MASS CUDA in a Multiple GPU Environment

Ben Pittman

CSS 600 Spring 2020 Term Report

TABLE OF CONTENTS
Purpose .................................................................................................................................................. 1
MASS cuda at present .............................................................................................................................. 1
Summary of Progress .............................................................................................................................. 3
  Multiple GPU Devices ......................................................................................................................... 3
  Unified Memory Management ............................................................................................................. 5
Future Research ................................................................................................................................... 6
References ........................................................................................................................................... 7

PURPOSE
The purpose of this research is investigation and implementation to extend Multi Agent Spatial Simulation (MASS) CUDA to work in a multi-GPU environment. MASS is an Agent-Based Modeling (ABM) library 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. The goal of this research is to extend the MASS CUDA implementation such that it utilizes multiple GPU’s to process simulations faster than either a pure C++ implementation or the current CUDA implementation.

The goal of my work this quarter was to establish a course for thesis research in MASS CUDA and prepare and implement peer-able multiple GPU device and unified memory addressing (UMA). UMA allows for one memory address space and abstracts away the host-to-device and device-to-host memory copy’s – this is handled by the CUDA runtime environment - and enables CUDA to deploy multiple identical GPU’s to process data.

MASS CUDA AT PRESENT
At the start of my research MASS CUDA was implemented to store data as classically done with CUDA - run simulations by moving data from host-to-device, running a CUDA kernel function on that data on the device, and then moving the modified data back to the host. In my research I
expand upon work done by two previous students as well as those whose work they expanded. Nate Hart’s research accomplished three main objectives. His work simplified the API for application developers, set MASS CUDA in a robust design pattern with a skeletonized lay-out for further implementation, and coded applications that utilize only places for simulations [3]. Lisa Koschienko’s work improved on the processing time of some MASS CUDA simulations through simplifying the class structure of the library, completing two kernel functions for each GPU memory copy - rather than one per - and leveraging constant memory to store read-only data of neighbor places. Finally, MASS CUDA is implemented for agents in the most current iteration, but not for applications that require repeated spawning and terminating [2][4].

**Figure1: Model, View, Presenter interaction**

MASS CUDA at the start of my research followed the Model View Presenter (MVP) design pattern, seen in Figure 1. This pattern separates the view and model and their interactions are managed by the presenter. In MASS CUDA, the view is the API through which application developers interact, the Model is the state and behavior of places and agents, while the presenter manages data calls on the host and device(s). This was a very suitable pattern for managing data separately. I expect my research to eventually break this pattern and become a Model-View-Controller (MVC) design pattern. With one data management solution it is not necessary have strict management.

When the MASS application is run it first initializes the Dispatcher class – the Presenter – which initializes the DataModel on the CPU and DeviceConfig classes for each CUDA device. Figure 2 is a sequence diagram that shows the interaction of the MVP design pattern after these initialization steps. Create places and create agents calls are made on the host, next these are initialized on the device, and finally, when calls are made by the application developers implementation through the API these are passed from the host to device and the results are copied back. From this point the user may results displayed or further calls and transfers will happen.
SUMMARY OF PROGRESS

My research this quarter is best viewed through two segments of work – Multiple GPU Devices and Unified Memory Management.

Multiple GPU Devices
To simplify my research, I added an additional NVIDIA RTX 2080 Super GPU to my home system. These GPU’s possess 8 GB of device memory, 3072 CUDA cores, and support 2-way NVLINK which allows them to directly share their memory. The system’s relevant resources are an Intel i7-7820X CPU, 64 GB of RAM, and two NVIDIA RTX 2080 Super GPU’s connected via NVLINK with Ubuntu 18.04 as the operating system. After initial testing is completed on this system research will be done deploying this work on a cloud-based system with more GPU’s.

Enabling peer device memory access is very straight-forward. First, ensure that the devices are the same – both the hardware and CUDA Compute Capability must be the same. Next, use the CUDA run-time provides library functions to check. Previous work iterated through GPU devices and placed them in a ranked list based upon highest and lowest compute capability for each...
device. In this work, we are interested in devices that are peer-able and as we know that CUDA places the device with the highest compute capability and hardware performance at the first position in the list of devices we add this to the peer-able list and check all other devices on the system against it. The remaining device list order is not guaranteed so my implementation uses the CUDA library function to check if peer-able –

```cpp
__host__ cudaError_t cudaDeviceCanAccessPeer ( int* canAccessPeer, int device, int peerDevice )
```

This function compares each device for compatibility (in our case with the device at position 0 of our list) and if so adds them to a sub-list of compatible devices. Once all devices are checked, the peer device memory access is enabled bidirectionally for all devices in the sub-list using the CUDA library function –

```cpp
__host__ cudaError_t cudaDeviceEnablePeerAccess ( int peerDevice, unsigned int flags ).
```

Figure 3 shows a function from the Dispatcher class of my MASS CUDA implementation that accomplishes this. This function is called from MASS main. After the peer-able devices are found, as shown in lines 12-16, each device is instantiated as a MASS::DeviceConfig object and added to a list in lines 20-22. Finally, each device enables peer memory access with every other device bidirectionally in lines 26-31.

```cpp
void Dispatcher::init() {
  if (!initialized) {
    initialized = true;
    Logger::debug("Initializing Dispatcher");
  }
  if (gpuCount == 0) {
    throw MassException("No GPU devices were found.");
  }
  vector<int> devices;
  devices.push_back(0);
  for (int d = 1; d < gpuCount; d++) {
    int canAccessPeer = 0;
    cudaDeviceCanAccessPeer(&canAccessPeer, 0, d);
    if (canAccessPeer)
      devices.push_back(d);
  }
  // instantiate DeviceConfig object for each device
  for (int i = 0; i < devices.size(); i++) {
    DeviceConfig d(devices[i]);
    deviceInfo.push_back(d);
  }
  // Establish bi-directional peer relationships for all peerable devices
  int peerCount = deviceInfo.size();
  for (int x = 0; x < peerCount; x++) {
    cudaSetDevice(devices.at(x));
    for (int y = 0; y < peerCount; y++) {
      if (x != y)
```
As all the necessary MASS CUDA changes are not completed and tested to enable multiple GPU devices, there may be some changes necessary to the code and process shared above. The paramount need to enable multiple GPU devices in CUDA lies with how memory is managed, and to enable multiple devices CUDA requires the host and all devices to share a memory address space. The CUDA run-time provides Unified Memory Management (UMA) to address this.

**Unified Memory Management**

The CUDA run-time provides a UMA scheme that allows all host declared memory to be accessed on devices that support UMA. Further, the UMA scheme is automatically enabled on Linux hosts with 64-bit processes. To declare UMA memory on the host cudaMallocManaged() is used to declare it rather than cudaMalloc(). Further, there is then no need to call cudaMemcpy() to then transfer the host declared memory to the device, and thus also no need to do the reverse to transfer the data back from device to host after computation. The UMA run-time library functions abstract this from the programmer.

UMA also provides functions that allow the programmer improve latency and hiding and further inform the run-time how to move and process data. cudaMemAdvise() and cudaMemPrefetchAsync() are two functions that allow the programmer to tell the run-time to stage data for following kernel function calls. This works like tiling strategies at a higher level. Where tiling takes advantage of register memory and process cycles, prefetching takes advantage of warps and the warp scheduler to interleave memory movement and computation.

One of the most important features of CUDA UMA is that it allows memory over-subscription. Previously, programmers had to take care to not over-subscribe memory – this can be seen in earlier iterations of MASS CUDA – where now that all memory is managed on the host and moved by the run-time. In research by Knap & Czarnel, it shows that large compute-to-communication ratios can benefit from UMA with prefetching over standard memory approach, but that in all other tested codes it showed worse performance [1]. The goals of this research match-up well with these findings as we look to enable multiple devices and have them access each other's memory, initialize, and maintain data on each device using UMA, and getting Agents to work in MASS CUDA.

To change MASS CUDA from standard memory management to UMA, many changes are required with some completed and others still to come. The first, and most fundamental, change is
removing the two memory models – one for the host and another for the device(s). To accomplish this the first changes were made to the class structure of the library. To start, the partitioning classes from Nate Hart’s implementation were reintroduced and their initialization furthered. The idea behind partitioning the data is to simplify mapping a chunk of data to a device. Nate Hart’s implementation contained ghost space for neighboring data chunks boundary data, allowing each device and data chunk to process independently and enabling the programmer to synchronize each time step via data transfers on the host. One of the goals to improve processing time for this implementation is to enable latency hiding even more greatly by simply relying on the weak-consistency model of MASS and the CUDA run-time for moving data and/or enabling the programmer to do so with prefetching. A full stop to swap all boundary is expensive when compared to the potential for only exchanging the requested boundary data and allowing the CUDA run-time and warp scheduling to process unrestricted warps while these data exchanges are occurring.

While coding the partitioning I decided a simpler solution would be better. One of the reasons to use partitioning is that it would be simpler to reintroduce boundary data management if deemed necessary. Otherwise, the data pointers on each device can be stored within the DeviceConfig object that represents each device. This opportunity still exists, but for now the simpler way. Therefore, when the Dispatcher object (the Presenter in our design pattern) calls to instantiate Places and Agents it calls only the DataModel object and passes the DeviceConfig objects to it. This allows us to keep all the CUDA functionality already coded in the DeviceConfig class. Further, all of the Places and Agents tracking and API calls are maintained in the DataModel. Finally, to initialize UMA data we first changed the calls in the DeviceConfig class to use cudaMallocManaged() and removed the cudaMemcpy() calls. To complete the data instantiation, the DataModel calls the AgentsModel and PlacesModel classes again passing the devices. Each of these then iterates over the devices and calls the associated DeviceConfig object functions to instantiate the data. This method takes further advantage as within UMA when data is declared the run-time places it on the active device provided it does not oversubscribe the device. I have not yet decided how the AgentsModel and PlacesModel will maintain pointers to each set of instantiated data – or if they will.

**FUTURE RESEARCH**

Next steps for this research are to complete the changes to fully implement UMA and run this initial version with the current Sugarscape implementation. All the MASS CUDA kernel functions after initialization need to be revised to work with UMA. Further, there is no working Agents class implementation in MASS CUDA. Therefore, a scheme will be designed that enables agents to initialize, process, and migrate. Initially, the UMA run-time will manage this, and we will provide the means for the implementation to get the pointers to the needed data. This may be tested alongside more purposeful migration scheme for agents if performance is not great.

When a working version of MASS CUDA from the preceding research is accomplished, research will be conducted to introduce CUDA streams and graphs. Streams allow for greater latency-
hiding by enabling on device scheduling control while graphs use multiple streams to allow for multiple kernel functions to be run sequentially on devices without transferring control back to the host. Again, the goal here is to improve latency-hiding by maximizing compute cycles for each memory transfer and keeping memory close to the processor that will use it. Non-uniform memory access is of concern for distributed data computation and presents in MASS CUDA [5]. The aim here is to remedy this issue by instantiating on each device as directly as possible, and dynamically providing CUDA thread and grid parameters such that migrating agents memory movements are interleaved with running warps.

Another goal for future research is to dynamically assign thread and grid parameters for kernel functions based on heuristics of the application running in MASS CUDA. Potential paths for this include running sampling kernels within the library at initialization to test and working with profiling implementations for data as well as functions on the data. CUDA Flux is a profiling implementation that shows great promise over the NVIDIA provided profiler nvprof [6].

Finally, to show the preceding research I will implement, at present, two applications. The first, Sugarscape, is based on previous MASS CUDA implementations and will be used as the test application. Once this is tested and complete I will implement BrainGrid in MASS CUDA. BrainGrid is a neural simulation framework created by Dr. Michael Stiber and his research group at UW-Bothell [7]. This framework aims to provide researchers with a simple yet high-powered framework to conduct neural simulation experiments. This framework has been implemented in C++ and CUDA, but not yet in MASS CUDA or another agent-based modeling library.

REFERENCES


