Heterogeneous Implementation of MASS CUDA to Scale up Agent Based Simulations

Ali Arslan Yousaf

A report

submitted in partial fulfillment of the

requirements for the degree of

Master of Computer Science and Software Engineering

University of Washington

2020

Project Committee:

Dr. Munehiro Fukuda, Chair

Dr. Erika Parsons

Dr. Michael Stiber

Program Authorized to Offer Degree:

Computer Science and Software Engineering

University of Washington

Abstract

Heterogeneous Implementation of MASS CUDA to Scale up Agent Based Simulations

Ali Arslan Yousaf

Dr. Munehiro Fukuda

Professor and Chair

Computer Science and Software Engineering

Agent based modeling has emerged as one of the foremost areas of research for tackling massively parallel simulation problems. In line with the broader move to special purpose processors and GPUs, ABM systems have been designed to take advantage of the massive amounts of parallelism available. Unfortunately, the amount of memory available on commodity GPUs has not scaled along with the increasingly larger amounts of data practitioners must work with and this has begun to impose limits on the size of problems, which can be solved.

We present an extension to the MASS framework, which allows it to run large models on oversubscribed GPUs. We show that it scales similarly in terms of performance to pure CUDA implementations of three representative algorithms and that the ease of use and programmability afforded by the MASS framework is favorably preserved.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge all the help, support and mentorship from my supervisor Dr. Munehiro Fukuda as well as all the members of the DS Lab.

DEDICATION

This project is dedicated to my parents, for always believing in me.

Chapter 1. INTRODUCTION

1.1 ABMs AND THE MASS FRAMEWORK

Agent-based modeling is a simulation technique used to simulate the actions and interactions of autonomous agents and their effects on the system as a whole. An agent-based model describes a system by representing it as a collection of agents often operating in a given space. Individually simplistic, these agents can collectively model behavior that is very complex.

MASS, or Multi-Agent Spatial Simulation, is a library that brings a multi agent paradigm to code migration based programming models for a variety of problems requiring large scale parallelism including physics simulation algorithms, economics models, social networks and graphs as well as any problem which can be represented in a spatial manner [8]. Developed by Dr. Fukuda et al., MASS centers around two central concepts: Places and Agents. To illustrate them, consider a generic problem of connectivity in a directed acyclical graph. To map this problem to MASS, the most logical mapping would be of the nodes of the graph to Places while Agents would be executable code exploring connectivity on the graph.

In terms of implementation, MASS is available as a Java library, a C++ library and a CUDA package. In all implementations, Places are statically assigned to threads while Agents move

autonomously over processes. The simulations run on multi-threaded processes over nodes and sockets are used for communication.

1.2 MASS CUDA

Mass CUDA is an implementation of the Multi-Agent Spatial Simulation framework on GPUs based on the CUDA platform by Nvidia. The key design goal of MASS CUDA is to abstract away the design and implementation of GPU programming specific concepts and allow a practitioner to benefit from the performance boost and massive parallelism available via the use of GPUs.

The MASS CUDA library is based around the Model-View-Presenter design pattern: the View is the application programming interface (API) accessed by the end user, the Dispatcher class serves as the Presenter by taking care of host-device data transfers and launching GPU kernels, and the model is the view of the data or state present on both the device and host. Figure 1 shows the layout of MASS CUDA components.

The maintenance of a data models on both the host and the device (GPU) is a key aspect of MASS CUDA's design and solves the problem of mutual execution by splitting the Agent's and Place's implementation into Agent/AgentState and Place/PlaceState.

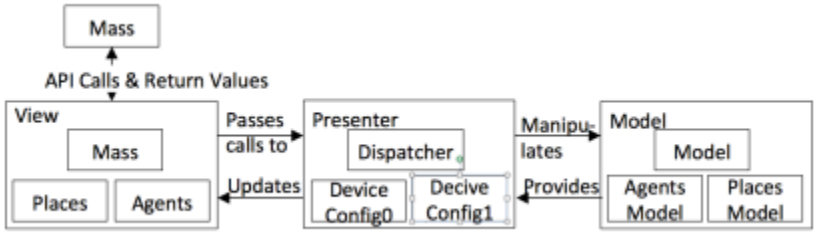

Figure 1 . Architectural design of MASS CUDA.

1.3 NEED FOR HETEROGENEOUS COMPUTING IN MASS

While graphics programming units (GPUs), field programmable gate arrays (FPGAs) and other application specific integrated circuit (ASIC) systems have seen large increases in performance and IPC counts, the memory has available on such systems has not grown on the same scale. Contrasted with the order of magnitude increases seen in the amountAs of data being collected in the world, it is not difficult to see how this situation might impose punishing limits on practitioners with respect to the size of problems they can realistically solve.

As a real-world example, consider the influenza epidemic simulation tool FluTE. FluTE is based on epidemic simulation models from Germann et al. and was designed to aid in understanding the spread of the H1N1 outbreak of 2009[REF]. FluTE requires around 80 MB of memory for every million individuals it simulates. Further, a simulation involving 10 million people takes about 2 hours depending on the R0 rate, intervention etc. Resultantly, practitioners find themselves in a situation where both memory and compute capability are bottlenecks: while a CPU simulation would be too slow to be useful for practitioners, a typical GPU would soon run

out of memory – hence limiting the suitability of Agent Based Modeling for a very crucial and time sensitive analytics task.

A further impediment to heterogeneous computing for practitioners is the steep learning curve involved in writing bespoke heterogeneous implementations of algorithms in CUDA and C. This goes beyond the already considerable difficulty of writing pure CUDA programs as there is very sparse developer support for heterogeneous computing and the process involves painstaking manual memory management, the selection and copying back and forth of data between the GPU and Host, and dependency management. Furthermore, ABM is an inherently stochastic process and fitting it to a heterogeneous mold is an exercise in the controlled adoption of tradeoffs; having to support all of this while also coming up with innovative solutions to the original problem would likely place too large a cognitive burden on researchers.

1.4 DESIGN CHALLENGES

Having established the need for a compelling ABM framework with support for heterogeneous CPU-GPU computing support, we examine some of the key design challenges in supporting said functionality in the context of ABM frameworks in general and MASS in particular.

Agent Based Modeling is an inherently stochastic process that aims to model aspects of the world to process large amounts of data by the use of active agents moving over static (fixed) places. As it is not possible to predict the movement of agents (it is after all dependent on the specific data being processed as well as the data scientist's implementation), this poses a challenge in framework design on data partitioning.

In general, simulations can be partitioned along two dimensions: domain decomposition (decompose data) and functional decomposition (decompose on functionality). Further, these partitions can be done statically or dynamically (at run time). While all these strategies have strengths and weaknesses, as a framework designer our job is pick a partitioning scheme that is characterized by low amounts of imbalances and high performance.

Another difficulty arises from the fact that an extremely large number of possible agent movements can occur, and it is difficult of supporting them all in a heterogeneous environment while still maintaining performance. While data shows that the vast majority of simulations tend to have agent movements in certain bands (citation needed), it is possible to imagine simulations where agents might jump from one edge of the ‘map’ to the other: i.e. a frog agent jumping across several stones across a water surface. Hence, design decisions must be focused on enabling support for the largest possible set of use cases without jeopardizing usability or performance.

Finally, the invariant of not requiring any CUDA or heterogeneous programming expertise on the part of practitioners must be maintained. In practical terms, this means ensuring that the ‘View’ layer (the end user API) remains maximally unchanged and requires a carefully thought out system architecture.

1.5 GOALS AND SCOPE OF WORK

Hence, our overarching goal is to provide a robust, usable and performant ABM framework with support for heterogeneous CPU-GPU computing. We adopt an acceptable set of compromises to achieve this end.

To minimize the impact on performance and reduce the level of indirection, we impose limits on agent migration depth: in each epoch, an agent cannot move more than X ‘paces’ (where a pace quantifies adjacent Places). This is necessary because in the context of heterogeneous computing, increasing the agent movement depth has a large performance penalty due to the increasing data transfer necessary at each epoch.

We decompose the problem along the data dimension and use a mixture of static and dynamic assignment in line with design goals of the MASS library. While Places distribution is predetermined for every run, Agents migration is dynamically decided at each epoch and this leaves the door open for further optimizations in this area.

Due to its intrinsically asymmetrical nature, adding heterogeneous support results in some architectural changes in MASS CUDA that overturn the designer’s choices. Most prominently, MASS CUDA laid heavy emphasis on maintaining an identical copy of the data (in the form of state arrays) on both the Host and the Device (GPU). Since the Host model is now much larger than the Device side data model, it’s role and application life cycle is now significantly different. Figure 2 shows the relationship between host and device memory in MASS CUDA and the heterogeneous version of MASS CUDA.

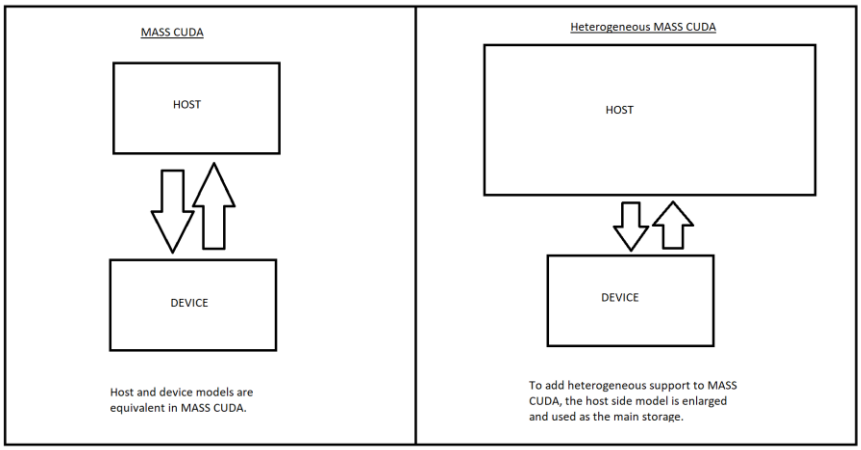

Figure 2 Data models in MASS CUDA vs. heterogeneous version

After adding support for heterogeneous computing in MASS CUDA to handle memory oversubscription, we perform certain performance optimizations. Firstly, we enable support for pinned and non-pageable memory in the host side model to allow the GPU to directly copy data to and from the system memory. Secondly, we use a reference only staging memory design to prevent double copying of data while preserving the extensibility and abstraction benefits of a staging host side buffer. Thirdly, we minimize data transfers to those absolutely essential for the computation, such as only transferring those Places which an Agent can immediately move to. Fourthly, we merge the functionality of agent migration and spawning without affecting the MASS API and this helps significantly to reduce the overhead of memory swapping.

Wherever the choice presents itself, we favor programmability, ease of use and a low barrier to entry over maximizing performance. This is in line with the goal of MASS to provide a general purpose, accessible and extensible ABM framework for data science practitioners. A large part of this comes down to maintaining the existing API and this is achieved by limiting changes in the library to those not requiring user assistance or guidance.

The contributions of this work are as follows: providing a general purpose ABM framework which can scale beyond GPU memory without requiring any specialized knowledge on the part of practitioners, adding heterogeneous functionality to the MASS project, optimizing the performance of our implementation, and providing an extensible base on which to add needed functionality such as multiple GPU support etc.

The rest of this paper is laid out as follows. Section 2 presents a review of the existing body of work on heterogeneous CPU-GPU systems in general and agent-based modeling in particular. Section 3 presents our detailed methodology and Section 4 delineates results of bench marking on the three representative algorithms presents a quantitative and qualitative evaluation of our work in terms of performance, scalability and programmability followed by our concluding words.

Chapter 2. LITERATURE REVIEW

Although initially designed for outputting video to displays, GPUs have been used for an increasing amount of general purpose and high-performance computing tasks. This was greatly accelerated with the release and adoption of the CUDA framework by Nvidia in the 2000s and can be attributed to the massive amount of parallelism offered which cannot be matched by general purpose CPUs.

As we have seen, the approaches may be categorized as working blindly to programmer (so no additional support is needed during the development process to manually manage resources), such as hardware level support for memory access, blind caching and prefetching of data etc., or

techniques which require some amount of manual resource management, such as compiler enhancements to decipher memory access patterns, effective data flow graph partitioning for work distribution etc.

However, ever since the dawn of GPU computing, memory oversubscription has always been a problem as GPUs are equipped with significantly faster and more expensive memory than normal DRAM. Solutions for this have focused on all levels of the stack from hardware/platform designs to high level application specific solutions.

A number of solutions have targeted the architecture and platform to come up with potential solutions to alleviate problems of GPU memory oversubscription. Kwon et al. provide an architecture level solution by aggregating a pool of memory modules locally within the device-side interconnect, which are decoupled from the host interface and function as a vehicle for transparent memory capacity expansion [7]. Matsouka et al. presented DRAGON, which allows the GPU to have direct access to NVMe based high speed memory which can scale much larger than even conventional DRAM [12]. This was accomplished by directly mapping the GPU memory to the NVM storage. Zheng et al. attempt to close the gap between automatic memory management between GPU/CPU and that done manually by programmers by treating page faults as long latency memory operations and utilizing intelligent prefetching [17]. GPUswap is a novel approach to enabling oversubscription of GPU memory that does not rely on software scheduling of GPU kernels and uses the GPU's ability to access system RAM directly to extend the usable memory [5]. Zorua is a framework to virtualize access to GPU resources to enable more controlled and finetuned oversubscription of resources. It uses information from the compiler

decipher resource usage patterns and prioritize those [14]. While these techniques have shown promise and generalize well, they require support from the platform and architecture which limits their applicability.

As the popularity of machine learning has exploded in recent years and it has become one of the most popular workloads for GPUs, a lot of solutions focusing on GPU memory oversubscription have focused on the learning tasks. Nonetheless, one caveat should be kept in mind regarding the generalizability of solutions designed for learning tasks: since gradient descent is a stochastic process, it is much more resilient to missed iterations and lost data as compared to deterministic workloads such as physics simulations or geometric algorithms. One of the first and influential works in this regard was vDDN by Nvidia which is a runtime memory manager that virtualizes the memory usage of DNNs so that both GPU and CPU memories can be used simultaneously [13]. Zhang et al. exploited the iterative nature of training algorithms to derive the lifetime read and write order of variables and hence exploit a memory pool with minimum fragmentation [16]. Similarly, Li et al. devise a memory scheme which does address translation between device and host memory while minimizing its performance impact [10]. The vDDN scheme was further improved by addressing PCI Express-bus contention problems and utilizing an intelligent prefetching algorithm [6]. Matsumiya et al. developed ooccuDNN, a computation blind scheme for processing data beyond GPU capacity by swapping target data whenever it is required for computation and overlapping communication and computation [4]. TOFU is a system to automatically partition dataflow graphs with minimum interdependencies to spread them across multiple GPUs for faster computation with minimum overhead [15]. In aggregate, work on probabilistic systems such as gradient descent has yielded good results, and this is partly due to

the inherently error resistant nature of these systems. However, the same does not apply to physics or geometric simulations which are highly precise in nature and cannot automatically recover from incorrect or missing data

Existing big data processing frameworks such as Spark and Flink have also seen work targeted towards accelerating them using heterogenous CPU-GPU usage. Chen et al. presented a pipelined multi GPU architecture for MapReduce which is able to exploit multi-tier memory to process datasets which might not fit on GPU memory [2]. Endo exploited locality of calculations in stencil computations to provide an effective way to scale to multi-tier memory systems including GPU VRAM, DRAM, NVMe etc. [3]. HeteroSpark, a GPU-accelerated heterogeneous architecture integrated with Spark, which combines the massive compute power of GPUs and scalability of CPUs and system memory resources for applications [9]. GFlink adapts the Flink framework to provide an in-memory computing architecture on heterogeneous CPU-GPU clusters for big data processing [1].

As we saw, the existing solution either require specialized platform or hardware support, require explicit support from the practitioner for specific algorithms, are not suited to a deterministic environment or are not set designed around taking advantage of the unique execution lifecycle of agent based modeling.

Chapter 3. METHODOLOGY

3.1 OVERVIEW

In this section, we detail our methodology and the engineering process undertaken to add heterogeneous computing support to the Multi Agent Spatial Simulation Library.

We begin by describing the architecture of MASS CUDA, with special attention to salient features relevant to our implementation, and then proceed to.

3.2 ARCHITECTURE OF MASS CUDA

MASS CUDA is a version of the MASS library designed to run on Graphics Processing Units (GPUs) and take advantage of the parallelism available.

It is designed to be functionally similar to the sequential versions of MASS and requires no specialized knowledge of CUDA from the practitioner. Figure 3 shows the major components of MASS CUDA:

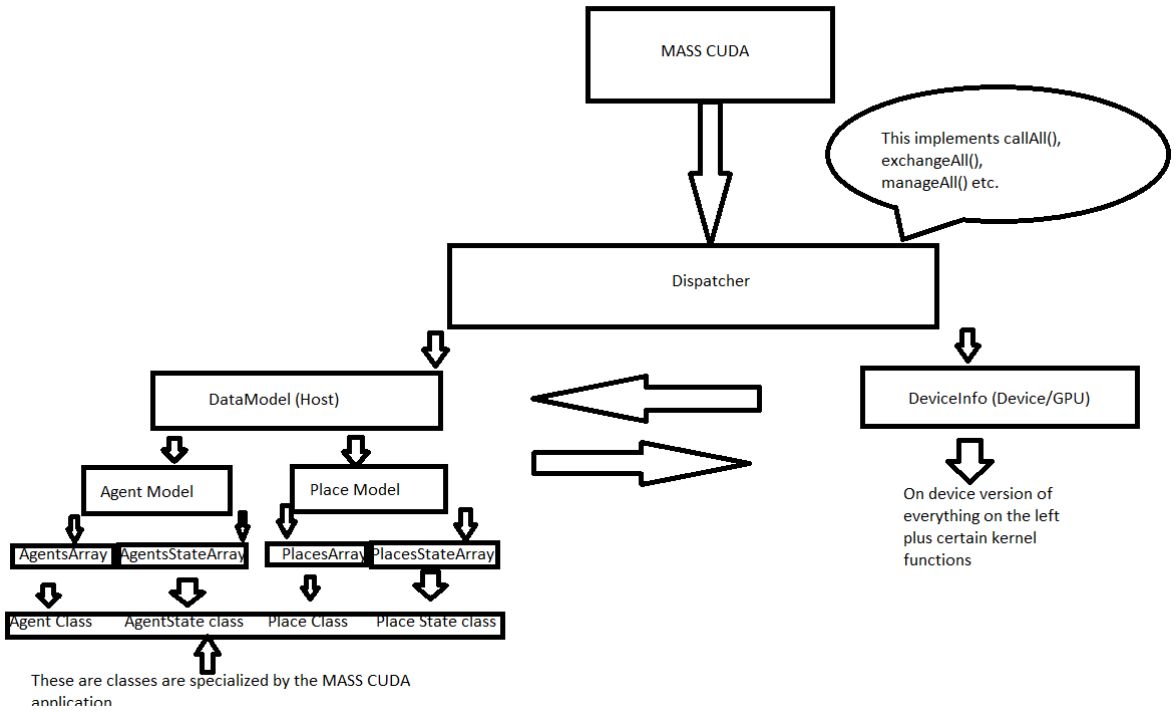

Figure 3 Major components and their hierarchy in MASS CUDA

An application first initializes the MASS environment by calling MASS::init(). This spawns an instance of the Dispatcher class which is the entry point for all library functionality.

MASS CUDA is designed around having an identical data model stored on both the host and the device (GPU) and this is achieved by separating the state of a Place/Agent from its behavior. Hence, classes for MASS Agent and MASS Place define the behavior of an Agent or Place and AgentState/PlaceState store their behavior. These classes are designed in an extensible manner and it is expected that users will extend these to implement the functionality required by their specific use cases.

To illustrate this further, we can look at a simulation of an Ant colony as an example. As in the sequential version of MASS, groups of agents/places are identified by an ID and stored in a higher-level abstraction called the AgentsModel or PlacesModel, as shown in Figure 4. A singular state array holds the state information for all Places/Agents corresponding to that ID; this makes copying data from the device to the host much easier as the entire state array is simply copied to the host.

```
class AgentState {
    friend class Agent;

public:
    int index;
    int size;
    Place *place;
    unsigned placeIndex;
    bool isAlive;
    short slice;

    Place *destPlace;

    int nChildren;  //numbe

    Place *childPlace;


};
```

```
class PlaceState {
    friend class Place;

public:
    Place *neighbors[MAX_NEIGHBORS];  // my nei
    unsigned index;            // the row-major
    int size[MAX_DIMS];   // the size of the Pl

    Agent *agents[MAX_AGENTS]; //agents residin
    unsigned agentPop; // the population of age

    Agent* potentialNextAgents[N_DESTINATIONS];

};
```

Figure 4 AgentState and PlaceState classes

It is expected that agents will be spawned simultaneously on both the host and device and that memory transfers will occur only when results need to be fetched.

3.3 Adding support for Places

The general idea for adding heterogeneous computing support is similar to running MASS across multiple nodes in a networked cluster, as illustrated in Figure 5:

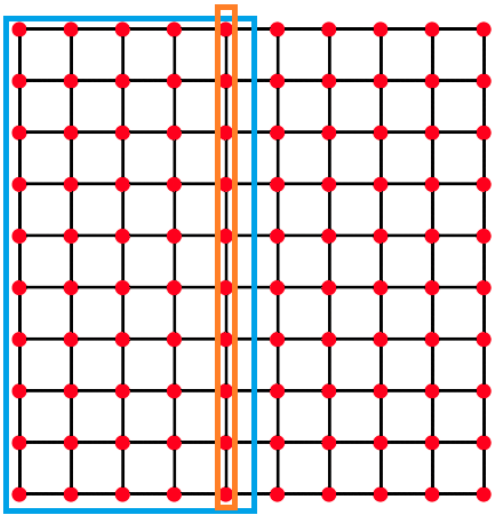

Figure 5 The blue rectangle depicts a slice being shared while the orange outline shows the inactive border areas being transferred to the GPU.

Essentially, we treat all the Places as a two-dimensional grid (currently, MASS CUDA is limited to 2D grids) and partition it into vertical stripes in accordance with the maximum available memory on the GPU. ‘Ghost’ space is included as read only border areas to allow continuity of computation between slices.

As the Dispatcher class is responsible for all communications, the allocation and transfer of data between host/GPU, and the invocation of GPU kernels, all changes are made at this level. As outlined in Figure 6, we make another larger host side data model to store the entire Places grid. Though this breaks the assumption of having an identical model on both the host and device present in the original version of MASS CUDA, it is intrinsic to heterogeneous computing to have asymmetrical models.

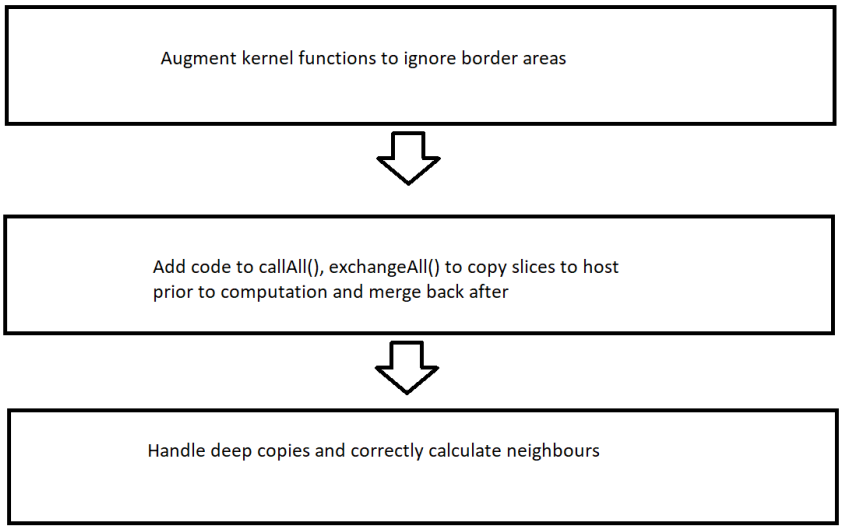

Figure 6 Principal changes at Dispatcher level to enable heterogeneous upport for Places.

Principally, all functionality for Places is provided through two methods: callAll() and exchangeAll(). Places.callAll() calls a user provided function on all the Places with a given ID and exchangeAll() results in all Places updating information about who their neighbors are.

We augment the main callAll() and exchangeAll() functions by adding code to calculate offsets to for slice indexing and copy the state data to a staging data model and subsequently invoke the HostToDevice() method which copies data from the staging array to the GPU. The same process occurs in reverse to copy data back to the staging array via a call to refreshPlaces() from where it is merged into the global data model.

```
__global__ void callAllPlacesKernel(Place **ptrs, int nptrs, int functionId,
        void *argPtr, int size_x, int size_y, int border) {
    int idx = getGlobalIdx_1D_1D();
    int derived_x = idx / size_x;
    int derived_y = idx % size_y;

    for (int i = 1; i <= border; i++) {
        if (derived_y == size_y-i || derived_y == i-1)
            return;
    }

    if (idx < nptrs  && derived_y != size_y-1  && derived_y != 0) {
        ptrs[idx]->callMethod(functionId, argPtr);
    }
}
```

Figure 7 Example of augmented kernel code for ghost space handling.

Once, the data has been copied to the device, kernel functions are invoked on it. The primary change needed in the kernel functions is to decompose the row major index to x and y coordinates and skip over any function calls for the ‘ghost’ space on either side of the actual data, as shown in Figure 7. To prevent performance degradation due to control divergence in the kernels, we do not have multiple branches in the kernel call and the overall implementation is kept very similar to the canonical CUDA programming paradigm of simply checking whether to execute the kernel functionality or not based on the index.

3.4 Adding support for autonomous agents

While much of the process described above also applies for adding heterogeneous Agents, there are several additional factors to take into consideration.

Agents have a lifecycle process (shown in Figure 8) of which each stage must be handled. Principally, an Agent has a callAll() method, which is similar to it’s counterpart in a Place, and a

manageAll() method, which takes care of agent spawning, migration and termination. Additionally, when a certain Agent is placed on the GPU, it's environment, which comprised of the Place it lives on and those it surrounds, must be copied to the device as well. Finally, unlike Places which are statically allocated, Agents are spawned and killed dynamically.

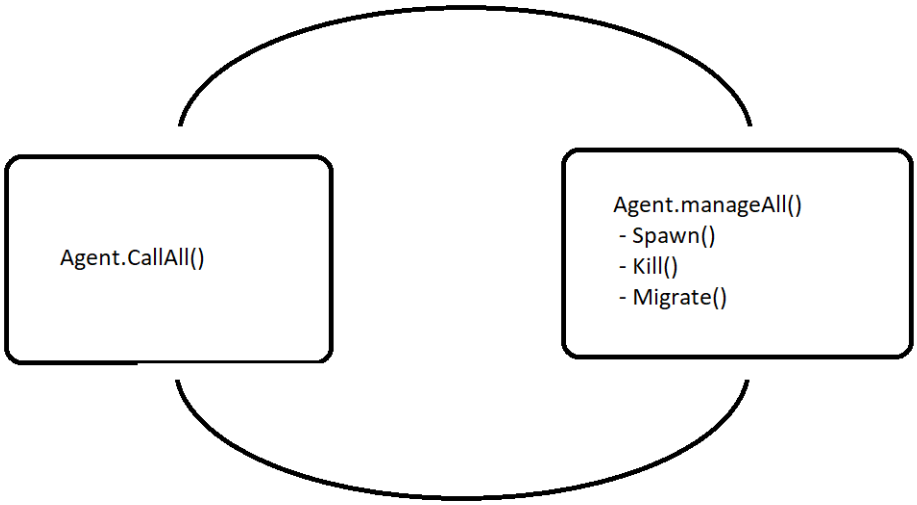

Figure 8 Life cycle of an autonomous Agent

As Agents are spread out over a two-dimensional grid of Places, there are multiple ways to split them into chunks for heterogeneous computing. Two of the most logical partitioning schemes, illustrated in Figure 9, are: split the Place grid into slices and copy all Agents within a specific slice, or alternatively, copy N number of Agents and a slice of Places immediately surrounding each of them. To maximize the number of Agents which can fit onto device memory at each epoch and minimize memory transfers, the latter scheme is chosen.

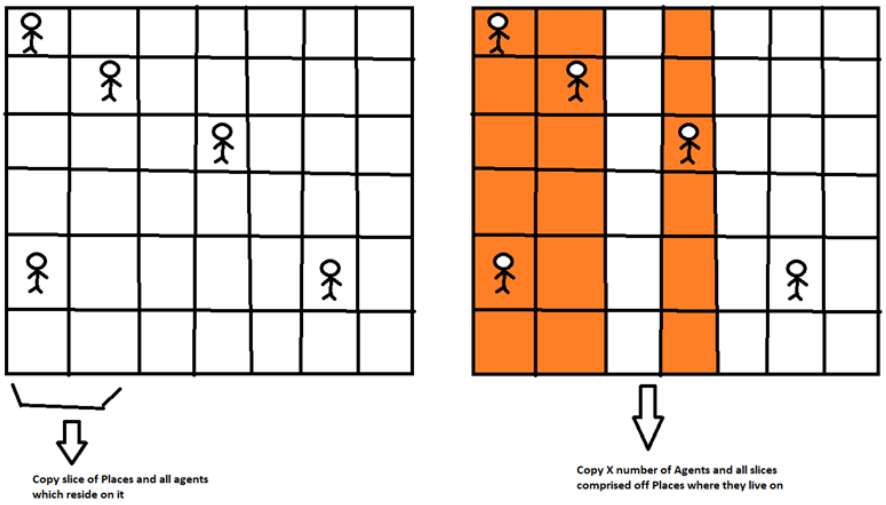

Figure 9 Two natural ways to distribute Agents to slices

For every epoch, the maximum number of agents which can fit onto the GPU memory are selected and copied into a host side staging container for copying. Following this, for each agent in the staging memory, the place it lives as well as its neighboring places are copied into another staging array for Places. Subsequently, host to device memory transfers are initiated and kernel methods are invoked.

To ensure slice to slice migration of Agents, we ensure that agents present in the ‘ghost space’ at the edge of a slice do not have their kernel methods invoked.

3.5 OPTIMIZATIONS, DESIGN CONSIDERATIONS AND EXTENSIBILITY

In this section, we present some of the performance optimizations we made, the design considerations underlying them and our overall architecture, and the emphasis on programmability and extensibility.

One key consideration in the design of our heterogeneous implementation of MASS CUDA was extensibility and the ability to build different features on top of the core functionality. As we form slices in each epoch and copy them to staging areas before a host to device copy for kernel invocation, it would require minimal changes to adapt this for utilizing multiple GPUs connected to the same host CPU (we can simply copy successive slices to different GPUs).

Additionally, granular sizing for individual slices is possible and this opens the door to having multiple devices with asymmetric compute capabilities. As an example, we can imagine a 4-core host with 2 GPUs connected. After analyzing the compute capabilities of all devices, we can push different sized slices to them per epoch from the main memory data model and hence optimally distribute work among them (in case of devices with the same capability, identical slices would be used).

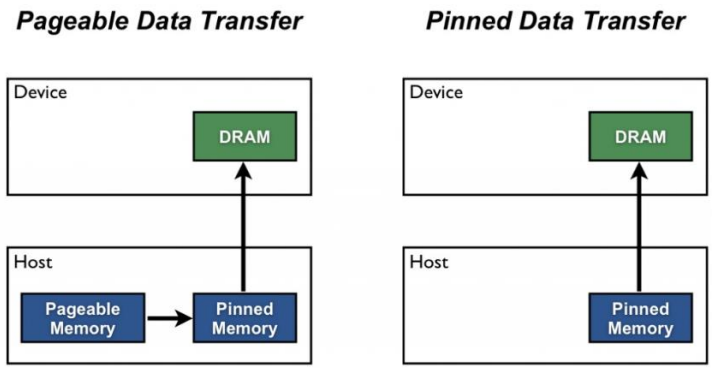

Figure 10 Pageable vs non pageable memory transfer.

Normally, system memory (RAM) is paged by the CPU and operating system. Resultantly, any page in the memory can be invalidated and evicted at any time by the CPU based on its eviction policy (i.e. due to running low on systems memory it might write sections of memory to a page

file on the disk). Due to this, memory transfers in CUDA are routed through the CPU which serves as a bottleneck and can reduce the transfer speed by as much as 50%. We optimize the memory transfers by adding support for making the host side memory buffers non pageable and pinned by using the CUDA API to register our host side data model as non-pageable memory. This allows the GPU to directly copy the data from the system memory and hence take advantage of the full bandwidth of the PCI bus. Using the Nvidia profiler, we find that the runtime is improved by 11% and the memory bandwidth, which is dependent on CPU performance in the case of pinned memory, increases from 3.9 GBPS to 6.1 GBPS. Figure 10 illustrates this change.

As described in the previous section, we initially designed the system around copying data from a slice to an intermediary staging memory buffer before flushing to the device. While this design was motivated to achieve gains in extensibility and abstraction (by essentially decoupling the logic for slice formation from that of workload assignment), it was found to have a performance overhead due to double copying (first from host to staging and then subsequently to the device). We optimize our design by modifying the staging memory to hold only references to the actual Agents and Places in the host side data model rather than storing complete copies of them. This preserves extensibility and abstraction benefits of a staging step while ameliorating the performance concerns. We are able to record a 13% performance improvement due to this change in terms of run time as measured by the Nvidia profiler.

As discussed in the previous section, agents have a manageAll() method in their API (in addition to callAll()) which in turns calls three methods: terminateAgents(), migrateAgents() and

spawnAgents(). As migrateAgents() and spawnAgents() do not cause side effects in each other's domains, we can safely merge them to realize performance improvements. Due to MASS CUDA (and hence Heterogeneous MASS CUDA) program lifecycle, there is a significant performance penalty for each invocation in terms of the memory management cycle. Hence, by merging them into a new migrateAndSpawnAgents() call, we minimize this overhead. The reduction in memory copying nets us a performance gain of 8.5% in terms of execution time and reduces the amount of memory transferred each epoch in the SugarScape user program (described in the next section) by 11%. Further, this change does impact the API as manageAll() abstracts away the individual calls to migrateAgents() and spawnAgents().

To minimize the amount of memory which needs to be transferred, in the case of Agent.callAll() and Agent.manageAll(), we copy only those Places which immediately surround an agent according to the look ahead setting as opposed to copying entire slices. This allows a larger number of agents to fit on the device memory and helps to minimize the memory transfers needed.

As different simulations have differing requirements for the depth of border area exchange needed, we add a parameter to the API to allow the user to manually set the exchange boundary for all Places. As an example, while a heat dispersion algorithm only needs to check the temperature of the adjoining Places, certain wave propagation simulations might need to view data from up 3 places away in each direction. The same principle applies in the case of Places where an agent might need to hop multiple Places in an epoch. To account for the vast majority

of use cases without incurring too heavy a performance penalty, we limit the look ahead to 3 in the case of both Agents and Places.

Finally, as discussed earlier, an important design goal of this work is to ensure that practitioners are able to benefit from heterogeneous computing on ABM systems with oversubscribed memory without needing specialized skills or knowledge of CUDA/memory systems. For example, a possible optimization would be to have users annotate functions which do not have temporal/spatial dependencies so the MASS library could optimize these into the same memory cycle. However, as it requires assistive information from the user, it, along with other such optimizations, was excluded.

Chapter 4. PERFORMANCE AND USABILITY EVALUATION

4.1 TESTING METHODOLOGY

In this section, we test and evaluate our work in terms of performance and ease of programmability. We select three representative algorithms and perform a qualitative and quantitative analysis for each of them.

4.2 EVALUATION ENVIRONMENT

To successfully test and evaluate our work, we require a modern x86 CPU, a Nvidia CUDA 9.0 compatible graphics processor (GPU), a Unix based OS and the NVCC compiler.

We utilize the following hardware and software to form our test bed, shown in Table 1.

Table 1 Summary of testing environment

CPU	Intel Skylake Xeon Platform
CPU Frequency	2.6/3.2 GHz
RAM	12 GB
CUDA Version	CUDA 9
GPU	Nvidia Tesla K80
CUDA Cores	4,992
Memory bandwidth	480 GB/s
CUDA Compute Capability	3.7
Graphics Memory	4 GB (limit set)

4.3 BENCHMARK SIMULATIONS

We select three representative algorithms to benchmark the performance of our work, namely, Heat 2D, SugarScape and Neural Network Growth Simulation.

Heat 2D models heat dispersion across a two-dimensional grid and is modeled using Places as areas on the grid. The problem is well suited for benchmarking performance across a large static system. SugarScape simulates the struggle for survival among agents as they compete to find

food and resources. We implement this using autonomous Agents which move around a 2-dimensional grid modeled with Places. This simulation is well suited for measuring performance for a dynamic system and testing our implementation's strengths and weaknesses. Finally, we run a Neural Network Growth simulation which models Axon and Dendrite growth from Neurons across a surface. This is a considerably more complex problem encompassing both Places and Agents in a stochastic environment and a good addition to our benchmark suite.

4.4 Evaluation outline

We begin by testing the performance of our heterogeneous implementation of MASS CUDA in 4.5-4.7 by comparing it to MASS CUDA, pure CUDA heterogeneous implementations, and sequential CPU versions. Here we set the memory oversubscription level to 2x to simulate the most common use case. Execution time is used as a suitable benchmark to compare performance as it provides a good quantitative comparison point across different implementations.

In 4.8, we illustrate how Heterogeneous MASS CUDA is able to scale past the device memory limitations in contrast to MASS CUDA by measuring the amount of memory used and time taken to solve a problem size. Section 4.9 takes a deeper look into how performance scales with the degree of oversubscription. 4.10 looks at the execution time distribution to gauge how much time is spent.

Finally, 4.11 presents a qualitative analysis of the programmability and ease of use of heterogeneous MASS CUDA.

4.5 HEAT 2D

The Heat 2D algorithm models the dispersion of heat across a two-dimensional surface. Each 'cell' in the grid is initialized with a temperature of zero degrees. Following this, heat is applied for a given amount of time to a subset of cells and the heat then diffuses to the remainder of the cells according to the Euler formula.

Our simulation runs for 100 iterations on an MxN grid. In addition to heterogenous MASS CUDA, results from a heterogeneous version of the algorithm written in Pure CUDA C, non-heterogeneous version of MASS CUDA, as well as a sequential implementation are given in Figure 11.

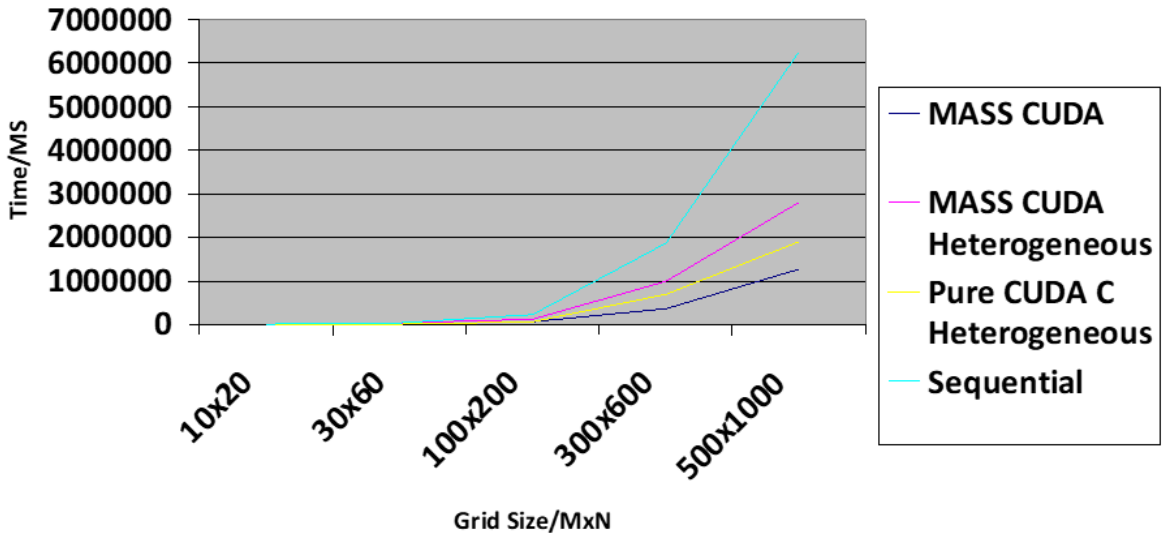

Figure 11 Performance testing results for Heat 2D

As shown by Figure 11, the implementation of the heat dispersion algorithm on our heterogeneous version of MASS CUDA provides scales up reasonably well when compared to the heterogeneous version written in pure CUDA. The performance difference can be explained by two main reasons: firstly, there is some overhead intrinsic to the MASS CUDA library, secondly, the pure CUDA C version is optimized to fold all functionality into a single kernel invocation needing one memory swap per epoch whereas the MASS CUDA version is limited by the API. Hence, while slower than a handwritten version, the additional memory copying is minimal due to there being only invocations of Places.callAll() in every epoch. In addition, the heterogeneous version manages to maintain a positive performance difference as compared to the sequential implementation.

4.6 SUGARSCAPE

Our second algorithm is SugarScape, a popular social survival simulation which first appeared in the 1990's. While the original author's aim to model ancient civilizations on it turned out to be too optimistic, it is nonetheless an interesting program and one which is well suited for our benchmark suite.

We model organisms as Agents in MASS while grids containing sugar are modeled as Places. The simulation begins with mountains of Sugar across the grid and randomly assigned agents having randomly assigned metabolisms. Based on the amount of food and pollution present, agents go to different grid points and either feed or die off.

Once again, we run the simulation for 100 iterations and tabulate our results, shown in Figure 12.

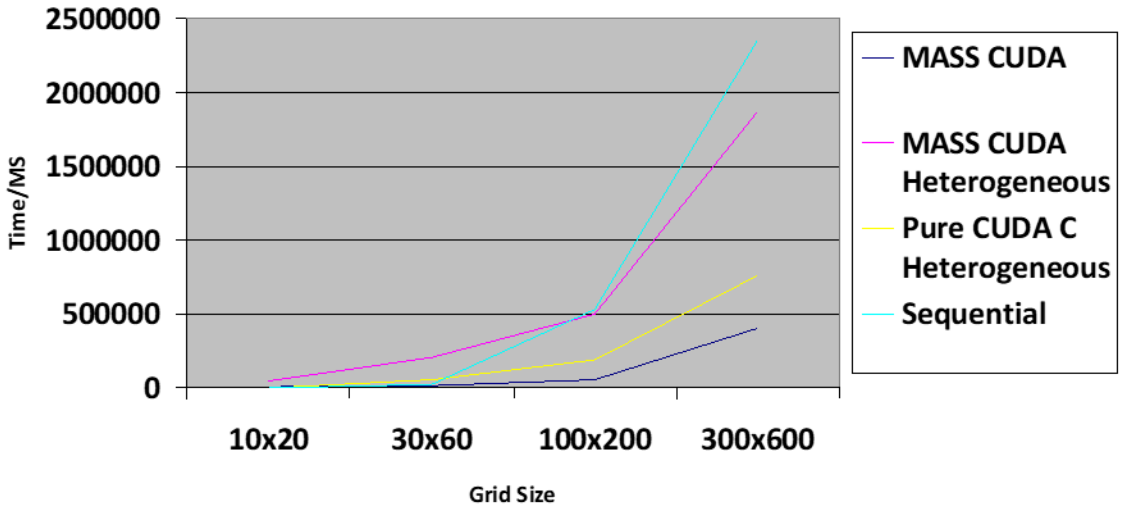

Figure 12 SugarScape benchmarking results

The results of benchmarking SugarScape show that our heterogeneous implementation of MASS CUDA is (relatively speaking) slower to start as compared to the rest of the implementations though it scales well and manages to close the gap, ending up faster than the sequential version for large problem sizes.

The discrepancy in performance can be explained by the implementation of SugarScape (Figure 13). As shown in the diagram below, SugarScape makes multiple calls to the MASS API per epoch. While this not significantly impact the normal version of MASS CUDA, in the case of the heterogeneous version it will lead to memory swapping for each of these calls which incurs a performance penalty.

```
for (; t < max_time; t++) {

    places->callAll(SugarPlace::INC_SUGAR_AND_POLLUTION);

    places->exchangeAll(&neighbors, SugarPlace::AVE_POLLUTIONS, NULL /*argument*/, 0 /*argSize*/);
    places->callAll(SugarPlace::UPDATE_POLLUTION_WITH_AVERAGE);

    places->exchangeAll(&migrationDestinations, SugarPlace::FIND_MIGRATION_DESTINATION, NULL /*argument*/, 0 /*argSize*/);

    agents->callAll(A+nt::MIGRATE);

    agents->manageAll();

    agents->callAll(Ant::METABOLIZE);
    agents->manageAll();


    // display intermediate results
    if (interval != 0 && (t % interval == 0 || t == max_time - 1)) {
        displaySugar(places, t, placesSize);
        displayAgents(agents, t);
    }
}
```

Figure 13 SugarScape main loop

4.7 Neural Network Growth

The third and last algorithm in our testing suite is a Neural Network Growth simulation. We model the problem as a two-dimensional grid comprised of Places where a predefined number of Neurons (modelled as Agents) live. At every epoch, there is a probability of the Neuron growing an Axon or up to 4 Dendrons. As MASS lacks the ability to have an Agent live over multiple Places, an Axon is modeled as both an Agent (head of the Axon) and a Place (tail of the Axon), while Dendrites are modeled entirely using Places. These Dendrites and Axons grow over the grid until one of two termination conditions is met: there is either no more space to grow (every Place is limited to hosting a set number of Axons/Dendrites) or when an Axon ‘fuses’ with a Dendrites having an opposite direction of growth.

Once more, we run the simulation for 100 epochs and set an occupancy limit of 5 Axons/Dendrites per Place. The results of our simulation are tabulated in Figure 14.

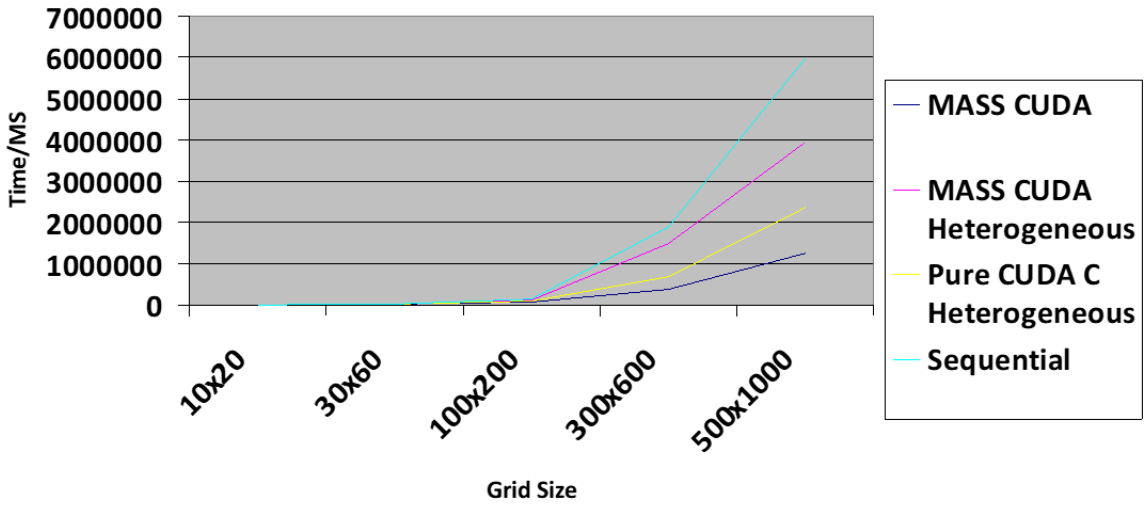

Figure 14 Neural Network Growth benchmarking results.

Despite being a relatively complex problem, neural network growth exhibits good performance on Heterogeneous MASS CUDA. Barring very small problem sizes, it is consistently faster than the sequential version and scales similarly to a pure CUDA C implementations.

4.8 Scaling past memory limits

We now turn our attention to scaling Heterogeneous MASS CUDA beyond the limits of device memory. We run Heat 2D and Neural Network Growth on both MASS CUDA and our heterogeneous versions and tabulate our results.

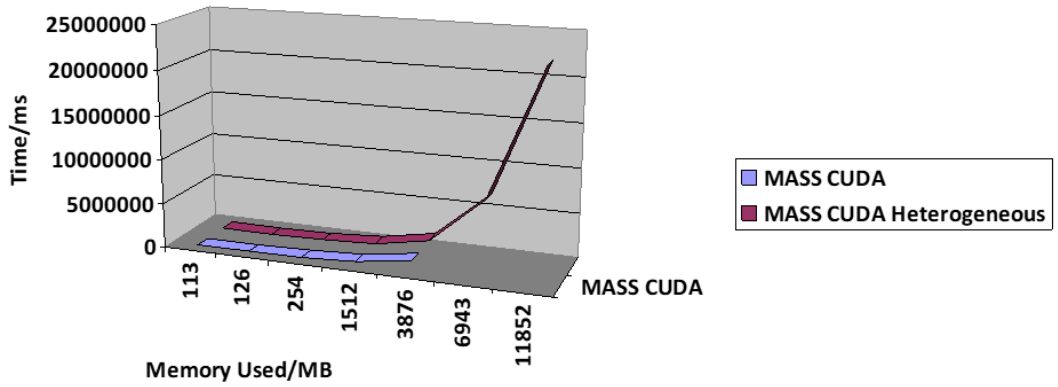

Figure 15 Heat 2D: Scaling past the GPU memory limit.

Figure 15 shows Heat 2D running on both MASS CUDA and Heterogeneous MASS CUDA. As shown, MASS CUDA exits with an out of memory runtime error while allocating 500,000 Places when the memory consumption begins to exceed 3 GB while our heterogeneous version can work in the oversubscribed environment. There is a deterioration in performance at 3.25 million Places as the degree of over subscription goes from 2x to 3x.

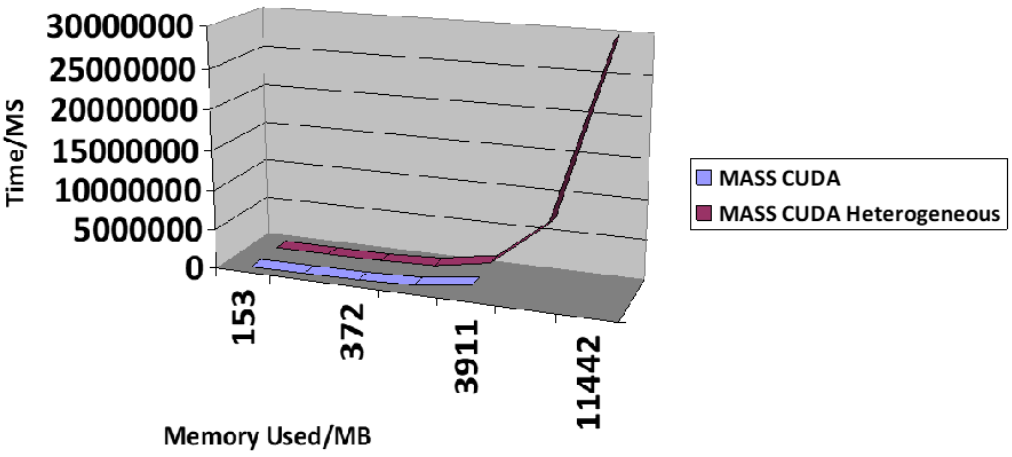

Figure 16 Neural Network Growth: Scaling past the GPU memory limit.

Figure 16 shows the results for running Neural Network Growth on MASS CUDA vs Heterogeneous MASS CUDA. The grid size is shown in the x axis and the number of Agents is set as the number of Places divided by 5. MASS CUDA runs out of memory at 320,000 places and 64,000 agents while our heterogeneous implementations continues to scale. As would be expected due to the more complex program, Neural Network Growth scales slightly worse than Heat 2D.

4.9 Scalability in terms of slices

While the size of a model our heterogeneous implementation of MASS CUDA can run is limited only by the system ram size (which can scale to terabytes), the degree of oversubscription does, of course, incur a correspondingly large performance penalty.

Figure 17 shows the rate at which the time taken to solve a given problem size scales in accordance with the number of times the GPU memory is oversubscribed for Heat 2D and SugarScape.

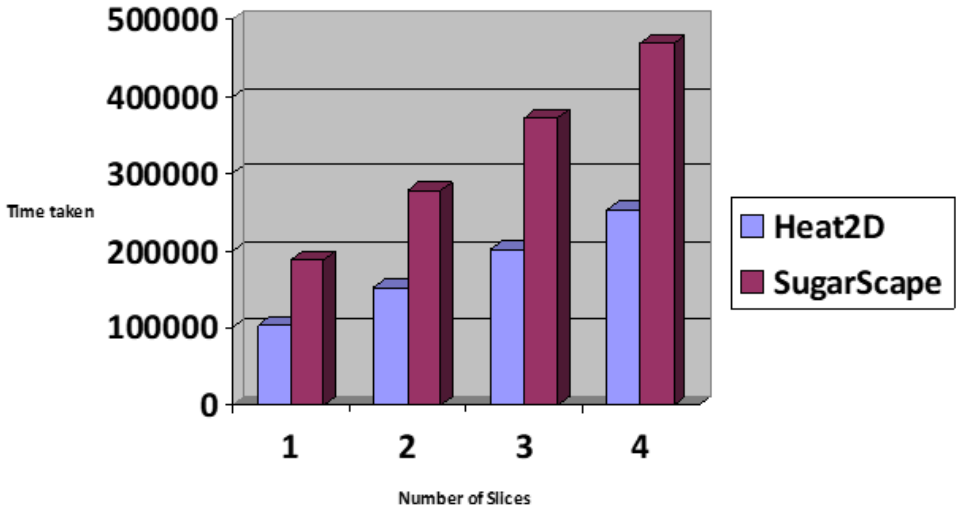

Figure 17 Scalability in terms of degree of memory oversubscription.

As heterogeneous MASS CUDA is primarily IO bound (that is, the performance bottleneck is generally caused by host to device memory transfers), the time taken to solve a given problem size increases significantly as the platform device becomes increasingly oversubscribed.

4.10 A DEEPER LOOK AT EXECUTION TIME DISTRIBUTION

We now take a deeper look at the execution lifecycle of heterogeneous MASS CUDA to decipher where the program spends the most time during execution. Figure 18 below presents the execution time distribution data obtained by profiling a run a of Heat 2D with 2000x2000 grid size and 50 epochs.

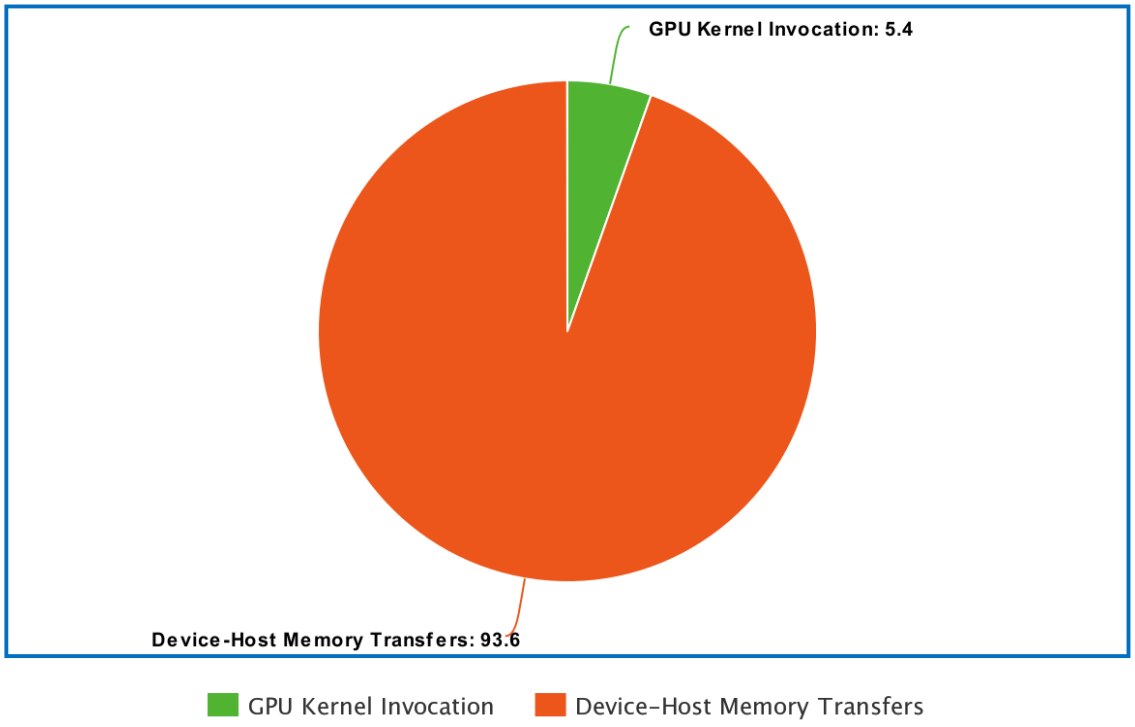

Figure 18 Execution time distribution for Heat 2D.

As shown in Figure 18, nearly 94% of the total execution time is spent copying data back and forth between the host and device. While our results show that heterogeneous MASS CUDA scales relatively well in terms of performance, these findings do raise questions about the possibility of further performance improvements without a platform/hardware level improvement (i.e. using NV Link rather than PCI etc.)

4.11 Programmability Analysis

Having presented an analysis of the performance characteristics of our work, we now qualitatively analyze it in terms of programmability. As mentioned in the introduction, one of the

main goals of heterogeneous MASS CUDA was to maintain the invariant of not requiring any CUDA or heterogeneous programming knowledge.

Based on a comparison of the pure CUDA implementations and our MASS CUDA heterogeneous implementations, we can surmise that we have favorably achieved this goal.

To implement Heat 2D in heterogeneous MASS CUDA, we derive child classes from Places and implement the Euler method to calculate heat diffusion from the neighboring Places. As all CUDA and heterogeneous functionality is abstracted away in the library, the user does not need any specialized knowledge. The only restriction imposed is that the C++ standard library cannot be used as it is not available in the device's execution environment. Contrasted with a pure CUDA heterogeneous implementation, we need to manually copies slices of memory for each iteration in addition to writing kernel functions to do the actual temperature dispersion calculation on the device.

Similarly, the case Neural Network Simulation is even more complicated as we have to account for the growth of Axons and Dendrites across Places, fusing them upon meeting etc. Once the cognitive overhead of manually implementing heterogeneous computing is added, it is likely to place too large a burden for the average practitioner.

Hence, we have managed to maintain the programmability of MASS CUDA and there are essentially no changes needed to run most MASS CUDA user programs on our heterogeneous version.

Chapter 5. CONCLUSION

5.1 CONCLUSION

This project presented a heterogenous implementation of MASS CUDA to scale up agent-based multiprocessing beyond GPU memory limits. By maintaining a host side data model and only selectively offloading Places and Agents to the GPU as needed for computation, we are able to limit memory transfers to those strictly necessary and minimize the impact to performance. Three representative benchmark programs, heat dispersion, sugar scape and neural network growth simulation were used to evaluate our work and demonstrated that it scales similarly in terms of performance to bespoke heterogeneous implementations in pure CUDA C. Further, we emphasize ease of use over performance optimizations where presented with a tradeoff and qualitatively show that the programmability of MASS CUDA is favorably preserved.

5.2 FUTURE WORK

There are several interesting directions to pursue with regards to future work.

In terms of improving the performance of heterogeneous MASS CUDA, we note that the majority of execution time is spent in transferring data to and from the device. A possible approach to reduce this would be to employ on device compression to compress to initially compress the data and then only transferring the compressed data to the GPU where it would be

decompressed, processed and recompressed before sending to the host. Recent advances in GPU based compression make this option more attractive [11].

Secondly, the next logical step would be to expand our work to encompassing multiple GPUs. We believe our overarching system design goes a long way to assisting this and the mechanism for assigning work to the GPU could relatively easily be extended to farm out work to multiple GPUs per epoch with dynamically shared border areas. This would allow multiple GPUs to run in an oversubscribed manner with the host data model being used to swap Agents and Places in and out of the devices as needed.

Thirdly, in conjunction with multiple GPU support as described above, CPU computation could be enabled as well so that host and device cores could work together to process slices with slice sizing per device dynamically calculated based on compute performance for the previous epoch.

Lastly, MASS CUDA (as well as our version with heterogenous computing support) is limited to working with two dimensional grids at the moment. As more and more of the data practitioners need to process is spatial in nature, adding a native internal graph representation would go a long to increasing its real-world usability and applicability.

BIBLIOGRAPHY

[1] Chen, Cen, et al. "Gflink: An in-memory computing architecture on heterogeneous CPUGPU clusters for big data." IEEE Transactions on Parallel and Distributed Systems29.6 (2018): 1275-1288.

[2] Chen, Yi, et al. "Pipelined multi-GPU MapReduce for big-data processing." Computer and Information Science. Springer, Heidelberg, 2013. 231-246.

[3] Endo, Toshio. "Realizing out-of-core stencil computations using multi-tier memory hierarchy on gpgpu clusters." 2016 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2016.L. Cohen, *Time-Frequency Analysis*, Prentice-Hall, 1995.

[4] Ito, Yuki, Ryo Matsumiya, and Toshio Endo. "ooc_cuDNN: Accommodating convolutional neural networks over GPU memory capacity." 2017 IEEE International Conference on Big Data (Big Data). IEEE, 2017.

[5] Kehne, Jens, Jonathan Metter, and Frank Bellosa. "GPUswap: Enabling oversubscription of GPU memory through transparent swapping." ACM SIGPLAN Notices. Vol. 50. No. 7. ACM, 2015.

[6] Kim, Youngrang, et al. "Efficient Multi-GPU Memory Management for Deep Learning Acceleration." 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W). IEEE, 2018.

[7] Kosiachenko, Lisa, Nathaniel Hart, and Munehiro Fukuda. "MASS CUDA: a general GPU parallelization framework for agent-based models." International Conference on Practical Applications of Agents and Multi-Agent Systems. Springer, Cham, 2019.

[8] Kwon, Youngeun, and Minsoo Rhu. "Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning." 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2018.

[9] Li, Peilong, et al. "Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms." 2015 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, 2015.

[10] Li, Shijie, et al. "A Novel Memory-Scheduling Strategy for Large Convolutional Neural Network on Memory-Limited Devices." Computational Intelligence and Neuroscience 2019 (2019).

[11] Lu, Li, and Bei Hua. "G-Match: A Fast GPU-Friendly Data Compression Algorithm." 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2019.

[12] Markthub, Pak, et al. "Dragon: breaking gpu memory capacity limits with direct nvm access." Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. IEEE Press, 2018.

[13] Rhu, Minsoo, et al. "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design." The 49th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Press, 2016.

[14] Vijaykumar, Nandita, et al. "Zorua: A holistic approach to resource virtualization in gpus." The 49th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Press, 2016.

[15] Wang, Minjie, Chien-chin Huang, and Jinyang Li. "Supporting very large models using automatic dataflow graph partitioning." Proceedings of the Fourteenth EuroSys Conference 2019. ACM, 2019.

[16] Zhang, Junzhe, et al. "Efficient memory management for gpu-based deep learning systems." arXiv preprint arXiv:1903.06631 (2019).

[17] Zheng, Tianhao, et al. "Towards high performance paged memory for GPUs." 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016.