MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks

Nathaniel B. Hart

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2015

Committee:

Munehiro Fukuda

Clark Olson

Michael Stiber

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

MASS CUDA: Abstracting Many Core Parallel Programming From Agent Based Modeling Frameworks

Nathaniel B Hart

Chair of the Supervisory Committee:

Associate Professor Munehiro Fukuda, Ph.D

Computing & Software Systems

Agent based modeling is the practice of simulating complex interactions by modeling the behavior of a single agent in the interaction, then observing the emergent behavior that occurs when many of those Agents interact. As simulation size increases, the computational requirements can become prohibitive, and parallelization of a simulation can be complex. As most users of Agent Based Models are not programmers by trade, most rely on frameworks to develop and parallelize processing of these models. NVIDIA's CUDA programming language is of interest as it can to harness the massively parallel capabilities of graphics processing units. This paper proposes an architecture for such a framework that uses the CUDA programming language to accelerate Agent Based Models while hiding the programming complexities from the end user.

TABLE OF CONTENTS

List of Figures

Acknowledgements

I would like to acknowledge the extraordinary patience and support of Dr. Munehiro Fukuda, who provided the vision for the MASS Library, the guidance necessary to help me through the complexities of producing a project of this scale, and the unfailing faith he had in the ability to find a workaround, even when bewildering obstacles presented themselves. This project would not exist without his presence and help.

I would also like to thank Robert Crovella, Solutions Architect Manager at NVIDIA, who provided prompt and accurate responses to my vague and confused questions whenever I posted on Stack Overflow. His willingness to share expertise in CUDA provided me with insight into my various problems and obstacles that I may never have obtained without him.

Most of all, I would like to thank my wonderful wife, Caitlin Hart, without whom none of this would have been possible. She put her own desires on hold to make time for my path to this thesis, never complaining about late nights or weekends spend working. Her support of me and my educational goals gave me the strength to see this enormous undertaking through to the end.

1 INTRODUCTION

It is common to see movies displaying dire predictions of disease transmission, zombification, or the spread of some toxic gas using computer models. Always shown are evocative graphics, maps with some red miasma spreading across the US, or 3D animations of a cloud spreading through a building. What is never shown is the effort, expertise, and hardware required to create the computer model used to make these dire predictions. Nobody sees the teams of subject matter experts in disease working with programmers to produce these models. Thus many people are unaware how difficult it can be to accomplish.

One method of producing this kind of prediction is agent based modeling, also called ABMs. Unlike mathematical models that attempt to quantify an entire system using equations, ABMs simulate complex behavior through the interaction of large quantities of objects, called Agents, that each contain the modeled behavior of an individual actor in a larger system. ABMs are interested in the emergent behavior that becomes apparent when thousands or even millions of agents interact to simulate the controlled chaos of real life. On the surface, it is intuitive: when modeling the spread of a human disease, just model a person, the probability the disease will spread on contact, and the time it takes the immune system to cure itself. Then copy it a million times to watch how disease spreads through the population in the program.

Even further from the silver screen are the incredible computation hurdles that are overlooked using cinematic slight-of-hand. Computing the interactions between massive numbers of agents requires a great deal of computational resources. Depending on the complexity of the simulation and the framework used to create it, run times can range from hours to weeks. ABMs can benefit greatly from parallel processing, but it is rare to find a subject matter expert who is also an expert in parallel programming. This means that subject matter experts must rely on programming teams or frameworks and libraries to create ABMs that also meet performance requirements. The MASS Library is one of these libraries.

The MASS Library has already been implemented in Java and C++. As CUDA [1] harnesses the massively parallel power of graphics processing units (GPUs), we are interested in discovering how well CUDA would be suited to parallelize the computation of the MASS Library. The version presented in this thesis focuses on a new version of the MASS Library that uses CUDA to parallelize processing instead of the current approach of distributing computation across a cluster of computing nodes. The architecture, requirements, and design decisions are all presented in this thesis.

1.1 CUDA OVERVIEW

CUDA, or Compute Unified Device Architecture, is a library developed by NVIDIA to allow GPUs to be used for high-performance computing. Like any problem sphere, CUDA comes with its own vocabulary, and two terms are critical to understanding any discussion: 'host' and 'device'. In general purpose GPU programming, it becomes necessary to differentiate between the computer and the GPU attached to it. The computer is referred to as the "host" and the GPU as the "device." CUDA developers consistently use this language to describe actions like host to device memory transfer. This thesis uses this convention as well.

GPUs are capable of massively parallel processing when properly harnessed. While a powerful desktop computer (at the time of writing) may have 16 CPU cores, a GPU may have several thousand cores that can execute in parallel. GPUs are capable of spinning up tens of thousands of threads at a time. Thread switching is easy, so there is little cost for mapping multiple threads to a single hardware thread, unlike CPUs where exceeding the number of hardware threads can actually damage performance.

Because of the number of threads GPUs support, CUDA excels at data parallelism, where data processing can be distributed to multiple threads. This contrasts with task parallelism, where the different tasks necessary for processing occur in parallel. This means CUDA is particularly well-suited to accelerating computations that are rigidly defined, well known ahead of time, and fall into the realm of embarrassingly parallel. Problems like matrix multiplication, where any value in the destination matrix can be calculated in any order without affecting performance or results, are ideal applications of CUDA's massively parallel

capabilities. Contrast this with a common task-parallel algorithm like quicksort, where each sub-problem executes in its own process or thread and the number of threads changes each step. These tasks would need to be executed in proper order to get the correct results.

The MASS Library has a mix of data parallel and task parallel concepts that is less well adapted to CUDA. Also, as it is a library, insight into the problems to be solved is limited only by the agent based modeling problem space. Unlike matrix multiplication, where carefully designed algorithms can take advantage of all possible hardware to achieve maximum acceleration, the MASS CUDA library is intended to be "good enough" to solve all types of Agent Based Models at the expense of optimal ABM performance in any one case.

1.1.1 Thread Model & SIMD

When GPUs execute, one thread is used per data element in some array of data. This array can be 1D, 2D, or 3D. In order to coordinate execution, threads execute in a series of thread blocks of no more than 1024 threads. If there are more than 1024 data elements, multiple blocks are used. Multiple thread blocks are called a grid. Thus a thread block can hold x * y * z threads up to 1024, and a grid can be made up of X * Y * Z thread blocks.

A thread block is further broken down into groups of 32 threads called warps. All threads in a warp execute a single instruction simultaneously on registers containing differing data. If the instruction is `x += 2` then this instruction will execute on multiple cores simultaneously, but the value of `x` may be different in the register on each core's register.

Flow of control passes from the host to the device via kernel functions. These are functions preceded by the `__global__` flag. Part of calling a kernel function is specifying the thread block and grid dimensions. The device then executes the kernel function code once per thread.

1.1.2 Thread Synchronization

As the number of threads frequently exceeds the number of processors on the GPU, some warps will execute before others. There is no guarantee of execution sequence of individual warps or of thread blocks. Traditional thread synchronization would handle this uncertainty with techniques like semaphores, mutex locks, and thread waiting. CUDA has less advanced synchronization because it is designed for embarrassingly parallel applications. The threads within a thread block can be synchronized using the `_syncthreads()` call built into CUDA. This will perform barrier synchronization for all threads within a thread block, preventing race conditions when using shared memory, but it is impossible to synchronize all thread blocks within a kernel function. The basic unit of synchronization in CUDA is the kernel function call. The only sequential guarantee in CUDA is that all thread blocks in one kernel function will finish before executing the next kernel function.

1.2 RESEARCH GOALS

The MASS Library, or Multi Agent Spatial Simulation library, is an agent based modeling library that allows a subject matter expert to create Agent Based Simulations without needing to engage in parallel programming, yet still reap the performance benefits of parallel processing. This thesis examines the hypothesis that the full functionality of the MASS Library can be implemented using CUDA without significant impact on the programmability or usability of the library. The purpose of this thesis is to test this hypothesis by implementing the MASS Library in CUDA, record the design of the MASS CUDA library, and examine the extent to which the MASS Library design can be successfully implemented using CUDA. The scope of this project is to implement the MASS Library in CUDA, allowing a simulation to run on a single host. The distributed nature of the other implementations of the MASS Library is outside the scope of this project, although it is worth considering how this single-machine implementation could be extended to function in a distributed environment.

This thesis will show that the MASS CUDA Library implements the existing MASS framework using GPUs to parallelize computation with only minor modifications, and will present a novel technique for enabling polymorphism and interfaces to be used for objects intended to be transferred between host and device.

1.3 SECTION DESCRIPTIONS

Chapter 2 will focus on the MASS Library, its goals, and the way in which it is intended to be used.

Chapter 3 will describe related work, including other Agent Based Modeling frameworks, both with and without GPU acceleration. It will also discuss the differences between previous attempts at implementing MASS CUDA and the current version.

Chapter 4 will present the architecture used for the MASS CUDA library and the requirements that drove design decisions. It will include additional details about CUDA that are relevant to design decisions.

Chapter 5 is a programming analysis that compares the code necessary to write ABMs using CUDA vs using the MASS CUDA library.

Chapter 6 covers current performance statistics as well as what work can be done to improve performance.

Chapter 7 addresses the next steps in research.

2 MASS CONCEPT

The MASS Library is intended to assist subject matter experts (SMEs) in creating Agent Based simulations that address subject specific problems. It provides a simulation environment and base classes that have implemented the basic functions necessary to run an Agent Based Model. It provides the ability to schedule interactions between agents by providing the ability to execute the same command on entire groups of agents in parallel with the `callAll()` function. Any number of functions can be called before the model

is updated with the `manageAll()` function. This allows SMEs to model problems like disease – drug interaction by carefully modeling what a disease behaves like (reproduction rates of organism, reactions to drug, nourishment requirements), and how the drug molecule acts (diffusion rates, number of organisms it can kill). Then the MASS Library will allow the user to specify how many of each are released into the simulation. The emergent behavior could help the SME determine minimum doses that reliably cure the disease, the maximum dose beyond which additional medication will have no effect, or any number of other factors the SME is concerned with.

2.1 HIGH LEVEL GOALS

The cornerstone of the MASS Library is the programmability and simplicity of the Library. Because the "plumbing" of the ABM is already provided, a user can focus on the problem rather than the implementation. This decouples the difficulty of parallel programming from the subject matter expertise required to run a useful simulation. As all implementation details such as clustering or simulation partitioning are hidden, the user works with a Single System Interface that allows ignorance of system details.

As the MASS Library is intended to run large ABMs that contain potentially millions of agents, the computational requirements of a simulation may be massive, and will benefit greatly from parallelizing the computation. Using the MASS Library should significantly shorten the time necessary to run any given simulation. While a custom written simulation co-authored by a SME and parallel programmer may perform better, the performance goal of the MASS Library is to perform close enough to a custom implementation that the extra effort would be deemed wasteful.

Because the parallel implementation is provided, all details regarding parallel programming should be hidden from the user. The user should code as if the simulation is sequentially executed. There should be complete encapsulation of multi-threading, process forking, GPU specific code, and code that compensates

for parallel implementation complexities. A user should never need to write a monitor class, worry about race conditions, or make a single call to the CUDA API.

The result of this encapsulation of parallel programming details is a very effective Single System Interface. The user is unconcerned with the underlying details of simulation distribution, device configurations, load balancing, or the myriad of other details that the Library coordinates. The user interacts with the simulation as a single, sequentially executed process.

2.2 PROGRAMMING MODEL

The user is intended to access the functionality of the MASS Library through inheritance. The Agent and Place class are provided, and inheriting from these classes provides the interface and functionality necessary to run a simulation. The user then uses the Library's API to instantiate collections of their derived Agent and Place classes and execute the functionality they added. The inheritance model is necessary not only to provide the built in functionality, but to allow the MASS Library to interact with user defined classes in a polymorphic manner.

This means the MASS Library relies heavily on the Open/Closed Principle [2] wherein code is "Open" for extension but "Closed" for modification. The code in the MASS Library is intended to be a black box, where the required functionality is accessed through class extension, but the user does not have access to alter the original Library functionality.

This results in a very modular design for the MASS Library. There is a clearly defined Library space, where parallel programming details live, and a user space, where the Library is used, but complex programming is absent. As the Library changes, updating to the latest version should be as simple as replacing the compiled library within the user's project with the latest version.

3 RELATED WORK

3.1 EXISTING ABM FRAMEWORKS

The Swarm Toolkit [3] is an early toolkit written in C intended to simplify creating ABMs. Like the MASS Library, the simulation progresses via function calls rather than a timer. Unlike MASS, it allows nested agents, where a group of agents is called a swarm (thus the name). Agents within a swarm can interact, and swarms can interact with each other as well. Furthermore, Swarm is sequential in nature, meaning it can handle smaller simulations only. The provided example has 65 agents only [4]. This is much smaller than the thousands or millions we hope to obtain with MASS. MASS also has the additional concept of Places, or an environment in which and with which the agents can interact.

Repast Simphony is a Java based ABM toolkit written and maintained by Argonne National Laboratory [5]. It integrates with the Eclipse IDE and provides a way to create simulations using Java. Unlike MASS, which uses inheritance to pass along functionality, Repast has users program their agents as plain old java objects, then register them with the simulation using XML. Users must also write the logic to instantiate and place agents using a provided simulation context object and placement grid. The environment in which the Repast agents operate is not open for extension or modification, and functions as a Cartesian coordinate tracking system for assisting agents to perform neighborhood searches. It is designed for execution on a single work station or small cluster of computers, and the samples show very small simulation sizes of only 100 – 200 agents [6].

Repast also has a version designed for large clusters and supercomputers called Repast HPC. While this may provide the scalability, it begins to require more knowledge of parallel programming, going so far as to require writing and compiling MPI code [7]. Though this software may be powerful, it does not have the same emphasis on programmability that is the MASS Library does.

MASON is a modern Java ABM toolkit created by Sean Luke at George Mason University [8]. It began as a rewrite of Repast before the Java version was available. Like MASS, agent classes are derived from provided classes. It provides some ability to customize the execution environment through selecting from several premade grids, but it has a severe limitation in that it is not a distributed application. It is intended to execute within a single multi-threaded process on one machine. This limits its scalability and performance gains to how many threads the most modern chipset can support.

D-MASON [9] is a distributed version of MASON maintained by the Università degli Studi di Salerno in Italy. It allows for clustered and load-balanced execution of MASON simulations. The only significant limitation of this framework is the limitation to two-dimensional simulation grids [10], while MASS can run simulations in an environment of any dimensionality.

3.2 GPU AGENT BASED MODELING

3.2.1 Bare simulations

While Agent Based Models have a rich history, it is only recently that ABM and general purpose GPU (GPGPU) have come together. Most work has been done in writing GPU accelerated versions of existing ABM scenarios. Scholarly articles examining the intersection of GPUs and ABMs have focused on custom implementations of specific problems like traffic modeling [11] or sugarscape [12]. Little effort has been put into creating reusable frameworks like Repast or D-MASON for GPU based ABMs. A survey of articles published from 2008 – 2013 yielded the following statistics:

- 12 Articles directly addressing agent based modeling on GPUs ranging from 2008– 2013 [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22]
- 9 showed an increase in performance for ABMs [11] [12] [13] [14] [15] [16] [18] [20] [22]
- Performance improvements ranged from 20x speedup [18] to 250x speedup [15]

The remaining 3 measured other factors regarding ABMs:

- Proof of concept [17] [19]
- Performance as a function of scale [21]

An important feature of MASS that rarely shows up in existing GPU ABMs is the concept of agent spawning. Most of the simulations used a fixed number of agents, and the simulation was either concerned with positioning of the agents (like the Molecular Diffusion benchmark simulation) or with agent state changing between some series of values (like FluTE, where disease spread is modeled with agents that are either infected or uninfected).

Of the articles, most focused on performance gains with non-reusable code, made no attempts to hide CUDA details, and ignored programmability entirely. Only two attempted to provide "hiding" of GPU details, and both were about the same framework: Flame GPU.

3.2.2 Flame GPU

Flame GPU is a GPU adaptation of an older ABM framework called Flame. It is unique among the existing GPU ABM frameworks in that it both hides GPU programming details and allows for agent spawning. Unlike MASS, it does not provide an execution environment the agents can interact with.

Flame GPU takes a very unique approach to handling the abstraction of GPU details from the user. Instead of using an Object Oriented inheritance approach to encapsulating framework functionality, it modularizes the Agents themselves. Agents, their functions, fields, messages, and memory allocations must all be defined using an XML schema. Agents field types are limited to `int`, `float`, and `double`. Each field requires 4 - 7 lines of XML, and arrays must also define a concrete length. The XML also maps the functions and scripting code to predefined stages in the Flame GPU framework, such as initialization.

The XML is then processed and combined with the actual code for agent functions to form the final simulation. Although the details of how Flame GPU actually works are not revealed, one can surmise that the simulation then uses the XML schema to arrange data on the GPU and execute a sequence of defined functions on that data.

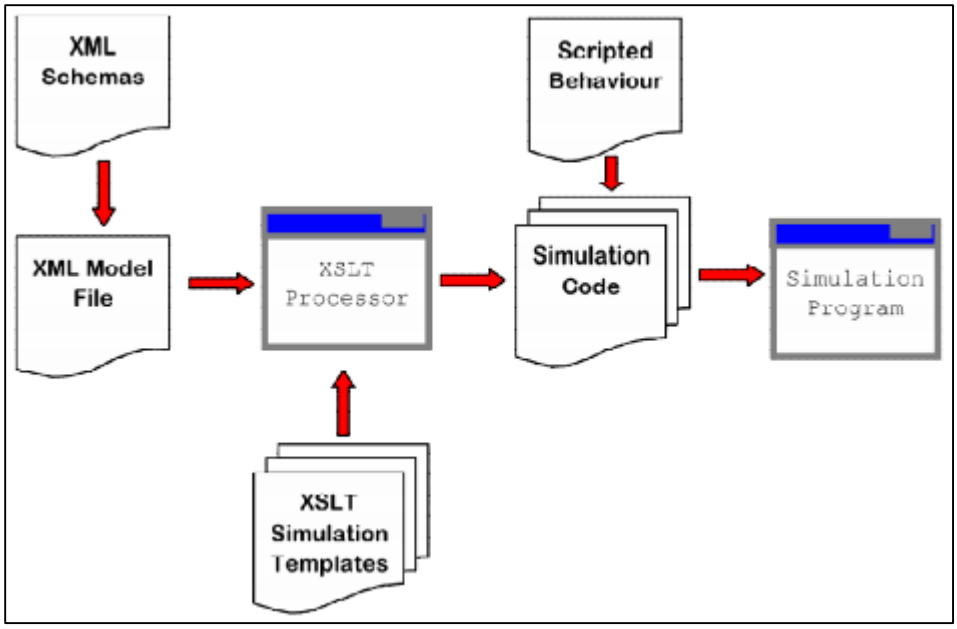

Figure 1 Flame GPU Modeling and Simulation Process [23]

While this approach works, it is not an intuitive programming model. It is an approach to modeling that is entirely driven by the architecture on which the model runs. A complex simulation would result in extensive XML, all of which must be hand written. Also, XML does not allow for debugging, resulting in code that can be difficult to fix if something is wrong. The MASS Library attempts to provide more flexibility on what type of data you want to process requiring only basic programming skills for effective use.

3.3 Previous Work on MASS CUDA

There have been three students who have made prior progress on CUDA versions of the MASS Library. From June 2011 to December 2012, Tosa Ojiru and Piotr Warczak worked on the MASS CUDA Library to run a Wave 2D simulation. Robert Jordan carried their work further by running a Wave 2D simulation across multiple GPUs.

In order to understand the differences between the previous versions of the MASS CUDA library and the present version, one must examine the code and the way in which the user uses it to produce a simulation.

3.3.1 Ojiru & Warczak

Ojiru's thesis and Warczak's project both showed significant performance gains by using CUDA, and their combined work left a significant body of code that should have provided a foundation on which to build. Their work, however, did not satisfy many of the requirements of the MASS Library design. The library failed to hide the parallel programming details from the user, and at times, even side-stepped the library's key feature: the API.

The following is an excerpt of code showing what a user would write to run a Wave 2D simulation [24] (code has been reformatted, commented-out code is deleted, and CUDA API calls/classes highlighted in bold):

```
1:  void Wave2D::doSimulation()
2:  {
3:    cudaEvent_t start, stop;
4:    cudaEventCreate( &start );
5:    cudaEventCreate( &stop );
6:    cudaEventRecord( start, 0 );
7:
8:    int width = size, height = size;
9:    size_t size1D = width * height * sizeof( float );
10:   size_t  sizeOfWave2D = sizeof( Wave2D );
11:   Wave2D* d_wave2d;
12:   float*  d_space0;
13:   float*  d_space1;
14:   float*  d_space2;
15:   cudaMalloc( &d_wave2d, sizeOfWave2D );
16:   cudaMalloc( &d_space0, size1D  );
17:   cudaMalloc( &d_space1, size1D  );
18:   cudaMalloc( &d_space2, size1D  );
19:
20:   dim3 dimBlock( threadsPerBlock, threadsPerBlock, 1 );
21:   dim3 dimGrid(  blocksPerGrid, blocksPerGrid, 1 );
22:
23:   for(int t = 0; t < iterations; t++ )
24:   {
25:     doSimulation_kernel<<<dimGrid, dimBlock>>>( d_wave2d,
26:         d_space0, d_space1, d_space2, size, t );
27:     cudaMemcpy(z0, d_space0, size1D, cudaMemcpyDeviceToHost);
28:     cudaMemcpy( z1,d_space1,size1D,cudaMemcpyDeviceToHost);
```

```
29:       cudaMemcpy( z2,d_space2,size1D,cudaMemcpyDeviceToHost);
30:     }
31:
32:     cudaEventRecord(stop,0);
33:     cudaEventSynchronize(stop);
34:     cudaEventElapsedTime( &exectime, start, stop);
35:
36:     //Select GPU  device with the most cores i.e. with Compute
37:     //Capability  of at least 1.3
38:     int deviceNum;
39:     cudaGetDevice( &deviceNum );
40:     cudaDeviceProp deviceProp;
41:     memset( &deviceProp, 0, sizeof( cudaDeviceProp     ) );
42:
43:     deviceProp.major =   1;
44:     deviceProp.minor = 3;
45:     cudaChooseDevice( &deviceNum, &deviceProp );
46:     cudaGetDeviceProperties( &deviceProp, deviceNum );
47:     cudaSetDevice( deviceNum );
48:
49:     //      iterations == # of iterations
50:     printf( "GPU(MASS)=%s, size=%d, iterations=%d,
51:         blocksPerGrid=%d, threadsPerBlock=%d, ",
52:         deviceProp.name, size, iterations, blocksPerGrid,
53:         threadsPerBlock );
54:
55:     cudaFree( d_wave2d );
56:     cudaFree( d_space0 );
57:     cudaFree( d_space1 );
58:     cudaFree( d_space2 );
59:  }
```

This code represents what is required to run a Wave 2D simulation using Ojiru's version of the MASS CUDA library. This is where a MASS Library user would be working, and should rely on the MASS API to hide parallel programming details. Instead, we see 15 unique CUDA classes and API calls. There is also no use whatsoever of the MASS API calls such as `callAll()` or `exchangeAll()`. Furthermore, there is actually a kernel function call on lines 25-26 which means the user would actually be writing the CUDA kernel functions that perform the Wave 2D simulation analysis.

While this is excellent CUDA programming by Ojiru and Warczak, it is in no way the MASS Library. Any MASS Library user with this level of ability in CUDA would not need a framework for creating an ABM, and would instead write a bare CUDA simulation that is tailored to their exact problem.

3.3.2 Jordan

Robert Jordan did some excellent work distributing the Wave 2D simulation across two GPUs. This involved sharing border data between them, much like data is shared between clustered computers using MPI. It also involved coming up with both a local and global indexing scheme that allowed a single GPU to process local data while maintaining understanding of a partition's position in the overall simulation.

Jordan's work came very close to translating the MASS Library API and programming model to a GPU platform in that it successfully hides the CUDA details and partitioning from the user. In order to do so, it makes some compromises to functionality and programmability.

The first compromise it makes is violating the "open/ closed principle" of Object Oriented Programming, where code should be open for extension but closed for modification. The following code from Robert Jordan's project [25] is the header file for the MASS Library's Place class (several boiler plate functions labeled "DO NOT MODIFY" have been removed at line 21 for brevity):

```
1:  /* DO NOT MODIFY */
2:  #include "range2d.h"  // Point2D
3:
4:  /* USER-DEFINED INCLUDES */
5:  // TODO: add any includes you need
6:
7:  class Place {
8:  public:
9:
10:   /* Device Constructor
11:    * DO NOT MOFIFY
12:    */
13:   __device__ Place(dim3 dimensions, Point2D coordinates, Place*
14:           neighbors, int index) {
15:     this->dimensions = dimensions;
16:     this->coordinates = coordinates;
17:     this->neighbors = neighbors;
18:     this->index = index;
```

```
19:    }
20:
21:  // several “DO NOT MODIFY” functions removed for brevity
22:
23:  private:
24:
25:    /* DO NOT MODIFY */
26:    dim3 dimensions;      // the 3D size of the Grid
27:    Point2D coordinates;  // the global coordinates of this Place
28:  /*  DO NOT MOFIFY  */
29:  __device__ Place* getNeighbor(int hOffset, int vOffset) {
30:      // boundary checks
31:      if (coordinates.x + hOffset < 0) return NULL;
32:      if (coordinates.x + hOffset >= dimensions.x) return NULL;
33:      if (coordinates.y + vOffset < 0) return NULL;
34:      if (coordinates.y + vOffset >= dimensions.y) return NULL;
35:      int offset = index + vOffset * dimensions.x + hOffset;
36:      return &(neighbors[offset]);
37:    }
38:
39:    /* USER-DEFINED VARIABLES */
40:    // TODO: add your variables here
41:
42:    /* Called while executing Places.updateAll(). Only the state
43:     * internal to this Place should modified during this method,
44:     * otherwise, race conditions occur.
45:     */
46:    __device__  void updateState() {
47:      // TODO: user implementation
48:    }
49:
50:    /* Called by MASS while executing Places.updateAll(). This is
51:     * your opportunity to safely update the message field.
52:     */
53:    __device__ void setMessage() {
54:      // TODO: user implementation
55:    }
56:  };
```

The expected use pattern in MASS would be for a user to extend this class and implement some abstract functions. Instead, the user is intended to modify the Place.h source file with the code necessary for the desired simulation, then recompile the library into the simulation. This places two serious limitations on this version:

- Modification of source code may result in errors: what happens if user modifies something marked "DO NOT MODIFY"?
- Future updates to MASS Library will be difficult to incorporate into a project of this type: you can't just replace the library module, you must copy/ paste your code into the new source files.

An additional problem is that a simulation is permanently limited to a single type of Place. Deep within the library code, where the user place instantiation actually occurs, places are created concretely using the `Place` constructor rather than a user derived class constructor (see line 21).

```
1: __global__ void gpuCreatePlaces( Place *p, Range2D* range,
2:       void *args, int argSize, dim3 gridSize ) {
3:
4:     Point2D local( blockIdx.x * blockDim.x + threadIdx.x,
5:         blockIdx.y * blockDim.y + threadIdx.y );
6:     Point2D global = range->converToGlobal( local );
7:
8:     // are we inside the range?
9:     if ( range->contains( global ) ) {
10:      // index calc
11:      int index = local.y * range->getWidth( ) + local.x;
12:
13:      // extract this Place's arg
14:      void *arg = (void *) ( (char *) args + argSize * index );
15:
16:      // execute
17:          // TODO allow for user derived Place objects
18:      p[ index ] = Place( gridSize, global, p, index );
19:      p[ index ].init( arg );
20:      p[ index ].update( );
21:    }
22:  }
```

If this model were also extended to the `Agent` class implementation, it would limit simulations to one type of agent as well. ABMs like Wator, which relies on both Shark and Fish agent types, would be impossible to create.

Deeper analysis of the code also revealed very tight coupling among the classes and the assumption that a simulation would run on two devices, making it difficult to build on top of this code. While it would be

difficult to add Agent functionality to this version of the library, it would be impossible to encapsulate the simulation execution to the point that simulations could be run on a cluster of GPU accelerated computers.

Jordan's MASS CUDA library clearly pushes boundaries in multi-GPU partitioning and makes excellent use of the MASS API to hide parallel programming details, but its architecture prevented it from running simulations that currently work even on the Java and C++ MASS Library versions. These limitations resulted in a decision to rewrite the MASS CUDA library from the ground up in an effort to implement the entire MASS API and functionality in a manner that renders the changes transparent from the user's perspective.

3.4 LATEST VERSION

The current version of the MASS CUDA library seeks to redress the shortcomings of earlier versions, hide all CUDA programming from the end user, allow the user to create any Agent Based Model by extending library classes, and implement all of the MASS API in a manner that works and feels identical to the Java and C++ MASS Libraries.

3.4.1 CUDA Details Hidden

The CUDA implementation details are completely hidden from the user, save three details:

1. The project must be compiled using the NVidia Cuda Compiler (NVCC) instead of gcc or g++
2. All files that would normally be a .cpp file must be a .cu file instead.
3. Any function in a user's Place or Agent implementations must be prepended with the macro `MASS_FUNCTION`. This macro contains the CUDA function flags to compile for both host and device functionality.

These are minor requirements and a user need not even know why they are doing it in order for them to work. The first detail is due to the fact that despite all the hiding of CUDA details, use of this library makes

any user project a CUDA project. Code the write for their Agent and Place class implementations will need to be complied for device-side execution. This forces the use of the NVCC compiler.

The second requirement is connected to the first due to the way function templates compile with NVCC, which compiles the template function for execution in the environment specified by the file suffix. A .cpp file will generate host code, and a .cu file will generate device compatible code. As the functions that create user-defined classes on the GPU must be instantiated within a .cu in order to compile, all function templates must be used within .cu files.

The third detail is simply a way of hiding a CUDA specific detail. Functions intended to be executed on the host and device must be prepended with the flags `__host__` and `__device__`. The `MASS_FUNCTION` macro simply inserts those flags.

3.4.2 MASS API Implemented

This version implements the full MASS API. Each simulation begins with a static call to `Mass::init()` and ends with a call to `Mass::finalize()`. The end user extends the Place and Agent objects to create concrete classes for their simulation that they the control through `callAll()`, `exchangeAll()`, and `manageAll()` function calls on their respective Places and Agents collections. Translating a simulation from MASS C++ to MASS CUDA requires minimal work because of these similarities. The code necessary to execute a Heat 2D simulation in MASS CUDA is almost indistinguishable from the same simulation in MASS C++ save the Places instantiation (lines 14-15). Instead of creating a Places collection with the `new` operator, it is done through a function template call that specifies a user type to instantiate.

```
1: void Heat2d::runMassSim(int size, int max_time, int heat_time,
2:          int interval) {
3:
4:      string *arguments = NULL;
5:      int nGpu = 1;
6:      int nDims = 2;
```

```
7:      int placesSize[] = { size, size };
8:
9:      // start a process at each computing node
10:     mass::init(arguments, nGpu);
11:
12:     // distribute places over computing nodes
13:     double r = a * dt / (dd * dd);
14:     Places *places = mass::createPlaces<Metal, MetalState>(0,
15:         &r, sizeof(double), nDims, placesSize, 0);
16:
17:     // create neighborhood
18:     vector<int*> neighbors;
19:     int north[2] = { 0, 1 };
20:     neighbors.push_back(north);
21:     int east[2] = { 1, 0 };
22:     neighbors.push_back(east);
23:     int south[2] = { 0, -1 };
24:     neighbors.push_back(south);
25:     int west[2] = { -1, 0 };
26:     neighbors.push_back(west);
27:
28:     // start a timer
29:     Timer timer;
30:     timer.start();
31:
32:     // simulate heat diffusion in parallel
33:     int time = 0;
34:     for (; time < max_time; time++) {
35:
36:           if (time < heat_time) {
37:                 places->callAll(Metal::APPLY_HEAT);
38:           }
39:
40:           // display intermediate results
41:           if (interval != 0 && (time % interval == 0 || time ==
42:                  max_time - 1)) {
43:                 displayResults(places, time, placesSize);
44:           }
45:
46:           places->exchangeAll(Metal::EXCHANGE, &neighbors);
47:           places->callAll(Metal::EULER_METHOD);
48:     }
49:
50:     // finish the timer
51:     Logger::info("Elapsed time with MASS = %.2f seconds.",
52:                 timer.lap() / 10000 / 100.0);
53:
54:     // terminate the processes
55:     mass::finish();
56: }
```

4 MASS CUDA ARCHITECTURE

The architecture breaks the MASS Library into three parts: the view, or API, the data model, and the command executor, or presenter. This is commonly known as Model-View-Presenter or MVP. It is similar to MVC, except the model and the view communicate only via the Presenter. This design attempts to achieve a true separation of concerns between the MASS API, the data model for the simulation, and the computational resources available to execute the simulation. This design not only simplifies the implementation of the MASS Library in CUDA, but maximizes reusability for future growth. When the data model is decoupled from the way in which it is used and the way in which results are computed, we begin to offer the computation as a service, allowing further applications to be built on top of this architecture without modification of the existing architecture.

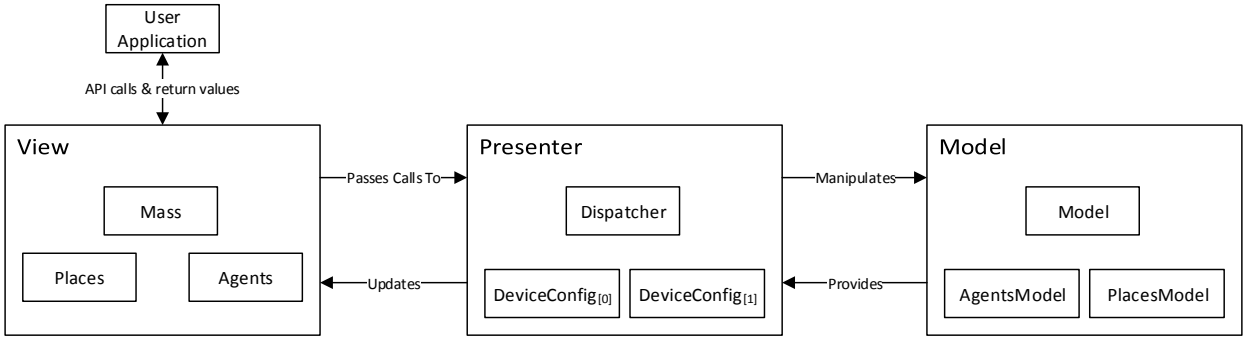

Figure 2 Model View Presenter Architecture

In this structure, the user application can create new Places and Agents collections, use the template functions in MASS, and make calls that control the simulations. The user application will be unaware of the data model partitioning, and the data model will be unaware of how its partitions are dispatched to device resources. As the user application requests access to the results of the simulation, the current

simulation cycle will complete, the most current state information will be copied from the devices to the host, and will be communicated to the user.

4.1 REQUIREMENTS

The design for the MASS CUDA library was driven by many requirements. First among them was to recreate the design, use patterns, and API already created in other versions of the MASS Library. A critical requirement was to minimize the extent to which a user would need to be aware of using the CUDA library vs the C++ library. Ideally, a user should be able to use the same code for both, but this was not quite achieved.

Limitations of the CUDA language drove the majority of the changes a user would notice. Foremost among these is the fact that the host and device are separate memory spaces that cannot access eachother. The only way to copy data between host and device via `cudaMemcpy()`. Like the conventional `memcpy()`, it writes a given number bytes from one memory address to another. This creates a key weakness in that `memcpy()` just copies bytes. The memory address of a pointer would be copied while the data it points to in the heap would be left behind. This forces us to avoid dynamically allocated memory structures like arrays or pointers to memory in the heap.

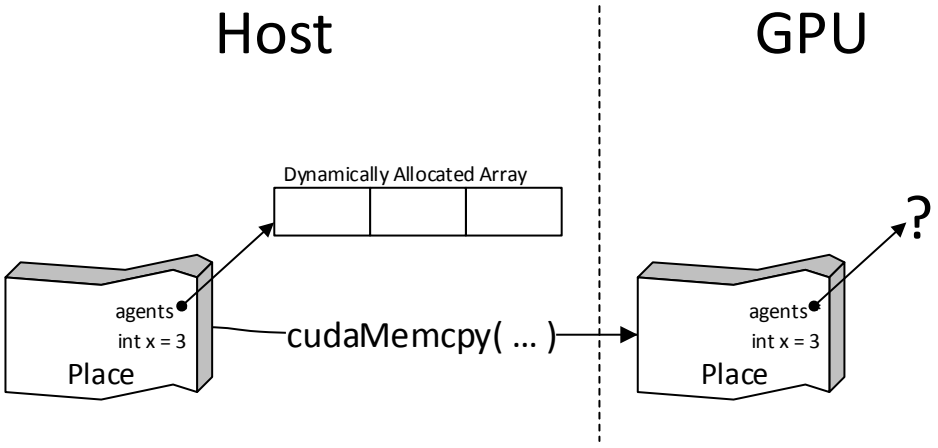

Figure 3 Implications of cudaMemcpy

The MASS Library provides an Agent and Place class in order to define the Agent and Place API as well as to provide basic functionality for Agent and Place behavior. Though some of the functions are intended to be overridden and implemented by the end user, others are intended to be used as is. This very common pattern, of defining an interface and interacting with polymorphic objects via that interface, presents a problem in CUDA: virtual function tables.

When an object is instantiated, the instance will contain a pointer to the virtual function table, where the executable code for a function resides. Because of this, an object that contains virtual functions will only work within the memory space where it is created. Using `cudaMemcpy()` to move an instance from host to device or back again will copy the address of the virtual function table, but not the table itself. This means that an object must be instantiated on the device in order for the virtual functions in the Agent and Place base classes to work. This is very limiting for two reasons:

1. A derived class that is instantiated on the host and is copied to the device will have no executable code. Attempts to call kernel functions on that object will fail.
2. A derived class that is instantiated on the device cannot be copied to the host and accessed programmatically, as the virtual functions will remain behind on the device.

This then begs the question of how to transfer state on and off the device. After all, the goal is to perform computation on the device, then view the results on the host. But, if the device instances are copied to the host, none of the field accessors will function properly. This led to the decision to split behavior and state into two different classes. The behavior class, Agent or Place, contains all the functions as well as a single pointer to a generic `AgentState` and `PlaceState` class that contains all the fields. A user will subclass both to create their own classes. An array of concrete behavior classes are instantiated both on the host and device, as well as an array of state classes of equal size. Each behavior instance is assigned a unique state instance. Memory transfer is achieved by copying the state array from host to device and back again.

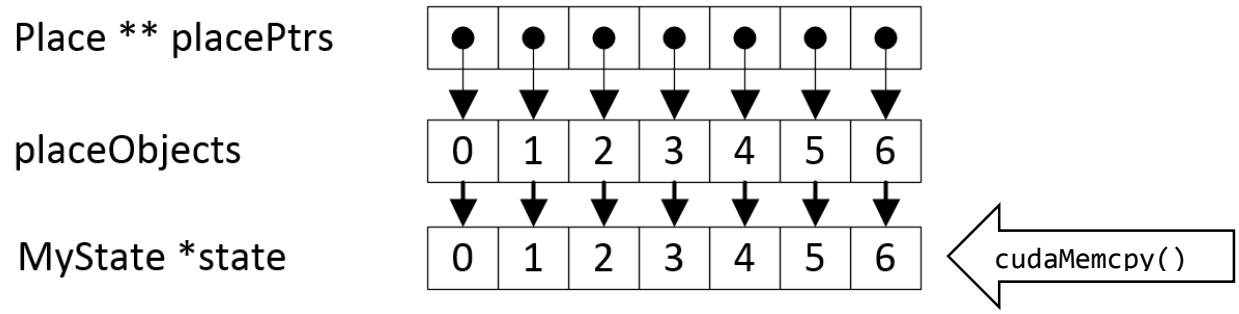

Figure 4 Separation of behavior and state

Another requirement that drove the architecture for this library is the need to be extensible to allow for new applications to be built on top it. A library that allows for running a simulation on one device is helpful, but limited. The internal workings of the architecture assume that a simulation will eventually run on multiple devices. This resulted in additional complexity, but modifying the library to allow such functionality can focus solely on implementing the algorithms rather than modifying the existing program structure and interfaces to allow for new assumptions.

A library that can use multiple devices on one host is better, but still caps the maximum size of a simulation. Ideally, the library should hide the device implementation so well that a machine can process a portion of a simulation, then share the results with other nodes in a cluster. Doing so would allow a simulation to be distributed across a cluster of GPU accelerated computing nodes, allowing extremely large ABMs to run on any number of computers. This removes the size cap and enhances the potential of the library. This requirement was one of the key reasons for the use of the Model-View-Presenter architecture.

4.2 WHY MODEL-VIEW-PRESENTER?

The Model-View-Presenter architecture more effectively hides the data model from the view than Model View Controller and provides the modularity necessary to extend this library to new applications and execution environments. As the uses for the library changes, it will be easy to replace the view with a

graphic user interface, a communication module, or any other necessary classes to facilitate the final goal. This makes the library modular and promotes future reuse.

An alternative architecture could have been MVC, but in MVC, the view is updated via event notifications from the data model. The MASS Library does not have events, as the program is entirely user driven using a main function that makes calls to the MASS API.

4.3 VIEW MODULE

The View is composed of the Mass, Places, and Agents classes. These are the only classes with which the user application can interact, and include the ability to start and stop the simulation, create new Agents and Places collections, and issue commands to those collections. The View layer is intentionally thin, and serves mainly to route user commands to the presenter and give controlled access to simulation values. The presenter is not exposed to the user because it has direct access to the data model and contains many device interaction details that do not concern the user. The View's API helps to hide the implementation details and simplify the user application.

4.3.1 MASS

The Mass module exists primarily to provide a programmatic way to interact with the simulation resources. The user can initialize and finish a simulation. These calls respectively instantiate and destroy the presenter and model layers. This is a very thin class.

4.3.2 Places & Agents

The Places and Agents modules gives a user the ability to issue commands to a particular collection of user-defined instances within the MASS simulation. Like the Mass module, they are very thin layers that pass commands on to the Presenter. A single command to a Places or Agents instance will execute that command on all partitions of the Data Model while hiding the partitioning scheme from the user. These classes help the user easily differentiate between the many different types of Places or Agents that may be required to run a given simulation. They also hide the complexity of partitioning the simulation, handling

communication between multiple GPUs, and coordinating the transfer of the most recent data from the device to the host upon user request.

4.3.3 Places and Agents Creation

Places and Agents are created via a call to the static template function `createPlaces<T,S>()` or `createAgents<T,S>()` in the Mass class. This call will make a similar call to the presenter, which will in turn create the PlacesModel instance with the correct number of partitions, initialize the new Place and State instances on the device (parallelizing object creation), and set the pointers to allow transfer of state data to and from this Places instance. Commands issued to an Agents or Places instance will not act directly upon the data contained within the instance. Instead, the command will be routed to the presenter, which will act upon and update the model.

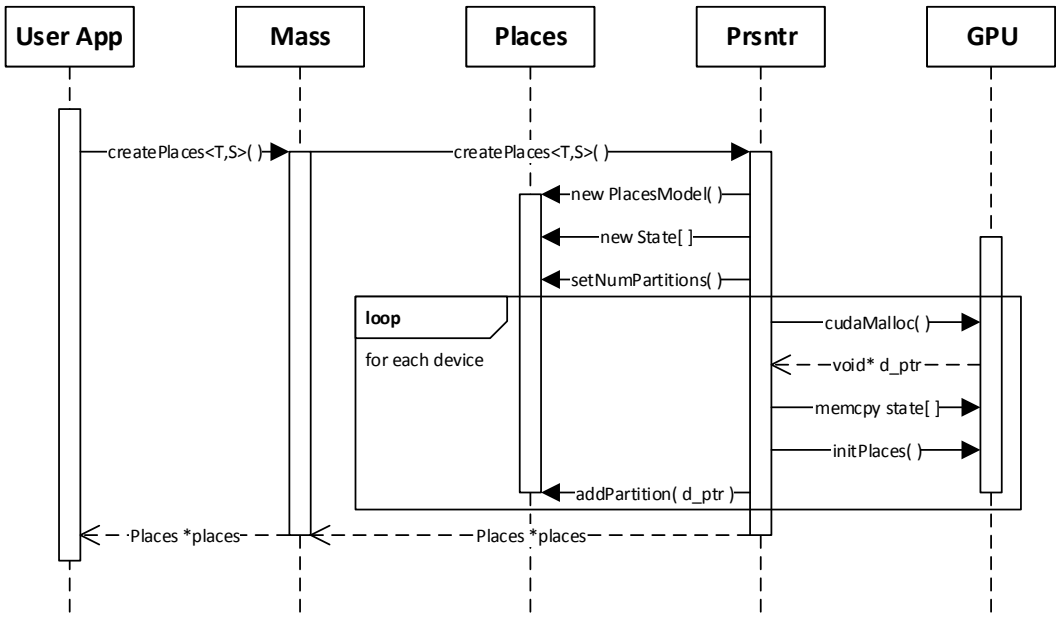

Figure 5 Collection Instantiation

4.4 MODEL MODULE

The data model comprises all PlacesModel and AgentsModel collections created for the given simulation, and it stores the raw data for the data model. This includes most of the Places and Agents specific data like

dimensionality, size, and type size. Its functions are exposed only to the presenter, which coordinates loading, manipulating, and unloading state from available devices.

Places and Agents appear to be a single collection to the user application, but are really composed of one or more data partitions. Taken together the partitions will hold all the data for a single collection, but in chunks that are "bite-sized" for the available devices. Each collection hides this partitioning from the user application by reassembling the partitions as necessary when the user requests access to the data. It also serves up the partitions to the Dispatcher module as necessary to perform allow computation. A single simulation may be composed of multiple Places and Agents collections, and a partition must coordinate the simultaneous loading of all Place elements and their corresponding Agents onto the same device. The goal of the partitioning is create divisions that cut across all Places and Agents collections in a model, allowing each partition to execute as a small simulation of its own.

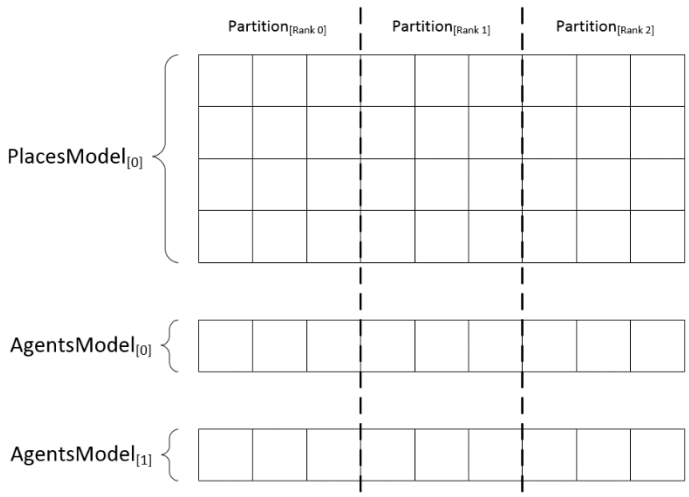

Figure 6 Data Model Partitioning

4.4.1 PlacesModel

A PlacesModel object stores information about the user-defined space, such as the number of dimensions, the size of each dimension, and the number of bytes in a single user-defined place. Note that there is an array of Place pointers and an array of void objects. The user-defined objects are hidden from the user in the private `placeObjects` array and presented to the user via the

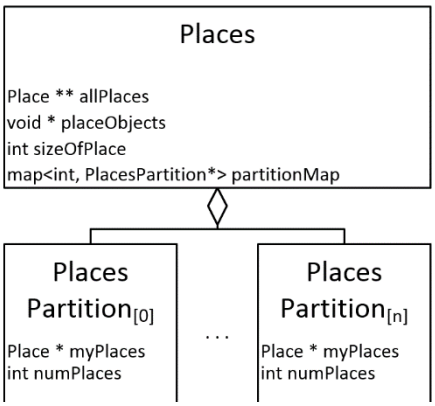

Figure 7. PlacesModel Design

`allPlaces` pointer array (Figure 7). The reason for this seemingly duplicated storage is twofold:

1. Only the user application can access all custom type information. The MASS Library cannot know or store user types, depending on polymorphism to interact with the objects.
2. CUDA depends on the `cudaMemcpy()` function to copy data from the host to the device. This function's performance depends on copying from large chunks of contiguously allocated memory. This means that we can't simply use a vector of Place pointers, as the objects are not guaranteed to be allocated in a contiguous memory block.

The array of places is divided into partitions of ranks numbered 0 to `n`. Each rank references a portion of the places objects via a careful indexing scheme, where each partition stores a pointer `myPlaces` to the beginning of its portion of the `placeObjects` array and the number of places in its segment. A pointer to the ghost space of adjoining partitions is also stored within each partition. This allows the single array of Place elements to be divided into partitions using only pointers and element counts instead of disjoint arrays (Figure 8). Data can then be copied to the device and back using these partition pointers. When copying a portion from the host to the device, the place elements from left_ghost_space to right_ghost_space go to each device. This is to allow all the places for a partition to correctly reference values in its neighbors that are in another partition. If there is only one partition, there is no ghost space.

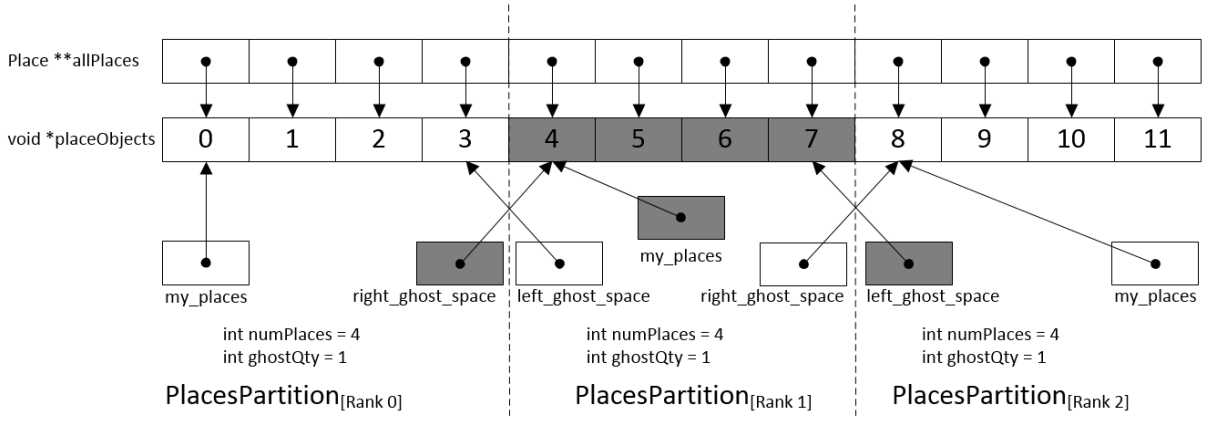

Figure 8 PlacesModel Partitioning Detail

A critical abstraction achieved by the Places collection is that of n-dimensional space. The n-dimensionality presents a small hurdle to device computation, which can only natively support up to 3-dimensional arrays. In order to work around this limitation and allow compatibility with the full range of expected uses, n-dimensional arrays are flattened into a 1-dimension array using row-major indexing. Places also provides functions to translate between row-major index and standard multi-dimensional array indexing based on the dimension sizes and dimensionality of the Places space. The number and size of dimensions cannot change in the lifetime of a Places instance.

4.4.2 AgentsModel

An Agents instance stores information about the user-defined agents, such as the places object upon which these agents work, the number of bytes in a single user-defined agent, or the number of living agents of this type in the simulation (dead agents are flagged as such and excluded from any calculations). An Agents collection requires a single Places object upon which to reside.

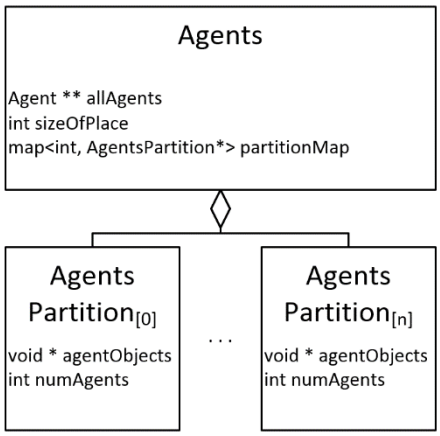

Figure 9. AgentsModel Design

Differences between Places and Agents roles forces a different storage design. Like Places, the Agents collection is composed ranks numbered 0 to `n`. Unlike Places, number of agents and the partition on which they reside can change. This means that the length of the `agentOjects` array may need to change during the lifetime of the simulation. As such, each agents partition needs to contain its own array of agents, and expand that array as the simulation demands. The `allAgents` pointer array in the Agents class can reference the objects contained within each partition, but it will need to be reconstructed each time any partition changes the size of its `agentsObjects` array. This reconstruction may be delayed until the user application requests the `allAgents` array.

An agents partition will contain an array of type void with enough bytes to store the Agent elements for a Places partition. Each partition can be copied by a `cudaMemcpy()` call copying `numAgents` elements beginning at the `agentsObjects` pointer.

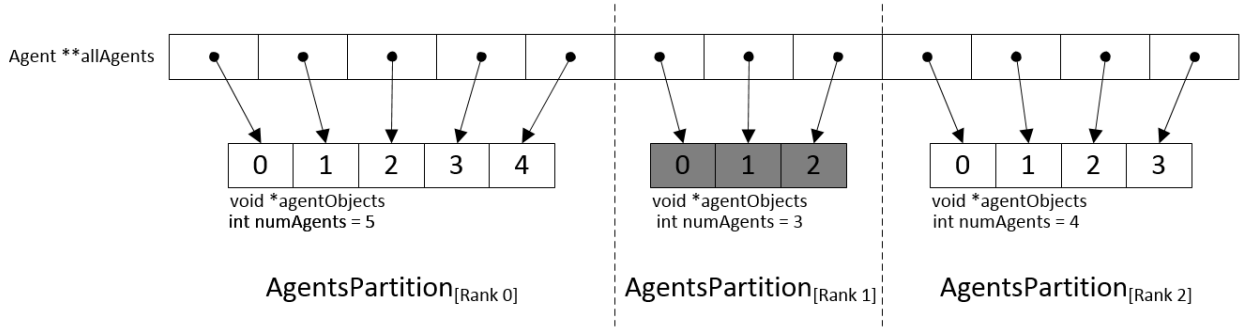

Figure 10 Agents Partitioning Detail

4.4.3 Implications of Partitioning

Rigid coordination must occur between an Agents collection its corresponding Places class. If `PlacesModel P` has three partitions, `AgentsModel A` must not only must have three partitions as well, but each partition must store only those agents that reside on the `Places` Partition of the same rank. This coordination is complicated by the fact that the individual agents are not stored in an order that reflects the order of the places upon which they reside. This means that each time the data model exchanges partition boundaries, the agents that reside in ghost space must be identified and either replaced by the correct agents from the next partition, or killed.

The partitioning implementation is especially critical to future growth and development. Prior work either assumed that all work would be done on a single device, so no partitioning was necessary, or that it would be done on two devices. In either case, the assumptions about GPU count with respect to simulation partitions was hard-coded into the program. The MASS CUDA Library provides a structure that can run a simulation with any number of partitions, and will even allow for dynamic re-partitioning of the collection if a simulation outgrows its existing partitions. The partitioning scheme will allow for new functionality to focus solely on the implementation of algorithms, not the restructuring of the entire library.

4.5 PRESENTER MODULE

The command and control logic resides the Presenter module, which can receive commands from the View Module. The dispatcher will carry out those commands by getting a partition of the data model and loading it on to the first available device, executing the command, then (if necessary) retrieving the data chunk from the device. This would mean that Model layer can be unaware of the device logic and be responsible only for "knowing" how to divide the various collections of `Agents` and `Places` into a specified number of partitions. Note that Figure 11 also depicts ghost space on each device. `Places` copies its ghost space from the other partition onto the local device in order to facilitate local calculations. In each case, the agent that resides on that ghost space is also copied into the other GPU as well.

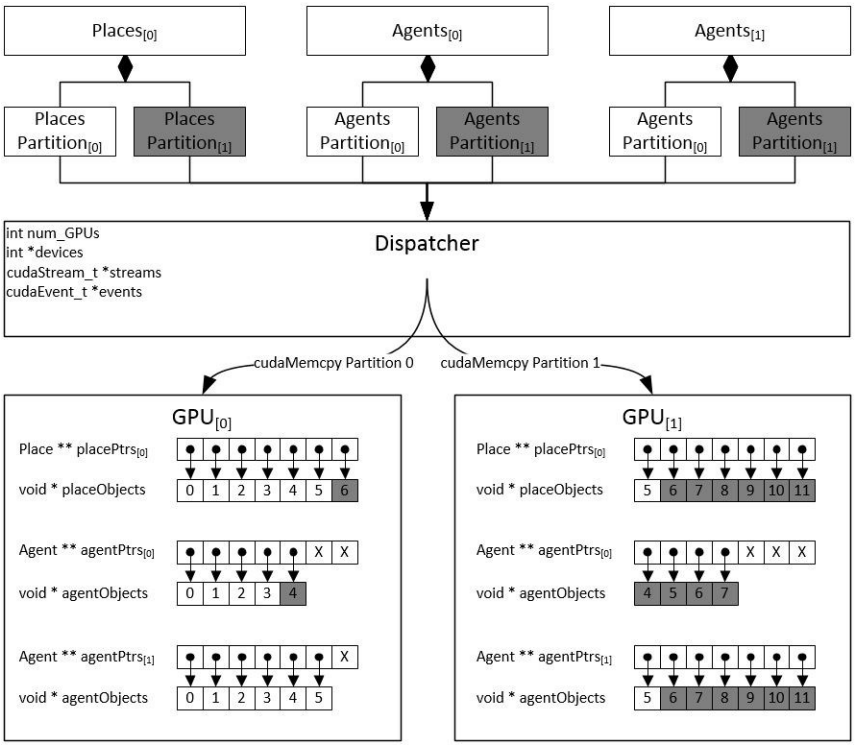

Figure 11 Dispatcher Copying Partitions to Multiple Devices

The presenter layer also allows for vast flexibility in simulation size. There are three basic situations that can occur in relation to simulation size vs. device configuration (Figure 12):

1. Simulation can fit on 1 device. Simulation executes on 1 device.

2. More than 1 device, simulation fits on all devices. Simulation executes 1 partition per device.
3. Simulation size exceeds available device space. Simulation is partitioned to device sized chunks. Partitions are paged on and off devices as they become available.

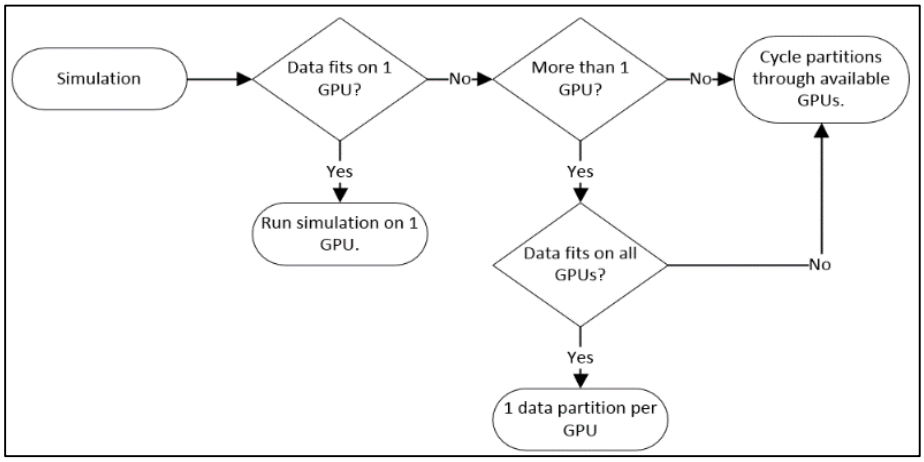

Figure 12 Simulation Size Decision Flow

4.5.1 Dispatcher

In CUDA, flow of control passes from the host to the device via special functions called "kernel functions." These functions execute within a device and work on data contained within the GPU. Launching a kernel function will generally launch one CUDA thread per element in an array, then execute the kernel function's algorithms simultaneously in all threads. The dispatcher is the only class that contains kernel functions. It implements the kernel functions that carry out the MASS Library commands like `callAll()`, `exchangeAll()`, and `manageAll()`. It also contains the AgentsModel and PlacesModel creation logic.

The dispatcher goes through three discrete phases: creation, execution, and destruction. Each phase contains distinct logic designed to maximize device utilization and model consistency.

4.5.1.1 Creation Phase

CUDA has powerful device discovery, allowing the dispatcher to search for either a specified number of devices, or to automatically discover all available devices. These devices can be queried for compute capability (the MASS Library currently requires 3.0 or 3.5) and available memory. All devices discovered that meet compatibility requirements can be tasked in the execution of user instructions, entirely without the user even knowing the host configuration.

4.5.1.2 Execution Phase

The execution phase begins with the creation of Places and Agents collections. As each is instantiated, the Dispatcher can instruct the model layer to re-partition itself if the expected simulation size alters a decision laid out in Figure 12

Once the simulation elements are created, the user application begins issuing instructions to the API. Each API instruction is replicated in the Dispatcher, where the number of model partitions is compared to the number of devices. If partitions are fewer or equal to devices, the dispatcher asynchronously executes kernel functions on each partition and coordinates boundary data exchange. Otherwise, it cycles through partitions, loading each on a device and executing a kernel function. Eventually, in order to prevent excessive partition cycling, implementing the command design pattern will allow user calls to be stored until a command that requires data exchange is issued, then executed in sequence on each partition as it is loaded.

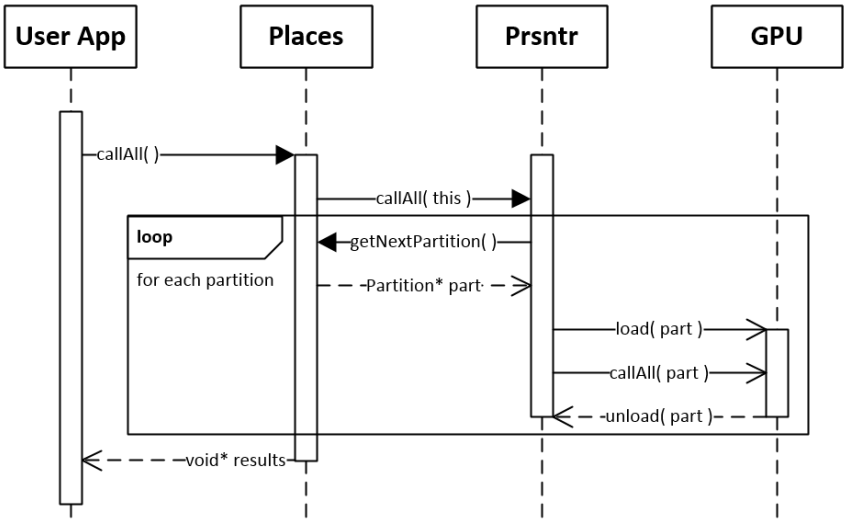

Figure 13 Command Execution Sequence

An example kernel function for callAll( ) might be:

```
1: void callAllKernel( Place **places, int nPlaces, int funcId ) {
2:       int idx = blockDim.x * blockIdx.x + threadIdx.x;
3:
4:       if ( idx < nPlaces ) {
5:           places[ idx ]->callMethod( funcId );
6:       }
7: }
```

In this kernel function, we pass in a pointer to the `placePtrs` array (see Figure 8), the number of place elements in this array, and the function ID as specified by the user. Some form of line 2 is contained within almost every kernel function, as a thread resides at some index (x, y, z) within a block of threads, and each thread block resides at some index (x, y, z) within a grid of thread blocks. Thus each thread calculates its own index through some built in variables. Sometimes, a kernel function launch must spawn more threads than there are elements. To prevent a fault, the calculated thread index must be checked for bounds before being used (line 4). If an index falls within bounds, then the thread executes its logic on the place element with the same index (line 5). This sequence executes in all threads simultaneously. Thus if there are 1,000 place elements, there will be 1000 threads executing in parallel, each with a different value of `idx`. Kernel

functions cannot have a return value, so any results must be copied back to the host with a `cudaMemcpy()` call.

4.5.1.3 Destruction Phase

Upon a call to `Mass::finish()` all device and host memory will be freed. Any further commands issued to the API will result in an exception or fault. Results of a simulation should be examined and saved prior to this call.

4.5.2 Device

The Device class is a representation of a single GPU. A host with multiple GPUs will detect these during the `Mass::init()` call and create a `Device` instance for each GPU. A Device instance is responsible for tracking the amount of memory used, the location and number of instances of all Agent and Place allocations on the GPU. It also maintains a pointer to a device-side object that contains global constants like Places dimensions, arguments, and other important simulation data. It also contains a function that can take a Partition instance and load it onto that particular device. This greatly simplifies the process of transferring data to and from any particular GPU.

4.5.3 Interaction between Devices and Data Model

The Presenter module is responsible for coordinating the interaction between the Devices and Data Model. Devices are unaware of Data Model implementation details, partitioning schemes, etc… The Data Model is unconcerned with any CUDA detail including the quantity or quality of any GPU. It simply knows how to partition the model and serve up pointers to the start of each partition.

The presenter will access the Data Model, request a partition, then determine if that partition is already loaded on a GPU. If not, it gets an available Device instance, loads the partition, and begins executing kernel functions on the active device. Because the device and data model are decoupled, it will make it very easy to expand the MASS CUDA Library to use multiple devices, perform load-balancing operations, and

account for the differences in available GPU resources. It also simplifies the data model, as it is unconcerned with CUDA implementation details.

5 PROGRAMMING ANALYSIS

5.1 HEAT 2D

In order to program a Heat 2D simulation using CUDA, a user would need to be familiar with CUDA, its API, and the implementation details. The following is the code used to run the GPU performance analysis on Heat 2D.

```
1: __global__ void euler(double *dest, double *src, int size,
2:          int t, int heat_time, double r) {
3:      int blockId = blockIdx.x + blockIdx.y * gridDim.x;
4:      int threadId = blockId * (blockDim.x * blockDim.y)
5:                  + (threadIdx.y * blockDim.x) + threadIdx.x;
6:      int idx = threadId;
7:
8:      // perform forward Euler method
9:      if (idx > size && idx < size * size - size) {
10:                 double tmp = src[idx];
11:                 dest[idx] = tmp + r * (src[idx + 1] - 2 * tmp +
12:                    src[idx - 1]) + r *
13:                    (src[idx + size] - 2 * tmp + src[idx - size]);
14:           }
15:     }
16:
17:     void Heat2d::runDeviceSim(int size, int max_time, int heat_time,
18:               int interval) {
19:     double r = a * dt / (dd * dd);
20:
21:     // create a space
22:     double *z = new double[size * size];
23:     double *dest, *src;
24:     int nBytes = sizeof(double) * size * size;
25:     CATCH(cudaMalloc((void** ) &dest, nBytes));
26:     CATCH(cudaMalloc((void** ) &src, nBytes));
27:
28:     int gridWidth = (size * size - 1) / WORK_SIZE + 1;
29:     int threadWidth = (size * size - 1) / gridWidth + 1;
30:
31:     dim3 gridDim(gridWidth);
```

```
32:     dim3 threadDim(threadWidth);
33:
34:     setCold<<<gridDim, threadDim>>>(dest, src, size);
35:     CHECK();
36:
37:     // start a timer
38:     Timer time;
39:     time.start();
40:
41:     // simulate heat diffusion
42:     int t = 0;
43:     for (; t < max_time; t++) {
44:           setEdges<<<gridDim, threadDim>>>(dest, src, size, t,
45:               heat_time, r);
46:           CHECK();
47:
48:           // display intermediate results
49:           if (interval != 0 && (t % interval == 0 || t ==
50:                     max_time - 1)) {
51:                 CATCH(cudaMemcpy(z, src, nBytes, D2H));
52:                 ostringstream ss;
53:                 ss << "time = " << t << "\n";
54:                 for (int y = 0; y < size; y++) {
55:                       for (int x = 0; x < size; x++)
56:                             ss << floor(z[(y % size) * size + x]
57:                                / 2) << " ";
58:                       ss << "\n";
59:                 }
60:                 ss << "\n";
61:                 Logger::print(ss.str());
62:           }
63:
64:           euler<<<gridDim, threadDim>>>(dest, src, size, t,
65:                heat_time, r);
66:           CHECK();
67:
68:           double *swap = dest;
69:           dest = src;
70:           src = swap;
71:     } // end of simulation
72:
73:     // finish the timer
74:     Logger::info("Elapsed time on GPU = %.2f", time.lap() /
75:           10000 / 100.0);
76:
77:     CATCH(cudaFree(dest));
78:     CATCH(cudaFree(src));
79:     delete[] z;
80: }
```

Though this is short, it requires a thorough understanding of CUDA, its threading model, how to write kernel functions, and how to manage memory. It is compact and fast, but simulations with agents running on multiple GPUs would get more complex very quickly.

This is contrasted with the main function presented in section 3.4.2 which requires the programming ability of a college sophomore. The remainder of the source code can be found in Appendix B, showing that nowhere is the user expected to know more than basic C++.

5.2 PROGRAMMABILITY SUMMARY

The programmability of the MASS CUDA Library when compared with the use of CUDA for agent based modeling shows that while the MASS implementation may, at times, be more verbose, it requires no specialized knowledge in order for it to work. A user can approach solving a problem using familiar Object Oriented Programming approaches, think about the problem in an intuitive sequential programming paradigm, and produce a functioning ABM even if programming is a secondary skill.

6 PERFORMANCE ANALYSIS

6.1 CURRENT PERFORMANCE STATISTICS

The simulation run was Heat 2D using a square matrix of values to run the calculation. The sizes provided are the length of a single side, so a simulation of size 125 actually has 15,625 elements. Each simulation was run at the various sizes for 3000 iterations with heat being applied for the first 2700 iterations. A timer was started after memory allocation was performed, and stopped immediately after the last iteration finished. Results were not displayed in order to isolate run time from other, unrelated I/O performance factors.

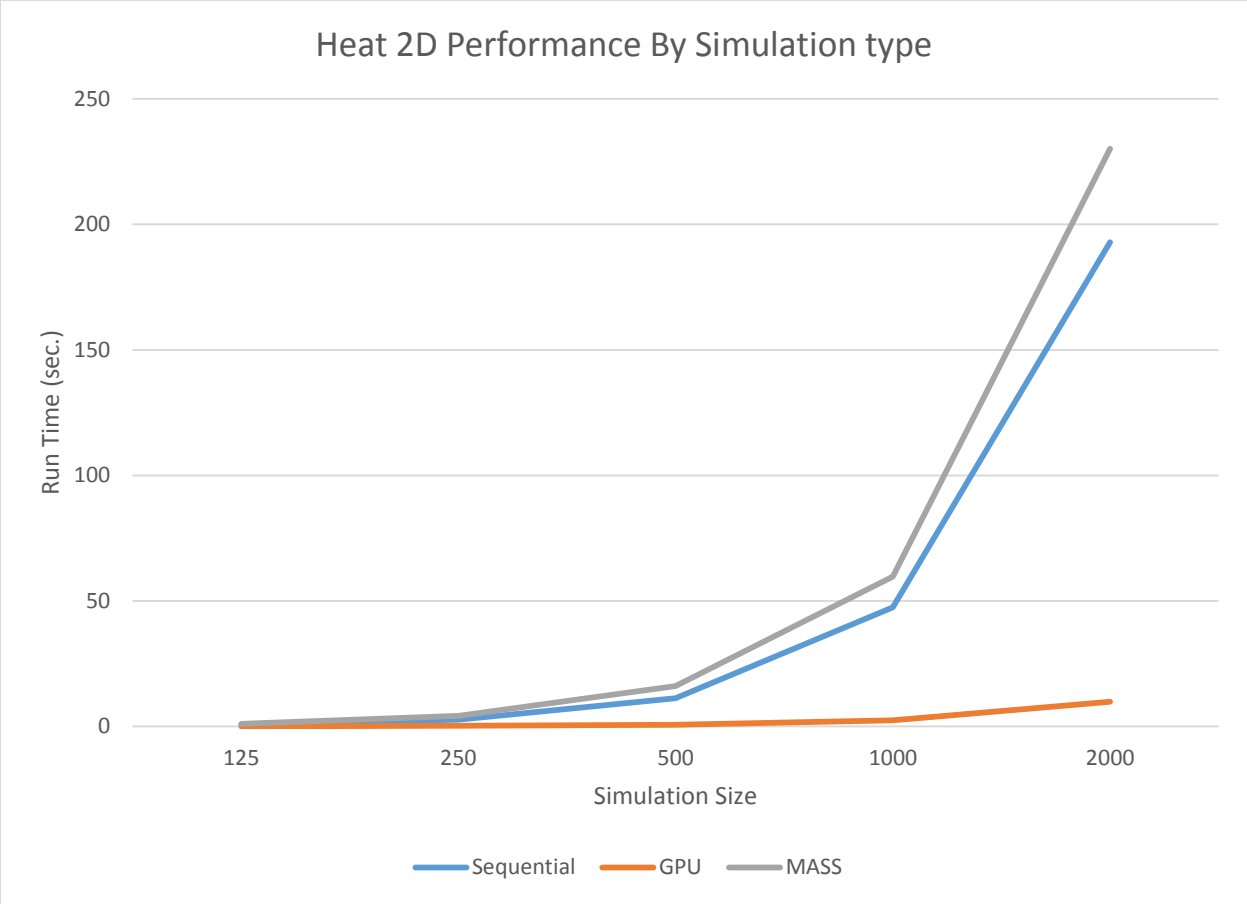

Figure 14 Performance Graph

Small run sizes shows a significant performance drop of MASS CUDA vs. sequential processing. This is expected as the overhead of parallelization can sometimes outweigh the benefits for very small problems. Simulations with larger sizes begin to converge on similar performance, being only 19% slower instead of 54% slower at its worst.

7 NEXT STEPS

7.1 POSSIBLE PERFORMANCE IMPROVEMENTS

7.1.1 Thread Coalescing and Multi-Dimensional Cuda Array Allocation

One potential cause of the slowdown is the lack of coalesced memory access. In a GPU, the global memory that stores the bulk of the simulation is approximately 100x slower than the shared memory caches, which are in turn slower than the L1 cache on each chip. When a warp of threads accesses global memory, the 32 memory accesses can be coalesced into a single memory access if all 32 fall into contiguous memory.

Currently, all arrays are a single dimension, and use index conversion to translate from the native X,Y,Z Cartesian index into a single row-major index. This means that during the Heat 2D simulation, the left and right queries to neighbors will be fast, but the top and bottom neighbors will be size elements to the left and right of the querying element. This may result in two trips out to global memory for each thread in the warp rather than the desirable single coalesced access.

CUDA supports 1D, 2D, and 3D arrays natively. Implementing multiple versions of the PlacesModel that allocate the correct native array type may help the GPU coalesce memory accesses.

7.1.2 Shared Memory

It is possible to further shorten memory accesses by additional use of shared memory. One technique to speed computation is to have each thread in a thread block copy its data into shared memory where it can be accessed by other threads very quickly. Computation is then performed using the shared memory copies, and results are written back to global memory. This goes further in preventing the many slow global memory accesses that appear to be slowing the MASS CUDA library.

7.1.3 Latency Hiding Schemes

When loading chunks of global memory into shared memory, the borders of that block may still need to query neighbors that do not fall within the thread block. As such, it may be beneficial to implement the B+2R latency hiding scheme proposed by Aaby, Perumalla and Seal [17]. This is a formal method of adding additional computation in exchange for lower communication overhead. It can be implemented at all levels of the GPU computation and involves padding the GPU or thread block with copies of neighboring elements that will be used in computation, then discarded.

7.1.4 Fully Templatized Library

Although ad hoc testing indicated that the use of virtual functions and interfaces in CUDA did not result in a significant slowdown, the negative effects of virtual functions may be worse in reality. If this is the case, it may be necessary to remove all virtual functions from the library and implement the entire library as class templates. This would allow statically typed Agent and Place implementations to be instantiated and used on the GPU. This is not an ideal solution for two reasons:

1. Templates throw confusing error messages when template requirements are not fulfilled perfectly. As these are compiled at runtime, it would be easy for a user to make small mistakes during implementation, then get lost in the error messages at runtime.
2. C++ and CUDA lack template covariance. This means that all template classes compile as unique and distinct classes that do not preserve concepts like inheritance. Unlike Java, in C++ you could not have a `Basket<Apple>` and a `Basket<Orange>` and interact with both at the interface level `Basket<Fruit>`. The fact that Apple and Orange inherit from Fruit is not preserved or accessible within a template class. You also could not have a function that takes a `Basket<Fruit>` parameter and pass in either a `Basket<Apple>` and a `Basket<Orange>`. You can use a template class at the interface level, but not the objects stored within the template. This makes it more complex to create and store multiple types of PlacesModels and AgentsModels.

7.1.5 Paradigm Shift Away from OOP

If none of the above modifications produce any significant performance improvements, it may be necessary to move away from an Object Oriented Programming paradigm for the MASS Library. One of CUDA's strengths is accelerating well known computations. It may be necessary to identify the range of functions performed within Agent Based Models and create functions that accelerate each type of problem. In Heat 2D, the problem is fundamentally modifying values within a matrix of `doubles` based on neighboring values.

It may be possible to restructure MASS in such a way that rather than loading objects containing functions and fields onto the device, we provide a library of device accelerated functions that can be used together to provide the range of functionality necessary to accelerate an ABM. The range of functions necessary for such a change is outside the scope of this thesis, and is left to future students to determine.

7.2 Clustered Computation

At some point, it is likely that this library will be used to run local computations of a simulation that is distributed over a cluster of host machines. Instead of running a user application, each host will run a cluster communication application, communicating border state with neighbors with each cycle of the simulation. In this use case, it may be that not all machines have an identical configuration of devices. In such a case, having a command & control layer that decouples the local hardware from the data model and the cluster communication layer will be a critical feature.

An implementation of this type would require the addition of some new API functions, like adding or removing agents from the simulation entirely. These changes could be added without changing the existing implementation.

The MASS CUDA library will ideally serve as a building block in a cluster of GPU accelerated hosts. If it appropriately hides the GPU details performing the processing on a host, it should be a small step to add the ability for a host to share data in a clustered environment, using the GPU to accelerate computation then sharing border data between hosts.

7.3 Automatic Heterogeneous Resource Detection and Host Configuration

Once clusters enter the picture, it will be necessary to extend the existing detection of local devices and memory. It would be very restricting to require that all nodes have the same number of GPUs or the same types of GPUs in order for a partitioning scheme to work or that all configurations be known ahead of time. The only other option would be to partition based on the weakest host in the cluster, another bad alternative.

Instead, the MASS CUDA library should detect local configurations, handle local partitioning and hide the processing details entirely from the rest of the cluster. This would dramatically increase the flexibility and robustness of the MASS CUDA library.

7.4 Load Balancing Among Heterogeneous Hosts

Once heterogeneous hosts begin working together, load balancing will be necessary. If a weak node is given equal amounts of work as a node with very powerful GPUs, the entire cluster will be stuck waiting on that node to finish before the next step of a simulation continues. Further study will eventually be required into optimizing the partition swapping algorithm, ensuring computational load balancing between clustered nodes (a host with one device should get half the simulation load of a host with two devices), and enhancing agent behavior. It will be necessary to develop an algorithm that adapts to the local performance of any single computing node, shifting data from the weak nodes to the strong nodes.

7.5 Large Simulation Partitioning

As the sizes of simulations increase, they will eventually exceed the memory available on all GPUs on a cluster of hosts. If this happens, a future improvement would be to partition the simulation into GPU-sized chunks, the number of which would exceed the number of GPUs. Then as commands are issued, these partitions are transferred on and off each host's GPUs to perform computation on the entire simulation.

This would require additional research into efficient ways to store and execute sequences of user commands to prevent unnecessary paging of partitions on and off the GPU.

8 Conclusion

The MASS CUDA Library currently shows a performance slowdown when compared to sequential computation of an identical simulation. While this is unfortunate, the data indicates that this may be a result of underlying implementation details rather than an unavoidable consequence of adapting the MASS Library to CUDA. Rather, it indicates that there is still work to be done beyond the initial effort of creating a MASS Library in CUDA that has a similar feel and programmability to the Java and C++ MASS Libraries. This also does not mean that significant progress has not been made in agent based modeling on GPUs. The MASS CUDA library allows a user to create and execute an ABM on a GPU without knowing anything about CUDA, its API, or its complexities. The programming skill required to write a simulation using MASS CUDA is equivalent to that of MASS C++, thus the primary goal of the MASS Library, encapsulating the details of parallel programming, has been accomplished.

The latest version eliminates the programmability limitations of the previous attempts at CUDA implementations and provides an excellent base for future students to build on this work. This work should be used by future students to explore the proposed performance improvements. As performance problems are solved, there are many possible future expansions to be made on the MASS CUDA Library that are supported by the existing partitioning and data model.

This thesis created an architecture that allows for GPU Agent Based Models to be created and executed using the common Object Oriented Programming principles of Encapsulation, Inheritance, and Polymorphism. No other GPU based simulation has provided a way to define and use objects at the interface level. The architecture proposed solves the problem of classes containing virtual functions breaking when copied from host to device or vice versa. It provides the foundation for research into clustered GPU computing and is designed to be adaptable to a wide variety of future applications. The novel manner in which it solves many of the limitations of CUDA, the degree to which it hides CUDA from the end user, and the opportunities for extension make this library a unique and valuable contribution to the computer science world.

BIBLIOGRAPHY

[1] NVidia, "CUDA C Programming Guide," 19 July 2013. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] R. C. Martin, "ObjectMentor," January 1996. [Online]. Available: http://www.objectmentor.com/resources/articles/ocp.pdf. [Accessed 23 Feb 2015].

[3] A. Lancaster, M. G. Daniels, G. E. P. Ropella and S. Christley, "Swarm - Summary [Savannah]," Swarm, 2015. [Online]. Available: https://savannah.nongnu.org/projects/swarm. [Accessed 20 Feb 2015].

[4] N. Minar, R. Burkhart, C. Langton and M. Askenazi, "University of Georgia Computer Science Website," 21 June 1996. [Online]. Available: http://cobweb.cs.uga.edu/~maria/pads/papers/swarm-MinarEtAl96.pdf. [Accessed 20 Feb 2015].

[5] Argonne National Laboratory, "Repast Suite," 2013. [Online]. Available: http://repast.sourceforge.net/. [Accessed 20 Feb 2015].

[6] N. Collier and M. North, "Repast Suite Documentation," 26 June 2014. [Online]. Available: http://repast.sourceforge.net/docs/RepastJavaGettingStarted.pdf. [Accessed 20 Feb 2015].

[7] Argonne National Laboratory, "Repast HPC Tutorial," 2013. [Online]. Available: http://repast.sourceforge.net/hpc_tutorial/RepastHPC_Demo_00_Step_02.html. [Accessed 20 Feb 2015].

[8] S. Luke, "MASON Multiagent Simulation Toolkit," George Mason University, August 2014. [Online]. Available: http://cs.gmu.edu/~eclab/projects/mason/manual.pdf. [Accessed 20 Feb 2015].

[9] Universita degli Studi di Salerno, "Distributed Mason by IsisLab," 2011. [Online]. Available: http://www.isislab.it/projects/dmason/. [Accessed 06 Mar 2015].

[10] Universita degli Studi di Salerno, "DSparseGrid2D," [Online]. Available: http://www.isislab.it/projects/dmason/DMason1.1_doc/index.html. [Accessed 06 Mar 2015].

[11] K. Wang and Z. Shen, "A GPU Based TrafficParallel Simulation Module of Artificial Transportation Systems," in *Service Operations and Logistics, and Informatics (SOLI)*, Suzhou, 2012.

[12] R. D'Souza, M. Lysenko and K. Rahmani, "Sugarscape on Steroids: Simulating Over a Million Agents at Interactive Rates," 2007. [Online]. Available: http://www.me.mtu.edu/~rmdsouza/Papers/2007/SugarScape_GPU.pdf.

[13] P. Richmond and D. Romano, "AgentBased GPU, a Real-time 3D Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the GPU," 2008. [Online]. Available: http://www.flame.ac.uk/pubs/pdf/10.1.1.144.734.pdf.

[14] G. Vigueras, J. Orduna and M. Lozano, "A GPU-Based Multi-Agent System for Real-Time Simulations," in *Advances in Practical Applications of Agents and Multiagent Systems*, Berlin, Springer, 2010, pp. 15-24.

[15] P. Richmond, S. Coakley and D. M. Romano, "A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA," in *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Budapest, Hungary, 2009.

[16] M. Lysenko and R. M. D'Souza, "A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units," *Journal of Artificial Societies and Social Simulation,* vol. 11, no. 4, p. 10, 2008.

[17] B. G. Aaby, K. S. Perumalla and S. K. Seal, "Efficient simulation of agent-based models on multi-GPU and multi-core clusters," in *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, Torremolinos, Malaga, Spain, 2010.

[18] J. M. Cecilia , J. M. García, A. Nisbet, M. Amos and M. Ujaldón, "Enhancing data parallelism for Ant Colony Optimization on GPUs," *Journal of Parallel and Distributed Computing,* pp. 42 - 51, 2013.

[19] G. Caggianese and U. Erra, "Exploiting GPUs for Multi-Agent Path Planning on Grid Maps," *IEEE,* pp. 482 - 488, 2012.

[20] J. Kumar, L. Singh and S. Paul, "GPU based parallel cooperative Particle Swarm Optimization using C-CUDA: A case study," in *Fuzzy Systems (FUZZ), 2013 IEEE International Conference on*, 2013.

[21] T.-Y. Liang and Y.-W. Chang, "GridCuda: A Grid-Enabled CUDA Programming Toolkit," in *Advanced Information Networking and Applications*, Biopolis, 2011.

[22] D. Strippgen and K. Nagel, "Multi-agent traffic simulation with CUDA," *High Performance Computing & Simulation,* pp. 106-114, 21 June 2009.

[23] P. Richmond, "FLAME GPU," The University of Sheffield, Department of Computer Science, 2011. [Online]. Available: http://www.flamegpu.com/documentation.php. [Accessed 21 Febbb 2015].

[24] T. Ojiru, "Implementing the Multi-agent spatial simulation (MASS) library on the Graphics Processor Unit," 2012. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/TosaOjiru_thesis.pdf.

[25] R. Jordan, "MASS: A Parallelizing Library for Multi-Agent Spatial Simulation," December 2012. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/RobJordan_au12.pdf.

[26] J. Fang, A. L. Varbanescu and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *International Conference on Parallel Processing*, 2011.

[27] W.-m. W. Hwu, "Heterogeneous Parallel Programming," 2014. [Online]. Available: https://www.coursera.org/course/hetero.

[28] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington, MA: Elsevier Inc., 2010.

[29] P. Richmond, S. Coakley and D. Romano, "Cellular Level Agent Based Modelling on the Graphics Processing Unit," in *International Workshop on High Performance Computational Systems Biology*, 2009.

[30] P. Warczak, "Coordinating Multiple GPU Devices to Run MASS Applications," 2012. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/PiotrWarczak_sp12.docx.

Appendix A - GPU MEMORY

A.1 MEMORY SPACE

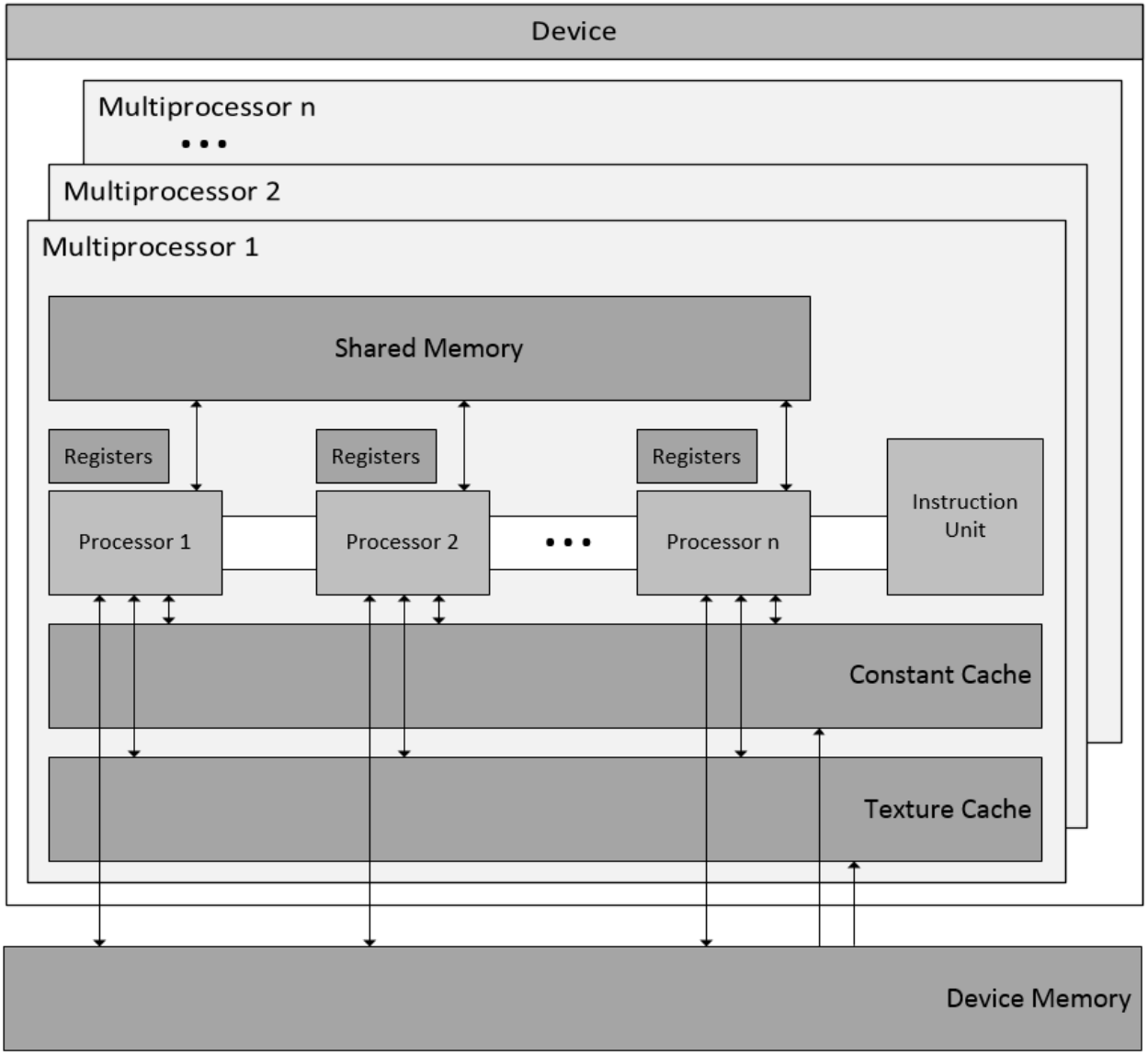

Figure A.15 GPU Memory

A GPU is composed of many multiprocessors. A multiprocessor contains, unsurprisingly, multiple processors and all receive the same instruction at the same time and execute that instruction on the data in

its own register. The data in each register may be different. A single thread at a time executes on a single processor. This is the Single-Instruction-Multiple-Data or SIMD model of parallel processing.

Each processor can access a very fast shared memory cache to share data between threads. Processors also have access to other device memory storage such as device memory. While this memory area is significantly larger (GB as opposed to KB), it is up to 100x slower to access than the shared memory.

Individual processors can share data using shared memory or global memory. Data can be shared between multiprocessors only using global memory. This is one of the reasons the B+2R [17] latency hiding scheme may provide excellent benefits. Data surrounding border regions could be loaded into shared memory to prevent the need for uncoalesced global memory accesses during execution.

Appendix B - HEAT 2D CODE

main.cu

```
1:  #include <iostream>
2:  #include <sstream>
3:  #include <stdio.h>
4:  #include <stdlib.h>
5:
6:  #include "../src/mass.h"
7:  #include "../src/Logger.h"
8:
9:  #include "Heat2d.h"
10:
11: using namespace std;
12: using namespace mass;
13:
14: int main ( ) {
15:     // test logging
16:     Logger::truncateLogfile ( "test_results.txt" );
17:     const int nTests = 6;
18:     int size[ nTests ] = { 125, 250, 500, 1000, 2000, 4000 };
19:     int max_time = 3000;
20:     int heat_time = 2700;
21:     int interval = 0;
22:
23:     Logger::info (
24:         "Running Heat 2D with params:\n\tsize =
25:         %d\n\ttime = %d\n\theat_time = %d\n\tinterval =
26:         %d", size, max_time, heat_time, interval);
27:
28:         Heat2d heat;
29:     for ( int i = 0; i < nTests; ++i ) {
30:         Logger::print ( "" );
31:         Logger::info ( "Running test for size %d", size[ i ] );
32:         heat.runMassSim ( 4000, max_time, heat_time, interval );
33:     }
34:
35:     return 0;
36: }
```

Heat2d.h

```
1:  #ifndef HEAT2D_H_
2:  #define HEAT2D_H_
```

```
3:
4:  #include "Metal.h"
5:  #include "../src/Places.h"
6:
7:  class Heat2d {
8:
9:  public:
10:     Heat2d ( );
11:     virtual ~Heat2d ( );
12:
13:     void runMassSim ( int size, int max_time, int heat_time, int
14:                       interval );
15:     void displayResults ( mass::Places *places, int time, int
16:                           *placesSize );
17:
18: };
19:
20: #endif /* HEAT2D_H_ */
```

Heat2d.cu

```
1:  #include "Heat2d.h"
2:  #include <ctime> // clock_t
3:  #include <iostream>
4:  #include <math.h>     // floor
5:  #include <sstream>    // ostringstream
6:  #include <vector>
7:
8:  #include "../src/mass.h"
9:  #include "../src/Places.h"
10: #include "../src/Logger.h"
11: #include "Metal.h"
12: #include "Timer.h"
13:
14: static const int WORK_SIZE = 32;
15: static const double a = 1.0;  // heat speed
16: static const double dt = 1.0; // time quantum
17: static const double dd = 2.0; // change in system
18:
19: using namespace std;
20: using namespace mass;
21:
22: Heat2d::Heat2d ( ) { }
23: Heat2d::~Heat2d ( ) { }
24:
25: void Heat2d::displayResults ( Places *places, int time, int
26:                               *placesSize ) {
27:     ostringstream ss;
28:
```

```
29:     ss << "time = " << time << "\n";
30:     Place ** retVals = places->getElements ( );
31:     int indices[ 2 ];
32:     for ( int row = 0; row < placesSize[ 0 ]; row++ ) {
33:         indices[ 0 ] = row;
34:         for ( int col = 0; col < placesSize[ 1 ]; col++ ) {
35:             indices[ 1 ] = col;
36:             int rmi = places->getRowMajorIdx ( indices );
37:             if ( rmi != ( row % placesSize[ 0 ] ) * placesSize[ 0 ]
38:                  + col ) {
39:                 Logger::error ( "Row Major Index is
40:                     incorrect:[ %d ][ %d ] != %d", row, col, rmi);
41:             }
42:             double temp = *( ( double* ) retVals[ rmi ]->
43:                              getMessage ( ) );
44:             ss << floor ( temp / 2 ) << " ";
45:         }
46:
47:         ss << "\n";
48:     }
49:     ss << "\n";
50:     Logger::print ( ss.str ( ) );
51: }
52:
53: void Heat2d::runMassSim ( int size, int max_time, int heat_time,
54:                           int interval ) {
55:
56:     string *arguments = NULL;
57:     int nGpu = 1; // # processes
58:     int nDims = 2;
59:     int placesSize[ ] = { size, size };
60:
61:     // start a process at each computing node
62:     mass::init ( arguments, nGpu );
63:
64:     // distribute places over computing nodes
65:     double r = a * dt / ( dd * dd );
66:     Places *places = mass::createPlaces<Metal, MetalState> ( 0,
67:                  &r, sizeof( double ), nDims, placesSize, 0 );
68:
69:     // create neighborhood
70:     vector<int*> neighbors;
71:     int north[ 2 ] = { 0, 1 };
72:     neighbors.push_back ( north );
73:     int east[ 2 ] = { 1, 0 };
74:     neighbors.push_back ( east );
75:     int south[ 2 ] = { 0, -1 };
76:     neighbors.push_back ( south );
77:     int west[ 2 ] = { -1, 0 };
78:     neighbors.push_back ( west );
```

```
79:
80:     // start a timer
81:     Timer timer;
82:     timer.start ( );
83:
84:     // simulate heat diffusion in parallel
85:     int time = 0;
86:     for ( ; time < max_time; time++ ) {
87:
88:         if ( time < heat_time ) {
89:             places->callAll ( Metal::APPLY_HEAT );
90:         }
91:
92:         // display intermediate results
93:         if ( interval != 0 && ( time % interval == 0 || time ==
94:             max_time - 1 ) ) {
95:             displayResults ( places, time, placesSize );
96:         }
97:
98:         places->exchangeAll ( Metal::EXCHANGE, &neighbors );
99:         places->callAll ( Metal::EULER_METHOD );
100:    }
101:
102:    // finish the timer
103:    Logger::info ( "Elapsed time with MASS = %.2f seconds.",
104:                   timer.lap ( ) / 10000 / 100.0 );
105:
106:    // terminate the processes
107:    mass::finish ( );
108: }
```

Metal.h

```
1:  #ifndef METAL_H_
2:  #define METAL_H_
3:
4:  #include "../src/Place.h"
5:  #include "MetalState.h"
6:
7:  class Metal : public mass::Place {
8:
9:  public:
10:
11:     const static int APPLY_HEAT = 0;
12:     const static int GET_VALS = 1;
13:     const static int EXCHANGE = 2;
14:     const static int EULER_METHOD = 3;
15:     const static int SET_BORDERS = 4;
16:     const static int NEXT_PHASE = 5;
```

```
17:
18:     MASS_FUNCTION Metal ( mass::PlaceState *state, void
19:                           *argument ); MASS_FUNCTION ~Metal ( );
20:
21:     MASS_FUNCTION virtual void *getMessage ( );
22:
23:     MASS_FUNCTION virtual int placeSize ( );
24:
25:     MASS_FUNCTION virtual void callMethod ( int functionId, void
26:                                             *arg = NULL );
27:
28:     MASS_FUNCTION void nextPhase ( );
29:
30: private:
31:
32:     MetalState* myState;
33:
34:     MASS_FUNCTION void applyHeat ( ); MASS_FUNCTION void
35:         *getVals ( );
36:     MASS_FUNCTION void *exchange ( void *arg );
37:     MASS_FUNCTION void eulerMethod ( );
38:
39:     MASS_FUNCTION void setBorders ( int phase );
40:     MASS_FUNCTION inline bool isBorderCell ( );
41: };
42:
43: #endif /* METAL_H_ */
```

Metal.cu

```
1:  #include <ctime> // clock_t,
2:  #include "Metal.h"
3:
4:  using namespace std;
5:  using namespace mass;
6:
7:  MASS_FUNCTION Metal::Metal ( PlaceState* state, void
8:              *argument ) : Place ( state, argument ) {
9:      myState = ( MetalState* ) state;
10:     myState->r = *( ( double* ) argument );
11:
12:     // start out cold
13:     myState->temp[ 0 ] = 0.0;
14:     myState->temp[ 1 ] = 0.0;
15:
16:     myState->p = 0;
17: } // end constructor
18:
19: MASS_FUNCTION Metal::~Metal ( ) {
```

```
20:     // nothing to delete
21: }
22:
23: MASS_FUNCTION void *Metal::getMessage ( ) {
24:     return exchange ( NULL );
25: }
26:
27: MASS_FUNCTION int Metal::placeSize ( ) {
28:     return sizeof( Metal );
29: }
30:
31: MASS_FUNCTION void Metal::callMethod ( int functionId, void
32:                                        *argument ) {
33:     switch ( functionId ) {
34:         case APPLY_HEAT:
35:             applyHeat ( );
36:             break;
37:         case GET_VALS:
38:             getVals ( );
39:         case EXCHANGE:
40:             exchange ( argument );
41:         case SET_BORDERS:
42:             setBorders ( ( myState->p ) ); //+ 1) % 2);
43:         case EULER_METHOD:
44:             eulerMethod ( );
45:             break;
46:         case NEXT_PHASE:
47:             nextPhase ( );
48:             break;
49:         default:
50:             break;
51:     }
52: } // end callMethod
53:
54: MASS_FUNCTION void Metal::applyHeat ( ) { // APPLY_HEAT
55:     int width = myState->size[ 0 ];
56:     int x = myState->index;
57:     // only heat first row
58:     if ( x < width ) {
59:
60:         if ( x >= ( width / 3 ) && x < ( width / 3 * 2 ) )
61:             myState->temp[ myState->p ] = 19.0;
62:     }
63: } // end applyHeat
64:
65: MASS_FUNCTION void *Metal::getVals ( ) { //GET_VALS
66:     return &( myState->temp[ myState->p ] );
67: }
68:
69: MASS_FUNCTION void *Metal::exchange ( void *arg ) { // EXCHANGE
```

```
70:     return &( myState->temp[ myState->p ] );
71: } // end exchange
72:
73: MASS_FUNCTION void Metal::eulerMethod ( ) { // EULER_METHOD
74:     int p = myState->p;
75:     if ( !isBorderCell ( ) ) {
76:         int p2 = ( p + 1 ) % 2;
77:
78:         // perform forward Euler method
79:         double north = *( ( double* ) myState->
80:                           inMessages[ 0 ] );
81:         double east = *( ( double* ) myState->
82:                          inMessages[ 1 ] );
83:         double south = *( ( double* ) myState->
84:                           inMessages[ 2 ] );
85:         double west = *( ( double* ) myState->
86:                          inMessages[ 3 ] );
87:
88:         double curTemp = myState->temp[ p ];
89:         myState->temp[ p2 ] = curTemp + myState->r * ( east
90:                             - 2 * curTemp + west ) myState->r
91:                             * ( south - 2 * curTemp + north );
92:     } else {
93:         setBorders ( p );
94:     }
95:
96:     nextPhase ( );
97: } // end eulerMethod
98:
99: MASS_FUNCTION void Metal::nextPhase ( ) {
100:     myState->p = ( myState->p + 1 ) % 2;
101: }
102:
103: MASS_FUNCTION void Metal::setBorders ( int phase ) {
104:     int x = myState->index;
105:     int size = myState->size[ 0 ];
106:     int idx = 1; // used for top and left borders
107:
108:     if ( x >= size * size - size ) {
109:         idx = 0; // use north value
110:
111:     } else if ( x % size == size - 1 ) {
112:         idx = 2; // no east, so west is 3rd element
113:     }
114:
115:     myState->temp[ phase ] = *( ( double* ) myState->
116:                                inMessages[ idx ] );
117: } // end setBorders
118:
119: MASS_FUNCTION inline bool Metal::isBorderCell ( ) {
```

```
120:     int x = myState->index;
121:     int size = myState->size[ 0 ];
122:     return ( x < size || x > size * size - size ||
123:              x % size == 0 || x % size == size - 1 );
124: }
```

MetalState.h

```
1:  #ifndef METALSTATE_H_
2:  #define METALSTATE_H_
3:
4:  #include "../src/PlaceState.h"
5:
6:  class MetalState : public MASS::PlaceState {
7:  public:
8:
9:      double temp[ 2 ];  // this place's temperature
10:     int p; // the index of current temp
11:     double r; // a coefficient used in Euler's method
12: };
13:
14: #endif /* METALSTATE_H_ */
```

MetalState.cu

This file is not necessary.