Implementing the Multi-agent spatial simulation (MASS) library on the Graphics Processor Unit

Tosa Ojiru

A thesis

Submitted in partial fulfillment of the

Requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2012

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

Implementing the Multi-agent spatial simulation (MASS) library on the Graphics Processor Unit

Tosa Ojiru

Chair of the Supervisory Committee:

Munehiro Fukuda, Ph.D.

Computing & Software Systems

The current frameworks for Agent-Based Models (ABMs) are mostly sequential, which causes a speed limitation in their execution. ABMs by nature require a large population of agents in order to show consistent patterns in the model, for example, epidemic modeling, airplane flight patterns, and crime rate analysis, all requiring millions of agents. To put this in perspective, there are 50,000 flights per day in the USA (18.25 million per year) while New York City has a population of over 18 million people as of 2011. This research effort implements a multi-agents spatial simulation library (MASS library) on the graphics processor unit (GPU) to achieve: 1) speedup by parallelization on the GPU and 2) implementing a general agent-based simulator on the GPU. Results obtained from implementing a wave simulation of 4 million array elements on the GPU showed a 15 times speedup over the corresponding sequential implementation while the CPU-only implementation (with the conventional multithreading) flattens at a 2 to 3 times speedup. This research explores other applications that use the MASS library and investigate their performance gains when compared to the same applications implemented using the CPU version of the library implemented in C++. Algorithms for efficient agent-to-agent communication and agent migration are proposed and evaluated as compared to existing algorithms.

Acknowledgement

I would to express my gratitude to my supervisor, Dr. Munehiro Fukuda who provided valuable feedback that was critical to the completion of my research work.

I would also like to thank my supervisory committee members: Dr. Michael Stiber, Dr. Kelvin Sung, and Dr. Charles Jackels for their thought provoking questions and constructively criticizing my work. I am honored to have my work criticized by the professors whom I hold to a very high regard for based on the high standard they expect from students.

I would like to thank Megan Jewell, the graduate supervisor who helped with guidance and critical deadline reminders. Josh Larios, the linux administrator was always available to help with the machines I performed my experiments on, thank you Josh.

Finally, I would like to thank my dad who provided personal advice and financial support without whom I would be deep in debt.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The Multi-Agents Spatial Simulation (MASS) library [1] is designed to facilitate Agents-Based Models (ABMs) in a way that makes it *easier* to parallelize ABM-based scientific research. The MASS library consists of Places and Agents, where Places map to the environment definition of an ABM while Agents map to the computation entities incidentally called agents in an ABM. In summary, the MASS library is an abstraction of an ABM while the actual implementation of an ABM is the concrete simulation that is executed in parallel. For example, the MASS library can be used to implement Conway's game of life [2], Sugarscape [2], and Boids [3], which are all ABMs.

1.1 Why ABM

Agent-Based Models [4] have been increasingly used in scientific communities and in practice to examine various emergent behaviors that exist when individual entities (or agents) interact with each other.

An agent here can be used to represent:

- A water particle in a wave simulation
- A human being (or a living organism) in an epidemic prediction simulation
- A supplier, a manufacturer, or a client in a supply chain simulation

In all the above cases, it can be deduced that the attributes of an agent have to be appropriately defined, in order words, when a library like MASS is used to implement an ABM, it is critical to inherit the Place and Agent computational entities and override the attributes that are specific to the domain of implementation. An important property of an ABM is that even a simple definition of an agent can still lead to useful emergent behaviors when combined with valid rules for agent-to-agent interaction, agent-to-environment interaction, and environment-to-agent interaction.

The problems that ABMs [3] help solve are:

- *Observing emergent behavior*. Agent-to-agent interaction can lead to emergent behavior that cannot be adequately modeled by mathematical equations. Conway's game of life is a variation of an ABM that shows consistent emergent behavior despite the simplicity of the rules that are defined on each agent.
- *Providing heterogeneity of agents*. In [3], Macal and North mentioned that the complexity of systems previously made it difficult to model using traditional techniques. They stated that "modeling economic markets has traditionally relied on the notions of a perfect markets, homogeneous agents, and long-run equilibrium because these assumptions made the problems analytically and computationally tractable."
- *Ability to model complex systems*. ABMs are a type of Complex Adaptive System (CAS), as defined by the interaction among agents as well as between agents and their environment. The

ability of an agent to adapt to its environment changes its state due to its environment, and affects change to other agents and its environment.

1.2 The need for more computing resources

ABMs have no restriction on the size of a population in the model. For example, it can range from a single agent model to hundreds of millions of agents in a single model. However, most ABMs require large population sizes for it to be practical. An example is an attempt to predict an epidemic outbreak for New York City that [5] has a population of over 19 million according to the 2011 Census Bureau. This would require an agent population in the tens of millions to adequately predict an epidemic dissemination, and a large amount of computational power and memory is needed to achieve faster performance so that multiple runs can be performed for allowing models to compute more accurate predictions.

Another example of a computational need is when modeling flight patterns. As of 2011, there is an average of 50,000 flights per day [6], resulting in over 1.8 million flights in a year. The dominant factor in ABMs is the size of the simulation space, in this case, number of agents. Even on an extremely optimized sequential algorithm, the runtime of running millions of agents serially would be worse than if these operations were done in parallel. For instance, running a four million agent simulation in a 4-core CPU by splitting agent execution such that one million agents would run in parallel on each CPU would reduce the runtime by approximately 75 percent.

CPUs have traditionally been used to implement parallel applications; however, the level of parallelism that can be achieved on the CPU is dwarfed by the parallelism provided by the GPU with a relatively lower cost in terms of price and scalability (i.e. adding more GPU cores). An example of a high-end CPU is the Intel Core i7-3960X with 6 total cores on the single multiprocessor for up to 12 threads with hyper threading and a price tag of about $1029. On the other hand, an above average GPU, the GeForce GTX-680 has 8 multiprocessors, 192 cores per multiprocessor totaling 1536 cores on this chip – with a cost of $440[1].

In summary, there is need for more concurrency, and from above, using a single GPU with a 1536 cores offers a more attractive option than combining multiple high-end CPUs with limits to the amount of cores that can fit on a single chip. The problem with current ABM tools is the limitation posed by the number of agents that can exist in the simulation. This is partly because popular ABM tools including NetLogo [7] and Repast [8] are implemented without parallelism, i.e. they are all sequential and do not take advantage of the resident computational power that they are running on. MASON [9] is an exception because it provides support for scheduling agents to run as parallel threads on the separate CPU cores. The MASS library however, attempts to maximize the local resources available to it by running the implemented ABM in parallel on the CPU cores, or in this case, the available GPU leading to improved performance results and faster simulation cycles.

1.3 GPU as a parallel powerhouse

According to the recent research and performance results from literature [10-22], many-core GPUs [14] have proven to outperform multicore CPUs especially in terms of the ability to run thousands of threads in parallel when the CPU is hitting a hardware limit in how many cores can fit on a chip.

[1] Prices are current on www.amazon.com as of 11/20/2012

The GPU is especially suited for applications where:

- The computational requirements are huge
- Parallelism would improve the current runtime
- The throughput is more desirable as opposed to latency. An application that puts more emphasis on the overall work performed by a group of threads rather than the latency incurred in performing individual operations.

To show how attractive the GPU is for implementing an ABM, Chen W. et al. [23] implemented a high-speed framework for blood coagulation by using NetLogo [7], Repast [8], and C. According to their experiments, the speedup of the GPU implementation of over a million agents was about 10 times faster than the C version, 100 times faster than the Repast version, and over 300 times faster than the NetLogo version. These results show that ABMs can benefit from implementation on the GPU.

1.4 The need for MASS library

The MASS library consists of Places and Agents [1]. "Places" is a grid of Place elements which form a location where an agent can reside. A place interacts with other place elements, agent elements, and act as a compute entity. An agent can migrate from one place to another, communicate with a single agent or multiple agents, and also change its state based on these interactions.

When an ABM is implemented using the MASS library, the underlying computational resources would be maximized to achieve the best performance that can be obtained on available computing cores. For example, the MASS library implements parallelism across multiple CPU cores, while utilizing available GPUs to achieve even better throughput and increase overall performance. All MASS library users have to do is:

- Inherit the MASS library Agent (and or Place) class
- Call the minimally required MASS library functions; this step requires evaluating a specific ABM design and identifies portions of the model that would benefit more from parallelization – these identified portions will be parallelized by calling MASS library functions.

The MASS library implements the basic functionality needed by an ABM so that the time to program a full ABM model is significantly reduced, and it gives users the ability to perform incremental improvements to the overall runtime of the target ABM. For example, executing a single function fooA using MASS library increased the performance by 10% when a for loop of the wave simulation was replaced by a call to the parallel CallAll. Implementing the same wave simulation using MASS (GPU version) gave a performance improvement of as much as 80% compared to the multicore version.

1.5 Goals

This research effort aims at:

- Implementing the MASS library on a single GPU by changing its library definition and designing so as to fit them to the GPU.
- Proposing and evaluating algorithms for:
    - Communication: agent-agent, agent-place, place-agent, place-place
    - Agent migration

    - Agent production (or spawning)
- Comparing the performance of MASS library implementation, using multicore CPUs versus a many-core GPU. Performance evaluation would be done by implementing the following two simulations:
    - Conway’s game of life (ABM)
    - Wave simulation (non-ABM)

2 Execution Model

The version of MASS library developed as part of this research is fully implemented on the GPU. This ensures that the available computational power of the GPU complements the CPU in running the simulation.

2.1 Processing flow

The follow section shows how data and processing goes from the CPU to GPU and back to CPU using a single threaded CPU process and a single GPU for parallel execution.

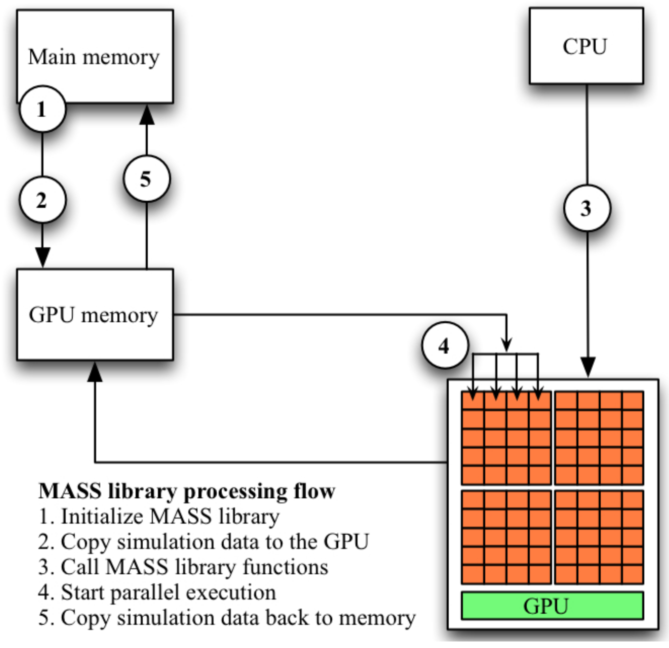

Main memory
CPU
1
2
5
3
GPU memory
4
GPU
MASS library processing flow
1. Initialize MASS library
2. Copy simulation data to the GPU
3. Call MASS library functions
4. Start parallel execution
5. Copy simulation data back to memory

Figure 2.1: MASS library processing flow

1. Initialize MASS library: The execution of a program implemented in MASS library starts out with calling the init() method. Here, the number of threads is specified, which maps to the total number of GPU threads that would be launched for each time step for parallel execution on the GPU cores. GPUs favor throughput over latency [20] so it is beneficial to get as much work done by launching as many threads as possible to hide the latency incurred by the slowest thread in a thread block.

2. Copy simulation data to the GPU: After the initialization step, an entire simulation dataset is copied to the GPU. As shown in Fig. 2.1, the GPU and CPU have separate memory, so the GPU needs to have a copy of the current simulation data in GPU memory. The dataset is copied from the GPU memory to CPU memory and vice versa across the PCIe bus, which constitutes a bus bottleneck because most PCIe bus have bandwidth [24] up to 6.4 GB/sec compared to GPU bandwidth that approach 100 GB/sec. Simulation data copied to the GPU should only be data that needs to be processed by the GPU to avoid unnecessary copy operations across the PCIe bus. After a successful copy operation from main memory to the GPU memory, the simulation data resident on the GPU is ready for processing.

3. Call MASS library functions: The third step to call MASS library functions is dependent on the application. In the case of Conway's game of life, a simulation time step might include a call to callAll(), exchangeBoundary(), and manageAll(). callAll() executes a user-specified function on all Places and Agents while exchangeBoundary() synchronizes parallel boundary data, and manageAll() updates the simulation state.

4. Start parallel execution: MASS library functions are executed using the data resident in GPU memory, and the function passed in as a parameter to the callAll() parallel function. Parallel execution is performed by assigning a group of threads (called a thread block) to execute different portions of the simulation space (in this case a 2D array). The number of threads launched at this step is proportional to the specified threads in the init() method at the initialization of the simulation.

5. Copy simulation data back to main memory: Here the state of the data is copied back to main memory to keep data updated on the CPU after each callAll() call. It is important to note that not all processing is done on the GPU so the data in main memory have to be in sync with the data in GPU memory to avoid bugs in the simulation due to data inconsistency. This emphasizes the need to only perform GPU operations on computationally intensive tasks that are parallelizable, and perform sequential operations on the CPU to avoid unnecessary trips across the PCIe bus.

2.2 Agents

Agents are execution instances with the ability to migrate, communicate with others, spawn, and terminate themselves. Agents in the MASS library are similar to agents in ABMs such that they have defined rules. These agents are separate from mobile agents, i.e., software proxies that roam around the World Wide Web to perform various user functions. In MASS library Agents are situated in an environment called Places.

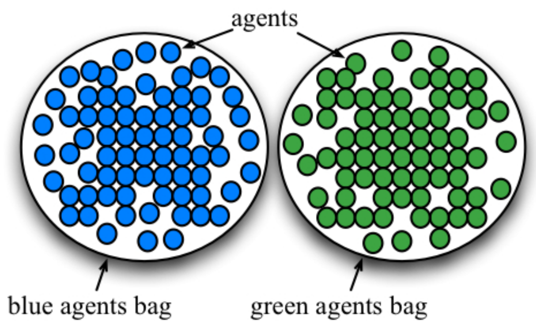


Figure 2.2: Agents in MASS library showing heterogeneity

There can be different types of agents in the same MASS-supported ABM simulation: blue agents, green agents, stock agents, and price agents, all depending on the ABM that is being modeled. In order to implement any ABM in MASS, the agent class needs to be overridden and specific agent properties would be defined which can include: behavioral rules, memory, resource attributes, etc.

2.3 Places

"Places" is the environment where agents reside. It is a distributed matrix whose elements can reside on separate GPU cores. Place elements make up a Places matrix such that a place can house a single agent or multiple agents.

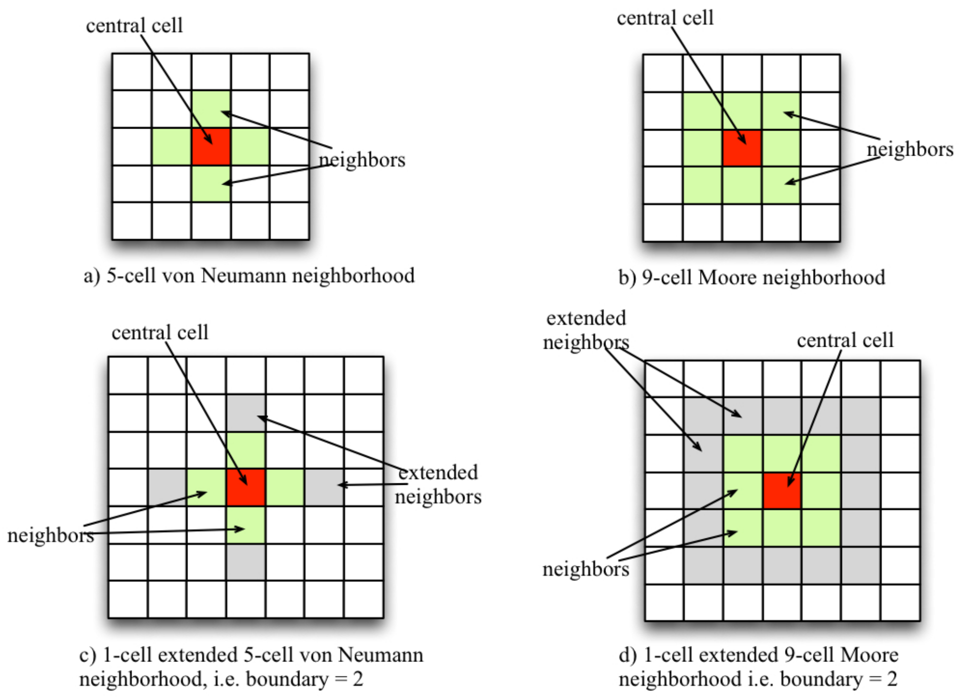
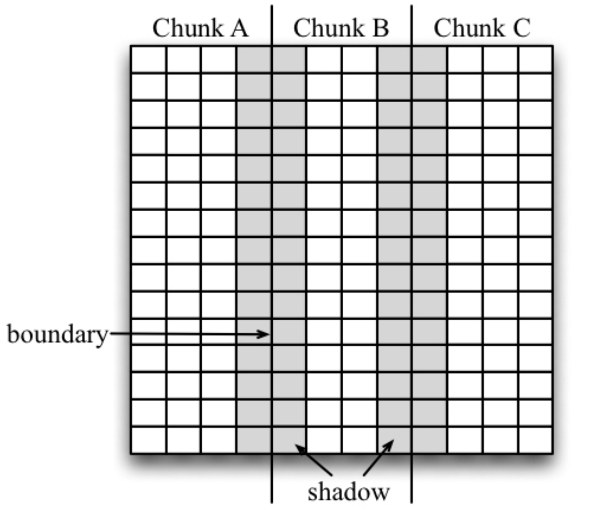

central cell
neighbors
a) 5-cell von Neumann neighborhood
central cell
neighbors
b) 9-cell Moore neighborhood
central cell
extended neighbors
neighbors
c) 1-cell extended 5-cell von Neumann neighborhood, i.e. boundary = 2
extended neighbors
central cell
neighbors
d) 1-cell extended 9-cell Moore neighborhood i.e. boundary = 2

Figure 2.3: Neighborhood configurations in MASS library

When a Places object is created in a MASS library simulation, the boundary can be specified; otherwise it defaults to a boundary of 1, which is the most efficient neighborhood configuration due to nearest neighbor communications. The higher the number of neighbors, the more complex the possible transitions from one state to another in a given neighborhood. For example, as illustrated in Fig. 2.3, a 5-cell neighborhood with two possible states, on and off, has 2^5 possible states that the neighborhood can be ranging from all on, all off, to the other 30 different combinations of states in between. A 9-cell neighborhood, however, would have 2^9 possible states that would have to be evaluated to decide the next state of the central cell.

The neighborhoods used in the MASS library are inherited from common neighborhoods [25] used in Cellular Automata (CA) which is an ABM [3]. These neighborhoods consist of the 9-cell Moore neighborhood and the 5-cell von Neumann neighborhood, while extended neighborhoods can be specified by specifying boundary > 1, and overriding the offset calculation from any given central cell.



A Places object divided into 3 chunks, A, B, C
where boundary = 1, size[0] = 12, size[1] = 15

Figure 2.4: Places in MASS library divided into parallel chunks

The shadow is calculated according to the boundary specified. From Fig. 2.4, Chunk A's shadow is on the right size of the boundary line where the cells are shaded grey. This shadow is shared with Chunk B, while Chunk B's left shadow is shared with Chunk A, Chunk B's right shadow is shared with Chunk C, and Chunk C's left shadow is shared with Chunk B. The concept of a shadow makes it possible for the right neighbor of Chunk A's boundary cells to be available to Chunk A's thread block during parallel execution.

2.4 Agents in their Places environment

Agents reside in Places. Upon an initialization, each agent needs to be mapped to their environment by specifying what Place element an agent maps to. An agent's place decides its environment, neighborhood, and what computing core the current agent resides in. MASS library provides a map() function which performs mapping by uniformly distributing the agents across the Places environment.

$$density = \frac{n_a}{n_p} \qquad 2.1$$

Where density is the number of agents per unit area of the Places array; n_a is the total number of agents assigned to the Places object; and n_p is the size of the Places matrix. The higher the density of agents present in a Places object, the more populated each place element becomes. A place element is processed by a single parallel thread in a thread block, which is the smallest unit of parallelism achievable by the MASS library, so when the number of agents present in a place is greater than one, they will be processed sequentially by the thread assigned to the current place.

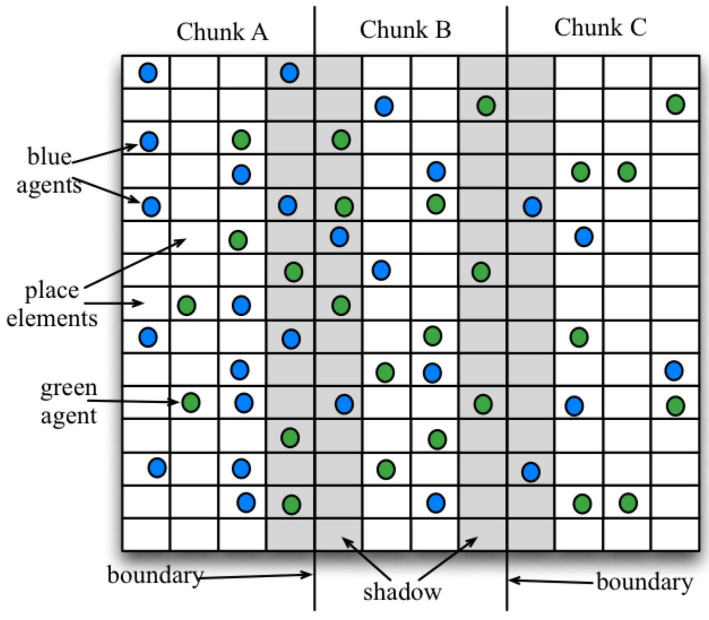


Figure 2.5: Agents distributed in a Places object in a MASS library simulation after performing a map of two agents' bag – a blue agents bag, and a green agents bag

When an agent is defined by a specific implementation, the MASS library map() function can be overridden to specify a different mapping desirable by the current application. The map() function specifies how each agent is distributed in the Places matrix. For instance, in an ABM simulation that requires at most one agent per Place, the map method would be invalid if it maps more than one agent per place. The default map behavior in this implementation of MASS library is to check all neighbors for an empty Place, and if none is found, then the whole Places matrix is walked though for a free space. If no free space is found, then the agent allocation fails on the agent.

Listing 2.1: MASS library implementation of the map function that executes on the CPU

```
Algorithm map( P, A, np, na )

Input:  P (a Places object of size np)

        A (an Agents object of size na)

Output: P' (a modified Places object containing Agents mapped to its Place elements)

        A' (a modified Agents object containing Place mapping for each mapped agent)

begin

        density := na/np          // determine the number of agent per place

        offset := 0

        // start distribution of each agent in A to the available place elements in P
        for ia := 0 to na do
            if A[ia].status = alive AND A[ia].index != EMPTY then
                if offset < np then
                    P[offset].agent := A[ia]
                    A[ia].index := P[offset].index
            // i.e. na = 16, np = 8, gives max of 2 agents per place
            // i.e. na = 8, np = 16, gives max of 1 agent for every 2 place element
            if density > 1 then
                offset := offset + density – 1      // na > np
            else if density == 1 then
                offset := offset + density          // na == np
            else if density < 1 then
                offset := offset + np/na – 1        // na < np
    end
```

2.5 CallAll

This is a parallel function that executes on the GPU. CallAll() executes a user-defined function on every Agent and Place elements in a Places matrix. The function name and arguments are passed in as a parameter thereby calling the function on every element in the simulation space. Prior to this function being executed, data should have already being allocated and copied to the GPU memory, and after the function exits, all data is copied back to main memory

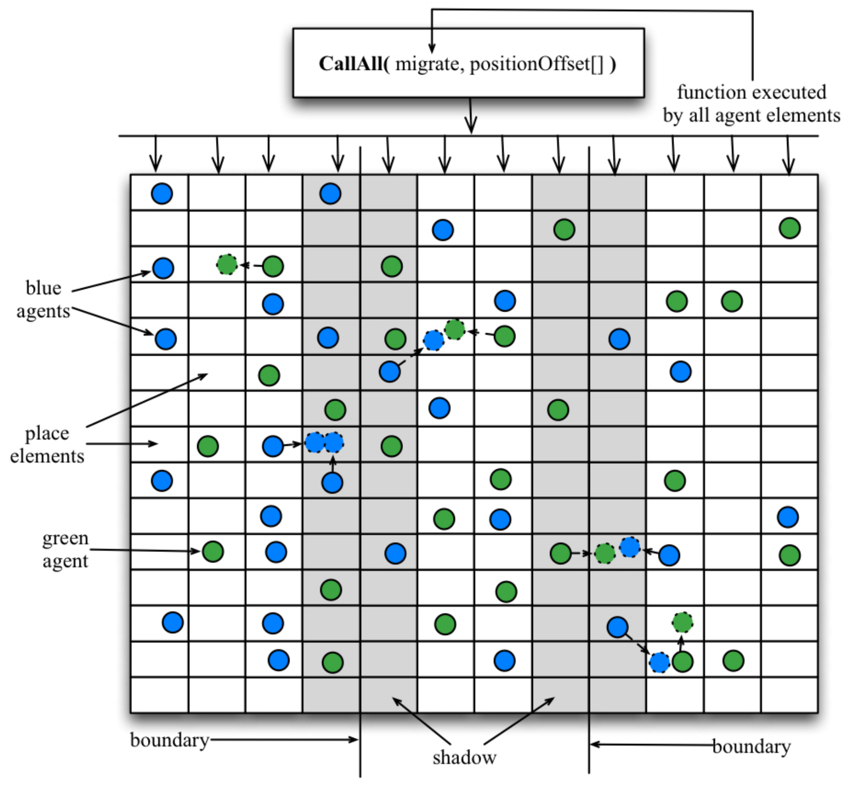

CallAll(migrate, positionOffset[])
function executed by all agent elements
blue agents
place elements
green agent
boundary
shadow
boundary

Figure 2.6: CallAll() executed on Places matrix with a rule that only a single agent can occupy a place

In some ABMs, at most one agent can reside at a given position, i.e., at most one agent per place. Figure 2.6 shows the parallel CallAll function is executed on the Places object, passing "migrate" as a pointer to the function to be executed on every agent resident in a place. Place elements do not have migrate() defined on them so this function applies only to the agents that reside in the environment. All agents in the specified Places object execute their migrate() function in parallel and decide their new location based on the positionOffset[] array passed in corresponding to size of the agents that are alive in the Places object.

After a CallAll() function has completed, the Places object (including resident agents), are in a dirty state because the shadow column defined is inconsistent across boundaries. Figure 2.7 shows the logical state of Places after a CallAll has been executed.

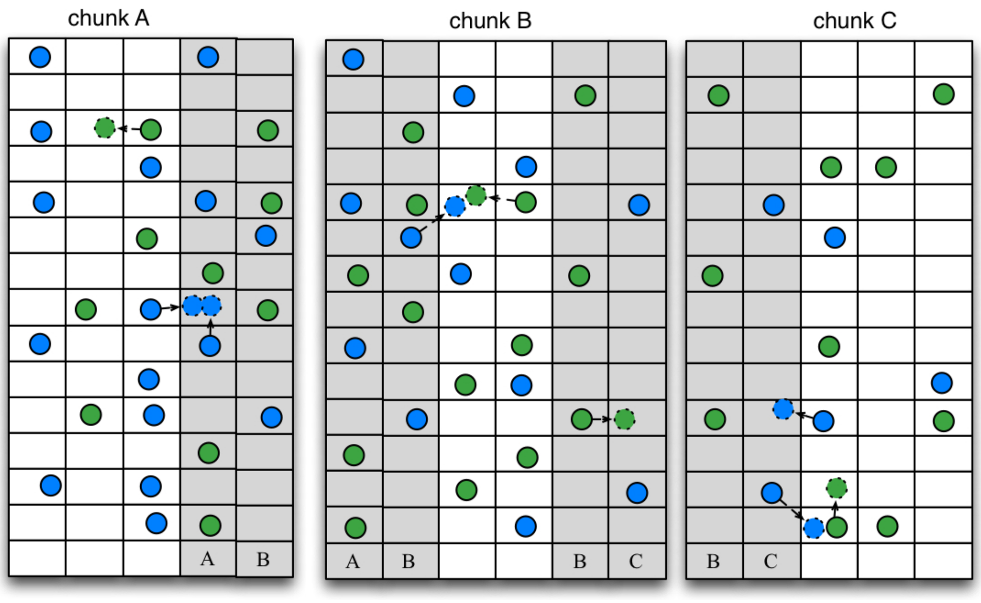


Figure 2.7: State of Places and Agents after CallAll(migrate, positionOffset[]) is executed given three parallel chunks: A, B, C

Each chunk contains a shadow copy of the adjacent chunk's boundary cells. For example, Figure 2.7 shows an extra column matrix appended to chunk A's parallel region, giving access to the last state of the neighbors on the boundary to allow for parallel execution. Chunk B shows a green agent attempting to migrate to its right neighbor (C's shadow copy); while chunk C has no knowledge of this migration attempt until boundary data has been synchronized. Also, chunk C has a blue cell migrating to the bottom left neighbor but this would not be known to adjacent chunks until the call to ExchangeBoundary().

An agent migration on a shadow cell does not complete until the sequence of ExchangeBoundary(), and ManageAll() functions have been executed, which is described in the following two subsections.

2.6 ExchangeBoundary

This function is used to synchronize the boundary elements of each chunk in the simulation to keep the shadow copies synchronized across the parallel chunks. This synchronization is required because after the call to CallAll has completed, the shadow copies of each chunk resident in adjacent chunks would be in an inconsistent state as explained in the previous section.

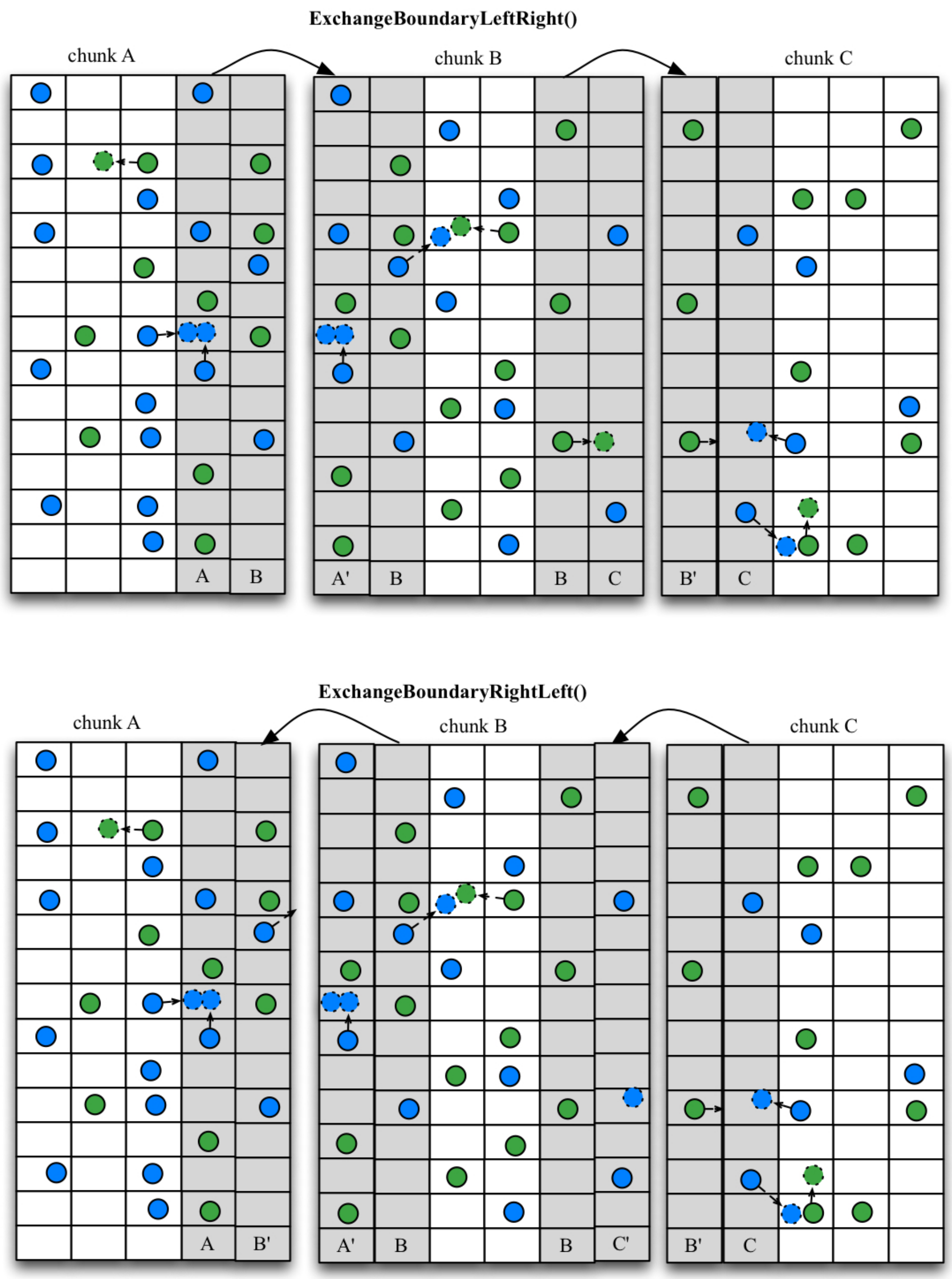


Figure 2.8: ExchangeBoundary() illustrating the effect of calls to ExchangeBoundaryLeftRight() and ExchangeBoundaryRightLeft()

The ExchangeBoundary() function call is divided into two steps: 1) a call to the ExchangeBoundaryLeftRight() function, 2) a call to the ExchangeBoundaryRightLeft() function. The order of these functions are arbitrary, however, these calls execute one after the order. ExchangeBoundaryLeftRight updates the left hand side of each parallel chunk by merging the states of the boundary cells on the left with the shadow copy present in an adjacent chunk. Figure 2.8 shows chunk A's shadow present in chunk B updated to from A to A' and B is updated to B' in chunk C.

ExchangeBoundaryRightLeft is similar to ExchangeBoundaryLeftRight except that it occurs in the opposite direction where updated shadows include B' in A's chunk, and C' in B's chunk.

2.7 ManageAll

This function updates the state of the Places environment and resolves conflicts where applicable based on the rules defined by the simulation.

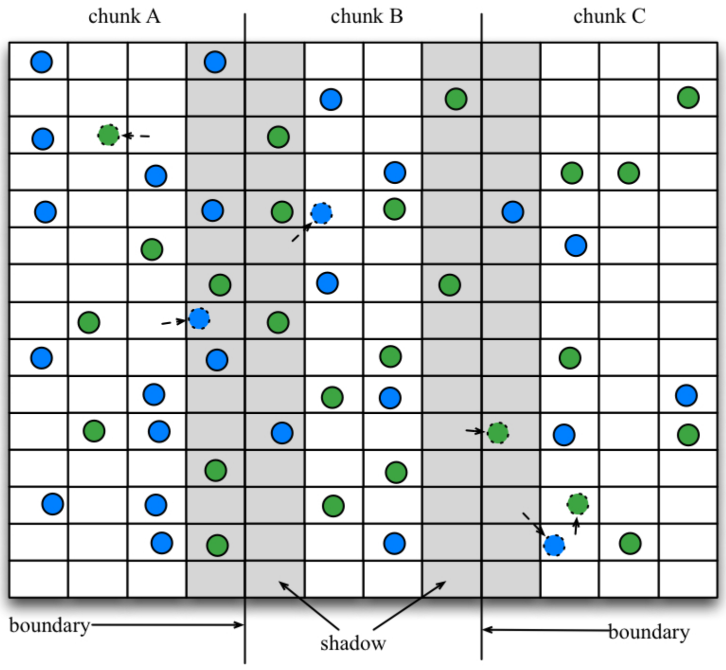


Figure 2.9: Places and Agents are updated with their current states after a call to ManageAll()

The state of the simulation space is updated by a call to ManageAll(). This function performs tasks that include: conflict resolution, removal of dead agents from simulation, and creation of new agents in new place location after spawn. An instance of a conflict in MASS library result when the agent density is greater than the current resident agent in any place P[i]. For example, when two agents A[0] and A[1] migrate to place P[2], given a density of 0, implying less than 1 agent per place. Here one of the agents would be chosen to rollback its migration by random selection.

3 Implementation

3.1 Compute Unified Device Architecture (CUDA)

CUDA is NVIDIA's proprietary language developed as extensions to the C programming language. These extensions tell the nvcc compiler (NVIDIA's CUDA compiler) when to execute a function on the CPU or when to execute a function on the GPU. For example, the keyword __global__ prepended to a function definition specifies that this function is to be executed on the GPU, while keyword __host__ prepended to a function definition specifies execution on the CPU (i.e. no data copy to GPU memory required). The popularity of CUDA combined with its success and availability of its hardware played a role in the decision to choose CUDA over other parallel computing APIs [14] like OpenCL, and DirectCompute with similar functionality.

3.2 MASS-C++

MASS-C++ is an implementation of the MASS library in the C++ programming language using object oriented programming concepts. A version of MASS-C++ was developed in this research effort to validate the functionality of MASS-CUDA by establishing a C++-ported version of the MASS library that can be easily ported and refactored into MASS-CUDA. Other versions of MASS-C++ exist in the distributed systems laboratory but in order to remove dependency on parallel work-in-progress versions of MASS-C++, the decision was made to rewrite a MASS-C++ version that can be easily extended to MASS-CUDA.

This version of MASS-C++ uses the same specification [26] developed by Professor M. Fukuda of the University of Washington, Bothell. This specification has been fully implemented as a multi-process, multi-threaded version in the Java programming language, (which this paper labels MASS-Java in the following discussions). The performance limitations of MASS-Java prompted this research effort into redesigning some MASS library functions (e.g. replacing ExchangeAll with ExchangeBoundary).

3.3 MASS-CUDA

MASS-CUDA is similar to MASS-C++ except that it has the ability to execute its parallel functions on the GPU. To avoid duplication of functionality and code, only functions that need to be run on the GPU are implemented, while maintaining the MASS-C++ codebase defined in section 3.2.

3.3.1 Data structure

The Places object is represented as a flattened one-dimensional array according to the row-major [13] addressing similarly implemented in the C language.

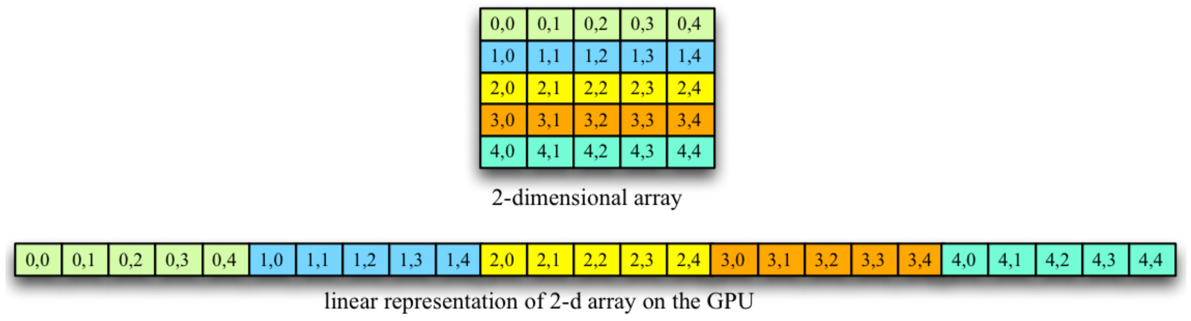


Figure 3.1: Linear representation of Places in MASS library

Each index is calculated using a row, and column index, for example, index at i, j represents the ith row, and jth colomn and is calculated with the formula below.

$$index = j + i \times size \qquad 3.1$$

Where j is the column offset; i is row offset into the Places object; and size defines the total number of Place elements in the Places object.

The data structure used to store agents is a vector implementation of a bag. Agents in a bag have no predefined order and can be added or removed using their unique agent identifier.

3.3.2 Shared memory

Shared memory gives a faster alternative to global memory, which resides off-chip [10]. The parallel functions defined in the MASS library have access to this memory in order to facilitate communication across parallel boundaries when data outside a boundary needs to be read or written to. During an ExchangeBoundary() call, data is made available for read/write by using the CUDA GPU shared memory. This is defined by allocating shared memory on the GPU device by using the CUDA-specific __shared__ keyword prepended to an array definition.

```
__shared__ Place boundaryData[sharedSize]
```

$$sharedSize = 2 \times nChunks \times hChunk \qquad 3.2$$

Where sharedSize is the size of the shared memory allocated; nChunks is the number of chunks in which the Places object has been divided based on the boundary defined on the Places (i.e. a default boundary size of 1); and hChunk is the vertical height of the defined chunk that is usually the height of the Places object which is the same as to the number of rows in a single Places column if Places where visualized as a matrix.

Shared memory is also used to store data for the current parallel chunk to facilitate inter-thread communication in cases where one thread needs access to data that is in another thread’s private memory. The statement below shows the definition of a shared chunk in MASS-CUDA.

```
__shared__ Place sharedChunk[sharedChunkSize]
```

$$sharedChunkSize = wChunk \times hChunk + (2 \times boundary \times hChunk) \qquad 3.3$$

Where wChunk is the size chunk width; hChunk is the chunk's height; and the boundary chunk is multiplied by 2 to account for most chunks being in the middle of the Places object having a left and right shadow as described in Chapter 2 with chunk B having chunk A and chunk C's shadows respectively.

3.4 Algorithms developed

The algorithms which this section defines make use of the CallAll(), ExchangeBoundary(), and ManageAll() parallel functions described in Chapter 2. Communication, migration, and reproduction are all accomplished by giving every agent and place an equal opportunity to perform its task using the CallAll() method, and conflicts are resolved by only allowing a single write operation to be performed based on who currently has the lock. In this case a write operation is performed by acquiring a lock, writing data, and then releasing the lock. Any agent or place element that attempts to write to a locked location, aborts its write operation.

3.4.1 Communication

A weak consistency model [27, 28] is used to guarantee mutual exclusion during parallel write operations. In order to avoid the need for busy waits, agent or place elements attempting to write to a locked location would abort their write operation. Read operations however, would succeed whether the location is locked or not.

3.4.2 Agent migration

Migration of an agent from one place element to another is performed after a parallel CallAll passes this migrate function as a parameter, and also specifying the offset location where the agent is migrating to.

Listing 3.1: Algorithm that describes agent migration in MASS-CUDA

```
Algorithm migrate( P, A, np, na, Offset )

Input:  P (a Places object of size np)
        A (an Agents object of size na)
        Offset (array of relative positions corresponding to the number of agents na)
Output: P' (a modified Places object with the new location of all migrated agents)
        A' (a modified Agents object with the positions of their target location updated)
Variables:   ip (index of the current place element)
             threadnum (GPU thread number)
             chunknum (parallel chunk number given sequential numbering of chunks from 0 to n)
             target (new position of agent after migration)
begin
```

```
    ip := threadnum × chunknum     // unique index into each thread block
    // migrate if there is room i.e. numAgents in current place is less than maximum acceptable
    if ip < np AND (P[ip].numAgents < density OR P[ip].numAgents = 0) then
        target := ip – Offset[ip]          // destination index
        P[ip].agent.newIndex := ip         // agent.newIndex will be assigned to agent.index
                                           // by manageAll after synchronization across boundaries
        P[target].addAgentToBag( P[ip].agent ) // add agent to destination Place
end
```

After the call to migrate completes, ExchangeBoundary() and ManageAll() manages the final position of each agent depending on the rules of the simulation i.e. if more than one agent may reside in a given place element (which depends on an ABM application), then all migrate operations would succeed. However, if at most only a single agent can reside in a place, then ManageAll() would remove all agents who failed their migration by performing a migrate operation from A[i].newIndex to A[i].index and deleting the agent from the current place's bag.

3.4.3 Agent reproduction

Agent reproduction is handled in the same way as agent migration such that a child agent gets a new position when it is created and this position is defined by randomly selecting a neighboring cell where the location of the parent agent is the central cell. Using the 9-cell Moore neighborhood, any one of the neighboring cells can be chosen at random.

```
i = random() % numNeighbors

childIndex = neighbors[i]
```

A child agent's status does not change to alive if its migration to its new position fails i.e. ManageAll does not persist this child's migration to its childIndex. Flipping the status of all child agents in a generation is performed by the ManageAll parallel function. An application can override the specific reproduction behavior by providing a destination for the spawned child agent to reside.

3.5 Experimental design

All experiments were conducted the UWB distributed systems laboratory. This environment simulated a practical scenario where the system being used is also servicing other requests and not dedicated to my experiments. My independent variables in this experiment are the number of threads used (GPU or CPU threads) and number of iterations per run. The dependent variable is the total runtime.

Table 3.1: Experimental setup experiments

	Conway's Game of Life	**Wave simulation**
Operating system	Red Hat Enterprise Client Linux release 5.8 (Tikanga)	Same
CPU	Intel® Xeon® CPU E5520 @ 2.27 GHz	Same

GPU	NVIDIA GeForce GTX-680	Same
Number of GPU cores	1536	Same
Number of CPU cores	16	Same
Application type	ABM	Scientific application

3.5.1 Conway's Game of Life

A summary of configurations used to run the experiment is:

Table 3.2: Experimental setup of Conway's game of life

C++, MASS-C++	**CUDA, MASS-CUDA**
Up to 16 parallel threads	Up to 1024 parallel threads
100 to 1000 generations with 100 generation increments	Same
100^2 up to 4000^2 agents size with increments of 100^2	Same

3.5.2 Wave simulation

A summary of the configuration used to perform the wave simulation experiment is given below:

Table 3.3: Experimental setup of Wave simulation

C++, MASS-C++	**CUDA, MASS-CUDA**
Up to 16 parallel threads	Up to 1024 parallel threads
1000 iterations	Same
100^2 up to 4000^2 places size with increments of 100^2	Same

4 Performance Evaluation

4.1 Metrics

The metrics used in measuring performance improvement is the runtime trend with increasing values of:

- Iterations (or generations for ABMs)
- Simulation size (size of Places object)

The baseline performance configuration was obtained in terms of the number of threads that gave the best performance in the C++ or CUDA version of the application.

4.2 Results

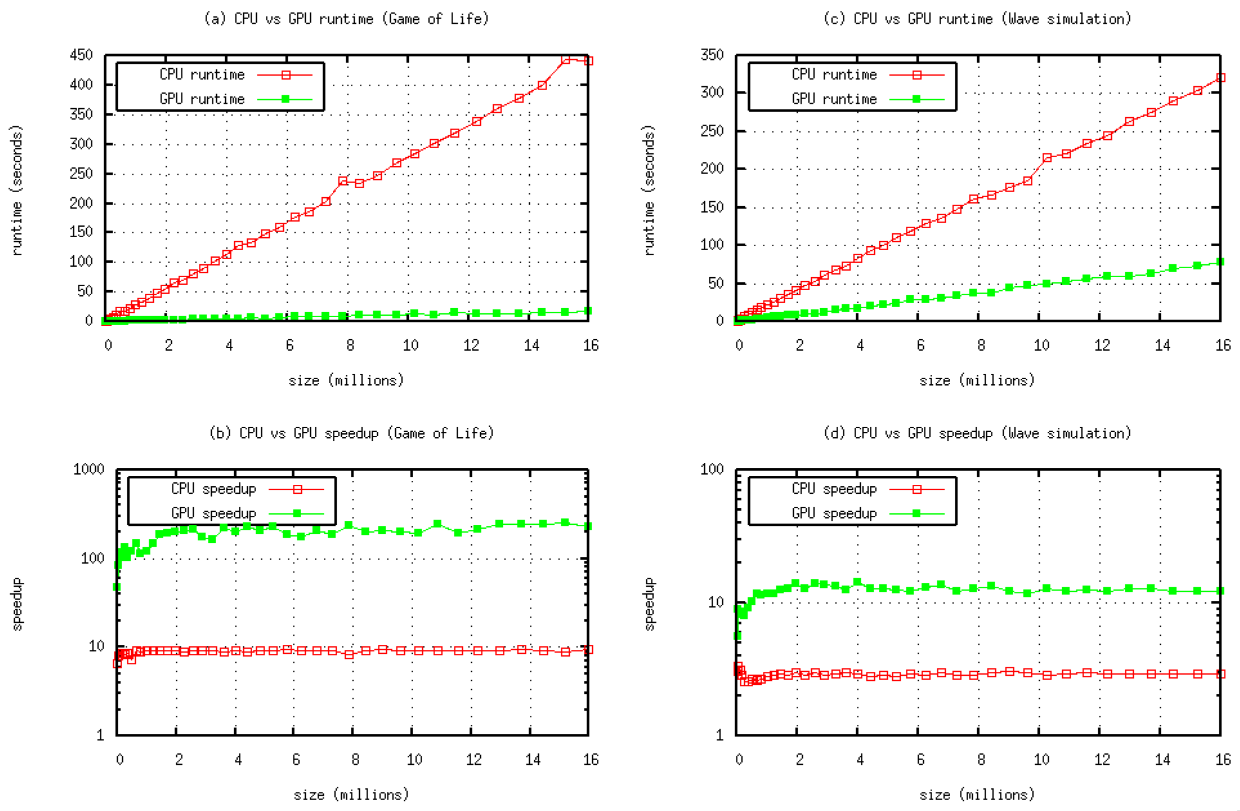


Figure 4.1: GPU versus CPU performance results

Figure 4.1-(a) compares CPU multithreading and GPU computing in the execution time of Conway's Game of Life. While the CPU version increases its execution time in proportion to the number of places, the GPU version performs fast regardless of the place size.

Figure 4.1-(b) compares CPU multithreading and GPU computing in speedup of Conway's Game of Life. The speedup result for the CPU version flattens at 10 times while the GPU version hovers above 100x even for larger place sizes.

Figure 4.1-(c) compares CPU multithreading and GPU computing in the execution time of Wave simulation. The CPU version increases proportional to the number of places showing a higher impact of the size on its execution time as opposed to the GPU version which shows less impact in execution time with increasing number of place elements.

Figure 4.1-(d) compares CPU multithreading and GPU computing in speedup of Wave simulation. The GPU version shows a higher speedup of over 10 times while the CPU version is less than half of that.

4.3 Discussion & limitations

The results in the previous section show that Conway's game of life shows close to 250x speedup while running on the GPU versus about 10x speedup recorded by a multithreaded CPU run. Also, for the CPU-only execution, the runtime increased linearly with increasing size of agents tapering at 450 seconds (or 7.5 minutes). The runtime of life on the GPU, however, stayed below 25 seconds for up to 16 million agents. In the wave simulation application there is a little over 14x speedup when running on the GPU versus CPU execution.

One reason for the difference in speedup between Conway's game of life and the wave simulation is that life is an ABM defined with three simple rules that determine the state of an agent. In the wave simulation however, the current wave height is calculated by the knowledge of the last two wave heights. The wave simulation also has three copies of the simulation space at any given time: current, previous, and height that precedes the previous height. This means that the GPU has to store three times more data for wave simulation as opposed to life.

The average memory size on the current GPU is 2 Gigabytes while the CPU used contain 16 Gigabytes of RAM. This memory gap becomes a problem when the size of the simulation being loaded onto the GPU exceeds 2 billion, in which case a higher memory GPU would be required, or multiple GPUs can be used to manage the simulation data. A multi-GPU solution to the memory problem would use the same method used split data across boundary by extending to ExchangeBoundary() function to manage data contained across two GPU boundaries as well as across thread-chunk boundaries.

5 Related Work

In this research, other ABM frameworks were surveyed which provide a range of ways to successfully implement an ABM. These frameworks are described below. The level of parallelism that can be achieved when implementing an ABM in these frameworks sets MASS-CUDA apart.

5.1 Swarm & Repast

This is one of the earliest ABM simulation frameworks and it was initially written in Objective-C. Swarm [29] is an ABM based on nested agent structures where agents can contain swarms of other agents. One difference between Swarm and the MASS library is the absence of an environment; Swarm only has the concepts of agents. Swarm is open source, and has analysis tools when running ABMs to keep track of how many agents are alive in the simulation at a certain generation for example. Another difference and limitation with Swarm is the sequential nature of underlying libraries. This sequential nature makes it painful to run practical ABMs that require large agent sizes in the range of the millions to the hundreds of millions.

Repast (recursive porous agent simulation toolkit) started out as a Java implementation of Swarm. It is also free and open source, and is currently being developed by Argonne National Library [8]. Repast focuses mostly on social simulation with C#.Net and Java implementations. Repast is different from MASS library in its sequential nature, thereby inherently limited in agent size and modeling practical systems that requires a large agent population.

5.2 NetLogo

Of all the ABM frameworks available, NetLogo is easier to use and is mainly for the novice ABM model designers with functionality to rapidly prototype different ABMs with the intention of using a more advance ABM framework to implement the actual model. In NetLogo, there is no agent defined, instead turtles [30] represent the individual entities of the simulation. Similar to Swarm and Repast, NetLogo is also sequential thereby inheriting the size limitations plagued by these frameworks.

5.3 MASON

MASON (Multi-Agent Simulator of Neighborhoods) is an ABM framework [9] that includes support for parallel and distributed execution. Multiple MASON instances can be used to run a simulation in parallel and improve the runtime as opposed to running the same simulation with a single instance as shown by [31]. MASON supports parallel thread execution in its model layer by providing "Steppable" wrappers which can group agents together and perform them in parallel on a separate thread [9]. There is still a size limitation however, because millions of agents would need more threads (in the thousands) to work cooperatively and therefore have a considerably lower runtime which the MASS library solution addresses with MASS-CUDA.

6 Conclusions

ABMs and many parallelizable scientific applications can be made more useful when there is little impact on the agents' size on the overall execution time. MASS-CUDA provides a massively parallel framework library that takes advantage of GPU accelerators to improve the overall execution time and enable ABMs to be run on commodity desktop or laptop computers that have a GPU.

6.1 Result statement

The GPU gives MASS library a platform to accelerate applications by providing a parallel ABM framework that can be used to implement ABMs and other scientific applications. This research has shown that a better runtime performance results when an ABM or scientific application is implemented using the MASS-CUDA version of the MASS library as opposed to implementing the same application using other ABM frameworks that depend on CPUs as accelerators.

6.2 Problems encountered & Future work

One desirable feature of MASS library is the ability to perform its acceleration on any computing device. However, the CUDA programming language only works on NVIDIA's GPUs, which makes it impossible to perform GPU acceleration on non-NVIDIA devices such as AMD. OpenCL is another alternative that has been developed by Khronos group, Apple, and NVIDIA, and it has the ability to run on any GPU device, including NVIDIA GPUs.

Although CUDA is in constant development and continually adds support for C and C++, and a growing list of other languages including Java, these new features require an updated hardware. For example, devices with compute capability less than 2.x do not have support for function pointers, which is a desirable feature of the MASS library parallel function. The function pointer problem is resolved by requiring the user to define the function to be passed into the GPU as a GPU function so that it can be executed on the GPU.

Another issue faced in this reresearch is decoupling the MASS-CUDA code from the user. It would improve the usability if the users of MASS-CUDA do not have to see any GPU code (or any CUDA-specific keywords) in the whole development cycle. The keywords used by CUDA should be hidden from the user and instead a wrapper to these underlying GPU functions should be provided. However, the fact that CUDA does not support the transfer of a CPU function to the GPU, this problem would have to be solved by creating a source-to-source translator [32] that parses a user-defined CPU function and generates a GPU version of the same function that can be run on the GPU. This gives room for more research in this area.

7 Appendix

7.1 Wave simulation – GPU version (Bare CUDA)

```
#include "util.h"
#define DEFAULT_SIZE 100
#define MAX_THREADS_PER_SM 2048
#define MAX_THREADS_PER_BLOCK 1024

class Wave2D
{
private:
       float* z0;           // previous - 1
       float* z1;           // previous
       float* z2;           // current
//     int nthreads;
       int threadsPerBlock;
       int blocksPerGrid;
       int interval;
       int iterations;
       float exectime;
public:
       int size;
       Wave2D();
       Wave2D( int _size, int _threadsPerBlock, int _blocksPerGrid, int _interval, int
_iterations );
       ~Wave2D();
       void doSimulation();
       float getExectime(){ return exectime; }
};

__global__ void doSimulation_kernel( float*, float*, float*, int size, int );
__device__ float compute_zt( float*, float*, float*, int size, int i, int j, int
time_unit );

int main( int argc, char* argv[] )
{
       if( argc < 6 ) return -1;
       int size = atoi( argv[1] );
       int iterations = atoi( argv[2] );
       int interval = atoi( argv[3] );
//     int threads = atoi( argv[4] );
       int blocksPerGrid = atoi( argv[4] );
       int threadsPerBlock = atoi( argv[5] );

       Wave2D wave2d( size, threadsPerBlock, blocksPerGrid, interval, iterations );

       // Perform simulation
       wave2d.doSimulation();
       printf( "exectime(ms)=%.2f\n", wave2d.getExectime() );

       return 0;
```

```
}


Wave2D::Wave2D()
{
//	printf( "Hello Wave2d in CUDA, " );
	size = DEFAULT_SIZE;
	int width = size, height = size;
	z0 = new float[ width * height ];
	z1 = new float[ width * height ];
	z2 = new float[ width * height ];
	threadsPerBlock = 1;
	blocksPerGrid = 1;
	interval = 1;
	iterations = 1;
	exectime = 0.0;
}
Wave2D::Wave2D( int _size, int _threadsPerBlock, int _blocksPerGrid, int _interval, int
_iterations )
{
//	printf( "Hello Wave2d in CUDA, " );
	size = _size;
	int width = size, height = size;
	z0 = new float[ width * height ];
	z1 = new float[ width * height ];
	z2 = new float[ width * height ];
	threadsPerBlock = _threadsPerBlock > 0 ? _threadsPerBlock : 1;
	blocksPerGrid = _blocksPerGrid > 0 ? _blocksPerGrid : 1;
	interval = _interval;
	iterations = _iterations;
}
Wave2D::~Wave2D()
{
	delete [] z0;
	delete [] z1;
	delete [] z2;
}
void Wave2D::doSimulation()
{
	cudaEvent_t start, stop;
	cudaEventCreate( &start );
	cudaEventCreate( &stop );
	cudaEventRecord( start, 0 );

	int width = size, height = size;
	size_t size1D = width * height * sizeof(float);
	float* d_space0;
	float* d_space1;
	float* d_space2;
	cudaMalloc( &d_space0, size1D );
	cudaMalloc( &d_space1, size1D );
	cudaMalloc( &d_space2, size1D );

//	int thrPerBlock = (int)sqrt(threadsPerBlock);
//	int blkPerGrid = (int)sqrt(blocksPerGrid);
//	thrPerBlock = thrPerBlock > 1 ? thrPerBlock : 1;
//	blkPerGrid = blkPerGrid > 1 ? blkPerGrid : 1;
	dim3 dimBlock( threadsPerBlock, threadsPerBlock, 1 );
```

```
       dim3 dimGrid( blocksPerGrid, blocksPerGrid, 1);
//     threadsPerBlock = thrPerBlock * thrPerBlock;
//     blocksPerGrid = blkPerGrid * blkPerGrid;

       cudaMemcpy( d_space0, z0, size1D, cudaMemcpyHostToDevice );
       cudaMemcpy( d_space1, z1, size1D, cudaMemcpyHostToDevice );
       cudaMemcpy( d_space2, z2, size1D, cudaMemcpyHostToDevice );

       for( int t = 0; t < iterations; t++ )
       {
              doSimulation_kernel<<<dimGrid, dimBlock>>>( d_space0, d_space1, d_space2,
size, t );
              cudaMemcpy( z0, d_space0, size1D, cudaMemcpyDeviceToHost );
              cudaMemcpy( z1, d_space1, size1D, cudaMemcpyDeviceToHost );
              cudaMemcpy( z2, d_space2, size1D, cudaMemcpyDeviceToHost );
//            if( size <= 20 && t % interval == 0 )
//                   print1d<float>( z2, size );
//            else if( t == iterations-1 )
//                   print1d<float>( z2, size );
       }

       cudaEventRecord( stop, 0 );
       cudaEventSynchronize( stop );
       cudaEventElapsedTime( &exectime, start, stop );

       // Select GPU device with the most cores i.e. with Compute Capability of at least
1.3
       int deviceNum;
       cudaGetDevice( &deviceNum );
       cudaDeviceProp deviceProp;
       memset( &deviceProp, 0, sizeof( cudaDeviceProp ) );

       deviceProp.major = 1;
       deviceProp.minor = 3;
       cudaChooseDevice( &deviceNum, &deviceProp );
       cudaGetDeviceProperties( &deviceProp, deviceNum );
       cudaSetDevice( deviceNum );

       // iterations == # of iterations
       printf( "GPU=%s, size=%d, iterations=%d, blocksPerGrid=%d, threadsPerBlock=%d, ",
       deviceProp.name, size, iterations, blocksPerGrid, threadsPerBlock );

       cudaFree( d_space0 );
       cudaFree( d_space1 );
       cudaFree( d_space2 );
}
__device__ float compute_zt( float* space0, float* space1, float* space2, int size, int
i, int j, int time_unit )
{
       float zt = 0.0;
       float c = 1.0, dt = 0.1, dd = 2.0;
       if( i == 0 || i == size-1 || j == 0 || j == size-1 )
       {
              return zt;
       }
       if( time_unit == 0 )
       {
              if( (i > 0.4*size && i < 0.6*size) && (j > 0.4*size && j < 0.6*size) )
```

```
            {
                   zt = 20.0;
                   return zt;    // Initializing disturbance in the middle of the water.
            }
            return zt;
     }
     else if( time_unit == 1 )
     {
            zt = space1[i*size + j] + c*c/2.0*(dt/dd)*(dt/dd)*(space1[(i+1)*size + j] +
                          space1[(i-1)*size + j] + space1[i*size + j+1] + space1[i*size
+ j-1] -
                          4.0 * space1[i*size + j]);
            return zt;
     }
     else if( time_unit >= 2 )
     {
            zt = 2.0 * space1[i*size + j] - space0[i*size + j] + c*c*(dt/dd)*(dt/dd) *
(
                          space1[(i+1)*size + j] + space1[(i-1)*size + j] +
space1[i*size + j+1] +
                          space1[i*size + j-1] -
                          4.0 * space1[i*size + j]);
            return zt;
     }
     return space2[i*size + j];
}
// NOTE: space0, space1, and space2 are 2D arrays in device memory
__global__ void doSimulation_kernel( float* space0, float* space1, float* space2, int
size, int time_unit )
{
     int x = threadIdx.x + blockIdx.x * blockDim.x;
     int y = threadIdx.y + blockIdx.y * blockDim.y;
     int i = x, j = y;
     int index = j + i*size;

     if( index < (size*size) )
     {
            space2[i*size + j] = compute_zt( space0, space1, space2, size, i, j,
time_unit );
     }
     __syncthreads();
     if( index < (size*size) )
     {
            space0[i*size + j] = space1[i*size + j];
            space1[i*size + j] = space2[i*size + j];
     }
     __syncthreads();
}
```

7.2 Wave simulation – GPU version (MASS-CUDA)

```
#include "util.h"
#define DEFAULT_SIZE 100

#include "places.h"

class Wave2D : public Places<float>
{
private:
       float* z0;           // previous - 1
       float* z1;           // previous
       float* z2;           // current
//     int nthreads;
       int threadsPerBlock;
       int blocksPerGrid;
       int interval;
       int iterations;
       float exectime;
public:
       int size;
       Wave2D();
       Wave2D( int _size, int _threadsPerBlock, int _blocksPerGrid, int _interval, int
_iterations );
       ~Wave2D();
       void doSimulation();
       float getExectime(){ return exectime; }
};

__global__ void doSimulation_kernel( Wave2D*, float*, float*, float*, int size, int );
__device__ float compute_zt( float*, float*, float*, int size, int i, int j, int
time_unit );

int main( int argc, char* argv[] )
{
       if( argc < 6 ) return -1;
       int size = atoi( argv[1] );
       int iterations = atoi( argv[2] );
       int interval = atoi( argv[3] );
//     int threads = atoi( argv[4] );
       int blocksPerGrid = atoi( argv[4] );
       int threadsPerBlock = atoi( argv[5] );

       Wave2D wave2d( size, threadsPerBlock, blocksPerGrid, interval, iterations );

       // Perform simulation
       wave2d.doSimulation();
       printf( "exectime(ms)=%.2f\n", wave2d.getExectime() );

       return 0;
}


Wave2D::Wave2D()
{
//     printf( "Hello Wave2d in CUDA, " );
       size = DEFAULT_SIZE;
```

```
	int width = size, height = size;
	z0 = new float[ width * height ];
	z1 = new float[ width * height ];
	z2 = new float[ width * height ];
	threadsPerBlock = 1;
	blocksPerGrid = 1;
	interval = 1;
	iterations = 1;
	exectime = 0.0;
}
Wave2D::Wave2D( int _size, int _threadsPerBlock, int _blocksPerGrid, int _interval, int
_iterations )
{
//	printf( "Hello Wave2d in CUDA, " );
	size = _size;
	int width = size, height = size;
	z0 = new float[ width * height ];
	z1 = new float[ width * height ];
	z2 = new float[ width * height ];
	threadsPerBlock = _threadsPerBlock > 0 ? _threadsPerBlock : 1;
	blocksPerGrid = _blocksPerGrid > 0 ? _blocksPerGrid : 1;
	interval = _interval;
	iterations = _iterations;
}
Wave2D::~Wave2D()
{
	delete [] z0;
	delete [] z1;
	delete [] z2;
}
void Wave2D::doSimulation()
{
	cudaEvent_t start, stop;
	cudaEventCreate( &start );
	cudaEventCreate( &stop );
	cudaEventRecord( start, 0 );

	int width = size, height = size;
	size_t size1D = width * height * sizeof(float);
	size_t sizeOfWave2D = sizeof(Wave2D);
	Wave2D* d_wave2d;
	float* d_space0;
	float* d_space1;
	float* d_space2;
	cudaMalloc( &d_wave2d, sizeOfWave2D );
	cudaMalloc( &d_space0, size1D );
	cudaMalloc( &d_space1, size1D );
	cudaMalloc( &d_space2, size1D );

/*
	// Max threads per multiprocessor = 2048, (hercules = 8 MP, max threads = 16384)
	// Max threads per block = 1024
	nthreads = nthreads < (int)sqrt(1024) ? nthreads : 32;
	dim3 dimBlock( nthreads, nthreads, 1 );
	int gridx = (size + dimBlock.x - 1)/dimBlock.x;
	int gridy = (size + dimBlock.y - 1)/dimBlock.y;
	gridx = gridx < (int)sqrt(1024) ? gridx : (int)sqrt(1024);
	gridy = gridy < (int)sqrt(1024) ? gridy : (int)sqrt(1024);
	//gridx = 1; gridy = 1;
	dim3 dimGrid( gridx, gridy, 1 );
*/

//	int threadsPerBlock = dimBlock.x * dimBlock.y;
//	int blocksPerGrid = dimGrid.x * dimGrid.y;
//	int totalThreads = threadsPerBlock * blocksPerGrid;

	dim3 dimBlock( threadsPerBlock, threadsPerBlock, 1 );
	dim3 dimGrid( blocksPerGrid, blocksPerGrid, 1);

	for( int t = 0; t < iterations; t++ )
	{
		doSimulation_kernel<<<dimGrid, dimBlock>>>( d_wave2d, d_space0, d_space1,
d_space2, size, t );
		cudaMemcpy( z0, d_space0, size1D, cudaMemcpyDeviceToHost );
		cudaMemcpy( z1, d_space1, size1D, cudaMemcpyDeviceToHost );
		cudaMemcpy( z2, d_space2, size1D, cudaMemcpyDeviceToHost );
//		if( size <= 20 && t % interval == 0 )
//			print1d<float>( z2, size );
//		else if( t == iterations-1 )
//			print1d<float>( z2, size );
	}

	cudaEventRecord( stop, 0 );
	cudaEventSynchronize( stop );
	cudaEventElapsedTime( &exectime, start, stop );

	// Select GPU device with the most cores i.e. with Compute Capability of at least
1.3
	int deviceNum;
	cudaGetDevice( &deviceNum );
	cudaDeviceProp deviceProp;
	memset( &deviceProp, 0, sizeof( cudaDeviceProp ) );

	deviceProp.major = 1;
	deviceProp.minor = 3;
	cudaChooseDevice( &deviceNum, &deviceProp );
	cudaGetDeviceProperties( &deviceProp, deviceNum );
	cudaSetDevice( deviceNum );

	// iterations == # of iterations
	printf( "GPU(MASS)=%s, size=%d, iterations=%d, blocksPerGrid=%d,
threadsPerBlock=%d, ",
	deviceProp.name, size, iterations, blocksPerGrid, threadsPerBlock );

	cudaFree( d_wave2d );
	cudaFree( d_space0 );
	cudaFree( d_space1 );
	cudaFree( d_space2 );
}
__device__ float compute_zt( float* space0, float* space1, float* space2, int size, int
i, int j, int time_unit )
{
	float zt = 0.0;
	float c = 1.0, dt = 0.1, dd = 2.0;
	if( i == 0 || i == size-1 || j == 0 || j == size-1 )
	{
		space2[i*size + j] = zt;
		return zt;
	}
	if( time_unit == 0 )
	{
		if( (i > 0.4*size && i < 0.6*size) && (j > 0.4*size && j < 0.6*size) )
		{
			zt = 20.0;
			space2[i*size + j] = zt;
			return zt;    // Initializing disturbance in the middle of the water.
		}
		space2[i*size + j] = zt;
		return zt;
	}
	else if( time_unit == 1 )
	{
		zt = space1[i*size + j] + c*c/2.0*(dt/dd)*(dt/dd)*(space1[(i+1)*size + j] +
				space1[(i-1)*size + j] + space1[i*size + j+1] + space1[i*size
+ j-1] -
				4.0 * space1[i*size + j]);
		space2[i*size + j] = zt;
		return zt;
	}
	else if( time_unit >= 2 )
	{
		zt = 2.0 * space1[i*size + j] - space0[i*size + j] + c*c*(dt/dd)*(dt/dd) *
(
				space1[(i+1)*size + j] + space1[(i-1)*size + j] +
space1[i*size + j+1] +
				space1[i*size + j-1] -
				4.0 * space1[i*size + j]);
		space2[i*size + j] = zt;
		return zt;
	}
	return space2[i*size + j];
}

// NOTE: space0, space1, and space2 are 2D arrays in device memory
__global__ void doSimulation_kernel( Wave2D* wave2d, float* space0, float* space1, float*
space2, int size, int time_unit )
{
	int x = threadIdx.x + blockIdx.x * blockDim.x;
	int y = threadIdx.y + blockIdx.y * blockDim.y;
	int i = y, j = x;
	int index = j + i*size;

	wave2d->callAll( compute_zt, space0, space1, space2, size, time_unit );

	__syncthreads();

	if( index < (size*size) )
	{
		space0[i*size + j] = space1[i*size + j];
		space1[i*size + j] = space2[i*size + j];
	}
	__syncthreads();
}
```

7.3 Wave simulation – CPU version (C++)

```
#include <stdio.h>
#include <stdlib.h>
#include "util.h"
#include <omp.h>
#define DEFAULT_SIZE 100

class Wave2D
{
private:
	int N;
	float** z0;		// previous - 1
	float** z1;		// previous
	float** z2;		// current
	int nthreads;
	int interval;
	int iterations;
	float exectime;
public:
	Wave2D();
	Wave2D( int _N, int _nthreads, int _interval, int _iterations );
	~Wave2D();
	int	getSize(){ return N; }
	int getNumThreads(){ return nthreads; }
	void doSimulation();
	double getExectime(){ return exectime; }
	float compute_zt( float**, float**, float**, int i, int j, int time_unit );
};

int main( int argc, char* argv[] )
{
	if( argc < 5 ) return -1;
	int size = atoi( argv[1] );
	int iterations = atoi( argv[2] );
	int interval = atoi( argv[3] );
	int threads = atoi( argv[4] );

	Wave2D wave2d( size, threads, interval, iterations );

	// Perform simulation
	wave2d.doSimulation();
	printf( "CPU, size=%d, iterations=%d, threads=%d, exectime(ms)=%.2f\n",
	wave2d.getSize(), iterations, wave2d.getNumThreads(), wave2d.getExectime()*1000 );

	return 0;
}

Wave2D::Wave2D()
{
//	printf( "Hello Wave2D, " );
	N = DEFAULT_SIZE;
	z0 = new float*[ N ];
	z1 = new float*[ N ];
	z2 = new float*[ N ];
	for( int i = 0; i < N; i++ )
	{
		z0[i] = new float[ N ];
```

```
            z1[i] = new float[ N ];
            z2[i] = new float[ N ];
        }
        nthreads = 1;
        interval = 1;
        iterations = 1;
        exectime = 0.0;
}
Wave2D::Wave2D( int _N, int _nthreads, int _interval, int _iterations )
{
        N = _N;
        z0 = new float*[ N ];
        z1 = new float*[ N ];
        z2 = new float*[ N ];
        for( int i = 0; i < N; i++ )
        {
            z0[i] = new float[ N ];
            z1[i] = new float[ N ];
            z2[i] = new float[ N ];
        }
        nthreads = _nthreads < 1 ? 1 : _nthreads;
#ifdef _OPENMP
        nthreads = nthreads <= omp_get_max_threads() ? nthreads : omp_get_max_threads();
        omp_set_num_threads( nthreads );
#endif
        interval = _interval;
        iterations = _iterations;
}
Wave2D::~Wave2D()
{
        for( int i = 0; i < N; i++ )
        {
            delete [] z0[i];
            delete [] z1[i];
            delete [] z2[i];
        }
        delete [] z0;
        delete [] z1;
        delete [] z2;
}

void Wave2D::doSimulation()
{
        double start = cpu_time_omp();
        int i = 0, j = 0;

        for( int t = 0; t < iterations; t++ )
        {
#ifdef _OPENMP
            #pragma omp parallel for \
            private( i, j )
//          shared( z0, z1, z2 )
#endif
            for( i = 0; i < N; i++ ) // Compute wave
            {
                for( j = 0; j < N; j++ )
                {
                    z2[i][j] = compute_zt( z0, z1, z2, i, j, t );
                    }
                }
#ifdef _OPENMP
                #pragma omp barrier
#endif

/*
#ifdef _OPENMP
                #pragma omp parallel for \
                private( i, j )
//              shared( z0, z1, z2 )
#endif
*/
                for( i = 0; i < N; i++ ) // Copy
                {
                        for( j = 0; j < N; j++ )
                        {
                                z0[i][j] = z1[i][j];
                                z1[i][j] = z2[i][j];
                        }
                }
        }

        exectime = cpu_time_omp() - start;
}

float Wave2D::compute_zt( float** space0, float** space1, float** space2, int i, int j,
int time_unit )
{
        float zt = 0.0;
        float c = 1.0, dt = 0.1, dd = 2.0;
        if( i == 0 || i == N-1 || j == 0 || j == N-1 )
        {
                return zt;
        }
        if( time_unit == 0 )
        {
                if( (i > 0.4*N && i < 0.6*N) && (j > 0.4*N && j < 0.6*N) )
                {
                        zt = 20.0;
                        return zt;    // Initializing disturbance in the middle of the water.
                }
                return zt;
        }
        else if( time_unit == 1 )
        {
                zt = space1[i][j] + c*c/2.0*(dt/dd)*(dt/dd)*(space1[i+1][j] +
                                space1[i-1][j] + space1[i][j+1] + space1[i][j-1] -
                                4.0 * space1[i][j]);
                return zt;
        }
        else if( time_unit >= 2 )
        {
                zt = 2.0 * space1[i][j] - space0[i][j] + c*c*(dt/dd)*(dt/dd) * (
                                space1[i+1][j] + space1[i-1][j] + space1[i][j+1] +
                                space1[i][j-1] -
                                4.0 * space1[i][j]);
                return zt;
        }
        return space2[i][j];
}
```

7.4 Game of Life – GPU version (Bare CUDA)

```
#ifndef _GAMEOFLIFE_H_
#define _GAMEOFLIFE_H_

// Constants
#define DEFAULT_SIZE 10
#define DEFAULT_GENERATIONS 10
#define ON 'O'
#define OFF ' '
#define LEFT        0
#define RIGHT       1
#define DOWN        2
#define UP                 3
#define UPLEFT             4
#define UPRIGHT            5
#define DOWNLEFT    6
#define DOWNRIGHT   7


#include <time.h>
#include <fstream>
using namespace std;

class Cell
{
private:
       char state;   // ON or OFF
public:
       Cell(){ state = OFF; }
       void SetState( char _state ){ state = _state; }
       char GetState(){ return state; }
};

struct Grid
{
       int size;
       Cell* cells;
};

// Life is based on Cellular Automata (CA). Macal and North mentioned that CA is the
simplest
// way to illustrate the basic ideas of agent-based modeling and simulation.

// Apply Rules (Macal and North, 2009)
// 1. The cell will be On in the next generation if exactly three of its eight
//            neighboring cells are currently On
// 2. The cell will retain its current state if exactly two of its neighbors
//            are On.
// 3. The cell will be Off otherwise

class GameOfLife
{
private:
       int size;
       int generations;
       Grid space;
       void Init( int _size, int _generations );
```

```
	// Mutators
	void Set( Grid, int i, int j, int _index, char state );

	// Accessors
	char Get( Grid, int i, int j, int _index );
	char Get( Grid grid, int i, int j, int _index, int direction );
	int GetOnCells();
	int GetOffCells();

public:
	GameOfLife();
	GameOfLife( int _size, int _generations );
	~GameOfLife();
	void Start();
	void Print( Grid );
	void WriteToFile( char*, float );
};

#endif

// Implementation

GameOfLife::GameOfLife()
{
	Init( DEFAULT_SIZE, DEFAULT_GENERATIONS );
}
GameOfLife::GameOfLife( int _size, int _generations )
{
	Init( _size, _generations );
}
GameOfLife::~GameOfLife()
{
	delete [] space.cells;
}
void GameOfLife::Init( int _size, int _generations )
{
	// Initialize random seed
	srand( time(NULL) );

	size = _size;
	generations = _generations;
	space.size = size;
	space.cells = new Cell[ size*size ];
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			int num = rand() % 2; // range 0 to 1
			space.cells[j+i*size].SetState( num == 1 ? ON : OFF );
		}
	}
}

void GameOfLife::Start()
{
	Print( space );
	int interval = generations / 4;
	for( int gen = 0 ; gen < generations; gen++ )
	{
		for( int i = 0; i < size; i++ )
		{
			for( int j = 0; j < size; j++ )
			{
				int countOn = 0;	// neigbhors who are On.
				int index = j+i*size;

				// Check neigbors: left, right up, down, upleft, upright,
downleft, downright
				if( ON == Get( space, i, j, index, LEFT ) ) countOn++;
				if( ON == Get( space, i, j, index, RIGHT ) ) countOn++;
				if( ON == Get( space, i, j, index, UP ) ) countOn++;
				if( ON == Get( space, i, j, index, DOWN ) ) countOn++;
				if( ON == Get( space, i, j, index, UPLEFT ) ) countOn++;
				if( ON == Get( space, i, j, index, UPRIGHT ) ) countOn++;
				if( ON == Get( space, i, j, index, DOWNLEFT ) ) countOn++;
				if( ON == Get( space, i, j, index, DOWNRIGHT ) ) countOn++;

				// Apply Rules
				// 1. The cell will be On in the next generation if exactly
three of its eight
				//		neighboring cells are currently On
				if( countOn == 3 )
				{
					Set( space, i, j, j+i*size, ON );
				}
				// 2. The cell will retain its current state if exactly two of
its neighbors
				//		are On.
				else if( countOn == 2 ) ; // Do nothing.
				// 3. The cell will be Off otherwise
				else
				{
					Set( space, i, j, j+i*size, OFF );
				}
			}
		}
		if( gen % interval == 0 || gen == generations-1 )
		{
			printf( "Generation %d:\n", gen );
			Print( space );
		}
	}
}
void GameOfLife::Set( Grid grid, int i, int j, int index, char state )
{
	if( i < 0 || i >= grid.size || j < 0 || j >= grid.size ) return;
	if( index >= (grid.size * grid.size) ) return;

	grid.cells[index].SetState( state );
}

// Accessors
char GameOfLife::Get( Grid grid, int i, int j, int index )
{
	if( i < 0 || i >= grid.size || j < 0 || j >= grid.size ) return '\n';
	if( index >= (grid.size * grid.size) ) return '\n';

	return grid.cells[index].GetState();
}
char GameOfLife::Get( Grid grid, int i, int j, int _index, int direction )
{
	switch( direction )
	{
		case LEFT:		return Get( grid, i, j-1, i*size + (j-1) );
		case RIGHT:		return Get( grid, i, j+1, i*size + (j+1) );
		case UP:		return Get( grid, i-1, j, (i-1)*size + j );
		case DOWN:		return Get( grid, i+1, j, (i+1)*size + j );
		case UPLEFT:	return Get( grid, i-1, j-1, (i-1)*size + (j-1) );
		case UPRIGHT: return Get( grid, i-1, j+1, (i-1)*size + (j+1) );
		case DOWNLEFT:		return Get( grid, i+1, j-1, (i+1)*size + (j-1) );
		case DOWNRIGHT:		return Get( grid, i+1, j+1, (i+1)*size + (j+1) );
	}
}
int GameOfLife::GetOnCells()
{
	int onCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].GetState() == ON ) onCells++;
	return onCells;
}
int GameOfLife::GetOffCells()
{
	int offCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].GetState() == OFF ) offCells++;
	return offCells;
}

// Display
void GameOfLife::Print( Grid grid )
{
	if( size > 50 ) return;
	printf( "On Cells: %d, Off Cells: %d\n", GetOnCells(), GetOffCells() );
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			printf( "%c ", grid.cells[j+i*size].GetState() );
		}
		printf( "\n" );
	}
	printf( "\n" );
}
void GameOfLife::WriteToFile( char* file, float runtime )
{
	ofstream myfile;
	myfile.open( file, ios::out | ios::app );
	if( myfile.is_open() )
	{
		myfile << size << "\t" << generations << "\t" << runtime << endl;
		myfile.close();
```

```
        }
        else
                printf( "Unable to open file\n" );
}
```

7.5 Game of Life – GPU version (MASS-CUDA)

```
#include "gameoflife_masscuda.h"


__global__
void callAll_kernel( Grid grid, int* d_functionId );
__device__
void compute_kernel( Grid grid, int tid, int i, int j, int stride );
void callAll( int _threads, int _threadspblock, Grid d_space, int* d_functionId );

// Implementation

GameOfLife::GameOfLife()
{
	init( DEFAULT_SIZE, DEFAULT_GENERATIONS, 1 );
}
GameOfLife::GameOfLife( int _size, int _generations, int _nthreads )
{
	init( _size, _generations, _nthreads );
}
GameOfLife::~GameOfLife()
{
	delete [] space.cells;
}
void GameOfLife::init( int _size, int _generations, int _nthreads )
{
	// Initialize random seed
	srand( time(NULL) );

	size = _size;
	generations = _generations;
	space.size = size;
	space.cells = new Cell[ size*size ];
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			int num = rand() % 2; // range 0 to 1
			space.cells[j+i*size].state = num == 1 ? ON : OFF;
		}
	}
	SetThreads( _nthreads );
}


void GameOfLife::Start()
{
	Print( space );
	int interval = generations / 4;
	int* functionId = new int[1];
	functionId[0] = COMPUTE;

	// Allocate memory on the GPU
	int* d_functionId;
	Grid d_space;
	d_space.size = space.size;
	size_t sizeofCells = size*size * sizeof( Cell );
	cudaMalloc( &d_space.cells, sizeofCells );
	cudaMalloc( &d_functionId, sizeof(int) );
	cudaMemcpy( d_functionId, functionId, sizeof(int), cudaMemcpyHostToDevice );

	for( int gen = 0 ; gen < generations; gen++ )
	{
		// Copy to GPU memory
		cudaMemcpy(   d_space.cells, space.cells, sizeofCells,
cudaMemcpyHostToDevice );

		// Perform computation on GPU
		callAll( blockspergrid, threadsperblock, d_space, d_functionId );

		// Copy results from GPU memory to main memory
		cudaMemcpy( space.cells, d_space.cells, sizeofCells, cudaMemcpyDeviceToHost
);

		if( gen % interval == 0 || gen == generations-1 )
		{
			printf( "Generation %d:\n", gen );
			Print( space );
		}
	}

	// Destroy allocated memory on the GPU
	cudaFree( d_functionId );
	cudaFree( d_space.cells );

	// Destroy memory on CPU
	delete [] functionId;
}

void GameOfLife::SetThreads( int _nthreads )
{
	nthreads = _nthreads < 1 ? 1 : _nthreads;
	blockspergrid = (nthreads + BLOCK_SIZE-1)/BLOCK_SIZE;
	threadsperblock = BLOCK_SIZE;
}

// Accessors
int GameOfLife::GetOnCells()
{
	int onCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].state == ON ) onCells++;
	return onCells;
}
int GameOfLife::GetOffCells()
{
	int offCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].state == OFF ) offCells++;
	return offCells;
}

// Display
```

```
void GameOfLife::Print( Grid grid )
{
	if( size > 50 ) return;
	printf( "On Cells: %d, Off Cells: %d\n", GetOnCells(), GetOffCells() );
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			printf( "%c ", grid.cells[j+i*size].state );
		}
		printf( "\n" );
	}
	printf( "\n" );
}
void GameOfLife::WriteToFile( char* file, float runtime )
{
	ofstream myfile;
	myfile.open( file, ios::out | ios::app );
	if( myfile.is_open() )
	{
		myfile << size << "\t" << generations << "\t" << nthreads << "\t" << 
runtime
		<< "\t" << "MASS-CUDA" << endl;
		myfile.close();
	}
	else
		printf( "Unable to open file\n" );
}
```

7.6 Game of Life – CPU version (C++)

```
#ifndef _GAMEOFLIFE_OMP_H_
#define _GAMEOFLIFE_OMP_H_

// Constants
#define DEFAULT_SIZE 10
#define DEFAULT_GENERATIONS 10
#define ON 'O'
#define OFF ' '
#define LEFT         0
#define RIGHT        1
#define DOWN         2
#define UP                  3
#define UPLEFT              4
#define UPRIGHT             5
#define DOWNLEFT     6
#define DOWNRIGHT    7

#include <omp.h>
#include <time.h>
#include <fstream>
using namespace std;

class Cell
{
private:
       char state;   // ON or OFF
public:
       Cell(){ state = OFF; }
       void SetState( char _state ){ state = _state; }
       char GetState(){ return state; }
};

struct Grid
{
       int size;
       Cell* cells;
};

// Life is based on Cellular Automata (CA). Macal and North mentioned that CA is the
simplest
// way to illustrate the basic ideas of agent-based modeling and simulation.

// Apply Rules (Macal and North, 2009)
// 1. The cell will be On in the next generation if exactly three of its eight
//            neighboring cells are currently On
// 2. The cell will retain its current state if exactly two of its neighbors
//            are On.
// 3. The cell will be Off otherwise

class GameOfLife
{
private:
       int nthreads;
       int size;
       int generations;
       Grid space;
```

```
	void Init( int _size, int _generations, int _nthreads );

	// Mutators
	void Set( Grid, int i, int j, int _index, char state );

	// Accessors
	char Get( Grid, int i, int j, int _index );
	char Get( Grid grid, int i, int j, int _index, int direction );
	int GetOnCells();
	int GetOffCells();

public:
	GameOfLife();
	GameOfLife( int _size, int _generations, int _nthreads );
	~GameOfLife();
	void Start();
	void SetThreads( int nthreads );
	int GetThreads(){ return nthreads; }
	void Print( Grid );
	void WriteToFile( char*, float );
};

#endif

// Implementation

GameOfLife::GameOfLife()
{
	Init( DEFAULT_SIZE, DEFAULT_GENERATIONS, 1 );
}
GameOfLife::GameOfLife( int _size, int _generations, int _nthreads )
{
	Init( _size, _generations, _nthreads );
}
GameOfLife::~GameOfLife()
{
	delete [] space.cells;
}
void GameOfLife::Init( int _size, int _generations, int _nthreads )
{
	// Initialize random seed
	srand( time(NULL) );

	size = _size;
	generations = _generations;
	space.size = size;
	space.cells = new Cell[ size*size ];
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			int num = rand() % 2; // range 0 to 1
			space.cells[j+i*size].SetState( num == 1 ? ON : OFF );
		}
	}
	SetThreads( _nthreads );
}
```

```
void GameOfLife::Start()
{
       Print( space );
       int interval = generations / 4;
       for( int gen = 0 ; gen < generations; gen++ )
       {
              int i = 0, j = 0;

#ifdef _OPENMP
              #pragma omp parallel for \
              private( i, j )
#endif
              for( i = 0; i < size; i++ )
              {
                     for( j = 0; j < size; j++ )
                     {
                            int countOn = 0;     // neigbhors who are On.
                            int index = j+i*size;

                            // Check neigbors: left, right up, down, upleft, upright,
downleft, downright
                            if( ON == Get( space, i, j, index, LEFT ) ) countOn++;
                            if( ON == Get( space, i, j, index, RIGHT ) ) countOn++;
                            if( ON == Get( space, i, j, index, UP ) ) countOn++;
                            if( ON == Get( space, i, j, index, DOWN ) ) countOn++;
                            if( ON == Get( space, i, j, index, UPLEFT ) ) countOn++;
                            if( ON == Get( space, i, j, index, UPRIGHT ) ) countOn++;
                            if( ON == Get( space, i, j, index, DOWNLEFT ) ) countOn++;
                            if( ON == Get( space, i, j, index, DOWNRIGHT ) ) countOn++;

                            // Apply Rules
                            // 1. The cell will be On in the next generation if exactly
three of its eight
                            //            neighboring cells are currently On
                            if( countOn == 3 )
                            {
                                   Set( space, i, j, j+i*size, ON );
                            }
                            // 2. The cell will retain its current state if exactly two of
its neighbors
                            //            are On.
                            else if( countOn == 2 ) ; // Do nothing.
                            // 3. The cell will be Off otherwise
                            else
                            {
                                   Set( space, i, j, j+i*size, OFF );
                            }
                     }
              }
              if( gen % interval == 0 || gen == generations-1 )
              {
                     printf( "Generation %d:\n", gen );
                     Print( space );
              }
       }
}
void GameOfLife::Set( Grid grid, int i, int j, int index, char state )
{
	if( i < 0 || i >= grid.size || j < 0 || j >= grid.size ) return;
	if( index >= (grid.size * grid.size) ) return;

	grid.cells[index].SetState( state );
}
void GameOfLife::SetThreads( int _nthreads )
{
	nthreads = _nthreads < 1 ? 1 : _nthreads;
#ifdef _OPENMP
	nthreads = nthreads <= omp_get_max_threads() ? nthreads : omp_get_max_threads();
	omp_set_num_threads( nthreads );
#endif
}

// Accessors
char GameOfLife::Get( Grid grid, int i, int j, int index )
{
	if( i < 0 || i >= grid.size || j < 0 || j >= grid.size ) return '\n';
	if( index >= (grid.size * grid.size) ) return '\n';

	return grid.cells[index].GetState();
}
char GameOfLife::Get( Grid grid, int i, int j, int _index, int direction )
{
	switch( direction )
	{
		case LEFT:		return Get( grid, i, j-1, i*size + (j-1) );
		case RIGHT:		return Get( grid, i, j+1, i*size + (j+1) );
		case UP:		return Get( grid, i-1, j, (i-1)*size + j );
		case DOWN:		return Get( grid, i+1, j, (i+1)*size + j );
		case UPLEFT:	return Get( grid, i-1, j-1, (i-1)*size + (j-1) );
		case UPRIGHT: return Get( grid, i-1, j+1, (i-1)*size + (j+1) );
		case DOWNLEFT:		return Get( grid, i+1, j-1, (i+1)*size + (j-1) );
		case DOWNRIGHT:		return Get( grid, i+1, j+1, (i+1)*size + (j+1) );
	}
}
int GameOfLife::GetOnCells()
{
	int onCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].GetState() == ON ) onCells++;
	return onCells;
}
int GameOfLife::GetOffCells()
{
	int offCells = 0;
	for( int i = 0; i < size; i++ )
		for( int j = 0; j < size; j++ )
			if( space.cells[j+i*size].GetState() == OFF ) offCells++;
	return offCells;
}

// Display
void GameOfLife::Print( Grid grid )
{
	if( size > 50 ) return;
	printf( "On Cells: %d, Off Cells: %d\n", GetOnCells(), GetOffCells() );
	for( int i = 0; i < size; i++ )
	{
		for( int j = 0; j < size; j++ )
		{
			printf( "%c ", grid.cells[j+i*size].GetState() );
		}
		printf( "\n" );
	}
	printf( "\n" );
}
void GameOfLife::WriteToFile( char* file, float runtime )
{
	ofstream myfile;
	myfile.open( file, ios::out | ios::app );
	if( myfile.is_open() )
	{
		myfile << size << "\t" << generations << "\t" << nthreads << "\t" <<
runtime << endl;
		myfile.close();
	}
	else
		printf( "Unable to open file\n" );
}
```

References

[1] J. Emau, T. Chuang and M. Fukuda. A multi-process library for multi-agent and spatial simulation. Presented at Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on. 2011, .

[2] J. M. Epstein, R. Axtell and 2050 Project. *Growing Artificial Societies: Social Science from the Bottom Up* 1996Available: http://cognet.mit.edu/library/books/view?isbn=0262550253.

[3] C. M. Macal and M. J. North. Agent-based modeling and simulation. Presented at Simulation Conference (WSC), Proceedings of the 2009 Winter. 2009, .

[4] S. F. Railsback and V. Grimm. *Agent-Based and Individual-Based Modeling: A Practical Introduction* 2011.

[5] NYC Department of City Planning. Current estimates of new york city's population for july 2011. [Online]. *2012(November 19th),* 2011. Available: http://www.nyc.gov/html/dcp/html/census/popcur.shtml.

[6] Federal Aviation Administration. Press release – FAA celebrates 75th anniversary of air traffic control. [Online]. *2012(November 20th),* 2011. Available: http://www.faa.gov/news/press_releases/news_story.cfm?newsId=12903.

[7] U. Wilensky. NetLogo. [Online]. *2012(November 19th),* 1999. Available: http://ccl.northwestern.edu/netlogo/.

[8] ARGONNE NATIONAL LABORATORY. The repast suite. [Online]. *2012(November 19th),* 2012. Available: http://repast.sourceforge.net/index.html.

[9] S. Luke, G. C. Balan, K. Sullivan and L. Panait. **M**ulti-**A**gent **S**imulator **O**f **N**eighborhoods... or **N**etworks. [Online]. *2012(November 19th),* 2012. Available: http://cs.gmu.edu/~eclab/projects/mason/.

[10] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming* 2011.

[11] J. Y. Shi, M. Taifi, A. Khreishah and Jie Wu. Sustainable GPU computing at scale. Presented at Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on. 2011, .

[12] Zhen Shen, Kai Wang and Fenghua Zhu. Agent-based traffic simulation and traffic signal timing optimization with GPU. Presented at Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on. 2011, .

[13] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach* 2010.

[14] J. Nickolls and W. J. Dally. The GPU computing era. *Micro, IEEE 30(2),* pp. 56-69. 2010.

[15] R. M. D'Souza, M. Lysenko, S. Marino and D. Kirschner. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. Presented at Proceedings of the 2009

Spring Simulation Multiconference. 2009, Available: http://dl.acm.org/citation.cfm?id=1639809.1639831.

[16] P. Richmond, S. Coakley and D. M. Romano. A high performance agent based modelling framework on graphics card hardware with CUDA. Presented at Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2. 2009, Available: http://dl.acm.org/citation.cfm?id=1558109.1558172.

[17] D. Strippgen and K. Nagel. Multi-agent traffic simulation with CUDA. Presented at High Performance Computing & Simulation, 2009. HPCS '09. International Conference on. 2009, .

[18] Enhua Wu and Youquan Liu. Emerging technology about GPGPU. Presented at Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on. 2008, .

[19] J. Nickolls, I. Buck, M. Garland and K. Skadron. Scalable parallel programming with CUDA. *Queue 6(2),* pp. 40-53. 2008. Available: http://doi.acm.org/10.1145/1365490.1365500.

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips. GPU computing. *Proceedings of the IEEE 96(5),* pp. 879-899. 2008.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. Presented at Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2008, Available: http://doi.acm.org/10.1145/1345206.1345220.

[22] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Comput. Graphics Forum 26(1),* pp. 80-113. 2007.

[23] W. Chen, K. Ward, Q. Li, V. Kecman, K. Najarian and N. Menke. Agent based modeling of blood coagulation system: Implementation using a GPU based high speed framework. Presented at Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE. 2011, .

[24] J. C. Phillips, J. E. Stone and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. Presented at High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for. 2008, .

[25] J. L. Schiff. *Cellular Automata: A Discrete View of the World* 2008Available: http://www.loc.gov/catdir/enhancements/fy0801/2007042557-d.html; http://www.loc.gov/catdir/enhancements/fy0806/2007042557-b.html; http://www.loc.gov/catdir/enhancements/fy0808/2007042557-t.html.

[26] M. Fukuda, "MASS: Parallel-Computing Library for Multi-Agent Spatial Simulation`," vol. 2012, pp. 19, May 7th, 2010, 2010.

[27] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA, Tech. Rep. 95/7, 1995.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput.Archit.News 18(3a),* pp. 15-26. 1990. Available: http://doi.acm.org/10.1145/325096.325102.

[29] N. Minar, R. Burkhart, C. Langton and M. Askenazi, "The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations," vol. 2012, pp. 11, June 21, 1996, 1996.

[30] S. Bandini, S. Manzoni and G. Vizzari. Agent based modeling and simulation: An informatics perspective. *Journal of Artificial Societies and Social Simulation 12(4),* pp. 4. 2009. Available: http://jasss.soc.surrey.ac.uk/12/4/4.html.

[31] M. Laclav\'\ik, Dlugolinsk\'y \vStefan, M. \vSeleng, M. Kvassay, B. Schneider, H. Bracker, M. Wrzeszcz, J. Kitowski and Hluch\'y Ladislav. Agent-based simulation platform evaluation in the context of human behavior modeling. Presented at Proceedings of the 10th International Conference on Advanced Agent Technology. 2012, Available: http://dx.doi.org/10.1007/978-3-642-27216-5_30.

[32] K. D. Cooper and L. Torczon. *Engineering a Compiler* (2nd ed.) 2012.