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

Benjamin Pittman

Term Report for CSS595

Master of Science in Computer Science & Software Engineering

University of Washington, Bothell

Fall 2020

Project Committee:

Munehiro Fukuda

Erika Parsons

Michael Stiber

Background

Multi-Agent Spatial Simulation (MASS) is an Agent-Based Modeling (ABM) framework that stores data in a distributed array (called places) and distributes processes (agents) to interact with the data. ABM libraries are efficient for simulating biological scenarios and some data science applications 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.

There are two primary considerations for the MASS framework – performance and programmability. While we endeavor to process large amounts of data accurately and quickly, it is a primary aim to furnish this framework for non-computing researchers. To accomplish this, MASS provides an application programming interface (API) for researchers to fit their models and leverage the computing power the framework provides.

Further, NVIDIA's Compute Unified Device Architecture (CUDA) provides a unified memory address (UMA) space that combines host and device addresses into a single virtual address space accessible by both. CUDA UMA 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 versus PCIe [1].

MASS CUDA adheres to the Model-View-Presenter design pattern. 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 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 this interact in the researcher's environment.

The objective of this research is to extend the implementation of MASS CUDA to use multiple GPU's to process simulations faster than either a pure C++ implementation or the current single device CUDA implementation. To accomplish this objective, we have set out goals. Refactoring current MASS CUDA to use multiple GPU's, managed memory, and a beginning dynamic agents' implementation were the goals for this quarter. The compute resources for this research are two NVIDIA RTX 2080 Super connected via NVLINK on a desktop computer running Ubuntu 18.04.

Progress

The first goal of this research was to refactor MASS to use managed memory. With managed memory, the CUDA runtime manages memory transfers between the host computer and the GPU device(s). Unfortunately, after refactoring MASS CUDA to use managed memory it was realized that this cannot be accomplished. As previously outlined, it is a primary goal of MASS

to enable non-computing researchers ease of use and to accomplish this the framework leverages inheritance and polymorphism. CUDA managed memory works with C++ classes, but it does not work with virtual classes or functions. In managed memory all memory is first allocated on the host and then transferred to the device (and , back, etc.), but it does not reproduce the virtual function table on the device(s) rendering any pure virtual classes and virtual methods un-initialized.

The failure with managed memory ended up being a failure on two fronts. First, the apparent failure of managed memory and the time spent refactoring to realize this. Second, a lot of the multiple-GPU code refactoring was done at the same time and unwinding and refactoring back to classic CUDA memory management took far longer than it should have. It appeared that MASS CUDA was initializing and allocating memory using managed memory, but when we tried to access data in the derived classes it became apparent nothing was fully instantiating. The lesson here is one can never read enough and it is imperative to quickly employ a development loop – write tests or interfaces, deploy small improvements, test, and, when they pass tests, refactor to remove redundancies – especially when the environment is complex. The goal with managed memory was to get to a simple starting point for MASS CUDA multiple GPU and progressively add features to improve performance. We expected to move back to traditional CUDA memory management to gain finer grained processing improvements at some point in this research.

This quarter I completed changes to how the MASS framework manages Place and Agent objects on devices and the host. First, we changed how we are tracking objects on the devices so that pointers are stored separately for each device in vectors. Then, all instantiation and kernel calling methods were updated to process on multiple devices. At the time of this writing MASS CUDA multiple GPU initializes Places and Agents and all Place functionality processes correctly, but the Agent functionality is not fully implemented, and no data is yet exchanged between devices during either places->exchangeAll( ) or agents->manageAll( ). Finally, while not all the kernel functions or kernel calling functions are finished towards an initial implementation of MASS CUDA on multiple devices, algorithms have been developed and are being implemented. These algorithms are outlined in the next section and code is available at - https://bitbucket.org/mass_library_developers/mass_cuda_core/src/ben.pittman/

Method

Refactoring MASS CUDA for multiple devices introduces the challenge of communicating across devices. In CUDA there are a few options for how to do this. First, and where our implementations begin, is to introduce the ghost Places and process more data than if everything was on one GPU and then update each device with the ghost Place data of the neighbor device(s). Other options to be explored include setting ghost Place data via kernel function calls by passing in pointers for each device's ghost Places.

Places

This research begins the multiple GPU framework by splitting Place objects amongst devices. MASS CUDA stores all data in one-dimensional arrays that are then processed and returned as output as though they are two-dimensional. To facilitate asynchronous processing on each device ghost Place objects are allocated on each device. The number of ghost Places kept on each device is equal to the furthest an Agent object may travel in one simulation step (at minimum one row) and contains the same values as the related object(s) on the neighbor device. One set of ghost Places is maintained on the first and last ranked device and two sets on each middle ranked device. This relationship is illustrated in Figure 1.

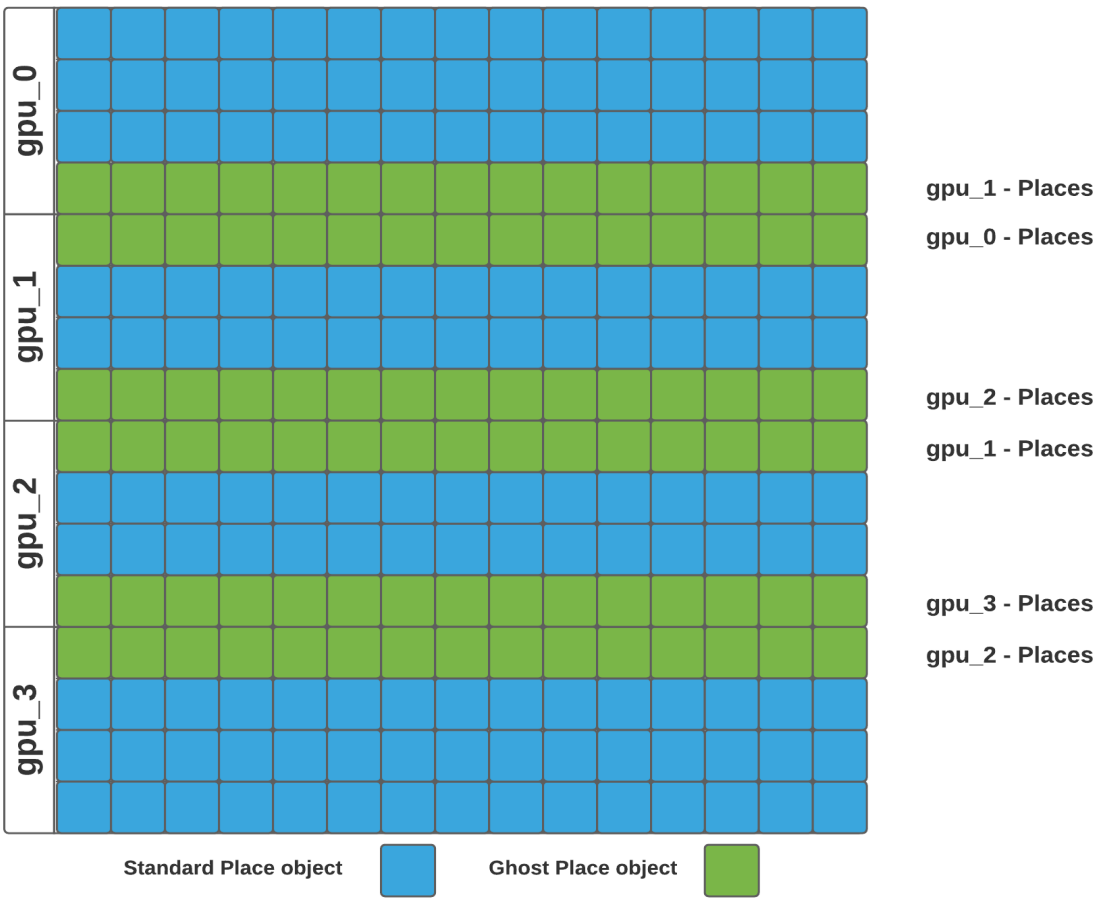

Figure 1. MASS Places

The Places neighbor exchange – not fully implemented – performs a collection of the data from each application developer defined neighboring Place – or none if the neighbor is outside the entire two-dimensional represented Place space. This algorithm is shown in pseudocode in Listing 1. The first step of the algorithm is to check that the passed in destinations vector

matches our neighbors, and if not we update the neighbor's vector and place in shared memory. Next, devices are looped over (lines 6-9) and the exchangeAllPlacesKernel function is called on each device – sequentially for now as we have yet to implement streams for each device. The kernel function is called on all valid Place indices (line 15), then iterates through the neighbors for each Place (lines 16-23) where if the neighbor Place is within the devices space it is added to the neighbor array of the state of the Place. Finally, the un-implemented portion of this function is to exchange data across devices. This can be accomplished with either a device-to-device memory copy (lines 10-11) or by calling a kernel function on each device with pointers from neighboring device(s) and copying the Place state (not shown).

As the aim of this research is to expand to multiple devices the only change to the MASS Place method callAll( ) is to put the kernel function in a loop that iterates over the devices similarly to exchangeAllPlaces( ) in Listing 1.

Listing 1. Places exchange with neighbor's algorithm

```
1. __constant__ int offsets_device[MAX_NEIGHBORS]
2.
3. void Dispatcher::exchangeAllPlaces( vector<int*> *destinations)
4.   if (destinations != neighbors) update neighbors to be destinations
   on each device
5.   retrieve device data parameters
6.   for (n : devices)
7.     cudaSetDevice(devices.at(n))
8.     exchangeAllPlacesKernel<<<>>> (params)
9.     cudaDeviceSynchronize()
10.      if (n != 0) cudaMemcpy(higher ranked device data to lower ranked
   device)
11.      if (n != devices.size() - 1) cudaMemcpy(lower ranked device data
   to higher ranked device)
12.
13.
14.  __global__ exchangeAllPlacesKernel(Place** ptrs, int nptrs, int
   nNeighbors)
15.    int idx = getGlobalidx_1D_1D()
16.    if (idx < nptrs)
17.      PlaceState *state = ptrs[idx]->getState()
18.      for ( n : nNeighbors )
19.        int j = idx + offsets_device[n]
20.        if (j >= 0 && j < nptrs)
21.          state->neighbors[n] = ptrs[j]
22.        else
23.          state->neighbors[n] = NULL
```

Agents

The Agents implementation – not completed – maintains a set of Agents on each device. At instantiation, the application developer may decide how to allocate Agents across Places, but if not this implementation first randomly generates indices over the entire Place space, then sorts them, assigns them to the corresponding device, and allocates them on each device. Agents interact with Places through a callAll() method by changing its data. These are performed concurrently on all Agents across all devices. As with the Place->callAll( ) function(s), the Agent->callAll( ) was refactored to have the kernel function called on each device with the appropriate data. After each callAll( ) a follow-up method, manageAll( ), is called to first terminate any agents marked as not alive, then migrate alive agents to different Place objects, and finally to spawn Agents at the Agents new Place, if applicable.

We define dynamic Agents in this research as meaning the framework will increase or decrease the underlying memory of the number of Agents spread across the Place objects throughout a simulation. Previous research instantiated arrays with twice the number of Agents declared at MASS initialization to allow for Agent spawning [2]. Further, no agent spawning or agent termination algorithms were developed for the MASS CUDA framework. This research extends Agent management by introducing algorithms for a starting point for garbage collection, agent spawning, and migrating agents across devices.

This research's garbage collection (agent termination) algorithm first adds a vector of Agent arrays to represent collected agents on each device that replaces the slacked array of the prior implementation. However, if the number of alive Agents drops below the number at instantiation the array of Agents size will not change in the same manner to avoid excessive memory allocations. This fits the implementation as the only objects that may need to be garbage collected or instantiated after initialization are Agent objects.

The terminateAgents() method shown in pseudocode in Listing 2 first calculates the number of Agents to collect by putting an integer on each device to track the count (lines 3-6). Next, the devices are iterated over, and each call a kernel function to atomically increment the count of Agents on each device that are set to not alive (lines 14-17). This count is then used to instantiate an array and a second kernel function is called to add each not alive Agent to the array using atomicity on each device to avoid race conditions (lines 19-28). Finally, each set of Agent pointers are pushed onto a vector representing the collected agents (line 11).

Following the terminateAgents( ) function as part of the Agents->manageAll ( ) function is a call to migrate agents to a different Place object. Agents at initialization are mapped onto Places located on each device, but not on any ghost Places. The migrate Agents algorithm runs through potential migration locations and always chooses the lowest possible index value to migrate. When a Place is chosen, Agents may migrate to either a Place or ghost Place and are then copied to the next ranked device if necessary. This MGPU implementation synchronizes Agent migration such that the lowest device first sets its Agents migration locations, transfers

any ghost Place located Agents to the neighboring device, waits on any Agents migrating back to it, and then removes any references to Agents in its ghost Places. This pattern repeats until all Agents across all devices have found a migration Place. Figure 2 shows an example of Agent migration on and across devices where potential Agent travel is the Von Neumann neighborhood of the Place it resides.

Listing 2. Agent termination algorithm

```
1. Void Dispatcher::terminateAgents()
2.   get devices and device parameters
3.   for (n: devices)
4.     cudaMalloc(devAgtTrmCount [n], sizeof(int))
5.     cudaMemcpy(devAgtTrmCount[n], hostAgtTrmCount[n], sizeof(int) *
   size, H2D )
6.     terminationCountKernel<<<>>> (params)
7.     cudaMemcpy(hostAgtTrmCount[n], devAgtTrmCount[n], sizeof(int) *
   size, D2H)
8.     Agent** a_ptr = NULL
9.     cudaMalloc(a_ptr, sizeof(Agent*) * hostAgtTrmCount[n])
10.     collectAgentPointersKernel<<<>>> (params)
11.     devCollectedPtrs[n].push_back(a_ptr)
12.      // update device array size tracking variables
13.
14.  __global__ terminationCountKernel(Agent** ptrs, int nptrs, int*
   devAgtTrmCount)
15.    int idx = getGlobalidx_1D_1D()
16.    if (idx < nptrs && !(ptrs[idx]->isAlive()))
17.      atomicAdd(devAgtTrmCount, 1)
18.
19.  __global__ collectAgentPointersKernel(Agent** ptrs, Agent** c_ptrs,
   int nptrs, int count, int trmC)
20.    int idx = getGlobalidx_1D_1D()
21.    if (idx < nptrs)
22.      if (!( ptrs[ idx ] ->isAlive())
23.        int idxStart = atomicAdd(nextIdx, count)
24.        if (idxStart < trmC)
25.          c_ptrs[idxStart] = ptrs[idx]
26.          // clear Agent params
27.      else
28.        // condense live agents in ptrs array
```

Spawn agents is run concurrently on each device once migration finishes on that device. For environments of more than two GPU devices these can commence once migration from the higher ranked neighbor device is finished. The implementation first checks if its Agent array has enough space, takes the collected array of Agents if needed and available, allocates new memory if needed, and, finally, combines the arrays.

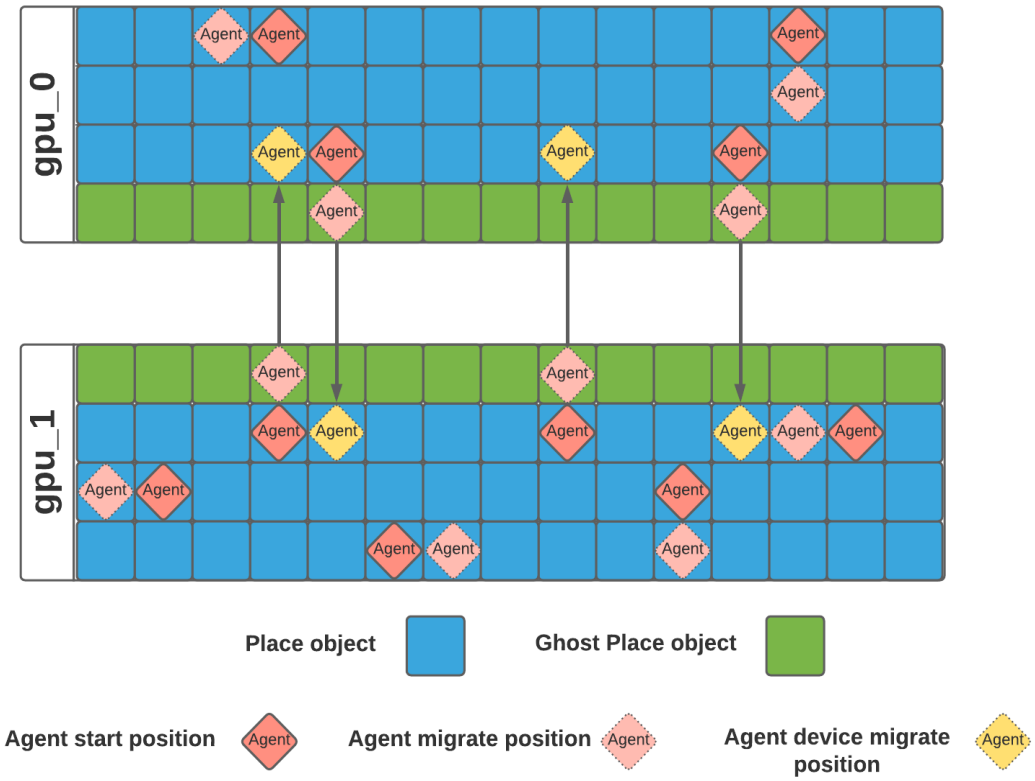

Figure 2. Agent Migration

Brain Grid

No research towards a Brain Grid implementation was completed this quarter.

Results

As the initial implementation for MASS CUDA multiple GPU was not finished this quarter I do not have results to compare against MASS CUDA single GPU.

References

[1] Kinghorn, D. (2019, January 11). P2P peer-to-peer on NVIDIA RTX 2080Ti vs GTX 1080Ti GPUs. Retrieved December 04, 2020, from https://www.pugetsystems.com/labs/hpc/P2P-peer-to-peer-on-NVIDIA-RTX-2080Ti-vs-GTX-1080Ti-GPUs-1331/

[2] Kosiachenko, Lisa, Nathaniel Hart, and Munehiro 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*, edited by Yves Demazeau, Eric Matson, Juan Manuel Corchado, and Fernando De la Prieta, 11523:139–52. Cham: Springer International Publishing, 2019. https://doi.org/10.1007/978-3-030-24209-1_12.