Agent Based GPU, a Real-time 3D Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the GPU

Paul Richmond
The Department of Computer Science
University of Sheffield, UK
Paul@dcs.shef.ac.uk

Daniela Romano
The Department of Computer Science
University of Sheffield, UK
D.Romano@dcs.shef.ac.uk

ABSTRACT
Traditional Agent Based Modelling (ABM) applications and frameworks lack the close coupling between the simulation behaviour and its visualisation that is required to achieve real time interactive performance with populations above a couple of thousand. The Graphics Processing Unit (GPU) offers an ideal solution to simulate and visualise the behaviour of high population ABM. The parallel nature of processing offers significant and scalable performance increases, with the added benefit of avoiding data transfer between the simulation and rendering stages. In this paper we demonstrate a framework for real-time simulation and visualisation of massive Agent Based modelling on the GPU (ABGPU).

Keywords
Agent Based Modelling (ABM), Graphics Processing Unit (GPU), GPGPU, Real-time Simulation, Visualisation, Flocking, Swarms, Spatial Partitioning, Parallel Algorithms

1. INTRODUCTION
ABM allows complex natural behaviour or various interacting entities to emerge from a set of simple individual rules. Phenomenon such as flocks of birds, schools of fish, and complex biological systems of cells are a good example of how systems with simple goals can demonstrate complex emergent behaviour as a result of communication with other neighbouring agents. Current agent based modelling techniques and frameworks, are mostly aimed at the Central Processing Unit (CPU) with the agents rendered on the GPU only after the agent has performed some serial communication and behaviour. Although this technique is simple and effective for small populations, the weak scalability of using serial processing for large amounts of computation combined with the slow transfer speeds from main memory to the GPU create obvious bottlenecks.

Fortunately both transfer bottlenecks and small population sizes can be avoided by considering the GPU as a simulation platform in addition to its usual role of graphics rendering. The shift from a fixed function GPU pipeline, to a number of programmable stages, has made this possible by allowing GPU's to be utilised as a general high performance parallel stream processor for consumer use. The use of General Purpose computation on GPUs (or GPGPU) is rising due to increasing popularity over recent years, however programmers must have a good understanding of the underlying hardware use it to its best potential and see significant performance increases. The *programmable fragment processor* is by far the most useful stage of the GPU rendering pipeline to GPGPU programmers. Its purpose in the graphics pipeline of processing a quad of fragments (or potential pixels) means that by filling an n × n, 2D orthogonal viewpoint the processor can be invoked to compute a parallel operation on each pixel in the n × n quad. By reading data from a number of bound read only textures the processor may then perform computations that can be fed back into texture memory for use in further computations. The process of programming general computation in the manor described above can be extremely complex and notoriously difficult to debug. The agent specification presented in this paper plays an important role in hiding the complexity of the underlying algorithms. This allows the user to focus on the individual agent's behaviour without explicit knowledge of the graphics pipeline itself.

Whilst recent work [1] has demonstrated how the GPU can lead to a significant performance increase for ABM in 2D, this paper describes a framework (ABGPU) which not only improves upon the performance of existing 3D GPU ABM implementations, but also tackles usability by providing an API similar to that of CPU libraries which allows agent specification and scripting. More specifically this paper describes a process in which agents can be mapped to the GPU, and a parallel communication algorithm, which allows agents to communicate across spatial partitions without time consuming read back operation to the CPU. The result of this is an architecture allowing massive and scalable models which by avoiding any slow transfer bottlenecks are able to

demonstrate incredibly high performance. Within this paper ABGPU is demonstrated through a Boids implementation which uses the libraries feedback routines to implement a Level Of Detail (LOD) rendering system of fish, allowing advanced visualisation at interactive rates.

2. RELATED WORK

ABM dates back to Reynolds [2] who first rendered a flock of 80 individuals (described as Boids) using offline calculations taking up to 30 seconds per second of footage rendered. Performance since then has increased substantially with the most recent ABM implementations boasting upwards of two million interacting agents. This section concentrates on describing recent ABM work which either has a focus on high performance GPU implementations or spatial partitioning techniques for parallel architectures.

In the most simplistic case of processing agent communication, each agent potentially communicates with every other agent in the system. Whilst this guarantees that any limited range communication between agents takes place, the $O(n^2)$ complexity results in large amounts of wasted computation that escalates non-linearly as the interaction radius is reduced. Despite this, the simplicity of implementing the *all pairs* technique using serial iterations through the population (for each agent) has made it extremely popular for CPU based libraries and toolkits [3, 4, 5]. The advantages of mapping an *all pairs* implementation to the GPU were first demonstrated by Nyland et al [6] who performed an N-body force simulation using an $N \times N$ (where N is the population size) communication space and parallel reduction for force averaging. The idea has been adopted recently by Drone [7] who applied it to the Boids flocking model by using automatically generated mip-maps to calculate average velocity and positions used to update and render a few thousand Boids in real time. Drone also describes a novel environment interaction technique that involves rendering geometry into a volume texture and using the geometry shader to calculate planer normals for each volume voxel.

Whilst *all pairs* implementations are suitable for Agent Based (AB) systems in the orders of hundreds to low thousands, the functionality to achieve higher populations is dependant on more scaleable algorithms. Spatial partitioning offers the most significant performance gain by reducing unnecessary communication between distant agents. Such a technique maps exceptionally well to parallel distributed systems such as processing grids networked by high speed Ethernet. Quin et al [8] demonstrated this technique using a SWARM cluster to update 10,000 evacuating pedestrians 50 times per second. The implementation consisted of partitions being split across 10 processors which used a message passing interface (MPI) library to handle communication of pedestrians across boundaries. The same technique is applied in the more recent FLAME [9] toolkit, which using formal agent specification techniques is primarily aimed at fast parallel simulation of large biological systems.

Erra et al. [10] describes an implementation of GPU ABM which incorporates spatial partitioning by using a sorting algorithm to assign individuals to spatial cells. Although the GPU is used in this implementation to perform agent updates, the sorting algorithm and nearest neighbour calculations are performed on the CPU. Erra et al. [10] highlights the cost of this phase but introduces a novel *'scattering matrix'* (not to be confused with the scatter matrix described later), which indicates the affinity of the flock in 27 cells. When the flock's movement is uniform the scatter matrix contains minimal values, however in the presence of a predicator or global obstacle the matrix values increase indicating a large movement of agents across spatial cells. Using the *scattering matrix* technique it is suggested that up to 20% of frames can avoid performing the CPU sorting, allowing up to 13000 agents to be modelled at up to 20 Frames per second (Fps), a slight improvement on previous work [11] of 8000 (at 20 Fps) using a similar technique.

Similarly to ABM simulation on the GPU, recent work by Reynolds [12] demonstrates how the PS3's Cell Processor can be used to efficiently render schools of fish. The PS3 architecture is somewhat different to that of traditional GPU or CPU, and although a NVIDIA RSX card is available for graphics processing the PS3 contains an additional IBM Cell Microprocessor capable of scheduling eight parallel Synergistic Processing Units (SPU's) with a high bandwidth (25.6 GBytes/sec) connection to the single cells Power Processing Unit (PPU). Reynolds [12] uses the architecture to batch a number of spatial buckets of fixed array size to the SPU's, which in turn calculate the nearest N neighbours for each individual in the bucket by considering neighbouring buckets through communication across the PS3's fast memory cache. Although architecturally different to the GPU, the PS3 implementation which is more similar to that of distributed parallel systems [8, 9] is able to render up to 10,000 low resolution fish (36 polygons) in 3D space (15,000 with a 2D crowd) at 60 Fps. When combined with more advanced underwater lighting effects and dynamic LOD up to 5000 fish of up to 400 polygons can be rendered at 30 Fps.

In the only example of agent based modelling entirely on the GPU, D'Souza et al. [1] describes the implementation of an ABM framework based on a 2D environment partitioned into a regular lattice small enough to contain single agents. Agent's positions are then scattered using a vertex shader into a separate buffer with collisions (of multiple agents per cell in the collision map) handled by multi pass priority system with efficiency dependant on the cell movement size of the agents themselves. In addition to this D'Souza describes a novel solution to agent birth and death through an iterative randomised scheme, which although singularly does not guarantee successful reproduction, converges quickly to a 95% likelihood after only five iterations. Whilst the technique described is successful in easily demonstrating real time performance of over one million agents, the framework is restricted to a 2D lattice which makes the technique unsuitable for 3D simulation of continuous valued agents, such as those presented in this paper.

3. AGENT MAPPING TO THE GPU

Most important to providing a library for ABM on the GPU, is hiding the underlying graphical concepts, the most obvious of these being texture data storage. ABGPU uses an agent specification as a means of generating a mapping function (F) to allow agent scripts to directly access memory variables without explicit knowledge of the underlying storage mechanisms. Within the mapping process agent variables are translated by F into '2t' textures, with dimensions of $\sqrt{N} \times \sqrt{N}$, where N is the population size, containing up to four variables (in each of the red, blue, green and alpha channels respectfully). Similarly GPU particle system implementations [13] agents can then be processed in parallel by invoking a fragment program to perform a read and write to texture space on the double buffered agent data textures, with the agent script being used to generate the output data in between. Multiple Render Targets (MRTs) allow the stacked textures to be updated in a single pass when the OpenGL Frame-Buffer Object (FBO) extension is used. One significant point to consider is that FBO's support multiple render targets only when the multiple textures are of the same internal texture format. For this reason ABGPU allows only 32 bit float values as agent memory, this limitation is expected to be avoided in future releases, which will use more direct access methods to GPU memory. Figure 1 demonstrates the mapping process of an agent specification into agent space at position 'i, j'. A simple state machine represents synchronisations after the communication algorithm, agent update phase and rendering.

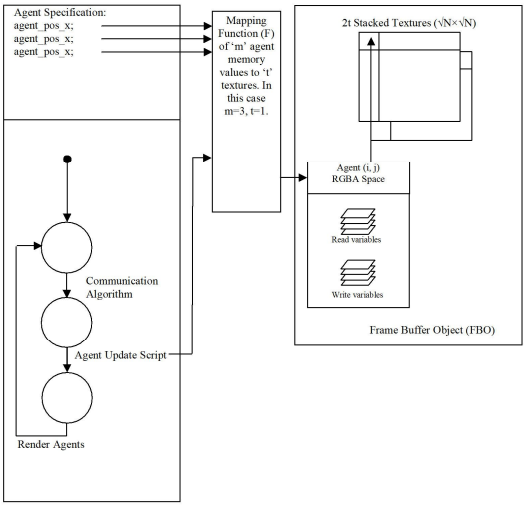

Figure 1 – The mapping of an agent specification into agent space at position 'i, j'.

4. AGENT COMMUNICATION

In order to allow the communication between agents to take place, a partitioning scheme is used, which splits the environment space into portions equal to that of the communication radius of the agents. This technique differs to that of D'Souza et al. [1] in that the significantly smaller partition space (with larger physical partitions) is used only to hold the indices of the agents occupying the partition rather than the actual agent data itself. For agents to reference neighbouring partitions it therefore requires the creation of a dynamic partition structure, each with an unlimited number of agents. This is achieved through first generating a positional partition identifier for each agent, along with a pointer to the agents position in 2D (agent) texture space. This identifier is then used to sort the pointers and reorder the agent data in order to increase the cache hit rate during later stages. A simulated scattering technique is then used to write to the boundary partition matrix, which is used during the update stage to determine the location of neighbouring agents.

Bitonic GPU sorting has received a wealth of interest in recent years leaving a good choice of suitable sorting algorithms. GPUSort [14] improves upon the performance of Purcels [15] original implementation as well as the performance of Kipfers [16] more cache efficient implementation and has hence been used as the basis for sorting within ABGPU. The improved bitonic network (i.e. an improved parallel comparison network for each rendering pass) offers two significant advantages despite the same $O(n\log_n)$ overall complexity of alternative methods. The first of these is that the sorting network enhances the GPU cache memory hit rate by increasing the number of texture lookups in close proximity. The second improvement is the

use of GPU blending functionality to perform the sorting steps. These processes balance the GPU much more than a pure fragment processor implementation. In its original state the GPUSort library is unsuitable for sorting non unique identifiers, consequently it has been modified slightly to allow this.

With the agents sorted by spatial partition it is easy to see how agents are able to perform a linear search between boundaries in order to consider neighbouring influences. More difficult is consideration of agents in neighbouring boundaries; assuming that the start and end position of agents within the sorted list can be calculated for each spatial partition the agents can perform a serial scan across agents with the same partition identifier for its own partition and the 26 neighbouring partitions. The method used for dynamically generating this partition boundary matrix is adopted from rigid body particles physics [17, 18] and requires scattering the first agent for each partition into a 3D partition matrix. Within ABGPU this is achieved by rendering N points (N agent population size) each with a texture coordinate between 0.5 and $\sqrt{N}+0.5$ in the x and y dimension. Vertex texture fetching then allows these coordinates to be used to lookup the partition values which are compared to the previous agents value to find the start of a spatial boundary. Rather than each agent in the update stage performing a linear search to find the end of each spatial boundary this is performed in the same vertex scatter program which scatters both the start and end index (position in agent space) of agents, using multi texture semantics, which can the be used to iterate between the two agent space positions during the update phase. To ensure that it is possible to scatter to each position within the partition boundary matrix a view port must be used which enables an output size equivalent to that of the partition boundary matrix itself. Using a traditional graphics API this gives the option of either rendering directly to a 3D texture or to render to a stack of 2D textures. As portability and performance is a key issue in ABGPU the 2D texture method is preferable as 3D textures have far less hardware support with no significant performance advantage. For an interaction radius '`i`' in an environment space '`e`' between `0` and `1` all partition boundary values can be stored with a 2D texture of size `ceil(`$\sqrt{(1/i^3)}$`)`with an over 4 million total partitions within; allowed by the maximum 2D texture size of 2048. Figure 2 demonstrates the complete steps of the algorithm including the agent update. Texture space inputs and outputs are indicated between each stage.

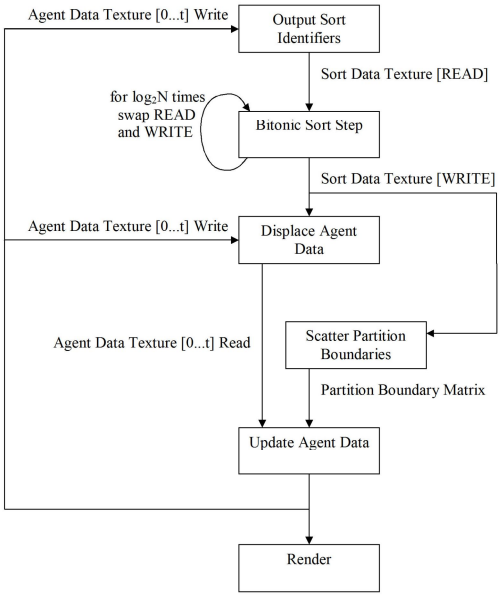

Figure 2 – Render passes and data bindings for a single update step.

5. USING ABGPU

ABGPU uses a number of C/C++ classes as well as an agent update script to create an AB simulation. The agent update script has a C like syntax and can be compiled with a C header file which contains place holders for key communication functions. Figure 3 demonstrates the simplicity of a simple agent based script in which an agent moves towards its perceived centre of the population. The structure of the update script is that an agents `main` function must be provided, which accepts an `agent` structure and a `GLOBALS` structure as arguments returning an `agent` structure. Upon setting the agent update script through an instance of an `Agent Population` class, any references to agent variables under go the same mapping process as any upload or download of agent data. The placeholders `FOR_EACH_AGENT_A` and `END_FOR EACH` are automatically replaced with the appropriate nested loop for retrieving each agent's data within the specified communication radius. The supporting C++ classes which are required to generate an AB simulation are as follows. Additionally a `SceneVisualiser` header is included which sets up a basic 3D viewpoint with mouse controls and basic point rendering of agents. This of course can be modified as in the case of the Boids fish example which uses more complex agent orientation and swimming animations.

```
struct agent{
	float agent_pos_x;
	float agent_pos_y;
	float agent_pos_z;
};

struct GLOBALS{
	float STEER_SCALE;
};

agent agentMain(agent IN, globals GLOBALS)
{
	float global_centre_x = 0.0f;
	float global_centre_y = 0.0f;
	float global_centre_z = 0.0f;
	float count = 0;

	FOR_EACH_AGENT_A
	{
		global_position_x += a.agent_pos_x;
		global_position_y += a.agent_pos_y;
		global_position_z += a.agent_pos_z;
		count += 1;
	}
	END_FOR_EACH

	//calculate average
	if (count >0){
		global_position_x /= count;
		global_position_y /= count;
		global_position_z /= count;
	}


	IN.agent_pos_x += global_position_x * GLOBALS.STEER_SCALE;
	IN.agent_pos_y += global_position_y * GLOBALS.STEER_SCALE;
	IN.agent_pos_z += global_position_z * GLOBALS.STEER_SCALE;


	return IN;

}
```

Figure 3 – A simple agent script using ABGPU scripting.

`AgentSpecification` – An agent specification is required to determine the GPU texture space required for agent data storage. Agent specifications currently support up to 32 internal agent values on a DirectX 10 series card with 8 MRTs, 16 values through 4 MRTs on most DirectX 9 cards.

`Agent` – An agent requires an agent specification and can be used to get and set values using the get and set agent variable methods respectfully.

`AgentPopulation` – An instance of the Agent Population class is responsible for the generation of GPU shader code by translating the agent update script into compilable cg code and by dynamically creating code for the sorting and displacement stages. Agent data can be set and retrieved by passing an array of agents to the `AgentPopulations` upload and download data methods and the simulation can be stepped by using the `step` or `StepN` (`N` being the number of steps) methods. Global variables are controlled by the agent population and can be got and set between simulation steps. Included within the global variables are a number of application variables which are accessible by default. These include the environment size, the agent data texture size and the number of spatial partitions. An additional distance sort method is also available allowing the agents to be distance sorted by any agent variable.

`AgentFeedback` – Agent feedback uses a number of parallel reduction functions to provide real time feedback of the agent population without reading back the entire agent population data. An instance of the `AgentFeedback` class can be used in conjecture with a number of `FeebackVariable` instances. Figure 4 demonstrates the C++ code for providing two feedback values of type FEEDBACK_MAX and FEEDBACK_SUM respectively.

```
FeedbackVariable variables[2];

variables[0].feedbackType = FEEDBACK_MAX;
variables[0].feedbackVariable = "agent_pos_x";

variables[1].feedbackType = FEEDBACK_SUM;
variables[1].feedbackVariable = "agent_pos_x";

AgentFeedback feedback = new AgentFeedback(2,
variables, agent_population);

float2 feedbackData;
feedback.getFeedback(feedbackData);
```

Figure 4 – An example of Agent Feedback using two `FEEDBACK_MAX` and a single `FEEDBACK_SUM` `FeedbackVariable`.

Additionally there is also a FEEDBACK_MIN and FEEDBACK_COUNTN feedback type. The COUNTN can be used to count all occurrences of the value N across the agent population.

6. BOIDS MODEL

The Boids model used to demonstrate the functionality of ABGPU is adapted from Reynolds original model [2] with the introduction of a *goal rule*, which is implemented using global variables. The agent specification consists of seven variables and x, y and z component for both position and velocity and a LOD variable used to hold the agents current detail level. The global variables controlling the goal point are potentially set after each simulation step by considering random variables. If this variable is below some threshold the goal point is moved to a new position within the environment bounds. Global values are also used to control agent interaction though the setting of a number of weights which control each of the Boids rules. The ultimate behaviour of the Boids is determined though a steer vector which is the summed weight of each rule which is also bound to a maximum threshold to preserve a maximum speed within the simulation. A simple menu system is used to control the rule weights, which in turn affect the behaviour in real time.

Rendering of the fish is achieved through two methods depicted in Figures 6 and 9. The first (Figure 6) uses the same technique as the inbuilt primitive point rendering, where instead of a point a low polygon count (66 faces) model for each fish agent is rendered into a single display list. As with the vertex scatter multi texture coordinates are used

to provide both visual texture map lookup values and lookup cords for the fish data in agent space. For higher resolution agent representations (Figure 10), rendering the entire population into a single display list is not an option as the memory requirements soon escalate. Additionally, to use a dynamic LOD system the number of each fish at each detail level changes continuously dependant on the user viewpoint and fish behaviour. In order to provide a high detail simulation the Boids update script uses global values for the eyes x, y and z position to calculate a detail level dependant on the agent position. The three decreasing detail levels of either 0, 1 or 2 (Figure 5) are then counted using three FEEDBACK_COUNTN feedback variables. With the total number of each detail level know the agents are then sorted by the LOD value so the CPU can draw the correct number of LOD models in order. Display lists are again used to improve performance, however a display list is used for each model at each detail level reducing the CPU to GPU communication to a single display list call for each agent in the system.

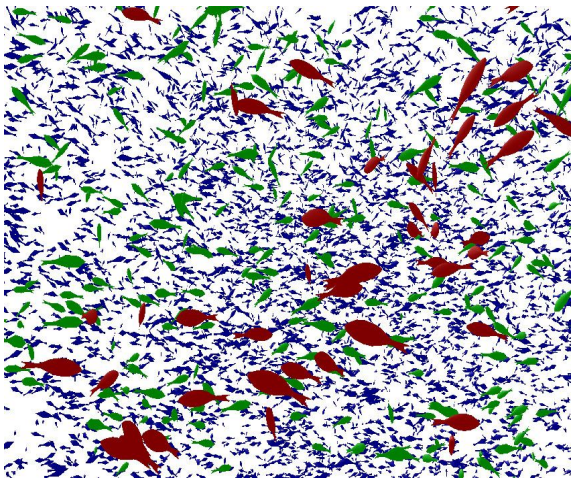

Figure 5 - A population of 16384 agents with rendered with the LOD system. Red, Green and Blue colours represent three detail levels 0, 1 and 2 respectively.

Animation of the fish is achieved through the use of a vertex shader. This uses texture indices passed from the agent population to allow the agent data to be used to first offset the agents to the correct position, and then orientate vertices and normals about two dimensions with a restriction on positive and negative vertical inclination. Finally the same vertex shader is used to provide some animation of the fish bodies with the body being displaced along its length through a sine wave giving the effect of swimming through the water.

7. RESULTS

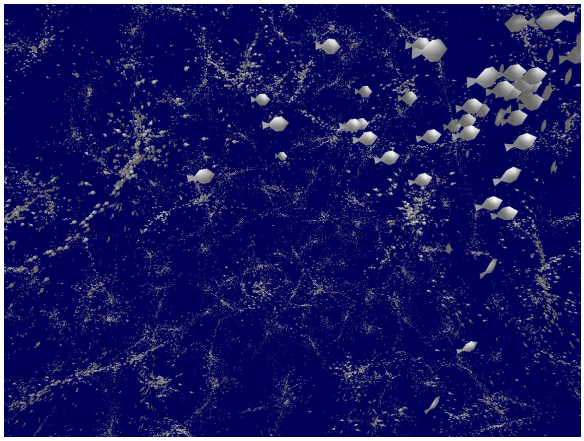

Figure 6 – 65,536 Interacting fish Agents at 30 Fps

The results obtained are based on a single PC with an AMD Athlon 2.51Ghz Dual Core Processor with 3GB of RAM and a GeForce 8800 GT. As the system performance is highly dependant on the complexity of the behaviour, the number of agent variables and the agent communication radius, comparisons with previous work are limited to that which most closely resembles the work in this paper. Despite the difference in underlying hardware both Reynolds [12] and Erra at al. [10] describe systems of agents in continuous valued 3D space and are consequently the most suitable for any direct performance comparisons.

Figure 7 shows the recorded performance of the Boids implementation with the varying colours representing variable communication radii in an environment clamped between the range of 0 and 1. The Fps readings were obtained after the simulation had run for some time allowing any local groups to form and to avoid influence of the Boids initial random positions and velocities. The performance results are capped at 200 Fps and not all values of communication radii attempt to simulate the larger population sizes. With an interaction radius of between 0.03125 and 0.0078125 it is possible to simulate 65536 Boid agents at 60 Fps. Erra et al. [10] simulated the same number of agents at less than 5 Fps. The system presented in this paper is, instead, able to render up to a million agents at 5Fps without visualisation, although it has to be noted that the simulation can only sustain this rate when the weight of the single goal rule is significantly small enough to allow multiple local groups to form. It must also be noted that Erra et al. [10] simulated interaction with 5 static scene objects and one dynamic one, which currently is not done in this simulation. Such behaviour will be incorporated at a later date through the use of global variables.

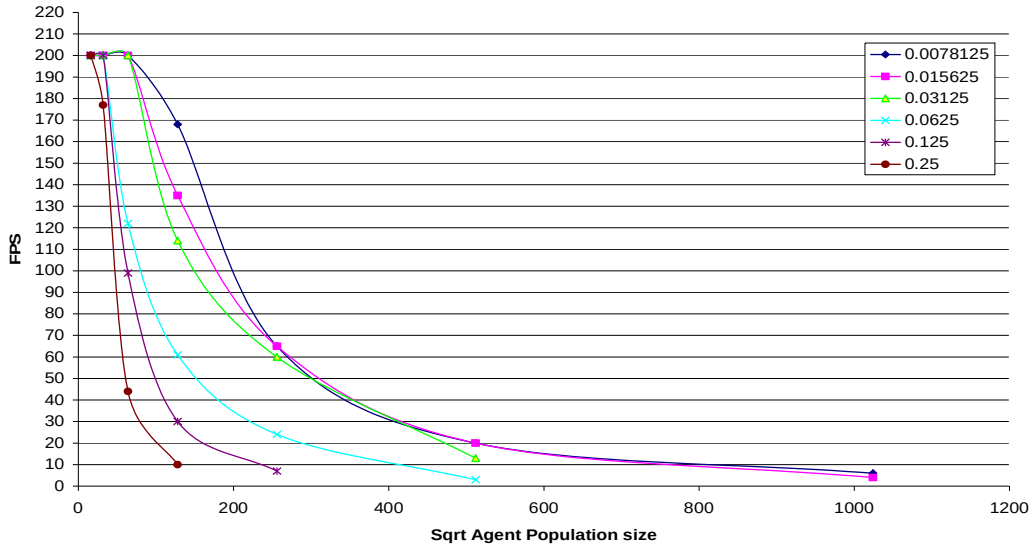

Figure 7 – Recorded performance with x axis representing the square of the population size and the y axis representing Fps recorded over multiple frames.

It is surprising how little effect rendering the population using both the point primitives and low polygon count models has on the performance of the system. In most cases a maximum of 5-10 % drop in frame rate is observed however with populations over 16384 the performance of the low polygon model rendering is reduced further. In fact with a communication radius of 0.03125, 65536 agents can be rendered as simple fish sustaining 30 Fps (Figure 6) and as point sprites at 50 Fps. In contrast this is substantially more than the reported performance of previous work, the most impressive being Reynolds [12] reporting 10000 agents at 60 Fps. When rendering using higher resolution agents with LOD system, the performance is obviously reduced due to the additional feedback and sorting stages performed before rendering (this effect is demonstrated by figure 8). Despite this, 16384 agents with a maximum detail level of 1500 polygons can be rendered at over 30 Fps. The majority of the performance slowdown is attributed to the secondary sort rather than the LOD feedback which makes little overall performance difference.

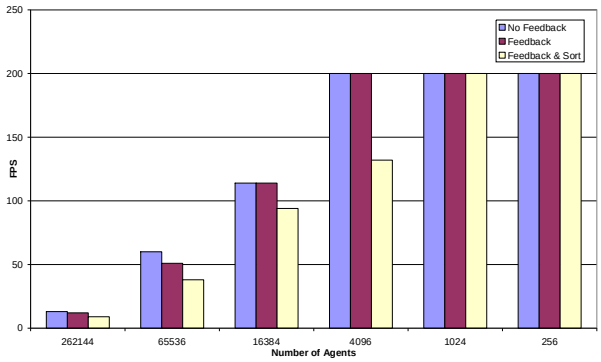

Figure 8 – Performance effect of agent feedback with a communication radius of 0.03125

As rendering is achieved with such high performance in all but the most extreme cases the agent update step is clearly the performance bottleneck of the system. In order to overcome this both Reynolds [12] and Erra et al. [10] avoided performing the full agent update step for each animation frame. Reynolds [12] simply decoupled the simulation and rendering stages with $1/8^{th}$ of the population being updated per rendering frame, whilst Erra et al. [10] used the earlier mentioned *scattering matrix* to vastly avoid recalculation when the flock was uniform. AGBPU does not implement either of these techniques and therefore it can be though of as considerably more brute force with this respect.

Arguably the most significant difference between ABGPU and that of both Erra et al. [10] and Reynolds [12] is the neighbourhood heuristic and consequently the number agents considered for communication. Both implementations favour an N-Nearest neighbour solution in opposition to the communication radius technique used within this work and that of the original Boids paper. Additionally ABM GPU work by D'Souza[1] and collision only based particle physics demonstrations by Green [18] perform significantly fewer communications than the agents described in this paper. In the case of D'Souza [1] a real time performance of 2 million agents is reported however agents are placed within a 2D lattice with only a 9x9 vision filter. The performance of Greens [18] physically based particle demonstrations which use the same boundary scatter technique are slightly higher than our own results (most likely as a result of utilising the CUDA radix sort which limits the algorithms to G80 hardware) however the ridged particle and partition size vastly reduces the number of particles considered during the more

simplistic update stage. Figure 9 shows both the number of agents considered for communication (Lookups) and those which are actually within the agents interaction radius and are hence communicated with. The results of this were obtained by using a communication radius of 0.0625 after 1000 iterations of the simulation. From this the efficiency of the spatial partitioning method can be considered, and on average is roughly 1/3rd, a figure far superior to that of the worst case *all pairs* solution which is the only other method which guarantees all interactions. In cases where hundreds of agents are serially considered during the update stage it is surprising the system is able to sustain real-time performance. This is attributed to the fast cache of the 8800 GT card and the displacement of agent data after the sort stage which increases the cache hit rate and dynamic branch coherency between pixel quads.

Communications				
N	Max	Min	Sum	Average
262144	16334	0	851476608	3248
65536	4244	0	61600140	940
16384	1530	0	5409149	330
4096	253	0	246636	60
1024	44	0	10726	10
256	15	0	684	3

Lookups				
N	Max	Min	Sum	Average
262144	29461	0	2431892736	9277
65536	8862	2	163508320	2495
16384	2638	1	12541857	765
4096	526	1	653152	159
1024	131	1	32572	32
256	29	1	2071	8

N	Efficiency
262144	35.0
65536	37.7
16384	43.1
4096	37.8
1024	32.9
256	33.0

Figure 9 – Texture Lookup Performance of a single frame with a communication radius of 0.0625 after 1000 iterations

8. CONCLUSIONS & FUTURE WORK

In this paper ABGPU has been presented as a fast efficient method of performing agent based modelling. Whilst the current performances are impressive the system would benefit from additional functionality demonstrated in alternative GPU work. The introduction of birth and death allocation [1] on the GPU is the most favourable of these, interaction with advanced or dynamic environment maps [7] is also highly desirable. Additionally the decision to sacrifice simplicity for portability may be reconsidered by producing a Compute Unified Device Architecture (CUDA) backend which will avoid some of the restrictions imposed by packing data into textures (most notably the agent variable count). An advanced extension to the current work could consider scaling the algorithm to multiple GPUs either within a single host machine or across a high performance GPU grid. These will be the topics of future work and will open ABGPU to a wider range of models beyond the continuous population systems which the current work has been so heavily influenced by.

9. References

[1] - D'Souza, R. M. Lysenko, M. Rahmani, K. (2007), SugarScape on steroids: simulating over a million agents at interactive rates, Proceedings of Agent2007 conference. Chicago, IL

[2] - Reynolds, C. W. (1987), Flocks, Herds, and Schools: A Distributed Behavioural Model, in Computer Graphics, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34.

[3] - Collier, N. (2002), RePast: An Extensible Framework for Agent Simulation

[4] - S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, Mason: A new multi agent simulation toolkit, Proceedingso of the 2004 SwarmFest Workshop.

[5] - Minar, N. Burkhart, R. Langton, C. Askenazi, M. (1996), The Swarm simulation system: a toolkit for building multi-agent simulations, Working Paper 96-06-042, Santa Fe Institute, Santa Fe.

[6] - Nyland, L. Prins, J. Harris, M. (2004), Rapid Evaluation of Potential Fields in N-Body Problems Using Programmable Graphics Hardware, Poster Presentation, ACM Workshop on GPGPU, Wilshire Grand Hotel, LA, California

[7] - Dronw, S. (2007), Real-Time Particle Systems on the GPU in Dynamic Environments, ACM SIGGRAPH 2007 course 28: Advanced real-time rendering in 3D graphics and games, pages: 80-96

[8] -Quinn, M. 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), pages 63–74

[9] - Adra, S.F. Coakley, S. Kiran, M.McMinn, P. (2008), An Agent-Based software platform for modelling systems biology, Epitheliome Project Report, Sheffield University

[10] - Erra, U. Chiara, R. Scarano, V. (2006), An Architecture for Distributed Behavioural Models with GPUs, Fourth Conference Eurographics Italian Chapter, pages 197-203

[11] - Chiara, R. Erra, U. Scarano, V. Tatafiore, M. (2004), Massive Simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance, VMV 2004, pages 233-240

[12] - Reynolds, C. (2006), Big fast crowds on PS3, In Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames (Boston, Massachusetts, July 30 - 31, 2006). sandbox '06. ACM, New York, NY, pages 113-121

[13] - Latta, L. (2004), Building a Million Particle System, In proceedings of Game Developers Conference, San Francisco, CA

[14] - Naga, K. Govindaraju. Raghuvanshi, N. Henson, M. Manocha, D. (2005), A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors, UNC Tech. Report 2005

[15] - Purcell, T. Donner, C. Camarano, M. Jensen, H. Hanrahan, P. (2003), Photon mapping on programmable graphics hardware, in Proceedings ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003, pages 41-50

[16] - Kipfer, P. Segal, M. Westermann, R. (2004), UberFlow: a GPU-based particle engine, In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (Grenoble, France, August 29 - 30, 2004). HWWS '04. ACM, New York, NY, pages 115-122

[17] - Harada, T. (2007), GPU Gems 3: Real Time Rigid Body Physics on GPUs, Addison Wesley, pages 611-632.

[18] Green, S. (2007), CUDA Particles, NVIDIA Whitepaper, November 2007.

Figure 10 – 16,384 agents with a maximum detail level of 1500 polygons rendered at over 30 Fps