©Copyright JASSS

Mikola Lysenko and Roshan M. D'Souza (2008)

A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units

Journal of Artificial Societies and Social Simulation vol. 11, no. 4 10
<http://jasss.soc.surrey.ac.uk/11/4/10.html>

For information about citing this article, click here

Received: 23-Jun-2008 Accepted: 24-Sep-2008 Published: 31-Oct-2008

Abstract

Agent-based modeling is a technique for modeling dynamic systems from the bottom up. Individual elements of the system are represented computationally as agents. The system-level behaviors emerge from the micro-level interactions of the agents. Contemporary state-of-the-art agent-based modeling toolkits are essentially discrete-event simulators designed to execute serially on the Central Processing Unit (CPU). They simulate Agent-Based Models (ABMs) by executing agent actions one at a time. In addition to imposing an un-natural execution order, these toolkits have limited scalability. In this article, we investigate data-parallel computer architectures such as Graphics Processing Units (GPUs) to simulate large scale ABMs. We have developed a series of efficient, data parallel algorithms for handling environment updates, various agent interactions, agent death and replication, and gathering statistics. We present three fundamental innovations that provide unprecedented scalability. The first is a novel stochastic memory allocator which enables parallel agent replication in O(1) average time. The second is a technique for resolving precedence constraints for agent actions in parallel. The third is a method that uses specialized graphics hardware, to gather and process statistical measures. These techniques have been implemented on a modern day GPU resulting in a substantial performance increase. We believe that our system is the first ever completely GPU based agent simulation framework. Although GPUs are the focus of our current implementations, our techniques can easily be adapted to other data-parallel architectures. We have benchmarked our framework against contemporary toolkits using two popular ABMs, namely, SugarScape and StupidModel.

Keywords:
GPGPU, Agent Based Modeling, Data Parallel Algorithms, Stochastic Simulations

Introduction

1.1

Agent based modeling is a technique which is becoming increasingly popular for describing complex natural phenomena such as crowd and swarm behavior. An Agent Based Model (ABM) describes a dynamic system by representing it as a collection of communicating, concurrent objects called agents. While each individual object may have a finite internal state, the combined interaction of all entities can create rich emergent behaviors (Bonebeau 2002). These techniques are crucial within many disciplines such as computational biology, population ecology, and epidemiology where they are being used to develop a rigorous mathematical basis for understanding collective behaviors.

1.2

Unlike equation-based methods, there are no formal mathematical tools such as stability analysis and phase plane analysis (Khalil 2002,Callier et al. 2002) that can be used to analyze asymptotic behaviors. ABMs are to be used in a manner similar to experimentation, i.e. brute force simulations have to be carried out to generate dense data-sets for statistical analysis (An 2008). Since macro-level behaviors are population sensitive, in modeling certain systems it may be desirable to have population sizes in the model that reflect reality (Bernaschi et al. 2005). In this paper we aim to address the problem of scalability with novel computational techniques.

1.3

One important class of agent based models is spatial ABMs. Spatial ABMs consist of a regular grid representing some ambient geometrically distributed quantities, a collection of mobile particles and a set of rules for updating both. In this paper, our approach is primarily directed at solving this particular problem as it represents one of the most important categories of

ABMs. Computationally speaking, these are also some of the most challenging systems to deal with as they consist of large numbers of agents with many intricate rules. As such, we feel that this forms a good proving ground for demonstrating the viability of the GPU as a platform for ABM simulations

Toolkits for ABM Simulations and Limitations

1.4

To simulate ABMs, several software frameworks have been developed that enable easy implementation and experimentation. So far, these frameworks include those than execute serially on desktops and in parallel on CPU clusters. Most ABM platforms follow a library based paradigm, where a host programming language is extended through generic procedures to facilitate ABM development. In this standard approach, the library includes a framework (a set of concepts for designing ABMs) along with tools for simulation and visualization. The first of these was Swarm (Burkhart 1994), written in Objective-C. Repast (North et al. 2006) began as a Java implementation of Swarm but diverged significantly. Most recently, MASON (Luke et al. 2005) is being developed as a new platform with emphasis on efficient execution on the JAVA virtual machine. The Logo family of platforms takes a more radical approach (Resnick 1994), defining a completely new programming language specifically for ABMs. The result is a high-level platform that enables users of all skills to rapidly prototype ABMs. The latest incarnation of the Logo family, NetLogo (Wilensky 1999), possesses many sophisticated capabilities including behaviors, agent lists, and graphical interfaces. Currently, NetLogo is one of the most widely used platforms for ABM simulation.

1.5

All the toolkits developed so far are essentially discrete-event simulators(Zeigler et al. 2000). Agent actions are modeled as discrete-events. A scheduler queues all events to be processed for a given time step and executes them one at a time in a serial fashion on the CPU. This is unlike natural systems where agents act in parallel. To eliminate execution bias, schedulers often randomly shuffle agent actions. For large model-sizes, the number of actions that have to be executed per time step can range into the tens of millions. In addition, agent spawning in certain models can result in hundreds for thousands of memory allocation calls. Due to their serial execution nature, the simulation of large sized ABMs with millions of agents is not feasible on these toolkits. Faster CPUs could improve performance, however, the physical limitations of Integrated Circuit (IC) fabrication has caused the performance gain of CPUs to taper off in the recent past (Keyes 2005). Consequently, current generation ABM toolkits will never be able to provide the requisite scalability for certain models.

Parallel Computing for Scalability

1.6

Some researchers have focused on using computer clusters to overcome these performance problems (Massaioli et al. 2005, Scheutz et al. 2006, Da-Hun et al. 2004,Quinn et al. 2003). Computer clusters consist of distributed memory processors communicating through high-speed network connections. Cluster based computing is effective only if the processors spend the majority of the time computing as opposed to communicating (Scheutz et al. 2006). This is not easy with ABMs, since they typically exhibit high levels of interconnectivity. To lower communication costs, researchers have devised various schemes such as spheres of influence (Logan et al. 2001), federations (Lees et al. 2003), strong groups (Som et al. 2000), and event horizons (Scheutz et al. 2006). All these attempt to distribute individual agent objects between different processors based on estimates of probable communications, either a-priori or dynamically. The idea behind these methods is that it is better to group agents which frequently communicate with each other on one processor to minimize inter-processor communication. Unfortunately, predicting the communication patterns within an ABM is at least as difficult as actually simulating the system. As a consequence, costly synchronization is inevitable (Som et al. 2000). To date, cluster based parallel computing has failed to deliver the requisite scalability. There is a point of diminishing return where adding more processors actually reduces performance because of communication overhead.

Scalability Using Agent Compression

1.7

Another solution that has been proposed for model scalability is agent compression (Wendell et al. 2007). The basic idea is to group similar agents into an aggregate agent that behaves as a single agent within the simulation. In static agent compression (Stage 1993), at the beginning of the simulation, each agent is a point in a multi-dimensional space of attributes. Agents are clustered into aggregate agents based on the values of the attributes. An aggregate agent's attributes represent the core of the cluster. An expansion factor is included to represent the number of agents. In dynamic agent compression (Wendell et al. 2007), an agent moves in and out of aggregate agents depending on how well it differentiates from other agents in the aggregate. The system itself contains a compression manager to compress and decompress agents, an agent container to represent aggregate agents, and individual agents. Agent compression works only in cases where model heterogeneity is even. Wendell et al. (Wendell et al. 2007) report that performance gain in computation time for a model containing a million individual agents is about 10 as compared to the uncompressed version.

Paper Overview

1.8

In this paper we present a set of algorithms that enable the simulation of large scale ABMs on data-parallel computer architectures. The focus of our implementation is the Graphics Processing Unit (GPU) since it is cost effective and ubiquitous. The same algorithms can be easily implemented on other data-parallel architectures such as the IBM-Cell and Clearspeed processors. Our methods achieve a speed up factor of over 9,000 relative to an implementation using a traditional CPU based ABM toolkit[1]. This is due to a combination of asymptotic algorithmic improvements and the massive parallelism of the GPU. The issues addressed in this research include manipulation of mobile agents, updating grid based environments, inter-agent communication, agent replication, visualization and methods for statistical measures. Our framework is composed of four essential parts. First, we describe a system for handling environments or static agents. Next, we show how to handle the interactions of spatial or mobile agents which move independent of the background grid structures. A novel method for gathering statistic in parallel during run-time is presented. Finally, we discuss the user interface components and visualization.

1.9

The following sections provide a brief overview of General Purpose Graphics Processing Unit (GPGPU) computing. Next, the data-parallel algorithms that enable ABM simulation on the GPU are presented. The results section present benchmark studies of the implementation of two ABMs, namely, SugarScape and StupidModel. We conclude the paper with discussion of the limitations of our method and directions for future research.

General Purpose Graphics Processing Unit (GPGPU)

2.1

The GPU is a specialized piece of computer hardware designed to efficiently perform computer graphics calculations for the shaded display of three-dimensional objects. Since 2003, GPU vendors such as NVIDIA and ATI introduced programmability to facilitate user defined specialized shading models (Purcell 2004). Researchers are now using this programmability to harness the computational power of the GPU for scientific computations (Strzodka et al. 2005). This technique is in general referred to as GPGPU (Owens et al. 2005).

2.2

The development of GPUs has been driven by the demands of the 3-D gaming industry. There are two factors that make the GPU uniquely suitable for large-scale computation. The first is the sheer computing power which is an order of magnitude greater than the top-of-the-line CPU (Figure 1(a)). The second is the memory bandwidth which is again, an order of magnitude faster than CPU memory (Figure 1(b)), and two orders of magnitude faster than the fastest high-speed interconnects used in cluster computing. Moreover, the GPU is a cost effective platform with the top-of-the-line GeForce 9800GX2 costing under $700.

Stream Programming Model

2.3

GPUs work on the Stream Programming Model (SPM) (Kapasi et al. 2003). The data in SPM is an ordered set of the same data type. For example, the data can be a stream of atomic data types such as integers, floats, etc. or aggregate objects such as triangles, vectors, etc. Stream computation is efficient if the streams have large lengths. Kernels are functions that operate and modify the streams of data (Harris 2006). In SPM, the processing of an element in the stream is independent of other elements. The advantages of this restriction are: first, the input data for the kernel is already known (kernel does not have to wait for data), and second, the independence of the computation between separate elements lends itself very well onto data-parallel hardware.

2.4

The primary challenge in simulating large scale ABMs on GPUs is reformulating ABMs in terms of stream computation. While certain advantages associated with the Object Oriented Programming models in traditional toolkits are lost, we are able to achieve better scalability. Moreover, the stream representation is much more memory efficient.

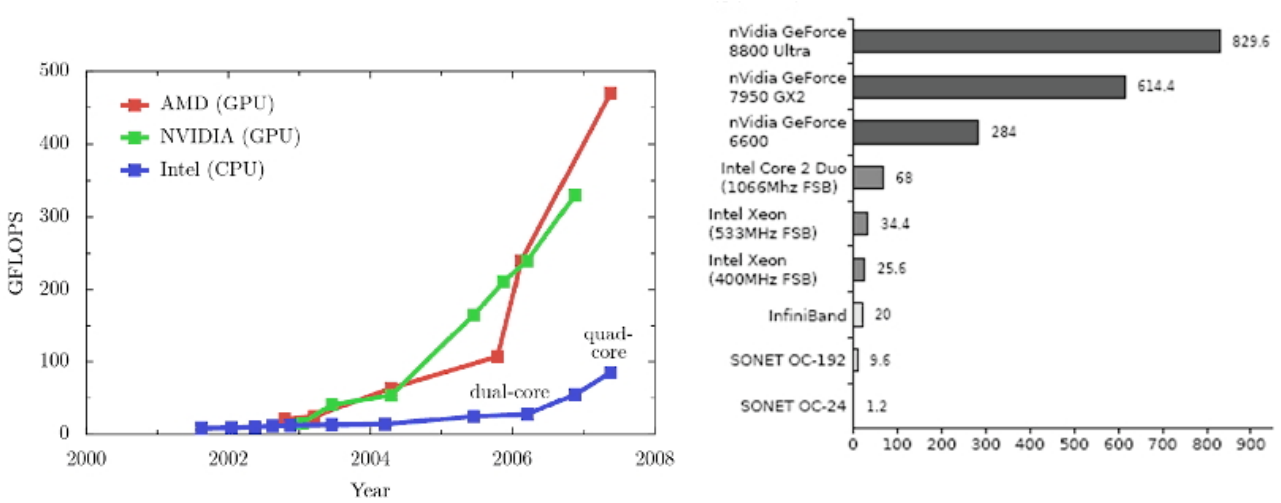

Figure 1. GPU Performance Specs. (a) Computing Power Comparison. (b) Memory Bandwidth Comparison. (Silberstien 2007 ,D'Souza et al. 2007)

Simulating ABMs on the GPU

3.1

In our SPM formulation of ABM simulation, all agent data is stored in large multi-dimensional arrays. Each element of the multi-dimensional array holds the complete state of an agent. State update functions that operate on the agent state data are programmed as kernels. Therefore, different update functions will have different kernels. Kernels operate one at a time on the entire data set. Thousands of individual threads are automatically launched with each thread executing the same kernel on a portion of the agent-state array. The threads are then barrier synchronized at the end of the computation. Depending on the ABM, several kernels may be invoked for the computation of one time step. In the current implementation, we use shader technology to implement our framework. Kernels are implemented as shaders and data is stored as textures (Harris 2006).

Handling Environments

3.2

Environments, or static agents, form the background for common agent simulations. Typically, they are stored as a lattice of values representing ambient quantities such as chemical concentration, heat or geographic structures. The type of values and the structure of the grid are determined by the agent based model. For discrete systems, cellular automata (CA) are often used (Wolfram 2002), while other systems may use convolution functions, such as the Laplacian for diffusion (Bonebeau 1997). These types of environment agents are distinctly well suited for data parallel execution, and have been extensively covered within existing literature (Harris et al. 2002, Harris et al. 2003, Singler 2004,Fialka et al. 2006) so we will discuss them only briefly.

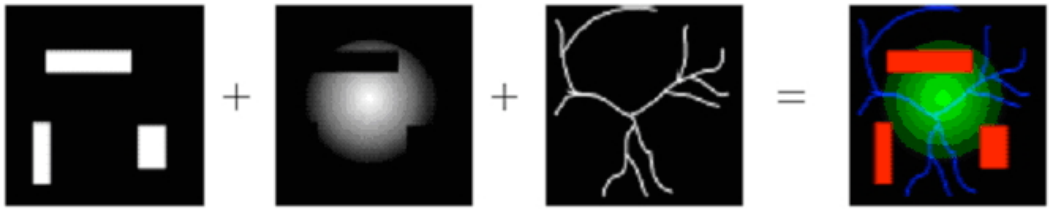

Figure 2. Various Environmental Parameters, Such as Obstacles or Scalar Fields, Are Stored in the Components of the Color Channels

3.3

On the GPU, textures are ideal for representing the kinds of environments used in common agent based models. Numerical quantities are packed into the red, green, blue and alpha channels within each texture element as illustrated in Figure 2. For more complex environments, additional channels can be layered using the extra color attachments within each frame buffer object. The frame buffer object is a construct that handles a stream. Updating the environment is accomplished using the classic ping-pong technique (Godekke 2005). The basic idea behind ping-ponging is to iteratively update some texture by repeatedly passing it between frame buffer objects using a fragment shader. A fragment shader is normally used for color computation of a pixel. In GPGPU, the fragment shader performs general computations. Figure 3 illustrates this process. In the first pass, the contents of buffer A are rendered into buffer B using a fragment shader. In the next pass, B is rendered into A and so on. In each pass, the state of the environment is updated a complete time step. For the purposes of this paper, the texture containing the environment data will be known as the environment texture. Indices into this texture are known as environment

coordinates.

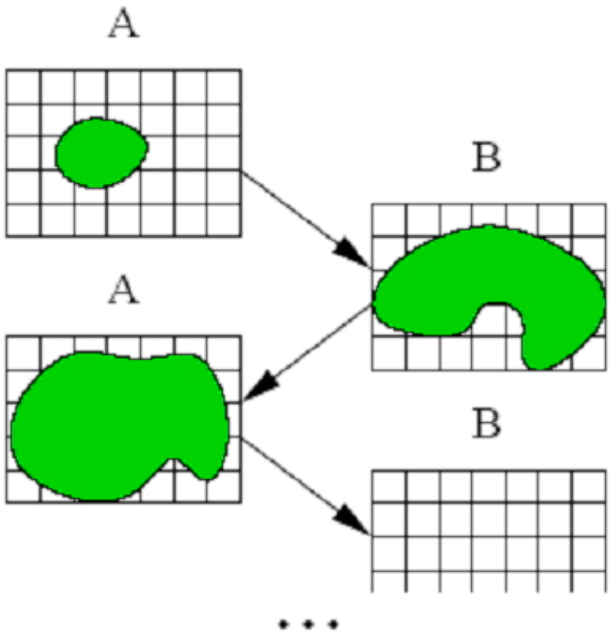

Figure 3. The Ping-Pong Technique Forms the Basis for GPGPU Processing. Images are Iteratively Updated as They Rendered From one Frame Buffer to the Next

3.4

These basic ideas have long been used to directly implement cellular automata on the GPU (Singler 2004). Harris et al. developed the concept of coupled lattice maps as an extension of CAs for the purposes of simulating clouds and natural phenomena (Harris et al. 2002). Convolution operations have already been heavily researched in the context of image processing (Fialka 2006). Broadly speaking there are three methods for evaluating a convolution operation using the GPU. The simplest method is direct convolution, which is optimal for small filter sizes (4x4 or less). For larger convolutions, there are two basic approaches. The first only works for what is known as a separable convolution. Separable convolutions can be expressed as the product of two smaller convolutions. The result is that they can be executed by simply running one convolution, then the other. For non-separable convolutions, the fast Fourier transform is best. Several GPU libraries exist for performing this operation (Moreland et al. 2003).

Handling Mobile Agents

3.5

Environments by themselves are not sufficient for handling mobile agents. Moreover, simulating mobile agents on the GPU is substantially more complicated than fixed position environments. The primary difficulty lies in connecting the agents to the external environment and their neighbors. Broadly, there are three basic tasks which must be performed, ideally entirely within the GPU:

- Store the mobile agent state
- Update the mobile agent state
- Connect the mobile agents to the environment

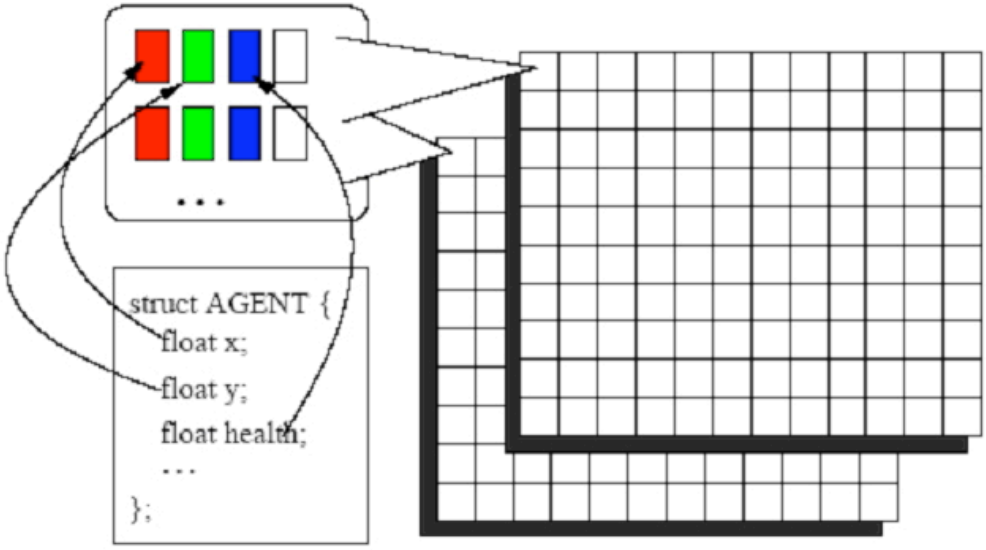

Figure 4. An Encoding of the Agent State Into the Agent State Texture

3.6

To solve the first issue, we adopt the standard GPGPU technique of encoding the agent information into a texture known as the state texture. Under this scheme, each mobile agent's state is packed into the texels' RGBA color values as shown in Figure 4. These color values are then interpreted as state variables, such as location or size. If the agent state cannot be squeezed into 4 floating point values, then it is possible to add additional color buffers. A side effect of this encoding scheme is that each agent is automatically given a unique identifier based on its coordinates within the state texture. This identifier is the agent ID.

3.7

The second issue, updating the mobile agents, is implemented with ping-ponging. Each individual type of behavior, such as movement, reproduction and communication must be handled separately. For randomized behavior, such as movement or searching, it may be necessary to implement a per agent random number generator. Doing so, can be effectively achieved using the techniques presented by Sussman et al. (Sussman et al. 2006). In the following sections, each of these issues will be discussed individually in greater detail. Some of these interactions may require multiple iterations of the ping-pong technique in order to resolve complex behaviors.

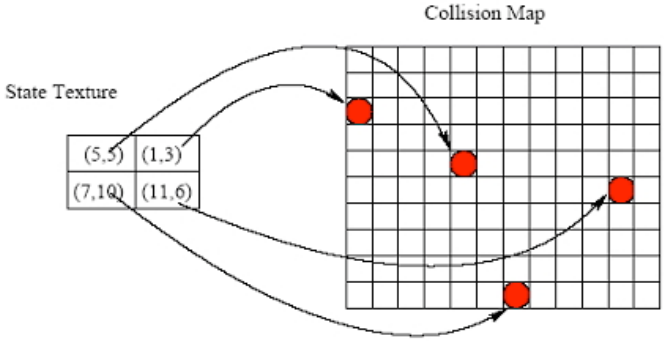

Figure 5. Scattering the Agents

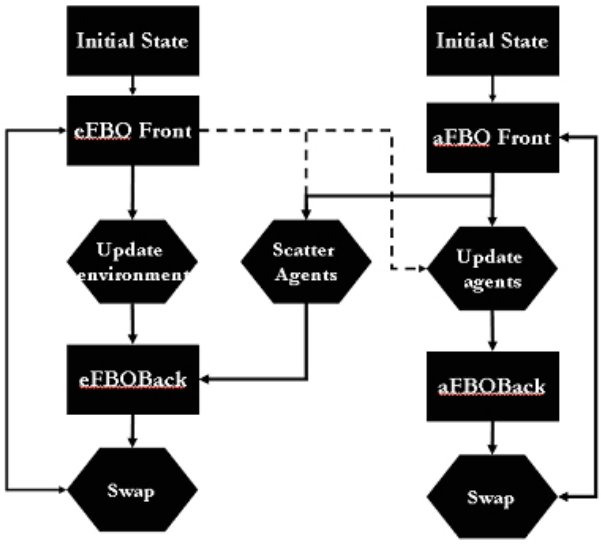

Figure 6. Connecting Mobile Agents to Environment Through Scatter

3.8

The third issue, connecting mobile agents to the environment, is performed differently for each direction. For the mobile agents, reading from the environment can be performed using a double look up into the environment texture using their position. Connecting the environment back to the agents is more difficult, and requires the use of a scatter operation. Scattering on GPUs has been described in several previous works on GPGPU programming (Owens et al. 2005). There are two primary methods for achieving this operation using shaders. The first is to use the render to vertex buffer technique to draw the desired positions for the agents into a buffer, and then use these positions to draw point primitives within environment texture. The second option is to use the vertex processor's read from texture feature to directly locate the agents within the environment. Either way, agent scattering is a costly operation and it is desirable to minimize its use. To do this, the agent positions are typically scattered only once into a separate buffer known as the collision map, as shown in Figure 5. The size of the collision map is the same as the size of the environment texture. Each fragment within the collision map stores the agent ID of the agent at that position, or a null value if it is empty. Using this data, all spatial queries can be translated directly into texture reads from the collision map.

3.9

The overall structure of the system is depicted in Figure 6. The loop shown in the right side of the figure shows the update kernel for the mobile agent state texture. There are four frame buffer objects, two for the environment and two for mobile agents ("aFBO Front" and "aFBO Back" for the mobile agents, and "eFBO Front" and "eFBO" Back for environment). At a given update step, one becomes the input and the other becomes the output. At the end of the update step, they are switched. This is done to avoid race conditions that can occur when simultaneously reading from and writing to the same frame buffer object. On the left part of the figure, the environment state is updated using a symmetric ping-ponging operation. The center of the figure shows how these components are connected through the agent scatter kernel.

Resolving Execution Orders and Collisions

4.1

One obstacle to parallel execution of current agent based models is that many of them specify an exact execution order. Doing so is at odds with the very nature of parallelism, and if taken to the extreme it becomes impossible to achieve any parallel speed up. To overcome this problem, it is best to split the simulation into several phases, each of which updates some subset of the agents in an arbitrary order. Each of these phases is then implemented using a separate kernel. As an example, a simulation might be pipelined into an environment update phase, a movement phase and a replication phase. Each phase would be executed in parallel on all agents, allowing for greater performance. If multiple phases within the pipeline can be executed independently, multiple GPUs could be used to further improve performance, such as updating the environment and moving agents simultaneously.

4.2

However, at times a finer grained level of ordering is absolutely required. An example is the StupidModel ABM, which requires that larger bugs preempt smaller bugs (Railsback et al. 2005).This is done by introducing the concept of an atomic action. Each action is essentially a behavior which is performed by an individual mobile agent within a single time step. Each action is independent, and no more than one agent may take a given action within a time step. An example of an action is moving into a cell where no more than a single agent is allowed at a given grid point.

4.3

Our method resolves the contention over actions between agents using a two-pass scheme.

Let the set of all possible actions be $T = \{ t_0, t_1, \ldots t_{m-1} \}$, and the set of all agents be A = { $a_0, a_1, \ldots a_{m-1}$}. Each agent, $a_i \in A$ is assigned an action, $t_i \in T$, and a corresponding priority $p_i \in$ □. Agents with higher priority always preempt those with lower priority when taking an action. Then, in a single pass over the set of all agents, each action is paired up with at most one agent through the map $ID : T \rightarrow A$ and the corresponding priority map $P : T \rightarrow$ □. These maps are internally represented as large arrays of size m, or on the GPU as textures. To compute the contents of *ID* and *P*, each agent uses an atomic operation to simultaneously set their priority for a given action. For the GPU, *ID* is treated as a color buffer storing the agent *ID*s, and *P* is stored implicitly within the depth buffer. Finally in the second pass, agents which had successfully stored a value to *ID* may then act. The entire process is summarized in Algorithm 1. In Algorithm 1, lines 3 to 15 are executed in parallel. Lines 4 to 9 use the depth buffer functionality.

Algorithm 1 Fine grained prioritized action algorithm

```
1:  P[0..m] ← −∞
2:  ID[0..m] ← NULL
3:  for all a_i in Agents do
4:     begin atomic
5:     if P[t_i] <= p_i then
6:        P[t_i] = p_i
7:        ID[t_i] = a_i
8:     end if
9:     end atomic
10: end for
11: for all a_i in Agents do
12:    if ID[t_i] = a_i then
13:       Perform action t_i
14:    end if
15: end for
```

4.4

As an illustrative example, consider the StupidModel movement problem. Let each movement action be indexed by the world coordinates of the destination. The collision map is interpreted to be the array, *ID*, and the depth buffer is used for *P*. When the vertex processor scatters the bugs, the value of the z-coordinate for the point primitive is set to the size of the bug. As a result, if two bugs move to the same space within the collision map, then the larger bug will be written over the smaller bug. If a bug is preempted, it may still be able to move, so the Algorithm 1 is repeated several times until each bug is in a valid spot. In the worst case, 80 iterations are necessary to resolve all possible conflicts due to the 9 × 9 movement radius. However, we have observed similar results with far smaller numbers of iterations. Using five seems to work successfully in all observed instances. The basic ideas in this technique can be used to compute both the collision map and resolve other priority problems. This phase of the simulation is by far the most expensive, and it is often preferable to try to rework a given model to avoid these issues if at all possible. While our current implementation restricts the maximum number of agents on a grid cell to one, it can Algorithm 1 can be trivially extended to handle multiple agents as long as there is a pre-defined upper bound on the number of agents. We are currently working on other methods to handle unbounded number of agents using dynamic lists. Our initial observation is that this method may not be as efficient as the one described in this paper for simulating large ABMs.

Agent Death and Replication

4.5

Agent death and reproduction are important features in population models. A classic example from the social sciences is Epstein and Axtell's SugarScape model (Epstien et al. 1996), which uses agent reproduction and inheritance to simulate the development of a society. Dynamic populations also play a central role in ecological simulations, such as predator-prey models. Finally, natural selection within evolutionary models is impossible without some level of death and rebirth.

4.6

Compared to reproduction, handling death is relatively simple. Each agent is given a flag to determine if it is alive or dead. Often this may be combined with other state variables through bitwise manipulation. If the agent is determined to have died on a given update, then this flag is turned off and consequently the agent dies. When the agent state texture is updated, all agents marked as dead are simply ignored, and dead agents are not scattered by linking processes. Agent birth is substantially more complicated. The main problem lies in allocating new agents. Allocation is typically viewed as a strictly serial operation, which presents serious barriers to parallel implementation. Furthermore, this allocation must be efficient, since potentially hundreds of thousands of agents could be giving birth at each update. In previous work on GPU particle systems, Kolb et al. (2004) used a CPU allocator to handle the creation of new particles on the GPU. However, this scheme is not suitable for agent systems, since the frequency of allocations is far too great. Even transferring the agent state texture to the CPU

once per update grinds the simulation to a halt.

4.7

Overcoming these difficulties requires a totally different strategy. In order to take advantage of data parallel execution, we relax the restriction that all allocations must be complete. This is not as unreasonable as it first sounds, since even CPU allocators may fail if their internal memory becomes filled or overly fragmented. It will be shown that our parallel allocator behaves similarly, with odds of success directly proportional to the amount of free memory. The allocator is not affected by memory fragmentation, and takes on average O(1) time. Moreover, the probability of successful allocation quickly converges to 1 within a few iterations, which is good enough for a large scale simulation. We begin first with an informal description of the algorithm's essential structure.

Figure 7. A Simplified View of the Agent State Texture. Black Cells are Empty (Dead), and White Cells are Filled (Alive). Cells with a Circle are About to Replicate (Gravid)

4.8

Agent replication is initiated by setting a flag within the agent state texture that signals that the agent is gravid, or about to reproduce. For the purposes of illustrating the reproduction process, Figure 7 represents a simplified version of the state texture. The basic goal of the allocator is to place each newly created agent into one of the empty cells. In other words, it must match each gravid cell to a unique empty cell. Additionally, the dual situation must also occur, in order for an empty cell to become alive, it needs to know about some unique gravid parent which will describe its initial state. One way to think about this problem is that we are searching for an invertible map, from the agent state texture onto itself. Additionally, evaluating this map must be easily done in parallel, otherwise the allocator will perform no better than the sequential counter part. To resolve this issue, we propose using an iterative randomized scheme. Failed mappings from gravid agents to live agents will be treated as failed allocations. For the sake of simplicity, assume a 1 dimensional agent state texture. Let $A = \{ a_0, a_1, \ldots a_{m-1}\}$ be the agent state texture and $f : A \rightarrow A$ an invertible function:

$$f(a_i) = a_{i+r} \quad (1)$$

$$f^{-1}(a_i) = a_{i-r} \quad (2)$$

where r is a random integer in the range $[1, n)$. The arithmetic in expression a_{i+r} is evaluated modulo n, so for example a_{n+3} maps to a_3. Moreover, it is trivial to evaluate both f and its inverse, f^{-1}. Using equation 1, it is now possible to construct the following parallel agent allocation algorithm.

Algorithm 2 Parallel agent allocation

```
1:  for i = 0 to NumPasses do
2:     r = a random integer between 1 and n-1.
3:     for all agent a_i do
4:        if a_i is gravid and a_{i+r} is empty then
5:           mark a_i not gravid.
6:        else if a_i is empty and a_{i-r} is gravid then
7:           mark a_i born.
8:        end if
9:     end for
10: end for
```

4.9

Algorithm 2 is not guaranteed to replicate all agents. However, it may be run multiple times in order to improve the odds of success. Suppose that l is equal to the number of non-empty cells. Then, the probability, $p(k)$, that any agent has successfully replicated after k iterations of Algorithm 2 is given by:

$$p(k) = 1 - (\frac{l}{n})^k \quad (3)$$

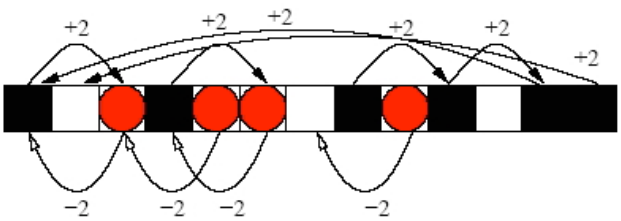

Figure 8. Connecting Gravid Cells to Empty Cells

Figure 9. The Result of one Iteration of the Stochastic Allocation Algorithm

4.10

Since $l/n<1$, $p(k)$ quickly approaches 1 as k increases. The argument for this is similar to that used in the birthday paradox. If half of the agent cells are currently filled, the probability of success reaches over 95% after only 5 iterations. The odds of success can be improved by allocating larger agent state textures, which in turn decrease the likelihood of an invalid map. To illustrate this behavior, consider Figure 8. For this example $r = 2$. Two of the gravid cells are successfully mapped to empty cells. The other two inadvertently collide with already filled cells, and are not able to replicate. At the end of the iteration, the agent state texture looks like Figure 9.

Run-Time Statistics Processing

4.11

Statistical measures are useful to analyze the macro-level behaviors of the system. Statistics are usually a function of the agent or environment state. The agent state vectors at a given time t is $A^t = \{ a^t_0, a^t_1, \ldots a^t_{m-1} \}$. The objective is to calculate some statistical measure S_t given a function:

$$S_t = \sum_{i=0}^{n} f(a_i^t) \qquad (4)$$

4.12

Since each of the terms in the summation can be evaluated independently, they can be done in parallel. In our current implementation, we use the alpha blending feature of graphics hardware to compute the sum in equation 3 by setting the blending function to add and setting the coefficients to one. We then draw a set of point primitives into a single pixel. Rather than draw one point per agent, we sum the particular agent attribute within the state texture. Algorithm 3 describes statistics gathering. The statistics generated using algorithm3 are stored sequentially within a separate buffer on the GPU. For run-time display, we draw a line-strip using the samples as y-positions and index (time) as the x-position using a vertex shader. The function f can be modified to compute any statistic. For example, if the statistic to be computed is the average of the i^{th} element of the state vector a, then f would return the value a_i/n, where n is the total number of agents. Similarly, for finding the standard deviation, we have to use a two pass scheme. The first computes the mean M_i. In the next pass, f returns the value $(a_i - Mi)^2$

Algorithm 3 Sample a single statistic from the agent state

1: Create vertex buffer containing n/k vertices
2: Turn on additive blending
3: Draw vertex buffer:
4: **for** $i = 0$ to n/k **do**
5: Compute sum $S = f(a^t_{ki}) + f(a^t_{ki+1}) + \ldots f(a^t_{k(i+1)})$
6: Draw point with color S
7: **end for**

4.13

Another class of statistics is histograms. Histograms can be used to divide agent populations in bins based on the value of some state variables. For every time step t, a histogram is defined as a set of summations $H = \{ b^t_0, b^t_1, \ldots b^t_{m-1} \}$ where each summation b^t_j is a map f such that:

$$b_j^t = |\{a_i^t : f(a_i^t) = j\}| \qquad (5)$$

4.14

To compute the sum in each b_j, we use the same alpha blending strategy as before, however we vary the position of each vertex rather than the color of the fragment based on the value of a_i. This procedure is given in algorithm 4. Once the histogram is constructed, we may display the contents using a scatter operation. In our implementation, we set the x-coordinate to the index of the histogram bin, and the y-coordinate to the number of elements in the bin.

4.15

Both our histogram computation and statistics gathering take place entirely within the GPU and are sufficient for a wide variety of graphical displays. Multiple dimension histograms can also be created by using buckets indexed by multiple quantities.

Algorithm 4 Computes a histogram over the agent state

1: Create vertex buffer containing n vertices
2: Create a frame buffer object with m pixels
3: Clear frame buffer
4: Turn on additive blending
5: Draw vertex buffer into frame buffer
6: **for** $i = 0$ to n **do**
7: Set position to $f(a_i^t)$ and color to 1.
8: Draw point.
9: **end for**

Visualization

4.16

The enormous performance increase given by the GPU implementation enables unprecedented levels of real-time user interaction. The GPU's capabilities make certain tasks such as 3D visualization much easier than on the CPU. The situation is further helped by the fact that our simulation methods are contained entirely within the GPU. As a result, all that needs to be done is to translate the internal data into user visual feedback using programmable GPU techniques. This visual feedback is especially important for agent based simulations. It is also critical when debugging and prototyping models, since vision provides the most natural method for studying the behavior of an evolving system.

4.17

For strictly 2-Dimensional simulations, one possible approach is to use a combination of masking and blending operations to combine the collision map and environment texture creating a directly viewable image using a single pass. Certain quantities in the environment can be re-colored to enhance the quality of the simulation and make it easier for user to discern the state of the system. In some simulations, the environment can be interpreted as a height map. In this situation, the agents are allowed to move in 3 dimensions, while the environment is effectively a 2 dimensional field. The display of height map data has been well researched in the context of GPU programming. One particularly efficient scheme is Lassao & Hoppe's Geometry ClipMaps (Lassao et al. 2004), which has the advantage that it executes entirely within the GPU. The ClipMap algorithm can be further simplified in this instance, since all necessary data is already within the GPU and therefore the required computation time can be minimized.

4.18

For volumetric models there are several options (Engel 2004). Perhaps the most direct method is to render volumetric slices back to front by sampling the 3 dimensional texture. More recently, ray-casting has become a popular option for GPUs, since it incurs a far lower overdraw penalty (Kruger et al. 2003). Both are capable of executing entirely within the GPU. Rendering the mobile agents in a 3D environment has been well researched in the context of crowd visualization. Extremely efficient rendering is possible using impostor methods (Millan et al. 2006) This strategy works well for both 2D and 3D visualization, since it is implemented directly in the GPU hardware.

Results

4.19

To test the performance of the GPU compared to other computing methods, we implemented several popular models. The models were implemented using C++, OpenGL and GLSL. For benchmarks, the system used was an AMD Athlon64 3500+ with 1 GB RAM and an NVidia GeForce 8800 GTX GPU. For the operating system, we used Ubuntu Linux 7.04. At the time of purchase, the total cost of this system was under $1,400. For comparison, several popular non-GPU platforms were also benchmarked, such as NetLogo (Wilensky 1999), Repast (North et al. 2006) and MASON (Luke et al. 2005).

Sugarscape Benchmark

4.20

Sugarscape is a social agent based model developed by Epstein and Axtel (Epstien et al. 1996). Despite its simplicity, Sugarscape is an extremely relevant model since many more complex social simulations are based on the same underlying principles. Agents in Sugarscape have a number of attributes such as vision, metabolism, age, etc. and are capable of adapting to varying environments. For the purposes of comparison, we implemented rules G, R and M, similar to Repast's demo program. Figure 10 shows the performance advantage of the GPU implementation over existing ABM toolkits. In this benchmark, the environment grid size is set to 100x100. With this performance increase, it is possible to handle previously inconceivable grid sizes and scales. An image of the GPU running a 2560x1024 resolution simulation of over one million agents is shown in figure 11. The average update rate is approximately 56 per second.The exact timings of the GPU's performance with respect to environment grid size are given in figure 12. With the GPU, it is possible to achieve over 16 million concurrent agents on grid sizes of up to 4096x4096. The effects of varying the number of mobile agents are shown in figure 13. Figure 14 shows the comparison between

statistics (population of live agents) generated by the GPU implementation and the MASON implementation (Bigbee et al. 2007). Figure 15 shows the histogram of wealth distribution generated by the GPU implementation and the MASON implementation. The results in figure 15 were generated from a single run. While they are not identical because of different initial conditions and the stochastic nature of time evolution, they show a similar trend.

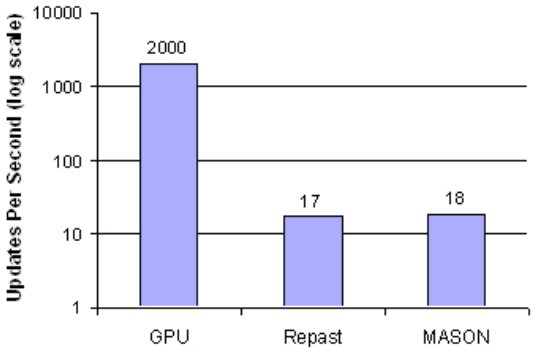

Figure 10. Performance Benchmarks vs Existing ABM Toolkits. The Environment was Set to 128x128. The Agent Population Was 500

Figure 11. Screen Shot of The Simulation. There are 2 Million+ Agents on a 2560x1024 Environment Grid

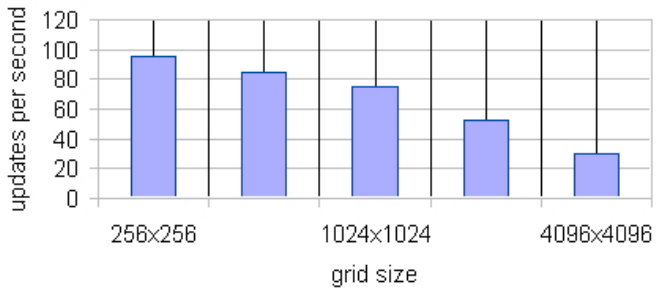

Figure 12. Sugarscape Implementation Scaling with Environment Grid Size. There are 2 Million+ Agents in This Benchmark

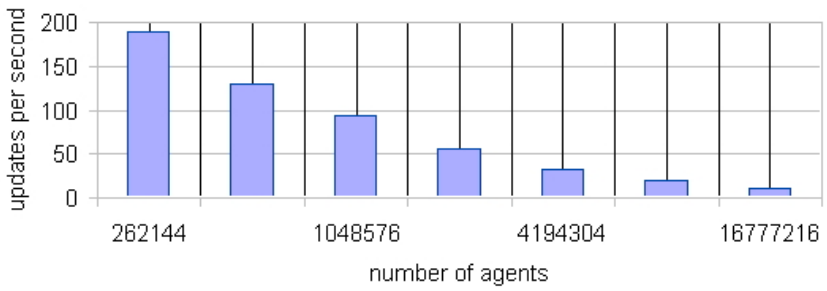

Figure 13. Sugarscape Implementation Scaling with Agent Population. The Grid Size for This Benchmark is 2560x1024

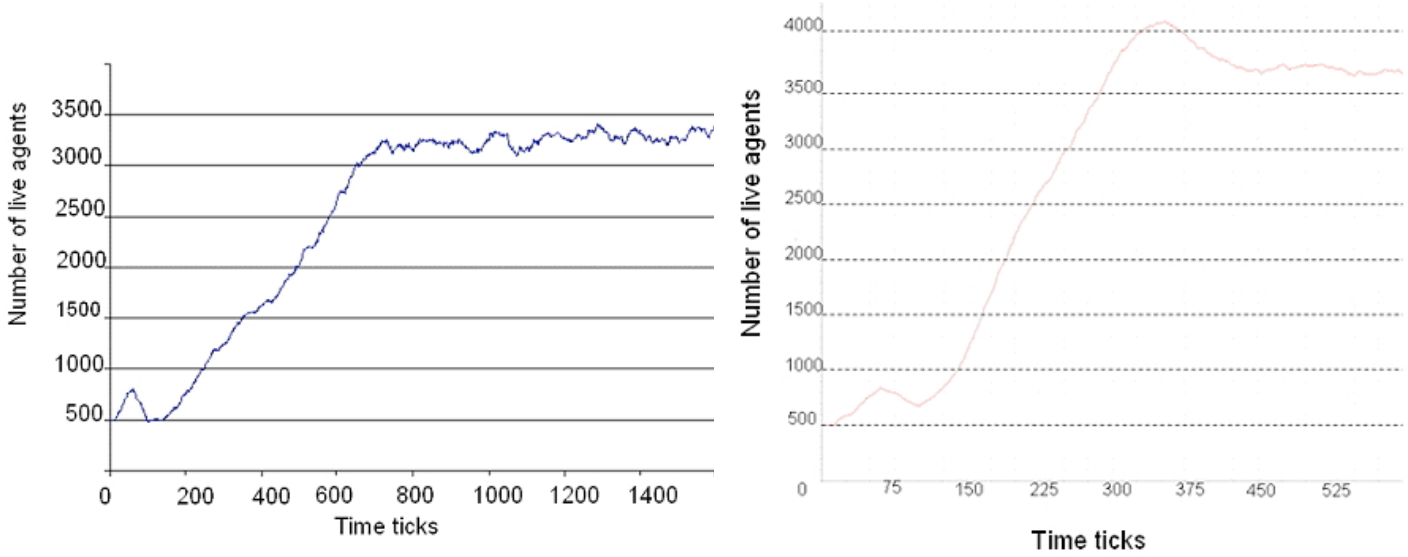

Figure 14. Live Agent Population over Time (a) Output From the GPU implementation, (b) Output From MASON Implementation

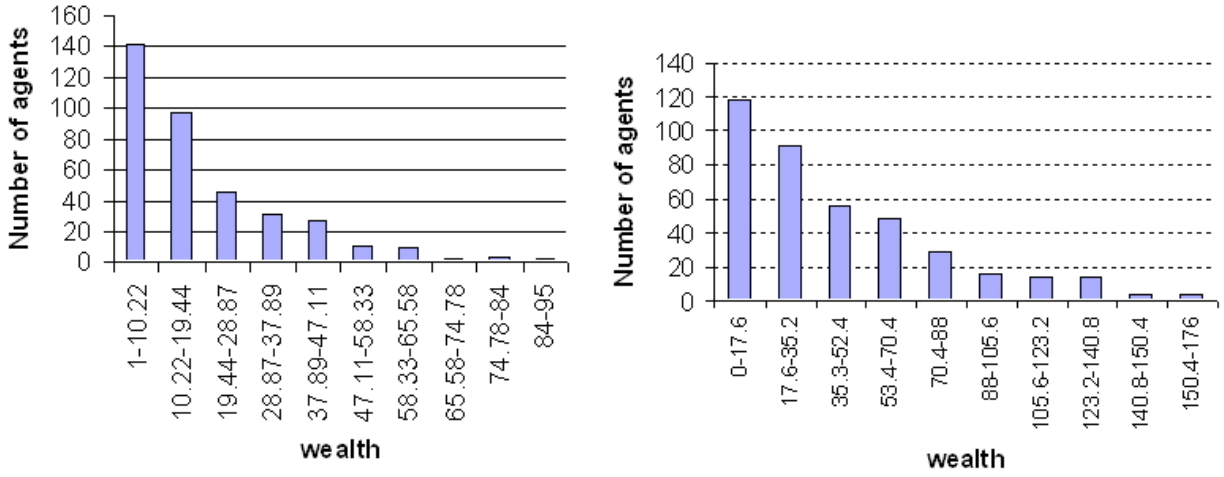

Figure 15. Histograms of Wealth Distribution (a) Output From the GPU implementation, (b) Output From REPAST Implementation

StupidModel Benchmark

4.21

The StupidModel, originally introduced by Railsback et al., is a benchmark for agent based modeling toolkits (Railsback et al. 2005). It functions as a kind of stress test, containing many costly functions such as 9x9 vision filters, rapid replication and multiple agent types. To test the limitations of our system, we implemented StupidModel version 16, the most complex version of the simulation. For comparison, we ran our implementation on a 256x128 grid with about 2000 bugs, which is slightly larger than that used by Railsback et al. (Railsback et al. 2005) in their survey. The relative performance of the GPU compared to other systems is shown in figure 16. Once more, the GPU dramatically out-performs other platforms. Going further, we once again try to push the GPU to the limit of what is possible. As shown in figure 17, the GPU can easily handle population sizes well over 1 million agents, and grid sizes of over 2048x2048. An image of StupidModel running with 1 million agents on a 2560x1024 grid is shown in figure 18. The performance of the GPU StupidModel implementation with respect to agent population size is shown in figure 19.

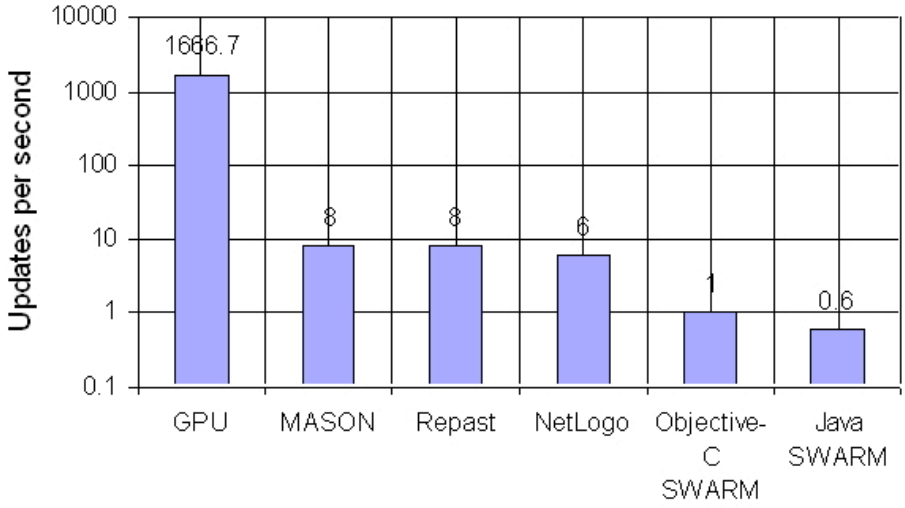

Figure 16. Performance Benchmarks Against Existing ABM Toolkits. The Updates per Second for the GPU is About 1666.7 Frames per Second. MASON, the Fastest Toolkit Runs the Same Model at About 8 Frames per Second

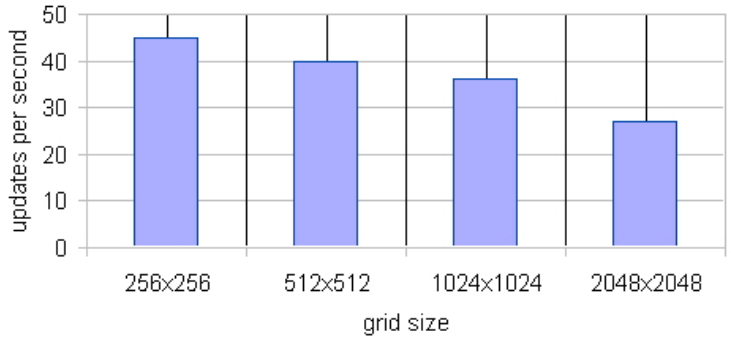

Figure 17. Scaling of the StupidModel with Environment Grid Size. There are 2 Million+ Agent in this Benchmark

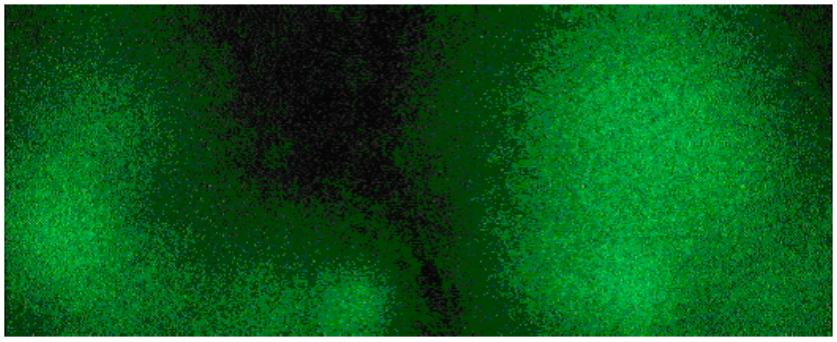

Figure 18. Screenshot of StupidModel Implementation. This Run has 1Million+ Agents on a 2560x1024 Environment Grid

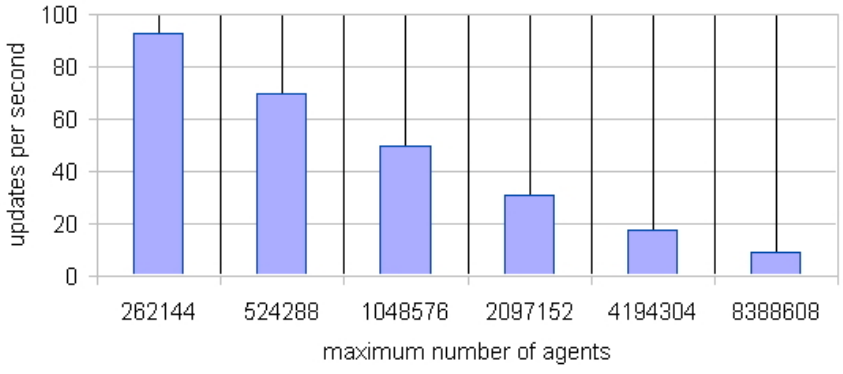

Figure 19. Scaling of the StupidModel Implement with Agent Population Size. This Simulation

Runs on a 2560x1024 Environment Grid

Conclusions

5.1

We have successfully developed and implemented data-parallel algorithms to enable ABM simulation on GPUs. The resulting framework is several orders of magnitude faster than current generation single computer state-of-the art toolkits. Anecdotal evidence from literature review suggests that our method will outperform cluster-based parallel implementations as well. The performance improvements are both due to the algorithms and the GPU computer architecture.

5.2

We are in the process of developing several improvements to the previously mentioned framework. The stochastic memory allocation algorithms require that agent state buffers must be at least twice the size of the maximum population of agents to achieve 90% effectiveness in five iterations. We are working on a deterministic algorithm that has O(nlog(n)) complexity where n is the number of agents. This will substantially reduce the memory requirements of the framework. Statistics gathering relies on specialized graphics routines. However, in the future we plan to use parallel prefix sum, which is more efficient and compatible with general data-parallel programming. Currently, agents in our framework do not have adaptive rules. We are working on algorithms based on neural networks and Bayesian classifiers for adaptive behaviors. Another interesting advancement we are working on is dynamic agent networks to represent social networks.

5.3

While GPUs have phenomenal performance advantages, programming them is counter-intuitive. NVIDIA's CUDA (Silberstien 2007) system provides a substantially better interface, and in the future we plan to develop CUDA based libraries for essential ABM functions to ease deployment of ABM simulations on GPUs. One of the areas we plan to investigate is the development of an agent language much in the manner of NetLogo to enable non-programmers to access the GPU platform. Given the expected growth in the computing power of GPUs, we envision that the GPUs will be the platform of choice for many ABM researchers.

Acknowledgements

This research was partially supported by the National Science Foundation under grant number IIS 0840666. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Notes

[1] Email communication with Russel Standish, developer of EcoLab ABM toolkit

---

References

AN G (2008) Introduction of an Agent-Based Multi-Scale Modular Architecture for Dynamic Knowledge Representation of Acute Inflammation. *Theoretical Biology and Medical Modelling*, in press.

BERNASCHI M, Castiglione F (2005) Computational Features of Agent-Based Models. *International Journal of Computational Methods*, 2(1).

BIGBEE A, Cioffi-Revilla C, Luke S (2007) Replication of Sugarscape Using MASON, Springer Series on *Agent Based Social Systems*, Springer Japan, 3, pp. 183-190. BONEBEAU E (1997) From Classical Models of Morphogenesis to Agent-Based Models of Pattern Formation. *Artificial Life*, 3 (3), pp. 191-211.

BONEBEAU E (2002) Agent-Based Modeling: Methods and Techniques to Simulate Human Systems. *PNAS*, 99(3), pp. 7280-7287.

BURKHART R (1994) The Swarm Multi-Agent Simulation System. OOPSLA '94 Workshop on "The Object Engine".

CALLIER F M, Desoer C A (2002) *Linear System Theory*. Springer-Verlag.

D'SOUZA R M, Lysenko M, Rahmani K (2007) Sugarscape on Steroids: Simulating Over a Million Agents at Interactive Rates. Proceedings of the Agent 2007 Conference, Chicago, IL.

DA-HUN T, Tang F, Lee T, Sarda D, Krishnan A, Goryachev A (2004) *Parallel Computing Platform for Agent-Based Modeling of Multicellular Biological Systems*. LNCS, 3320, pp.5-8.

ENGEL K, Hadwiger M, Kniss J M Lefohn A E, Salama C R, Weiskopf D (2004) Realtime Volume

Graphics, In SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, ACM Press, New York, NY, USA, 2004, p. 29.

EPSTEIN J M, Axtell R L (1996) *Growing Artificial Societies: Social Science from the Bottom Up*, MIT Press.

FIALKA O, Cadík M (2006) FFT and Convolution Performance in Image Filtering on GPU. In Proceedings of the conference on Information Visualization, IEEE Computer Society, Washington, DC, USA, 2006, pp. 609-614.

GODEKKE G (2005) Gpgpu Tutorials - Basic Math. Webpage at http://www.mathematik.uni-dortmund.de/ goeddeke/gpgpu/tutorial.html

HARRIS M J, Coombe G, Scheuermann T, Lastra A (2002) Physically-Based Visual Simulation on Graphics Hardware, In HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 109-118.

HARRIS M J, Baxter W V, Scheuermann T, Lastra A (2003) Simulation of Cloud Dynamics on Graphics Hardware. In HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 92-101.

HARRIS M (2006) Mapping computational Concepts to GPUs, *GPU Gems 2*, Addison-Wesley Publishers, pp. 493-508.

KAPASI U J, Rixner S, Dally W J, Khailany B, Ahn J H, Matson P, Owens J(2003) Programmable Stream Processors. *IEEE Computer*, pp. 54-62.

KEYES R W (2005) Physical Limits of Silicon Transistors and Circuits. *Rep. Prog. Phys*., 68, pp. 2701-2746.

KHALIL H K (2002) *Nonlinear Systems*. Prentice Hall.

KOLB A, Latta L, Rezk-Salaman C (2004) Hardware-Based Simulation and Collision Detection for Large Particle Systems. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware, ACM Press, New York, NY, USA, pp. 123-131.

KRUGER J, Westermann R (2003) Acceleration Techniques for GPU-Based Volume Rendering. In Proceedings of the 14th IEEE Visualization 2003, IEEE Computer Society, Washington, DC, USA, 2003, p. 38.

LASSAO F, Hoppe H (2004)F. Losasso, H. Hoppe, Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. In SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM Press, New York, NY, USA, pp. 769-776.

LEES M, Logan M, Oguara T, Theodoropoulos G (2003) Simulating Agent-Based Systems with HLA: The Case of SIM AGENT Part-II. Proceedings of the 2003 European Simulation Interoperability Workshop, Stockholm, Sweden.

LOGAN B, Theodoropoulos G (2001) The Distributed Simulation of Multi-Agent Systems. *Proceedings of the IEEE*, 89(2), pp.174-186.

LUKE S, Cioffi-Revilla C, Sullivan K, Balan G (2005) MASON: A Multiagent Simulation Environment. *Simulation*, 81(7), pp. 517-527.

MASSAIOLI F,Castiglione F, Bernaschi M (2005) OpenMP Parallelization of Agent-Based Models. *OpenMP*, 31(10), pp.1068-1081.

MILLAN R, Rudomín I (2006) Impostors and Pseudo-Instancing for GPU Crowd Rendering, In Proceedings of the 4th International Conference on Compute Graphics and Interactive Techniques in Australasia and Southeast Asia, ACM Press, New York, NY, USA, 2006, pp. 49-55.

MORELAND K, Angel E (2003) The FFT on a GPU. In Proceedings of the ACMSIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 112-119.

NORTH M J, Collier N T, Vos J R (2006) Experiences in Creating Three Implementations of the Repast Agent Modeling Toolkit. *ACM Transactions on Modeling and Computer Simulations*, 16(1), pp. 1-25.

OWENS J D, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn A E, Purcell T (2005) A Survey of General-Purpose Computation on Graphics Hardware. In: *Eurographics*, State of the Art Reports, pp. 21-51.

PURCELL T (2004) Ray Tracing on a Stream Processor. Ph.D. Dissertation, Stanford University.

QUINN M J, Metoyer R, Hunter-Zaworski K (2003), Parallel Implementation of the Social Forces Model. Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics, pp. 63-74.

RAILSBACK S F, Lytinen S L and Grimm V (2005) StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling Platforms. Webpage at http://condor.depaul.edu/~slytinen/abm/StupidModelFormulation.pdf

RESNICK M (1994) *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworld*. MIT Press, Cambridge, MA, Webpage at http://education.mit.edu/starlogo/

SCHEUTZ M, Schermerhorn P (2006) Adaptive Algorithms for Dynamic Distribution and Parallel Execution of Agent-Based Models. *Journal of Parallel and Distributed Computing*, 66(8), pp.1037-1051.

SILBERSTIEN M (2007) High Performance Computing on GPUs Using CUDA, GPGPU Tutorial at SIGGRAPH07.

SINGLER J (2004) Implementation of Cellular Automata Using a Graphics Processing Unit. In proceedings of ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, CA.

SOM T K, Sargent R G (2000), Model Structure and Load Balancing in Optimistic Parallel Discrete Event Simulation. Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation, pp. 147-154.

STAGE A R, Crookston N L, and Monserud R A (1993) An Aggregation Algorithm for Increasing the Efficiency of Population Models. *Ecological Modelling*, 68(3-4), pp. 257-271.

STRZODKA R, Doggett M, Kolb A (2005) Scientific Computing on Programmable Graphics Hardware. *Simulation Modelling Practice and Theory*, 13(8), pp. 667-681

SUSSMAN M, Crutchfield W, Papakipos M (2006) Pseudorandom Number Generation on the GPU. In Proceedings of ACM EUROGRAPHICS/SIGGRAPH Conference on Graphics Hardware.

WENDELL S, Dibble C (2007) Dynamic Agent Compression, JASSS, 10 (2) 9 http://jasss.soc.surrey.ac.uk/10/2/9.html

WILENSKY U(1999) *Netlogo*. http://ccl.northwestern.edu/netlogo/

WOLFRAM S (2002), *A New Kind of Science*. Wolfram Media Inc., Champaign, Illinois, US, United States.

ZEIGLER B P, Praehofer H, Kim T G(2000). *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. Second edition. Academic Press

Return to Contents of this issue

© Copyright Journal of Artificial Societies and Social Simulation, [2008]