 0};
7  unsigned char* agentColor = new unsigned char[3]{192, 0, 0};
8
9  ...
10
11 if (n_agents > 0) {
12   vizFile.write((char*)agentColor, simviz::NumRGBBytes);
13 } else if (curSugar > 2) {
14   vizFile.write((char*)sugarColor3, simviz::NumRGBBytes);
15 }
16
17 ...
```

Multi-GPU Agents

Unlike Places, which represent static state, Agents are dynamic in that they can actively move around a simulation space. This poses a few challenges unique to them that are not present in the multi-GPU implementation of places.

The first challenge is to co-locate the agent with the same device as the place on which it resides. We want to do this to minimize communication between devices. If an agent resides on a place that lives on device 0, then that agent should also live in the memory of device 0. Because agents can move around the simulation space, it's possible for all of them to reside on places that live on a single device. In order to account for this, enough memory is allocated on each device to provide space for all available agents. See Figure 3 below.

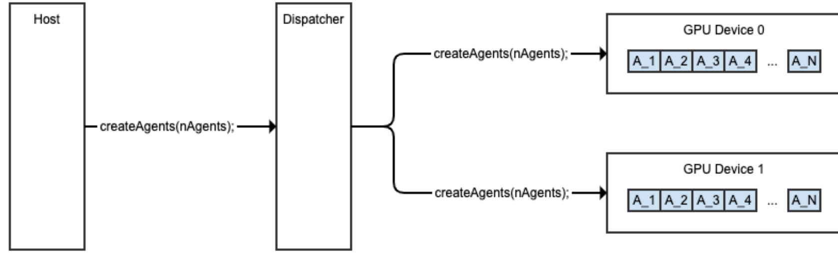

Figure 3: Multi-GPU Agent Construction.

A `resident` boolean is stored on each agent to determine whether or not it is resident on a device. This allows callAll functions to target only agents that resident on the device in which the callAll method is being executed.

```
1  // Agent struct
2  template <typename T, typename P>
3  struct AgentV2 {
4      size_t index;
5      bool isAlive;
6  —> bool resident;
7
8      PlaceV2<P>* place;
9      size_t destPlaceIdx;
10     ...
11 };
12
13 // callAll Kernel
14 template <typename A, typename T, class F>
15 __global__ void callAllAgentsKernel(mass::AgentV2<A, T>* agents, size_t numAgents, F func)
16     size_t idx = mass::getGlobalIdx_1D_1D();
17     if (idx >= numAgents) { return; }
18 —> if (!agents[idx].resident) { return; }
19     if (!agents[idx].isAlive) { return; }
20
21     func(&agents[idx]);
22 }
```

When an agent migrates from a place on one device to a place on another, a kernel is invoked that updates the device location of all agents. It does this by looking at the index of the place in which it resides and determining, based on that index, which device it should be on. It then sets the `resident` flag appropriately. This approach enables the ability to do this with a single kernel function executed in a parallel manner across all agents on both devices.

0.1 Agent Termination

Agents have the concept of life in that they can be alive, meaning they are active, can migrate, spawn other agents, and callAll functions can be invoked to manipulate their state.

They can also be *dead* which means they are inactive, cannot migrate, cannot spawn or be processed by callAll functions, and their state can be overwritten by newly spawned agents. Terminating an agent effectively means settings its state such that it is *dead*, or inactive.

This is done via a `terminateAgent` function that can be invoked and provided with the agent in which the caller wishes to terminate. When this is done, the provided agent is marked for termination by setting its `isAlive` field to `false`.

```
1  // Agent struct
2  template <typename T, typename P>
3  struct AgentV2 {
4      size_t index;
5  --> bool isAlive;
6      bool resident;
7      ...
8  };
```

This maintains the terminated agents state so that any parallel processing happening at the time of termination can continue without error. While preventing all subsequent processing for that agent from occurring as no further callAll functions will be executed over it.

0.2 Agent Spawning

Agent spawning is one of the more complex actions an agent can perform. This is because spawning can happen in parallel and there is a finite, static, amount of space allocated for agents to live in. To add to this, places need to be updated with references to the agents that reside on them. This introduces the potential for more agents to be spawned than there is memory for, as well as for race conditions to occur when updating pointers to these agents from within the place in which they are residing.

There are a couple of things done to mitigate these issue. The first is that `spawn` invocations do not create new agents within the `callAll` functions in which they are invoked. Instead, when `spawn` is invoked, an atomic counter, kept in global memory, is incremented. This counter tracks the number of alive agents. If its value, after being incremented, is below the max number of agents, then there is guaranteed to be available space for the new agent.

Once the `callAll` in which the `spawn` function was invoked returns, a separate kernel is executed to create the agent state for the newly spawned agents. Because there is a one to one relationship between threads and agents, race conditions are not a concern during agent construction. However, places also maintain pointers to the agents that reside on them. Because more than one agent can reside on a place at a given time, the function logic that updates the place must be thread-safe.

To facilitate concurrent updates to a places `agents` array, an atomic compare-and-swap (CAS) function is used. When we want to add an agent to the array, an `addAgent` function is invoked that enumerates it, looking for an available space. When it encounters one, as indicated by a null pointer value, it attempts to claim that space for the agent, using an `atomicCAS`, as shown in the excerpt below. Because this is an isolated function call, no instructions from other threads can be interleaved between it. Meaning that if a thread successfully updates the pointer value, then we can conclude that no other thread will be able to. Thus, no race conflicts can occur.

```
1  ...
2  unsigned long long int null = 0;
3  agentAddr = atomicCAS(
4      (unsigned long long int*)&agents[i],
5      null,
6      (unsigned long long int)newAgent
7  );
8
9  // If the address of the array index matches our
10 // newAgent address, then it was successful.
11 if ((unsigned long long int)newAgent ==
12     (unsigned long long int)agents[i]) {
13     added = true;
14 }
15 ...
```

Future Work

Over the course of Winter quarter, two more benchmark applications have been created by Christopher Sumali. These are intended to assist with measuring the performance of dynamic agents in scenarios where more than one agent can reside on a place. Christopher implemented them using the single-GPU version of the MASS CUDA library. As such, they will need to be ported to the multi-GPU version. Additionally, I will need to complete the remainder of my thesis paper and give my final defense.

References

[1] "OpenGL - The Industry Standard for High Performance Graphics." [Online]. Available: https://www.opengl.org/

[2] "GLAD: Multi-Language Vulkan/GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs." [Online]. Available: https://github.com/Dav1dde/glad

[3] "An OpenGL library | GLFW." [Online]. Available: https://www.glfw.org/

[4] "Heat2D README." [Online]. Available: https://bitbucket.org/mass_library_developers/mass_cuda_core/src/develop/applications/heat2d/README.md

[5] "SugarScape README." [Online]. Available: https://bitbucket.org/mass_library_developers/mass_cuda_core/src/develop/applications/sugarscape/README.md

[6] "BrainGrid README." [Online]. Available: https://bitbucket.org/mass_library_developers/mass_cuda_core/src/develop/applications/braingrid/README.md