MASS CUDA: Parallel-Computing Library for Multi-Agent Spatial Simulation

Nathaniel Hart

1 Introduction

This document is written to define the existing features of the CUDA version of the MASS library, a parallel computing library for **M**ulti-**A**gent **S**patial **S**imulation. As envisioned from its name, the design is based on multi-agents, each behaving as a simulation entity on a given virtual space. The library is intended to take advantage of the massive parallel computing capabilities of GPUs to parallelize a simulation program that particularly focuses on multi-entity interaction in physical, biological, social, and strategic domains. The examples include major physics problems (including molecular dynamics, Schrödinger's wave equation, and Fourier's heat equation), neural network, artificial society, and battle games.

2 Programming Model

2.1 Components: Places and Agents

"Places" and "agents" are keys to the MASS library. "Places" is a matrix of elements that are allocated over a several GPUs on a single machine. Each element is called a place, is pointed to by a set of matrix indices, and is capable of exchanging information with any other places. On the other hand, "agents" is a set of execution instances that can reside on a place, migrate to any other places with matrix indices, (thus as duplicating themselves), and interact with other agents as well as multiple places.

An example of places and agents in a battle game could be territories and military units respectively. Some applications may need only either places or agents. For instance, Schrödinger's wave simulation needs only two-dimensional places, each diffusing its wave influence to the neighbors. Molecular dynamics needs only agents, each behaving as a particle since it must collect distance information from all the other particles for computing its next position, velocity, and acceleration.

Parallelization with the MASS CUDA library assumes one or more NVidia GPUs with compute capability 3.0 or better running on a single computer as the underlying computing architecture, and thus uses a CUDA kernel functions to load data into available GPUs and perform parallel processing. The library spawns a thread for each place and agent, and coordinates data transfer and agent migration using barrier synchronization. In order to minimize divergence of flow of control within warp execution units, Agents and Places will be executed by unique threads that are created within separate calls to function specific kernel functions.

Places are mapped to CUDA threads on available GPUs, as specified by the user. The MASS library divides places into smaller stripes in vertical or in the X-coordinate direction, each of

which is then allocated to and executed by a different GPU. Contrary to places, agents are grouped into vectors, each being allocated to a different GPU where CUDA threads will execute all active agent upon a call to Agent.callAll().

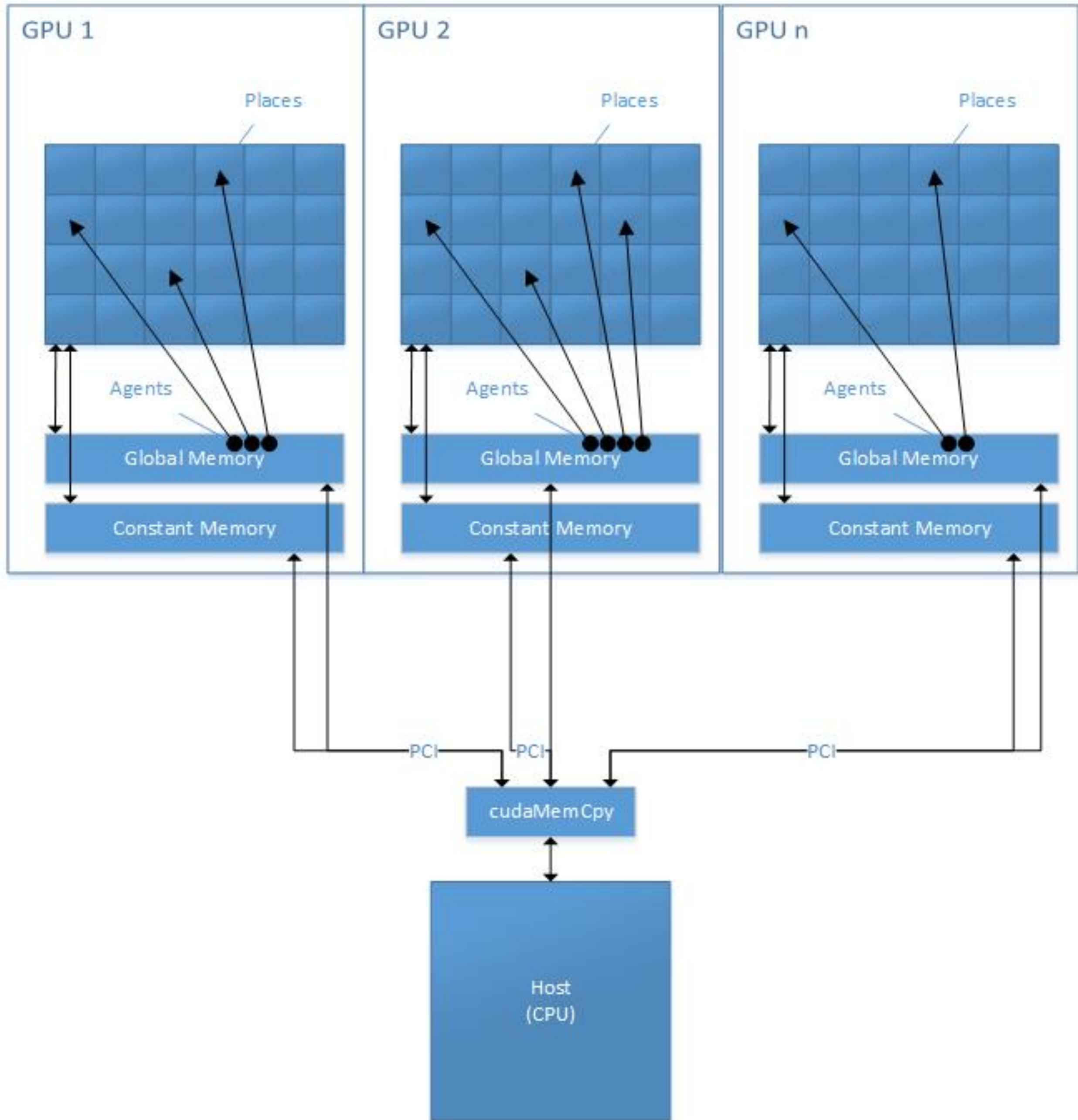

Figure 1 Place and Agent Distribution

2.2 Programming Framework

The following code shows a CUDA programming framework that uses the MASS library to simulate a multi-agent spatial simulation.

Example 1:

```
1:     #include <cstdio>       // fprintf()
2:     #include <cmath>        // ceil(), floor()
3:     #include <unistd.h>     // getopt()
4:     #include <sys/time.h>   // gettimeofday(), timeval
5:
6:     #include "cudaUtil.h"   // CATCH(), CHECK(), syncDevices(), getDevices()
7:     #include <string>       // string
8:
9:     #include "grid2d.h"     // Grid2D
10:    #include "stripe2d.h"   // Stripe2D
11:    #include "range2d.h"    // Range2D
12:
13:    include "mass.h"
14:    #include "places.h"
15:    #include "place.h"
16:
17:    int main(int argc, char *argv[]) {
18:
19:      Parameters_t parms;
20:      processParms(argc, argv, &parms);
21:      int n = parms.size;  // system size
22:      int t = parms.time;  // simulation time
23:      char *filename filename = parms.filename;  // output filename
24:
25:      Mass mass;
26:      mass.init(parms.ngpu, parms.devices);
27;
28:      // build arguments to pass to each place
29:      int *args = new int[n * n * 1];
30:      for (int i = 0; i < n * n; i++) {
31:        args[i] = 1;
32:      }
33:
34:      dim3 bounds(n, n, 1);
35:      Places *waves = new Places<DerivedPlace>( 0, size, nGpu, params.devices,
            (void *)args, sizeof(*args), params.streams, params.events);
36:
37:      // start simulation
38:      for (int tick = 0; tick < t; tick++) {
39:        int funcId = 0;
40:        waves->callAll(funcId, (void *)args, sizeof(*args));
41:        waves->updateAll();
42:
43:        elements = waves->getElements();
45:
46:        printPlaces2D(stdout, elements, n);
47:      }
48:
49:      // print results
50:      elements = waves->getElements();
```

```
51:       printf("Places after simulation:\n");
52:       printPlaces2D(stdout, elements, n);
53:
54:       if (filename != NULL) {
55:         FILE *fp = fopen(filename, "wb");
56:         if (fp == NULL) {
57:           fprintf(stderr, "error opening file.\n");
58:         } else {
59:           printPlacesCsv(fp, elements, n);
60:           fclose(fp);
61:         }
62:       }
63:
64:       mass.finalize();
65:
66:       return(EXIT_SUCCESS);
67:     }
```

The behavior of the above code is as follows: defines all available GPUs with MASS::init( ) (line 26). The code thereafter maps a matrix of n x n “Waves” places and distributes them over the available GPUs (lines 34 – 35). The host process then falls into a cyclic simulation (lines 38 – 47) where all GPU threads repeat calling the following four functions in a parallel fashion:

- callAll( ) of the “Waves” places to update each place object’s status
- updateAll( ) of the “Waves” places to exchange data among place objects

At the end, all the threads are terminated and GPU and memory resource reclaimed get (line 64).

In the following sections, we will define the specification of “MASS”, “Places”, “Place”, “Agents”, and “Agent”

3 MASS

All processes involved in the same MASS library computation must call MASS::init( ) and MASS::finish( ) at the beginning and end of their code respectively so as to get started and finished together. Upon a MASS::init( ) call, each GPU, running on the local machine, is discovered, enumerated, and communication streams to and from each are opened up to facilitate cross-GPU data transmission. Upon a MASS::finish( ) call, all devices are released, streams destroyed, and memory released.

public void	**init(int ngpu, int* devices)** Initializes the MASS environment. Must be called prior to all other MASS methods.
public static void	**finish( )** Shuts down the MASS environment, releasing all resources. Finishes computation.
public static Places*	**getPlaces(int handle)** Retrieves a “Places” object that has been created by a user-specified handle and mapped over multiple GPUs.

public static Agents*	**getAgents(int handle)** Retrieves an "Agents" object that has been created by a user-specified handle and mapped over multiple GPUs.
public static void	**createLaunchDimensions(dim3 &calcArea, dim3 &bd, dim3 &gd)** Creates CUDA launch dimensions for the given calculation area.
public static void	**printLaunchDimensions(dim3 bd, dim3 gd)** Prints out the given launch dimensions (mainly for debugging and optimization).

4 Places

"Places" is a distributed matrix whose elements are allocated across different GPUs, and to independently executing thread blocks within each GPU. Each element, (termed a "place") is addressed by a set of device independent matrix indices. Once the main method has called MASS::init( ), it can create as many places as needed, using the MASS::createPlaces( ) function. A "Places" instance (simplified as "places" in the following discussion) is partitioned into smaller stripes in terms of coordinates[0], and is mapped over a given set of GPUs and thread blocks. Places is a template class that is designed to handle the class type Place or any class that extends Place. Any other class type will result in undefined behavior.

4.1 template<typename T> public class Places

The class represents an array of Place objects distributed across available GPUs. Array elements are accessed and processed by GPUs in parallel. The type represented by 'T' must be derived from Place or this will result in undefined behavior

template< typename T> public	**Places<T>(int handle, dim3 dimensions, int nGpu, int* devices, void* args, int argSize, cudaStream_t streams, cudaEvent_t *events)** Constructs a Places object with the given identifier 'handle' and size 'dimensions' across the specified number of GPUs. nGpu must match the number of elements in devices. The remaining parameters may be NULL if there are no arguments required for place creation or use of CUDA streams and events is not necessary. Type 'T' must extend Place.
template< typename T> public T*	**getElements( )** Returns an array of the Place elements contained in this Places object. This is an expensive operation since it requires memory transfer from the GPU.
public int	**getHandle( )** Returns the handle associated with this Places object that was set at construction.

public dim3	**getDimensions( )** Returns the 3D size of this Places object.
public void	**callAll(int funcId)** Executes the given funcId on each Place element within this Places.
public void	**callAll(int funcId, void *args, int argSize)** Executes the given funcId on each Place element within this Places. Args is the arguments for this given function.
public void	**updateAll( )** Executes the update() function on each Place element within this Places.
public void	**eschangeAll(int handle, int functionId, Vector<int*> *destinations)** Calls the method specified with functionId of all destination cells, each indexed with a different Vector element. Each vector element of destination[] is an array of integers where destination[i] includes a relative index (or a distance) of the coordinate i from the current caller to the callee cell. The caller cell's outMessage is a continuous set of arguments passed to the callee's method. The caller's inMessages[] stores values returned from all callees. More specifically, inMessages[i] maintains a set of return values from the ith callee.

4.2 public class Place

"Place" is the abstract class from which a user can derive his/her application-specific matrix of places. An actual matrix instance is created and maintain within a "Places" class, so that the user can obtain parallelizing benefits from Places' callAll( ) and updateAll( ) methods that invoke a given method of each matrix element and exchange data between each element and others. All functions defined in this class are preceded with the flag "__device__" which means that this is code that can be called from within a GPU. This is the class to derive from in order to obtain a type 'T' to use in the Places template class.

__device__ public	**Place(dim3 dimensions, Point2D coordinates, Place* neighbors, int index)** This is the device constructor. It will only execute on a GPU. DO NOT MODIFY.
__device__ public void	**update( )** Called by MASS while executing Places.updateAll(). Also called when creating Places, immediately following init. When overridden, this method must use fence synchronization to prevent repeated sharing of place state. Each place must get and store their neighbors' state in local variables, and only once ALL places have done so, should each Place's internal state be modified with that data.

__device__ public void*	**getMessage( )** Returns the message that this Place is publishing. DO NOT MODIFY.
__device__ public Point2D	**getCoordinates( )** Returns this Place's global coordinates. DO NOT MODIFY.
__device__ public void	**init(void *args)** Called by MASS immediately following the constructor. This is the user's chance to perform any initialization and setup. Neighbors should not be accessed yet. ABSTRACT FUNCTION.
__device__ public void	**callMethod* int funcID, void* args)** Called by MASS while executing Places.callAll(). This is intended to be a switch statement where each user implemented function (in addition to those listed here) is mapped to a funcID, and is passed 'args' when called.
__device__ public vector<Place*>	**getAgents( )** **The agents residing locally on this place**

4.3 A Framework of Application-Specific Place-Derived Class

An application-specific "Place"-derived class, (thus whose objects are instantiated upon a Places instantiation), should have the following programming framework as shown in example 2. First of all, it must include "Place.h" and inherits the Place class (lines 5 and 7). The constructor must be defined to receive a void pointer as its argument (line 13). The place-derived class must then implement callMethod( ) that receives an int-type functionId to invoke the corresponding method and to pass a void pointer to it as its argument (lines 19 – 26). The actual functions invoked from callMethod( ) and should be implemented as private method members (lines 29 – 33).

All functions in the derived class must be preceded by the flag '__device__' in order to function properly. See sample code for guidance. Functions should be defined within the header file.

Example 2:

```
1.  #ifndef DERIVEDPLACE_H
2.  #define DERIVEDPLACE_H
3.
4.  #include <iostream>
5.  #include "Place.h"
6.
7.  class DerivedPlace : public Place {
8.  public:
9.    // 0: FUNCTION IDS
10.   static const int FUNC_NAME = 0;
11.
12. // 1: CONSTRUCTOR DESIGN
13. __device__ DerivedPlace( void *argument ) : Place( argument ) {
14.     // START OF USER IMPLEMENTATION
15.     // END OF USER IMPLEMENTATION
16. }
```

```
17.
18. // 2: CALLALL DESIGN
19. __device__ void *callmethod( int functionId, void *argument ) {
20.     switch( functionId ) {
21.         // START OF USER IMPLEMENTATION
22.         case FUNC_NAME: return func_name( argument );
23.         // END OF USER IMPLEMNTATION
24.     }
25.     return NULL;
26. };
27.
28. private:
29.   // 3: EACH FUNCTION DESIGN
30.   // START OF USER IMPLEMENTATION
31.   __device__ void *func_name( void *argument ) {
32.     return NULL;
33. }
34.
35. // END OF USER IMPLEMENTATION
36. };
37.
38. #endif
```

Example 3 shows how to instantiate a 100 by 100 objects from the above DerivedPlace class (line 8) and to call the function( ) of each object in parallel (line 9).

Example 3:

```
1.  #include "MASS.h"
2.  #include "DerivedPlace.h"
3.
4.  int main( int argc, char *argv[] ) {
5.    Parameter_t params;
6.    processParms(argc, argv, &params);
7.    MASS mass;
8.    mass.init( params.ngpu, params.devices );
9.    int *args; // initialize program specific args her
10.   dim3 size(100, 100, 1);
11. Places *places = new Places<DerivedPlace>( 0, size, nGpu, params.devices,
12.   (void *)args, sizeof(*args), params.streams, params.events);
13.
14.   places->callAll( Wave2.FUNC_NAME, "message", 7 );
15.   Mass.finalize( );
16. }
```

5 Agents

“Agents” is a set of execution instances, each capable of interacting with a place, migrating or cloning themselves to any other place(s) with matrix indices, and interacting with any other agents indirectly through a shared place interaction.

5.1 template<typename T> public class Agents

Once the main method has called MASS.init( ), it can create as many agents as needed. Unless a user supplies an explicit mapping method in his/her “Agent” definition (see 5.2 public class

Agent), "Agents" distribute instances of a given "Agent" class (simplified as agents in the following discussion) uniformly over different GPUs. The type 'T' must be a derived class of Agent (see section 5.2).

template<typename T> public	**Agents(int ngpu, int *devices, void *args, int argSize, cudaStream_t *streams, cudaEvent_t *events, int initPopulation)** Instantiates a set of agents from the "className" class, passes the "argument" object to their constructor, associates them with a given "Places" matrix, and distributes them over these places, based the map( ) method that is defined within the Agent class. If a user does not overload it by him/herself, map( ) uniformly distributes an "initPopulation" number of agents. If a user-provided map( ) method is used, it must return the number of agents spawned at each place regardless of the initPopulation parameter. Each set of agents is associated with a user-given handle that must be unique over GPUs.
public int	**getHandle( )** Returns the handle associated with this agent set.
public int	**nAgents( )** Returns the total number of active agents.
public void	**callAll(int functionId)** Calls the method specified with functionId of all agents. Done in parallel among all GPU threads.
public void	**callAll(int functionId, void *argument, int argument_size)** Calls the method specified with functionId of all agents as passing a (void *) argument to the method. Done in parallel among all GPU threads.
public *void	**getValues (void *arguments[], int argument_size, int return_size)** Receives a return value from all agents into (void *) arguments[i] whose elements' size is return_size. Done in parallel. The order of agents depends on the index of a place where they reside and starts from the place[x][y][z], and gets increased with the right-most index first and the left-most index last.
public void	**manageAll( )** Updates each agent's status, based on each of its latest migrate( ), spawn( ), and kill( ) calls. These methods are defined in the Agent base class and may be invoked from other functions through callAll and exchangeAll. Done in parallel among all GPU threads.

5.2 public class Agent

"Agent" is the abstract class from which a user can derive his/her application-specific agent that migrates to another place, spawns copies, suspends/resumes activity, and terminates itself. All Agents template classes must use a class derived from this class in order to function properly.

`__device__ public`	**`Agent( void *args )`** Is the default constructor. A contiguous space of arguments is passed to the constructor.
`protected Place*`	place Points to the current place where this agent resides.
`protected dim3`	**`index`** Is a dim3 that maintains the coordinates of where this agent resides. Intuitively, index.x index.y, and index.z correspond to coordinates of x, y, and z, or those of i, j, and k.
`protected int`	**`agentId`** Is this agent's identifier. It is calculated as: the sequence number * the size of this agent's belonging matrix + the index of the current place when all places are flattened to a single dimensional array.
`protected int`	**`parented`** Is the identifier of this agent's parent.
`protected int`	**`newChildren`** Is the number of new children created by this agent upon a next call to Agents.manageAll( ).
`protected vector<void*>`	**`arguments`** Is an array of arguments, each passed to a different new child.
`protected bool`	**`alive`** Is true while this agent is active. Once it is set false, this agent is set to inactive upon a next call to Agents.manageAll( ).
`protected int`	**`agentsHandle`** Maintains this handle of the agents class to which this agent belongs.
`protected int`	**`placeHandle`** Maintains this handle of the agents class with which this agent is associated.

__device__ public int	**map(int maxAgents, vector<int> size, vector<int> coordinates)** Returns the number of agents to initially instantiate on a place indexed with coordinates[]. The maxAgents parameter indicates the number of agents to create over the entire application. The argument size[] defines the size of the "Place" matrix to which a given "Agent" class belongs. The system-provided (thus default) map( ) method distributes agents over places uniformly as in: maxAgents / size.length The map( ) method may be overloaded by an application-specific method. A user-provided map( ) method may ignore maxAgents when creating agents.
__device__ protected bool	**migrate(vector<int> index)** Initiates an agent migration upon a next call to Agents.manageAll( ). More specifically, migrate( ) updates the calling agent's index[].
__device__ protected void	**spawn(int numAgents, vector<void*> arguments, int arg_size)** Spawns a "numAgents' of new agents, as passing arguments[i] (with arg_size) to the i-th new agent upon a next call to Agents.manageAll( ). More specifically, spawn( ) changes the calling agent's newChildren.
__device__ public void	**kill( )** Terminates the calling agent upon a next call to Agents.manageAll( ). More specifically, kill( ) sets the "alive" variable false.
__device__ public Object	**callMethod(int functionId, void *arguments)** Is called from Agents.callAll. It invokes the function specified with functionId as passing arguments to this function. A user-derived Agent class must implement this method.

5.3 A Framework of Application-Specific Agent-Derived Class

An application-specific "Agent"-derived class, (thus whose objects are instantiated upon an Agents instantiation), should have the following programming framework as shown in example 4. First of all, it must include "Agent.h" and inherits the Agent class (lines 5 and 7). The constructor must be defined to receive a void pointer as its argument (line 13). The agent-derived class must then implement callMethod( ) that receives an int-type functionId to invoke the corresponding method and to pass a void pointer to it as its argument (lines 19 – 26). The actual functions invoked from callMethod( ) and should be implemented as private method members (lines 31 – 40). They may call the "Agent" base class' migrate( ), spawn( ), and kill( ) methods to control the invoking agents (lines 34 and 38). Note that actual migration, spawning, and termination will be performed with the following Agents.manageAll( ) invocation.

Example 4:

```
1.  #ifndef DERIVEDAGENT_H
2.  #define DERIVEDAGENT_H
3.
4.  #include <iostream>
5.  #include "Agent.h"
```

```
6.
7.  class DerivedAgent : public Agent {
8.  public:
9.  // 0: FUNCTION ID
10. static const int function_ = 0;
11.
12. // 1: CONSTRUCTOR DESIGN
13. __device__ DerivedAgent( void *argument ) : Agent( argument ) {
14. // START OF USER IMPLEMENTATION
15. // END OF USER IMPLEMENTATION
16. }
17.
18. // 2: CALLALL DESIGN
19. __device__ void *callmethod( int functionId, void *argument ) {
20.     switch( functionId ) {
21.         // START OF USER IMPLEMENTATION
22.         case function_: return function( argument );
23.         // END OF USER IMPLEMNTATION
24.     }
25.     return NULL;
26. }
27.
28. private:
29. // 3: EACH FUNCTION DESIGN
30. // START OF USER IMPLEMENTATION
31. __device__ void *function( void *argument ) {
32.     vector<void*> arguments;
33.     arguments.push_back( "hello" );
34.     spawn( 1, arguments, 5 );  // spawn one child agent.
35.     vector<int*> destinations;
36.     int next[2] = { place->index[0] + 1, place->index[1] - 1 } // go NW
37.     destinations.push_back( next );
38.     migrate( );
39.     return NULL;
40. }
41. // END OF USER IMPLEMENTATION
42. };
43.
44. #endif
```

Example 5 shows how to uniformly distribute 4000 agents from the above DerivedAgent class over a Places array (line 9), to call the function( ) of each object (line 10), and to control these agents in parallel (line 11).

Example 5:

```
1.  #include "MASS.h"
2.  #include "DerivedPlace.h"
3.  #include "DerivedAgent.h"
4.  #include <vector>
5.
6.  int main( int argc, char *argv[] ) {
7.
8.    // convert arguments
9.    int nTurns = atoi(argv[0]);
10.
11.   int x = atoi(argv[1]);
12.   int y = atoi(argv[2]);
```

```
13.    int z = atoi(argv[3]);
14.    dim3 size(x, y, z);
15.
16.   // initialize MASS
17.   int nGpu = atoi(argv[4]);
18.   int* devices = (int *) calloc(nGpu, sizeof(int));
19.   MASS.init( nGpu, devices );
20.
21.   Places *places = new Places<DerivedPlace>( 1, size, nGpu, devices, null, 0,
22.       null, null);
23.
24.   Agents *agents = new Agents<DerivedAgent>( nGpu, devices, "hello", 5, null,
25.       null, 4000 );
26.
27.   // compute results
28.   for(int j = 0; j < nTurns; j++){
29.       agents->callAll( DerivedAgent.functionConstant, "message", 7 );
30.       agents->manageAll( );
31.
32.       places->callAll( DerivedPlace.functionConstant, "message2", 8 );
33.       places->updateAll( );
34.    }
35.
36.    // use elements to output or store results of computation
37.    DerivedPlace *elements = places.getElements( );
38.
39.    MASS.finish( );
40. }
```

6 Implementation Status and Plan

MASS CUDA is currently available for using Places only. Places are currently limited to 2D places only, and current plans involve building 3D spaces out of a series of 2D grids.

Agents cannot yet be instantiated. We are planning to complete the initial implementation of Agents by the end of June 2014 so that MASS CUDA can begin testing of full functionality and performance gains.