

# SUGARSCAPE ON STEROIDS: SIMULATING OVER A MILLION AGENTS AT INTERACTIVE RATES

R. M. D'SOUZA \*, Dept. of MEEM, Michigan Tech. University  
M. LYSENKO, Dept. of Computer Science, Michigan Tech. University  
K. RAHMANI, Dept. of MEEM, Michigan Tech. University

## ABSTRACT

In this paper we present a new technique for simulating mega-scale Agent-Based Models (agent population sizes exceeding one million) at interactive rates. We achieve this performance by leveraging the computing power of Graphics Processing Units (GPUs). To test our system, we implemented SugarScape, a simple model with many common ABM features. We are able to achieve over 50 updates per second with agent populations exceeding 2 million on an environment with a resolution of 2560x1024 with visualization.

**Keywords:** ABM Simulation toolkit, GPGPU, Parallel Computing

## INTRODUCTION

Due to the emergent nature of Agent-Based Models (ABMs), it is critical that the population sizes in the simulations match the population sizes of the dynamic systems being modeled [1]. In domains such as social modeling, ecology, and biology, the agent population can exceed several million. However, the performance of current agent simulation frameworks is inadequate to handle such large population sizes. Single core CPU performance has stagnated due to physical limitations. This fundamentally limits the performance of all serial frameworks for ABM simulation. Parallel computing frameworks designed to run on computing clusters suffer due to the bandwidth limitations [2]. Issues such as load balancing and synchronization can severely degrade performance [3]. Moreover, visualization is inefficient in distributed systems because of the amount data that must be communicated to the computer node that handles the display.

In this paper we investigate Graphics Processing Units (GPUs) as an alternative platform for ABM simulations. GPUs are powerful parallel processors designed to perform graphics functions such as rigid body transformations and special effects such as lighting. Driven by the 3D gaming industry, GPU computing power has been growing at a rate far exceeding Moore's law [4]. New generations of hardware have opened up more features for programming allowing GPUs to perform tasks other than graphics computations.

GPUs are appealing for large scale ABM simulations for two reasons: one is the sheer number crunching computational power. An NVIDIA GeForce 8800GTX has an average throughput of 512 GFlops [5], while a top-of-the-line Intel Xenon 2.6 Ghz quadcore processor has a theoretical maximum throughput of only 63 GFlops. More importantly, the memory bandwidth of the GeForce 8800GTX GPU is rated at 820.9 Gbps, while the Xenon is rated only at about 68 Gbps. The second major advantage is cost. The GeForce 8800GTX GPU retails at \$600 while the Xenon processor costs over \$1000. Additionally, Xenon processor suffers from cost overheads due to custom mother boards. Graphics cards on the other hand can be mounted on much cheaper computer hardware.

Because of the high degree of parallelism, the computational model of GPUs is very different from traditional programming. Taking advantage of the new data-parallel architecture is non-trivial and requires radically new algorithms. In this paper we investigate ABM simulation using the GPU. To the best of our knowledge, this is the first attempt at using GPUs for ABM simulation. The following sections

will briefly describe the implementation of SugarScape [6], an ABM model that captures most of the behaviors of social sciences ABMs.

## APPROACH

The act of forcing a GPU to perform computational labors beyond computer graphics is known as General Purpose GPU (GPGPU) programming. This technique requires a radical shift from traditional serial programming techniques [7]. While older GPUs only supported a limited set of behaviors, modern graphics cards have rich programmable functionality. By exploiting these capabilities, it is possible to run general numerical computations on specialized GPU hardware. GPUs follow a single instruction multiple data programming model, which does not fit conventional programming methods.

In GPGPU terms, textures (used to store image patterns) act as memory. Color channels in each texel (smallest data element) are used to store values of variables. Updating data values of textures is typically accomplished using shaders. In our implementation, variables for agent states are stored in agent state textures (Figure 1). Variables representing the environment states are stores in environment state textures. At each step, the environment and agent state textures are updated using pixel shaders. To make the updates iteratively continue throughout the simulations, a method called ping-ponging is used [8]. Using framebuffer objects, results of the updates is written back to another texture without communicating with the CPU. Since all computations are done by the GPU, our method is not hampered by the slower bandwidth of the connection (PCI express) between the CPU and GPU. CPU is involved only in processing some user input, and issuing rendering commands to the GPU.

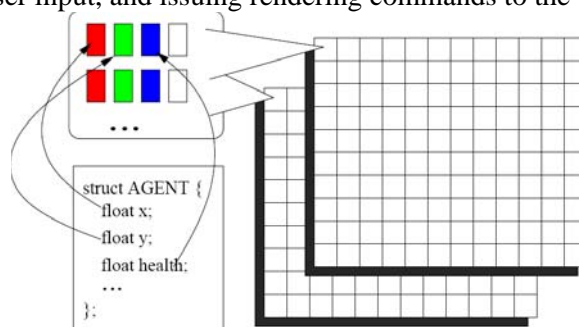


Figure 1 Agent state textures

## SUGARSCAPE ON THE GPU

In this paper, we implement SugarScape to show that GPUs can be used to simulate large scale ABMs efficiently. SugarScape is an extremely relevant model since its has most of the important components of social simulation. Agents in SugarScape have a number of attributes such as vision and metabolism, and are capable of adapting to varying environments. We implemented rules G (sugarscape growback), R (agent replacement), M (agent movement), S(agent mating), P (pollution formation), and D (pollution diffusion). In the following sub-sections, we briefly explain the implementation of each of these rules.

### Sugarscape Grow Back (G)

Sugarscape grow back is by far the simplest rule to implement on the GPU. To do this, we store the current level of sugar inside one of the color channels within the world texture, and the maximum sugar level within another channel. Re-growing the sugar then becomes an image processing operation, where the sugar-level channel is replenished at a given rate until it reaches saturation. Supposing that the

sugar-level is stored in the red channel, while the limit is stored in the green channel, we get the following simple shader written in GLSL shader language:

```
vec4 next_sugar(vec4 prev_sugar, float regrowth)
{
    return vec4(min(prev_sugar.r + regrowth, prev_sugar.g), prev_sugar.gbr);
}
```

## Movement (M)

Updating an agent's position can be performed using a pixel shader applied to the agent state texture with the current world state as input. The basic idea for this operation is similar to that used in various GPU particle system [9], with the added twist that the agents are moving according to the state of the sugar-level in the outside world texture. Assuming that the sugar is stored in the red channel of the texture, and that the agent vision range is given by the constant VISION, the following code updates the agent's position.

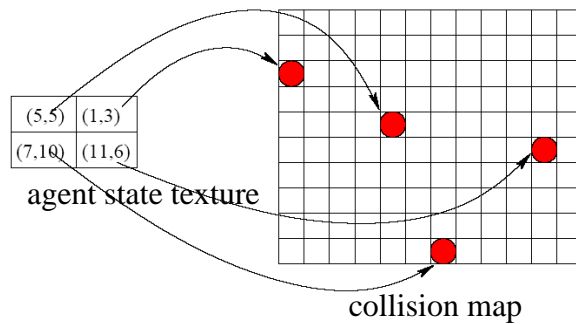
```
vec2 next_position(vec2 prev_position, sampler2DRect world)
{
    vec2 best_position;
    float best_sugar;

    for(int i=-VISION; i<VISION; i++)
    for(int j=-VISION; j<VISION; j++)
    {
        vec2 p = prev_position + vec2(i, j);
        float s = texture2DRect(world, p).r;

        if(s > best_sugar)
        {
            best_position = p;
            best_sugar = s;
        }
    }

    return best_position;
}
```

A somewhat more difficult task is the problem of performing environment agent interactions. To do this, we must use a separate rendering pass to perform a scatter operation [10]. The idea is to write the agents into a separate agent collision map using a separate rendering pass [11]. From this collision map we can locate agents directly based on their spatial position, which makes environment and agent-agent interactions possible.



**Figure 2** Agent scatter

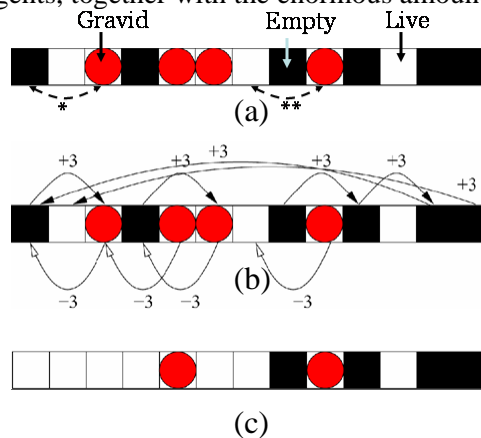
Scattering the agents is typically performed using a vertex shader. A vertex array of indices with the same dimension as the agent state texture is initialized. Using a series of shaders, this array is then drawn into the collision map to determine the positions of each agent. There are two basic methods for scattering using a vertex shader. The most primitive is to use the render-to-vertex-buffer extension, and directly scatter the agents in such a fashion. A much simpler and faster method is available on the latest GeForce8 cards using vertex-shader-read-from-texture. With this feature, the vertex array is allocated and initialized once, and subsequent scatter passes simply read from the agent state texture as they are scattered. Figure 2 illustrates the scatter operation.

## Replacement (R)

Handling agent death is accomplished using a state flag. If set to dead, then the agent is simply not updated, and not scattered during other phases. Using conditional branching, this test can be made extremely efficient. For simple replacement, the position and attributes of the agent can be randomized upon death, rather than killing off the agent. However, this strategy is not compatible with mating. Therefore, we only implement the death aspect of replacement, and allow mating to create a dynamic population.

## Mating and Reproduction (M)

Agent replication is one of the most difficult aspects of any agent based model to implement properly. Making it work on the GPU is one of the key difficulties inherent in realizing efficient models. The basic problem is analogous to memory allocation. Given a new agent, we wish to place it within an empty (dead) agent cell within the state texture. A simple sequential algorithm to perform this replication is to traverse the set of all agents until an open space is found, then place the new agent into the first available memory location. Unfortunately, this is not likely to perform well for any realistic models given both the enormous number of agents, together with the enormous amount of replications per update.



**Figure 3** Stochastic memory allocation process (a) initial state (b) mapping (c) state after 1 iteration

Further improvements on the basic sequential allocation technique are possible, using objects such as freelists, but none of these are suitable for parallel allocation. In order to gain the necessary amount of speed, we use a novel stochastic parallel allocation strategy. The key to this approach is to relax the assumption that all allocations must succeed immediately. Agent replication is initiated by setting a flag within the agent state texture that signals that the agent is *gravid* or about to reproduce (Fig. 3(a)). 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 (Fig. 3(b)). This can be accomplished

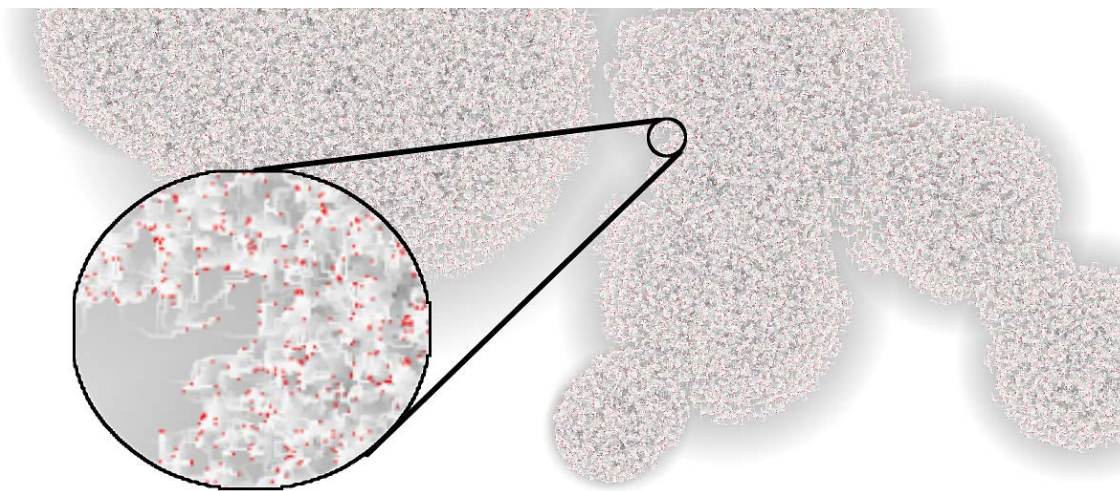
by defining a random invertible map, from the agent state texture onto itself. For the purposes of the GPU, a linear shift is sufficient. A single iteration of this technique with an offset of 3 is shown in Fig. 3. In this iteration, the map “\*” is successful while as the map “\*\*” is unsuccessful (Fig. 3(a), Fig 3(c)). Subsequent iterations with different offsets may solve this. As the number of iterations increases, the probability of success quickly converges to 100%.

### Pollution Formation and Diffusion (P &D)

Implementing pollution once again requires use of the collision map. Assuming that the concentration of the pollution is stored within a separate color channel of the world texture, it is possible to blend the collision map together with the information contained in the agent state texture to add more pollution to the environment. Pollution diffusion is handled using finite differences over the background. Both processes can be carried out simultaneously within the world texture, and evaluated at the same time as the environment growback.

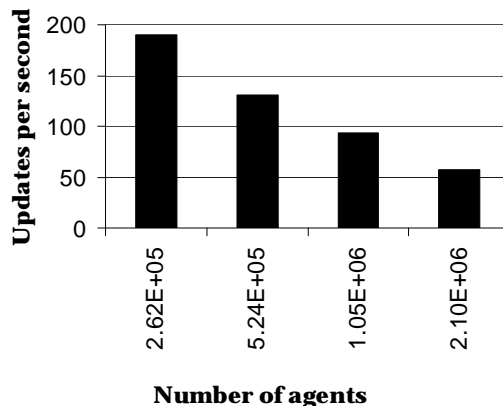
## RESULTS

In our prototype implementation, we have achieved over 50 updates per second with agent population size exceeding two million on an environment with a resolution of 2560x1024 with visualization. Figure 4 shows a screen shot of the visualization. Figure 5 shows the scalability of our system. We are able to freely interact with the simulation as it runs, and dynamically change model parameters without any perceptible degradation in performance.



**Figure 4** Sugarscape screen shot

All simulations were carried out on a single desktop. The computer hardware consisted of AMD Athlon 64 bit CPU with 1GB of main RAM running Ubuntu7.0 operating system. The graphics card is an NVIDIA GeForce 8800 GTX. The total cost of the system is under \$1,400.



**Figure 5 Scalability**

## CONCLUSIONS

We have successfully implemented an ABM simulation on the GPU. Our simulation runs entirely on the GPU and takes full advantage of the ultra high memory bandwidth and computational power. To the best of our knowledge, there are no single computer ABM frameworks that can deliver the performance of our prototype system. We suspect that our prototype will outperform High Performance Computing (HPC) clusters as well. Currently, statistics calculation is not implemented. However, we are working on an algorithm that is based on image histogram generation, a topic well researched in computer graphics [12]. While the simulation performance of GPUs is phenomenal, programming them is completely counterintuitive. In the future we plan to develop libraries for essential ABM functions to ease deployment of ABM simulations on GPUs.

## REFERENCES

- [1] Gilbert, N., Banks, S., 2002, Platforms and Methods for Agent-Based Modeling, *PNAS*, 99(3) :7197–7198.
- [2] Quinn, M. J., Metoyer, R., Hunter-Zaworski, K., 2003, Parallel Implementation of the Social Forces Model, in *Proceedings of the Second International Conference in Pedestrian and Evacuation Dynamics* (August 2003), pp. 63-74.
- [3] 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):1037-1051.
- [4] Pharr, M., Fernando, R., 2006, GPU Gems2: Programming Techniques for High-Performance Graphics and General Purpose Computing, Addison-Wesley Publishing.
- [5] nVidia, (2007), GeForce 8800 Specifications/Performance.
- [6] Epstein, J. M., and Axtell, R. L., 1996, *Growing Artificial Societies: Social Science From the Bottom Up*. MIT Press.
- [7] Göddeke, D., 2005, Gpgpu Tutorials - Basic Math, available at <http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html>
- [8] Pharr, M., Fernando, R., 2004, GPU Gem 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison Wesley.
- [9] Kruger, J., Kipfer, P., Kondratieva, P., Westermann, R., 2005, A Particle System for Interactive Visualization of 3D Flows, *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744-756.
- [10] Sheuermann, T., and Hensley, J., 2007, Efficient Histogram Generation Using Scattering on GPUs, To appear in proceedings of *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (SI3D '07)*.

- [11] Millán, E., and Rudomín, I., (2006), Impostors and Pseudo-instancing for GPU Crowd Rendering. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, 49–55, New York, NY, USA, ACM.
- [12] Fluck, O., Aharon, S., Cremers, D., and Rousson, R., 2006, GPU Histogram Computation. In *SIGGRAPH '06: 18 ACM SIGGRAPH 2006 Research posters*, page 53, New York, NY, USA, ACM Press.

