# FLEXIBLE HIGH PERFORMANCE AGENT BASED MODELLING ON GRAPHICS CARD HARDWARE

by

Paul Andrew Richmond

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy at
the University of Sheffield
Department of Computer Science

February 2010

# Abstract

Agent Based Modelling is a technique for computational simulation of complex interacting systems, through the specification of the behaviour of a number of autonomous individuals acting simultaneously. This is a bottom up approach, in contrast with the top down one of modelling the behaviour of the whole system through dynamic mathematical equations. The focus on individuals is considerably more computationally demanding, but provides a natural and flexible environment for studying systems demonstrating emergent behaviour. Despite the obvious parallelism, traditionally frameworks for Agent Based Modelling fail to exploit this and are often based on highly serialised mobile discrete agents. Such an approach has serious implications, placing stringent limitations on both the scale of models and the speed at which they may be simulated. Serial simulation frameworks are also unable to exploit multiple processor architectures which have become essential in improving overall processing speed.

This thesis demonstrates that it is possible to use the parallelism of graphics card hardware as a mechanism for high performance Agent Based Modelling. Such an approach is in contrast with alternative high performance architectures, such as distributed grids and specialist computing clusters, and is considerably more cost effective. The use of consumer hardware makes the techniques described available to a wide range of users, and the use of automatically generated simulation code abstracts the process of mapping algorithms to the specialist hardware. This approach avoids the steep learning curve associated with the graphics card hardware's data parallel architecture, which has previously limited the uptake of this emerging technology. The performance and flexibility of this approach are considered through the use of benchmarking and case studies. The resulting speedup and locality of agent data within the graphics processor also allow real time visualisation of computationally and demanding high population models.

# Declaration

The work presented in this thesis is original work undertaken between September 2006 and September 2009 at the University of Sheffield. Some of this work has been published elsewhere:

- Richmond Paul, Walker Dawn, Coakley Simon, Romano Daniela (2010), "Parallel Cellular Level Agent Based Modelling with FLAME", Invited Submission for Publication in the special issue: "Parallel and Ubiquitous methods and tools in Systems Biology" of the international journal: Briefings in Bioinformatics (*under review*).
- Richmond Paul, Coakley Simon, Romano Daniela (2009), "Cellular Level Agent Based Modelling on the Graphics Processing Unit", To appear in Proc. of HiBi09 - High Performance Computational Systems Biology, 14-16 October 2009,Trento, Italy
- Romano Daniela, Lomax Lawrence, Richmond Paul (2009), "NARCSim An Agent-Based Illegal Drug Market Simulation", Proc. of The International IEEE Consumer Electronics Society's Games Innovations Conference 2009 (ICE-GIC 09), London in UK, 25th-28th August 2009
- Richmond Paul, Coakley Simon, Romano Daniela (2009), "A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA", Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009), May, 10–15, 2009, Budapest, Hungary
- Richmond Paul, Romano Daniela (2008), "A High Performance Framework For Agent Based Pedestrian Dynamics On GPU Hardware", Proceedings of EUROSIS ESM 2008 (European Simulation and Modelling), October 27-29, 2008, Universite du Havre, Le Havre, France
- Richmond Paul, Romano Daniela (2008), "Agent Based GPU, a Real-time 3D Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the GPU", Proceedings of International Workshop on Supervisualisation 2008 (IWSV08), Part of ICS08, Kos Island, Greece. June 2008.

As a result of preliminary work into automatic city generation, the PhD process has also resulted in the following publication which falls outside the scope of the final thesis.

- Richmond Paul, Romano Daniela (2007), "Automatic Generation of Residential Areas using GeoDemographics", 2nd International Workshop on 3D Geo-Information: Requirements, Acquisition, Modelling, Analysis, Visualisation

# Acknowledgements

The work within this thesis would not have been possible without the help and support of other people. I particularly like to thank;

- My supervisor Dr Daniela Romano who offered support and guidance as well as encouraging me towards a career in academia with help and advise in completing this thesis work and achieving my Fellowship awards.
- My internal PhD panel comprising of Dr Steve Maddock and Mike Holcombe whose expertise no doubt guided me towards my chosen topic.
- The computer graphics research group and systems biology group who have provided feedback on my work through informal discussion and research seminars. I would particularly like to thank Dr Simon Coakley and Dr Dawn Walker who have always made themselves available to discuss issues regarding both FLAME and cellular biology. I would also like to thank Mark Burkitt for testing FLAME GPU during early releases, where he highlighted a number of issues and bugs.
- The Royal Academy for my Student Development Research Fellowship which has been particularly generous in offering financial support, allowing me to travel internationally and encouraging me take part in schools outreach activities. Also ESPRC who have offered me a Fellowship position and more importantly a focus to complete my thesis on time.
- Anyone who has been involved directly and indirectly with providing outreach activities. In particularly, Ed Morgan, Lewis Gill, Duncan Payne and Twin Karmakharm. Thanks also to Mick Humble with help fixing the PCs in the Reflex lab on a regular basis.
- My parents Steve and Anne Richmond who have kindly provided much needed proof reading.
- Rankin Barr and Chris Garret from the former Lincolnshire Drugs and Alcohol Action Team (DAAT) for their input into the original PhD brief and for providing the sponsorship for this EPSRC case funded PhD.
- My dog Sparkey and the guys from Sheffield Steelies football team for ensuring I remained physically active and offering the opportunity to leave my desk once in a while.

- Finally my girlfriend Shelley Hughes who has provided proof reading of both this thesis and research papers throughout the last 3 years. More importantly, she has offered unquestionable support throughout the more difficult times of the PhD process, and without her encouragement I would most likely have given up on my research long ago.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

AB - Agent Based

ABGPU - Agent Based GPU

ABM - Agent Based Modelling

API - Application Programming Interface

CA - Cellular Automaton

CAL - Compute Abstraction Layer

CPU - Central Processing Unit

CTM - Close to the Metal

CUDA - Compute Unified Device Architecture

CUDPP - CUDA Parallel Primitives Library

DRAM - Dynamic Random Access Memory

DTD - Document Type Definition

EBM - Equation Based Models

EIB - Element Interconnect Bus

FBO - Frame Buffer Object

FLAME - FLexible Agent Modelling Environment

FLAME GPU - FLAME for the GPU

FPS - Frames Per Second

FSM - Finite State Machine

GFLOPS - Giga Floating Point Operations Per Second

GLSL - OpenGL Shading Language

GPGPU - Purpose computation on the GPU

GPU - Graphics Processing Unit

GPUXMML - A GPU extension of XMML

HLSL - High Level Shader Language

HPC - High Performance Computing

IO - Input Output

LOD - Level of Detail

MAS - Multi Agent System

MPI - Message Passing Interface

MRT - Multiple Render Target

ODE - Ordinary Differential Equation

OO - Object Orientated

OOP - Object Orientated Programming

OS - Operating System

PDE - Partial Differential Equation

RAM - Random Access Memory

RGBA - Red Green Blue Alpha

SDK - Software Development Kit

SIMD - Single Instruction Multiple Data

SPH - Smoothed Particle Hydrodynamics

SPMD - Single Program Multiple Data

SPU - Synergistic Processing Unit

SQL - Structured Query Language

SXM - Stream X- Machine

TBO - Texture Buffer Object

UML - Unified Modelling Language

VBO - Vertex Buffer Object

XMDL - X-Machine Description Language

XMML - X-Machine Mark-up Language

XSLT - Extensible Stylesheet Transformations

# Chapter 1

# Introduction

Agent Based Modelling (ABM) is a powerful simulation technique used to assess and predict group behaviour from a number of simple interacting rules between communicating autonomous agents. Traditional ABM toolkits are primarily aimed at a single CPU architecture, with an inherent lack of parallelism as a result of the design methodologies and choice of programming language. This has serious implications with regards to the scale of models that can be simulated, as well as suitability of such frameworks to exploit multi-core architectures. ABM methods specifically targeting parallelism have taken a task-parallel approach, aimed at high performance computing (HPC) architectures, such as processing clusters or grids. Such specialist hardware is generally expensive and unavailable to the majority of Agent Based (AB) modellers.

The Graphics Processing Unit (GPU), which is primarily designed to stream graphics primitives through a rendering pipeline, is a massively parallel device offering the potential for cost-effective supercomputing. The performance advantages of utilising the GPU are only realised by considering the hardware's performance in contrast with that of the CPU (Figure 1). Unlike more generic and flexible CPUs, the GPU's architecture is task specific, making it highly optimised for data parallel programming applications. Technically, the GPU not only exceeds the transistor count of modern CPUs, but a significantly higher portion of transistors are available for data

processing, rather than data caching and flow control. In addition, the GPUs memory bandwidth exceeds that of system memory bandwidth by roughly a factor of 10. In the past General Purpose computation on the GPU or (GPGPU) has focused on utilising graphics libraries to exploit the data parallel architecture. More recently, GPGPU computing has benefited from the introduction of improved programming interfaces implemented by hardware vendors, making the architecture more accessible. Despite this, major performance gains are often achieved only through careful optimisation, requiring advanced knowledge of the hardware's capabilities and optimal operating conditions. As a result of the difficulty of programming GPU devices, the uptake of this parallel hardware architecture has had a relatively low impact on the field of ABM. The few examples of ABM on the GPU are limited to specific examples of discrete models or swarm systems.

The work within this thesis addresses the performance and architecture limitations of previous work by presenting a flexible framework approach to ABM on the GPU. The aim of this thesis is to provide a technique that allows modellers to harness parallelism and the power of the GPU, without requiring background knowledge of parallelism or the hardware.

The advantages of this work include a method for high performance ABM using consumer hardware. Performance rates equalling, or bettering that of HPC clusters can easily be achieved, with obvious cost to performance benefits. Massive population sizes can be simulated, far exceeding those that can be computed (in reasonable time constraints) within traditional ABM toolkits. The use of data parallel methods ensures that the techniques used within this thesis are applicable to emerging multi-core and data parallel architectures that will continue to increase their level of parallelism to improve performance. Finally the use of the GPU allows AB models to be visualised in real time, which further widens the application of ABM to real-time simulations. This is the result of both the speedup achieved and the avoidance of transfer bandwidth costs by maintaining model data on the GPU device.

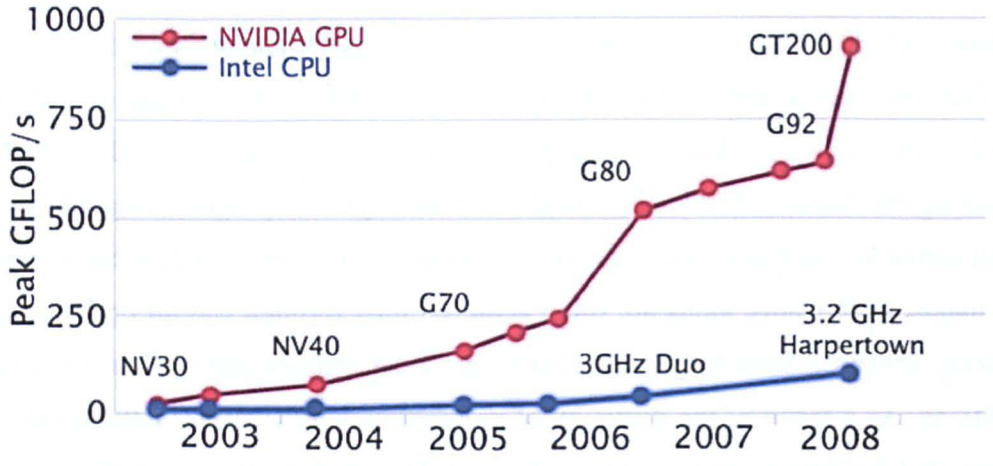



**Figure 1 - Peak Performance of NVIDIA GPU Hardware (Red) vs. Intel CPU Hardware (Blue). Data from [NVI07]**

## 1.1 Contribution to Knowledge

This thesis explores the potential of consumer GPU processors as a technique for flexible ABM. More specifically the novel contributions of this thesis are as follows:

- Agent Based GPU (ABGPU), a technique for high performance swarm modelling (ABGPU) is presented. This automatically maps agents and behaviour to computer graphics APIs, abstracting the process of GPU programming. This is demonstrated with the simulation of flocking and pedestrian dynamics with real time rendering of agents.
- The conceptualization, implementation and testing of FLAME GPU[1], a FLexible Agent Modelling Environment (FLAME[2]) inspired, ABM framework that allows simulation of a wide range of general AB models through high performance parallel computation on the GPU.

In particular FLAME GPU introduces the following novel techniques:

- A flexible technique for Object Oriented extension of a formal specification mark up language within FLAME GPU. A robust templating system, which complements this ensures syntax validation of model specifications.

[1] www.flamegpu.com
[2] www.flame.com

- Non mobile discrete system support to FLAME GPU. It is subsequently demonstrated that can be used to efficiently simulate mobile discrete systems.
- The concept of a global function condition is introduced into the underlying X-Machine representation of FLAME GPU. This is used to achieve non linear simulations steps and is shown to be suitable for parallelising high accuracy recursive force resolution and resolving conflict within mobile discrete agent systems.
- The real time visualisation of FLAME GPU models including a complex cellular level systems biology model which was previously inconceivable and required offline computation taking several hours.

## 1.2 Outline of the Thesis

**Chapter 2**

Chapter 2 provides background literature on ABM with emphasis on top down equation based alternatives and the techniques which are used for AB specification. The GPU is introduced, as are techniques for harnessing the GPU's parallel processing power for general purpose use. The languages and techniques for GPU programming, which are used within this thesis are also introduced. The implementations of work on the GPU which have agent based functionality are reviewed. These mostly consists of particle systems with a degree of communication or interaction, however techniques for discrete agent and pedestrian dynamics modelling are also discussed.

**Chapter 3**

Chapter 3 considers the use of traditional GPGPU techniques to implement a high performance swarm based library for the GPU. It describes a process where agents can be mapped to the GPU and a parallel communication algorithm, which allows agents to communicate across spatial partitions. 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. The library is demonstrated through the implementation of a flocking algorithm and a model of pedestrian behaviour. Both implementations make use of real time feedback routines

to implement a Level of Detail (LOD) rendering system. This allows high fidelity visualisation at interactive rates.

**Chapter 4**

Chapter 4 describes a more flexible architecture for ABM on the GPU. This extends an existing template based framework based upon formal specification techniques. A method for flexible and extendible specification is presented, as is an improved templating system. The process of converting models from the original framework is also discussed.

**Chapter 5**

Chapter 5 describes the implementation of the flexible ABM framework described in the previous chapter. The implementation of various ABM aspects such as data storage, birth and death allocation, agent heterogeneity, visualisation and communication are described. Communication is further broken down into three differing patterns, which are optimised in each case to provide maximum performance depending on the type of model which is being implemented.

**Chapter 6**

Chapter 6 evaluates the performance of the flexible agent framework described in the two proceeding chapters. A performance evaluation between the use of the FLAME framework for the CPU, HPC and FLAME GPU is presented using a benchmarking model. It is demonstrated that FLAME GPU can outperform FLAME on any other architecture. This is validated through the implementation of a more complex cellular systems biology model which shows a massive performance improvement.

**Chapter 7**

Chapter 7 concludes the main body of the thesis giving suggestions for future work.

# Chapter 2

# Background

This chapter is divided into three sections which introduce ABM, the GPU and examples of agent like systems on the GPU.

## *2.1 Agent Based Modelling*

This first section reviews background material relating to ABM. The origins and history of ABM are reviewed, together with the origin of swarm systems and the use of ABM within biology. The specification of AB systems is discussed, with emphasis on the use of both objected orientated and more formal techniques for systems specification. Existing ABM frameworks are also reviewed as is the use of the X-Machine for AB specification.

### 2.1.1 Equation Based vs. Agent Based Modelling

Top-down modelling involves representing observed system behaviour with Equation Based Models (EBM), such as differential equations (either ODE's over time or PDE's over time and space) [DSR98]. In such systems, observables represent

changeable quantities, such as population sizes or concentrations of a particular entity. Models are often designed to match real world observations and then used to make predictions or a hypothesis of the system under differing conditions. Often such predictions can be validated through observation or experimentation in the real world. Despite the advantages for macroscopic simulation, EBM offers little insight into the micro level behaviour of the individual interactions within the system. Where global observations are made, these represent average values and assume homogeneity and perfect mixing of system components. As a result, important low level details of the system may be simply ignored.

By contrast, ABM utilises a bottom up approach to simulation that does not explicitly attempt to model aggregate characteristics of a system. As with Multi-Agent Systems (MAS), AB models can be described as a "system of interacting parts". The notable difference being that agents (or individuals) within ABM are simulated as autonomous individuals, whereas MAS may use a more generic agent representation. Typically an AB model consists of a number of agents, an environment and a set of rules governing agent behaviour. Agents themselves are self contained entities containing states and a set of behavioural rules. They may represent spatial entities such as molecules or cells, in which case they may reside within a continuous or discrete spatial environment. In addition, agents may interact directly, or through an environment where they may compete for resources. By specifying rules at a local individual level, complex 'emergent' system behaviour can be observed through the result of agent interactions. The specification of individual rules also makes ABM inherently capable of representing heterogeneity, as each agent can possess its own individual attributes and behaviours. Such system-wide diversity is important as many systems of real life phenomenon cannot be expressed as simple uniform entities.

### 2.1.2 Origins of Agent Base Modelling

Historically, ABM can be traced back to work by Von Newman and Stanislaw Ulam and their work on self replicating machines which became known as Cellular Automaton (CA). Further improvements to this in John Conway's model 'The Game of Life' [Gar70] became an inspiring demonstration of how autonomous agents can

produce complex behaviours from simple rules. CA is a subset of ABM where cells are located only within discrete space, with a finite number of states and operating in discrete time. Most commonly, CA environments consist of a regular grid of cells, however hexagonal environments are not uncommon. In all cases CA are homogeneous and are processed asynchronously with cells states being determined as a function of their current state and their neighbour's states, at the previous discrete time step. From these simple rules a grid of cells with an initial random state, form complex evolutions of patterns, demonstrating emergence and indefinite growth. As a result of this, the Game of Life and other such CA have fascinated mathematicians and philosophers, who draw parallels with phenomenon such as free will and consciousness as emergent elements spawned from the physical laws governing our universe [Den91].

### 2.1.3 Swarm Modelling

Reynolds early work on Flocking [Rey87] is an example of complex agent modelling. It demonstrates a technique for spatially explicit continuous valued mobile agents, before the term agent was widely adopted. The model itself demonstrates the power of emergence within biological systems by achieving complex flocking and swarming through the use of only three simple steering behaviours. Although more complex behaviour can be demonstrated through the additional of obstacle avoidance, the systems general behaviour can be expressed on an individual level by the following simple rules;

- Separation – each individual avoids local flock-mates.
- Alignment – each individual matches its other flock-mates velocity.
- Cohesion – each individual moves towards its perceived centre of the flock.

Within each of the above the Boids algorithm takes into account an individual model of perceived perception. Practically, this limits the interaction between the agents to within a localised neighbourhood. Although this neighbourhood can be expressed as a simple radial function, Reynolds demonstrates how it is also possible to limit this to an individual's line of sight. Whilst the Boids algorithm rules are relatively simple, they (as with similar avoidance models) have been widely adopted, particularly for

use with animation techniques in games and movies, and more recently for large scale pedestrian dynamics [SOHTG99, HM97, Rey99, QMHz03].

### 2.1.4 Applications of Agent Based Modelling in Systems Biology

Aside from simple swarm behaviour, biological systems in general are an ideal and common candidate for ABM. Entities in biological systems, from a cellular level [WSH+04, SMC+07, WS09] to large ecosystems [Dor01], can be represented directly as agents (with some level of abstraction). Of particular interest to this thesis is the use of ABM to simulate cellular level systems, which offer a middle out alternative to the 'top down' or 'bottom up' approaches which are more common [WS09]. By modelling cells, the basic unit of biological function, predictive models based on the large amounts of data described at the cellular level are able to provide insight into larger biological systems. Unlike discrete continuum alternatives [GMST99, JEB+06], ABM of cellular level systems allows individual cells to be tracked throughout a simulation. Whilst this creates an enormous amount of data this can be easily visualised to allow comparison between in-vitro/vivo and in-silico experiments for simple visual model validation. Such a technique is invaluable to biologists as in-silico simulation offers potential to hypothesise about the effects of external stimuli, reducing or complimenting in-vivo alternatives.

### 2.1.5 Object Orientated Agent Representation

Object Orientated Programming (OOP) is widely adopted as the most common paradigm for ABM frameworks [IGCG99]. This is mainly due to the strong analogy between agents and objects which, although dissimilar [Ode02], have advantages of representation in this form. The most significant of these is the popularity of Object Orientated (OO) methodology and the common adoption of design principles such as the Unified Modelling Language (UML) [OPB00, BMOO01, PKK07]. OOP offers a natural and simple technique for modelling which is easily understood by software engineers who are familiar with OO design. Agent Objects are represented as static objects which control their state and execution and communicate through message passing. The majority of popular ABM frameworks are based on OO design

principles, some even use concepts such as UML for agent and system specification. Of these frameworks most are accessible through an Application Programming Interface (API), which provides tools for building and describing models and common routines such as agent communication and scheduling of agent behaviour and interactions. Table 1 shows a comparison of common AB frameworks and compares critical aspects, such as the underlying programming language and dynamics used for agent representation.

**Table 1 - Comparison on Object Orientated ABM Platforms**

| **Name** | **SWARM** [MBLA96] | **RepastS** [NHCV05] | **NetLogo** [Wil99] | **MASON** [LCRP+05] | **Cormas** [BBPP98] |
|---|---|---|---|---|---|
| Language | Objective C, Managed C++, Java | Java, Python, .NET | Scripting | Java | SmallTalk, Scripting |
| Tutorials | Yes | Yes | Yes | Yes | Unknown |
| Parallel | No | No | No but the original Star Logo was targeted at a parallel machine | Yes – each agent is stepped with the same function through a scheduler | No |
| Raster Space | Yes | Yes | Yes | Yes | Yes |
| Vector Space | Yes | Yes | No | Yes | No |
| Visualisation | 2D | 2D & 3D | 2D, basic 3D | 2D, limited 3D | 2D Grid |
| Open Source | Yes | Yes | Yes | Yes | Yes |

Swarm is often considered the first reusable and most mature tool for ABM, with projects such as Repast originating from branches of development adding their own focus or platform support. Repast Symphony (RepastS) is the most recent version of Repast and replaces older platform specific implementations with an improved, yet strictly Java, specific interface. MASON stems also from Repast and is designed with performance in mind. Despite its multithreaded implementation, MASON and the other OO frameworks listed in Table 1 are not designed with parallelism as a key requirement. Part of the reason for this is the pedigree of the frameworks, which have origins in ecology where highly serial models in discrete environments are often used. This imposes an obvious limitation in both model scale and performance, which is addressed by this thesis.

Aside from the lack of support for HPC architectures, OO techniques have additional criticisms which mostly stem from the differences between agents and objects. Agent autonomy is also a central issue. Method invocation between objects is incapable of capturing the dynamic nature of agent interactions which may involve complex protocols and negotiation. Wooldridge [WJK00] makes the suggestion that Objects are too fine grained and static to represent agents and operate at an inappropriate level of abstraction. Central to this argument is the fact that Objects are unable to capture the relationship between agents and their interactions. Likewise, agents' internal states are not necessarily suitable for representation as attributes, a view which is shared by Iglesias et al. [IGCG99]. Finally, OO principles, such as inheritance and abstraction, have little meaning with respect to agent representation and can only add confusion.

In summary, OO principles offer a modelling technique that has been well adopted due to familiarity of the methodology. Whilst intuitive, this introduces problems which can be avoided through alternative or more formal representations which are discussed in the following section.

### 2.1.6 Formal Modelling of Agent Based Systems

Formal techniques are advantageous as they provide a method for both specification and validation. Whilst formal specification is useful in the generation of system implementations, validation is invaluable as it allows automatic verification and error checking of systems. In the case of high integrity or mission critical systems the guarantee of reliability and correct behaviour is not only advantageous but essential. Whilst useful for software design, not all formal specification techniques are suitable for use with ABM. Many lack the ability to represent indefinite simulation or express the dynamic behaviour or communication.

Many formal techniques appropriate for multi agent systems are based around automaton and state based representation of agents. For example, CA can be described as a grid of interacting Finite State Machines (FSMs) where a FSM is formally defined as the 5-tuple $(Q, \sum, q_0, \delta, A)$ where;

Q is a finite set of states

$\sum$ is a finite alphabet of input symbols

$q_0$ is the initial (or starting) state where $q_0$ is a member of Q

A is the set of accepting (terminal) states where A is also a subset of Q

$\delta$ is the state transition function from $Q \times \sum$ to Q

This provides a powerful technique for simple models, as states can be specified and rules defined as transition functions which describe the agents control flow from one state to another. Input symbols represent output from neighbouring cells and in the case of the Game of Life [Gar70] can be represented as the binary sets of 00000000 through to 11111111 (where 1 and 0 represent dead or alive for each neighbouring cell). Whilst feasible for simple CA, the lack of any internal memory leads to a combinatorial explosion of stages when considering even simple communication. As a result of this, it is no surprise that in order to represent more complex non-trivial systems a more powerful representation is required. Aside from state based formal techniques, process algebras such as Z [ASM80] and Communicating Sequential Processes [Hoa78] have also been used for specification of concurrent systems and sequential processes respectively. These techniques tend to become quite complex when applied to ABM [dL00], with the generation of simulation code being far from straightforward [RTRH05].

### 2.1.7 Agent Based Modelling with the X-Machine

X-Machines are a concept first introduced by Eilenberg [Eil74] and extended upon by Holcombe [Hol88] which unlike FSMs contain an internal memory. This memory therefore extends upon the mapping '$\delta$' of FSMs by introducing a function that manipulates an X-machine's internal memory. This extension overcomes the exponential growth of systems by allowing the number of states to be greatly reduced. The uptake of this formal method has previously been limited but since its use of formally validating swarms [HRRT05, RTRH05] there has been increased interest in its use to model biological systems [Ghe05, KSG05]. The FLAME framework [CSH06, Coa07, ACKM08] remains the most advanced use of the X-Machine for formal modelling and forms the inspiration for work presented later in this thesis. The formal definition of a Stream X- Machine (SXM), a particular class of X-machine,

where the input and output are streams of symbols is defined formally as a 9-tuple $(\sum, \Gamma, Q, M, \Phi, F, q_0, m_0, T)$ [LL93], where;

$\sum$ and $\Gamma$ are the input and output finite alphabet respectively;
Q is the finite set of states;
M is the (possibly) infinite set called memory;
$\Phi$ is the type of the machine SXM, a finite set of partial functions $\phi$ that map an input and a memory state to an output and a new memory state, $\phi: \sum \times M \rightarrow \Gamma \times M$;
F is the next state partial function that, given a state and a function ($\phi$) from the type $\Phi$, provides the next state, $F: Q \times \Phi \rightarrow Q$ (F is often described as a transition state diagram);
$q_0$ and $m_0$ are the initial state and memory respectively; and
T is the set of terminal states.

Whilst significantly more expressive than FSM, SXMs are limited by their inherent lack of communication. The Communicating X-Machine is a deviation from a standard X-Machine which addresses this limitation. There are numerous formal definitions, Bălănescu et al. [BCG+99] describe one variant, Communicating SXMs (CSXMS) which are specified formally as the 3-tuple $((P_i)_{i=1,\ldots,n}, CM, C_0)$, where;

$P_i = (X_i, IN_i, OUT_i, in_i^0, out_i^0)$ is a component of the system, i = 1,..., n;
$X_i$ are stream X-machines, i=1, ..., n;
$IN_i$ and $OUT_i$ are two sets called the input port and the output port respectively of component i. The elements belonging to these sets are values from $M_i$ or the undefined value $\lambda$, which is $IN_i$, $OUT_i$ is a subset of $M_i$ union $\{\lambda\}$ and $\lambda$ is not a member of $M_i$;
$in_i^0$ is a member of $IN_i$ and $out_i^0$ is a member of $OUT_i$ and are the initial input and output port values respectively;
CM is can be defined as set of matrices of order n × n used for communication between the X-machines $X_i$; and
$C_0$ is the initial communication matrix.

This definition constitutes a number of X-Machines communicating through messages arranged in a static matrix. Whilst suitable for certain agent models the inflexibility of a fixed communication matrix is far from ideal. Firstly, the inability to change the communication matrix size during simulation makes this definition suitable only for static population sizes. Additionally, the exponential growth of the communication matrix is (with respect to the population size) unnecessary, as many agents interact over a limited range making many of the cells in the matrix redundant. These issues are addressed within FLAME, which uses message lists rather than a static matrix and within work by Kefalas et al [KHEG03], which allows communication directly through additional input/output streams.

## 2.2 Graphics Processing Units

Before a review of ABM on the GPU can be considered, the success of GPU hardware as a parallel co-processor, to accelerate computationally expensive algorithms must first be reviewed. Although programmable GPU hardware functionality was primarily aimed at techniques to improve shading, the speed and parallel processing power has and continues to attract a wider range of programmers than from the pure graphics domain. In particular, the use of GPU hardware for general high performance computation has spawned an area of research known more commonly as GPGPU. The popularity of this area of research has no doubt been highly influential in the release of more recent architectures and APIs which give direct access to underlying hardware. This has been made possible by the shift from a fixed function pipeline to the evolution of programmable processors. This section introduces the GPU, with some history and description of classical GPU architecture use before discussing emerging hardware architectures and functionality which are used later in this thesis.

### 2.2.1 The Graphics Pipeline

Although a brief history and discussion of future architectures and APIs is presented later in this chapter, Figure 2 presents the graphics pipeline of a current generation GPU where each section of the pipeline is described below;

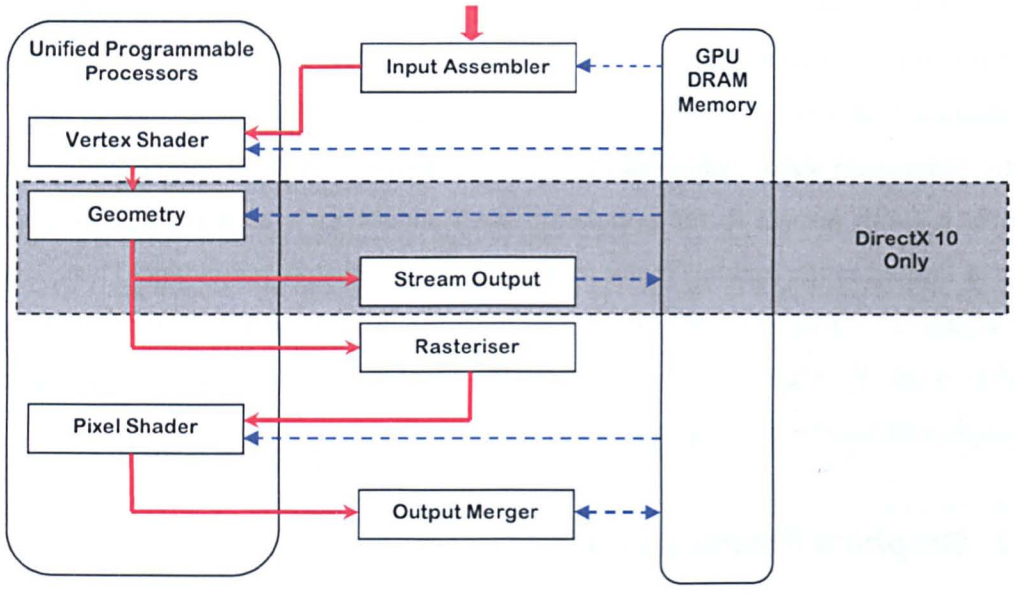



**Figure 2 - The DirectX 10 (Shader Model 4) Graphics Pipeline**

*Input Assembler*

The input assembler is responsible for scheduling 3D positional data (vertices) from graphics programs into the vertex processor. Such data occurs from calls within graphics libraries such as OpenGL or DirectX.

*Vertex Shader*

As the vertex data is scheduled into the vertex processor from the input registers each vertex is processed using a list of instructions from the vertex program instruction memory, which describe the vertex program. Typically these transform vertex attributes, such as the 3D position into world space, perform per vertex lighting (which is interpolated between vertices) as well as operating on the colour and texture coordinate attributes of each vertex.

*Geometry Shader*

The geometry shader is a recent addition to the graphics pipeline which introduces the functionality to process and create graphics primitives (lines, triangle or quads). During invocation the geometry shader inputs the small sets of vertices making up the graphics primitives. In addition to this, access to neighbouring edge primitives such as adjacent triangle or adjoining lines are available. The ability to amplify a single vertex to zero or more strips of connected primitives adds the functionality to generate primitives, which can be either fed back directly to the primitive assembly stage or into Dynamic Random Access Memory (DRAM) via the stream output stage.

*Stream Output*

The steam output stage works closely with the output from the vertex or geometry (if active) processor stages enabling vertices making up complete primitives to be stored in GPU memory via a buffer. From here they can be either used as input for the input assembler in subsequent render passes or used as input in any of the programmable stages.

*Primitive Assembly*

The processed vertices are passed along with the primitive batching mode into the primitive assembly stage where a number of tests determine their visibility in screen space. Clipping is performed which ensures primitives outside the view frustum (the area between the near and far view planes of the viewpoint) are disposed of and culling performs the process of dropping primitives facing away from the viewpoint.

*Rasterisation (and Interpolation)*

The rasteriser takes the continuous primitive coordinates in 3D space and converts them into a set of discrete fragments. Although fragments are often confused with pixels, they are somewhat different; they are in fact more accurately the discrete set of pixel-sized elements representing the primitive objects in screen space. This therefore implies a pixel may consist of more than one fragment. After the rasterisation the primitives vertex attributes (for a triangle there would be three, a quad four, etc.) are interpolated into the fragments allowing each to have a colour representation, a

number of texture coordinates and a possible new depth value (which may or may not result in it being discarded at this stage).

*Fragment (Pixel) Shader*

This programmable stage is responsible for operating on each fragment by processing them using a set of instructions from the program instruction memory that makes up the fragment program. Although different hardware profiles (discussed later in Section 2.2.3) allow for varying instruction sets to be used, the primary purpose of the fragment processor is to apply any non-linearly interpolated values, such as the texture value lookups, to determine a fragments final colour. In addition to this, the (write only) depth buffer can optionally be written to in the same way.

*Output Merger (Blending and Fragment Operation Tests)*

The final stage in the rendering pipeline involves a number of fixed function operations, which are responsible for using fragment data to write to the frame buffer. Within this stage a number of operations are performed per fragment such as depth testing and alpha blending; which discards hidden surfaces and displays semi transparent objects.

### 2.2.2 Evolution of the GPU

The above description of the graphics pipeline represents the basic stages of today's programmable graphics hardware. Earlier hardware for 3D graphics was far more rudimentary. In particular, the fixed pipeline cards of the early 1990s, such as the ATI range and 3dfx Voodoo, lacked any kind of programmable support that is so common in today's hardware. It wasn't until the early 2000s when the first programmable hardware became available supporting shader model 1, and even then the hardware (NVIDIA GeForce 3 and 4 Ti and ATI's Radeon 8500) supported only a programmable vertex processor with pixel configuration support, therefore lacking full pixel programmability. It wasn't until the introduction of shader model 2 hardware that true programmable graphics cards with floating point support became

available, in fact Kruger [Kru06] highlights that, following the release of shader model 2, the SIGGRAPH[3] 03 conference dedicated an entire session to GPU programming techniques.

Shader model 2 essentially paved the way forward for GPU programming through the introduction of floating point textures/arithmetic, in addition to the full shader model 3 instruction set (despite the hardware to support it coming substantially later). Important in the new functionality was dynamic branching, multiple render targets and texture access within the programmable vertex stage. In addition to this, the instruction count was in theory limited only by the finite memory, despite early DirectX 9 APIs limiting this substantially more.

The unified shader model was introduced in DirectX 10 as shader model 4 (SM4) creating a common instruction set used by all programmable stages of the graphics pipeline. Whilst hardware may continue to provide separate physical processors for each stage, the majority of cards supporting SM4 adopt the unified shader architecture, where a single more flexible processor uses dynamic load balancing between the programmable stages. In addition to this, SM4 added the geometry shader and stream output pipeline stages along with integer and bit wise operations, allowing a new class of algorithms such as Marching Cubes [JC06, FQK08] and Progressive Mesh Refinement [HSH09] to be implemented.

### 2.2.3 Graphics APIs and GPU Programming

The previous section discussed evolutions of graphics hardware architectures with respect to DirectX versions. Whilst DirectX is one option for programming the GPU, OpenGL is an alternative, which is often favoured outside of the computer games industry. The difference between these two graphics APIs is primarily with respect to the target platform. DirectX is supported only in windows environments (and on the Xbox and Xbox 360) where as OpenGL is cross platform with common and up to date driver implementations for both Windows and Linux. Additionally, both DirectX and OpenGL introduce new functionality very differently. DirectX fully specifies a new feature set in advance of supporting hardware, where as OpenGL allows hardware

[3] http://www.siggraph.org/

vendors to specify 'extensions' which give programmers early access to new hardware functionality.

With respect to programming the various stages of the GPU, this can be achieved using various high level languages such as Cg [FK03], the OpenGL Shading Language (GLSL) [gls08] and the High Level Shader Language (HLSL) [PM04]. These higher level shading languages offer an abstraction from the physical hardware, which offers significant advantages with respect to both simplicity and portability. Avoiding the use of lower level assembly language is highly beneficial, as code is far more expressive, readable and generic with respect to differing hardware platforms. Compilers have played an important role in shader languages and ensure that generic higher level code is mapped optimally to underlying hardware. Profiles play a major part in achieving this and act as a method of defining the hardware functionality which the shader should be compiled for. For this reason, dynamic compilation through runtime APIs is encouraged and offers the potential to optimise shader programs, depending on the hardware at runtime.

### 2.2.4 The GPU Programming Model

Traditional GPGPU techniques focus on utilising the graphics libraries and shader languages (discussed in the previous section) to exploit the hardware architecture [OJL+07]. When using this technique the GPUs parallelism is harnessed by expressing algorithms in a data parallel fashion, with application flow following a stream like programming model [BFH+04, Buc06]. At the heart of the stream programming model is the notion of a 'kernel' which takes as input and output one or more read or write only data streams. As a kernel's output is a function only of its input, a single kernel's computation is independent of any others, allowing a high degree of parallelism. This therefore allows a kernel application to run in parallel on every input simultaneously. Whilst this may suggest that the GPU therefore acts using a Single Instruction Multiple Data (SIMD) model, it in fact utilises a technique somewhere between SIMD and Single Program Multiple Data (SPMD). In a purely SIMD architecture every parallel processor executes the same instructions and in the case of conditional code branching, masking is used to ensure all processors to follow the same instruction path. On the GPU this SIMD behaviour is only observable within

small groups of processors (or subgroups). Each subgroup is able to follow different instruction paths through the kernel program, therefore demonstrating SPMD behaviour across groups.

### 2.2.5 Traditional General Purpose GPU Programming

For the purposes of stream programming using the graphics based shading techniques describes in Section 2.2.3, the fragment processor offers the best conceptual match. Computational tasks map well to the 2D grid based semantics of this processing stage, where streamed input and output data are stored within the four colour channels of one or more stacked 2D textures.[4] In order to invoke a kernel four vertices must be passed into the graphics pipeline to fill an 'n x n', 2D orthogonal viewpoint. This causes the rasteriser to produce 'n x n' fragments that use the bound fragment program to perform processing in parallel. In the most basic case, if the input textures are of the same dimension, each parallel instance of the fragment program can then lookup a single value from the input textures(s) to operate on (using SIMD subgroups of 4 fragment blocks) before finally outputting to a 'n x n' frame buffer. Whilst the output frame buffer is most commonly bound to the display screen, the use of an off-screen buffer allows output to be used in subsequent calculations. The obvious limitation of this technique, and of the stream programming model in general, is that data may only be bound as input or output but never both. This can be overcome by utilising a ping-pong technique, which enables the roles of the texture to swap after each render pass (output becomes input and vice versa). A simple example of this is a linear algebra program which is recursively called using a two textures (input and output) which alternate their functionality after each frame pass. The implementation of this requires that N data elements are initialised within a data array of size 'n x n', with a quad being rendered of equal dimensions. As the vertex program plays no part in the algebra calculation, vertices are simply passed through (where a particular stage is non essential the term pass-through is often used to describe it) to the rasteriser, which in turn invokes the programmable fragment stage. With the destination (write) texture attached to the frame buffer, the fragment program then performs a single texture lookup (on the source input texture) and outputs the input value plus some

[4] Whilst 3D textures can be used as an alternative to a number of stacked textures, there is generally less hardware support and in most cases offer poorer performance.

linear value. The ping-pong technique is then used to swap the source and destination textures so that the output becomes the input for the next frame pass, which is invoked by the repeated quad drawing. This ping-pong technique can continue indefinitely, always overwriting the previous input at each output stage.

### 2.2.6 GPGPU Programming Languages

Despite the performance benefits offered by traditional GPGPU techniques they are at best unintuitive, and at worst incredibly tedious, even for programmers familiar with graphics programming. Whilst some of the inherent weaknesses, such as the lack of debugging support have been partially addressed [gde, Mer07], higher level languages which avoid traditional Graphics APIs are significantly easier to use for general purpose programming. Broadly GPGPU languages can be broken down into two subcategories. Those which utilise graphics APIs, by translating general purpose code into the necessary graphics calls, and on a lower level, those which map code directly to hardware without use of the graphics driver.

Brook for GPUs [BFH+04] is a compiler and runtime implementation which exclusively uses the stream programming model. Its focus is on data parallel GPGPU programming and actively encourages the use of high arithmetic intensity to maximise GPU performance. The Sh [MQP02] metaprogramming langue (which is now commercialised as RapidMind with extended Cell processor support) is similar to Brook in respect that it supports stream programming, however it has additional support for graphics programming. Like Brook, Sh supports the writing of shaders/kernels directly in C++ applications. This allows access to common variables on both the host and device without parameter binding, which are required using traditional techniques. Both Brook and Sh rely on compiler technology to generate code for GPU hardware. In both cases GPU assembly is generated, which is optimised depending on the available hardware and chosen graphics API.

Close to the Metal (CTM), which has been superseded by the ATI Compute Abstraction Layer (CAL), is a lower level vendor specific programming interface suitable for GPGPU. It avoids the use of graphics APIs by providing a runtime environment with driver support. This allows direct access to the unified processors on ATI hardware and gives programmers the opportunity to highly optimise code. In

order to provide high level support, ATI provides the AMD Stream SDK [ATI09] which most importantly includes a modified Brook compiler (Brook+) with support for CAL output. In addition to this, performance profiling and optimised core math libraries are available.

The Compute Unified Device Architecture (CUDA) is the NVIDIA alternative to Stream SDK which similarly provides vendor specific low level hardware support. As with the Stream SDK, high level support is offered through the use of the C programming language with extensions. As well as being far more widely adopted than the Stream SDK library, the low level access to per processor shared memory offers potential to achieve far greater performance than with the purely stream oriented alternative. Likewise the availability of lightweight synchronisation primitives, and similarity with expected standardised future libraries for data parallel programming (see section 2.2.8), makes CUDA the GPGPU language of choice for more flexible GPU programming presented later in this thesis.

### 2.2.7 CUDA Architecture and Programming Model

The CUDA GPU programming model is described in detail in the CUDA Programming Guide [NVI07] where the device is presented as a highly parallel, multithreaded, many-core co-processor. The key to the model is the use of a hierarchy of thread groups, where each thread represents a fine grained data parallel execution of a program (or kernel). The concept of a grid of thread blocks is used to allow a transparently scalable (to multiple hardware implementations with varying parallel capabilities) model with independently parallel blocks containing cooperating threads. At a hardware level the CUDA architecture consists of a varying number of streaming multiprocessors (Figure 3) each with eight scalar processing elements, a multithreaded instruction unit and on chip shared memory. On chip shared memory allows thread cooperation through a lightweight synchronisation primitive however global synchronisation can only be ensured after a complete kernel execution. In addition to shared memory each multiprocessor has access to an on chip constant and texture memory cache as well as the devices global DRAM memory.

For each data parallel kernel, the CUDA API provides a template based semantic for grid and block size specification. During execution thread blocks are split into

smaller units of 32 threads, called warps. Blocks are then optimally distributed amongst multiprocessors with the amount of blocks limited by either the hardware specific maximum number of warps per multiprocessor, or resource limited by the total register or shared memory usage per block. Warps are always processed a single instruction at a time (SIMD), with the multiprocessor switching between warps which are ready for execution. This warp switching (or interleaving) allows global memory access latency to be effectively hidden, providing the multiprocessor can be kept busy with non latent arithmetic instructions i.e. a high enough ratio of computational arithmetic to bandwidth (arithmetic intensity). As each instruction across a warp is executed in parallel, any conditional branches between threads must follow the same path to attain maximum performance. In the case of divergent branches (or warp serialisations) between threads in the same warp, instructions must either be serialised or multiple paths evaluated by every thread.

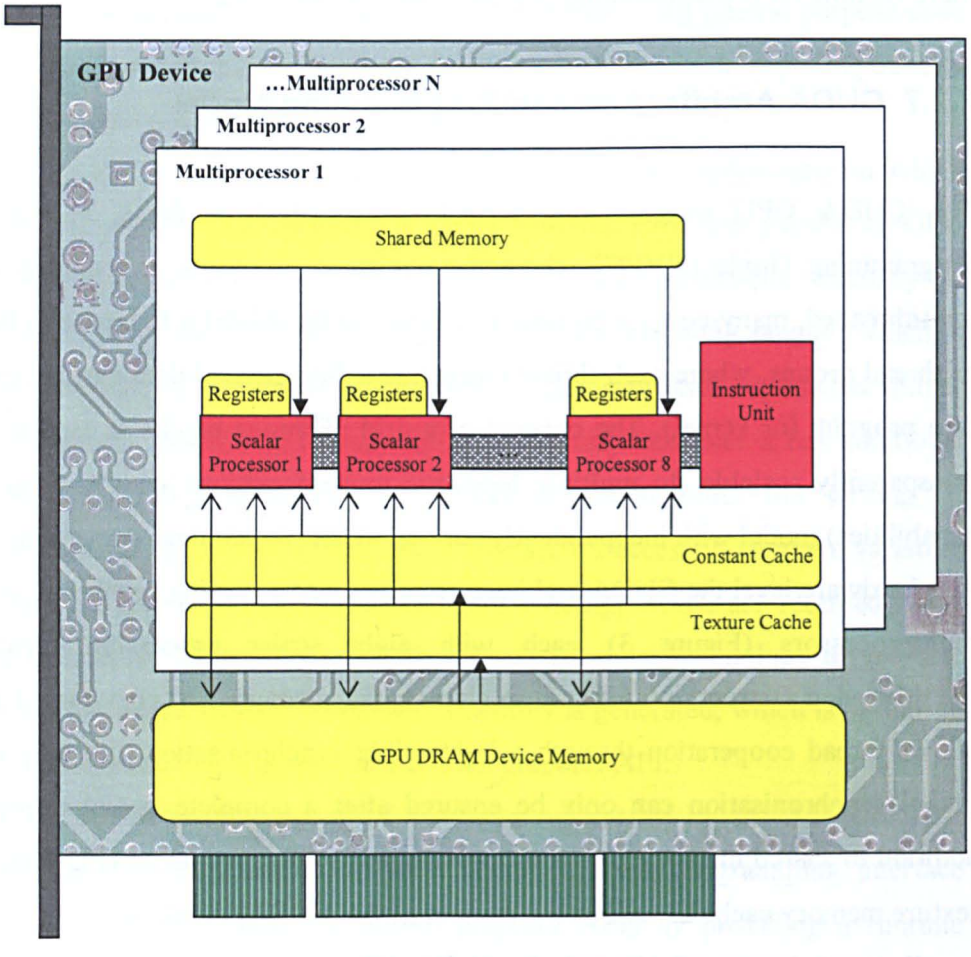



**Figure 3 - The CUDA architecture hardware model**

### 2.2.8 The Future of Data Parallel Programming

Stream SDK and CUDA represent major advancements in GPGPU programming, which highlights vendor commitment to providing support for general purpose usage of GPU hardware. Likewise, the competition between the two major hardware vendors has forced a significant push forward in GPGPU technology which is advantageous to users, as both try to establish themselves as the leading provider of GPGPU hardware. The significant disadvantage of this has been the unavailability of standardised tools for GPGPU which forces programmers to essentially choose between the two. OpenCL [ope09] is a recent development in beta stages, created by the Khronos Group (responsible for the standardisation of OpenGL and OpenAL), which tackles this problem head on. In addition to gaining support from both major GPU hardware vendors, OpenCL is supported by a huge number of industry partners, suggesting that it will become the new standard for data parallel programming not only on the GPU but on multi-core CPU and Cell-type architectures. Technically OpenCL is very similar to CUDA and uses an extension to the C language and the same model of a grid of thread blocks (with shared memory).

Whilst OpenCL seems to offer advancements in standardisation this has not effected the announcement of alternative GPGPU technologies. Far from abandoning CUDA, NVIDIA have announced continued support, with the suggestion of optional OpenCL output from the CUDA compiler. Likewise, Microsoft has announced support for GPGPU programming within DirectX 11 which introduces a number of new programmable stages to the graphics pipeline. Most import of these is the addition of a compute shader at the end of the pipeline, which can be for both post processing and general purpose usage. Despite being limited to Windows environments, compute shaders are sure to make a big impact, particularly within the games industry where physics libraries and advanced lighting effects are likely to gain performance advantages.

Finally, other emerging technologies such as Larabee [SCS+08], and the far more established Cell processor, are certainly worth consideration for data parallel processing. Whilst they both support their own APIs, both the technology manufactures are onboard with the OpenCL specification, making applications transferable between architectures.

## 2.3 Communicating Systems on the GPU

The use of the GPU techniques (described in the previous section) in accelerating computational algorithms covers a very broad spectrum. Despite this, the work presented within this thesis is, as far as the author knows, the only fully functioning ABM framework for the GPU. There are however numerous works which have been influential, or arisen during the development stages of this thesis. This section reviews this work which is mainly limited to particle systems, discrete modelling methods and the implementation of GPU assisted swarms and pedestrian populations.

### 2.3.1 Particle Systems

The particle system as an animation technique has been around since early 1960s video games, which used pixels to represent explosions. The formal introduction by Reeves [Ree83] as a method for modelling fuzzy objects such as fire, clouds and smoke was however important in demonstrating the effects in 3D computer graphics. Since then the GPU has been used in various instances to provide increasing complex particle behaviour. In the most simplistic case, a stateless, or Euler particle system, involves particles updating their output position through a system of closed form linear equations. This technique lent itself particularly well to shader model 1 series of graphics card, with only a programmable vertex stage. As each particle's position is calculated as a function of time, there is no storage requirement and hence could be implemented within a vertex shader in a single pass, with the particles being rendered as primitives such as points. The obvious advancement to this was made possible by the pixel shader, which allowed state preserving particle systems using off-screen rendering to write to persistent texture memory. Until recently pixel shaders supported outputting only a single four channel colour value, particle systems typically consisting of both a position and velocity had to be calculated using a multi-pass approach.

In addition to particles having a greater degree of variability (as a result of updating themselves dependant on their previous time step position and velocity), state preserving systems offered the ability to change throughout the scene by introducing collision detection and response. As a result of GPU assisted particle

systems Latta [Lat04] described the possible implementation of a million particle system with collision detection and response, even before the hardware was available to support it. Central to this was the use of Über-buffers (or Super-buffers), which arose as a method of uniting texture data with vertex arrays (due to architecture changes, both were physically stored in the same area of memory) without a slow CPU read back. This technique is demonstrated by work from Kolb et al [KLRS04] which is able to render one million particles interactively at 10 Frames Per Second (FPS) entirely on the GPU, using the CPU only for particle birth and death allocation. Since then there have been a number of advancements in hardware architecture which have been useful to particle simulations. The first of these, vertex texture lookups, effectively replaced the Über-buffer by allowing vertex programs similar functionality to fragment programs, in that they could use standard texture lookup to displace vertices. The rendering of particles in modern GPU implementations [KKKW05, SKKW08] involves simply rendering primitives for each particle in arbitrary locations, which are then displaced by a vertex program. The addition of multiple render targets also allows the position and velocity values of a particle system to be updated during the same rendering pass.

### 2.3.2 Lattice Based Communication Methods

After considering particle systems, where particles act as non-communicating individual elements, the next progressive step is to consider communication between particles. The simplest scenario to first consider is Cellular Automaton, which due to the relatively limited communication makes an ideal candidate for GPU implementation. Early work by NVIDIA SDK examples demonstrated the most famous CA example of Conway's Game of Life and a CA implementation of Madelbrot fractals. Since then more complex adaptations have emerged [Gre05]. One example of this is the use of CA for studying excitable media, and in particular, the study of Atrial Fibrillation [TJL04], a heart disorder, which uses the Gerhardt model, a CA with a variable neighbourhood interaction radius. Similar, is the coupled map lattice [HCSL02], which replaces the discrete state variables of CA by continuous ones, allowing the simulation of many physically based systems reliant on partial differential equations (PDEs). Harris proposes this technique as a method to animate

realistic real time clouds on the GPU, as well as demonstrating the use of the CML in boiling simulations. Further to this, the CML has been used for a wider range of simulations, including chemical reaction-diffusion. A computationally similar concept, the Lattice Boltzmann Method (LBM), has been used in computational fluid dynamics (CFD) as a macroscopic solution to the Navier-Stokes (NS) equations governing fluid flow [LWK03, WLMK04]. Central to both cloud formation and fluid flow simulations are the extension of 2D to 3D environments. Although graphics card hardware does support the use of 3D textures, it has emerged that stacking a number of 2D textures slices has proved more efficient. In boiling examples using the CML technique [HCSL02] (implemented on GeForce 4 hardware) 3D grids of up to 128x128x128 can been rendered on GPU hardware at a rate of 8 iterations per second, with 2D examples running substantially quicker than this.

### 2.3.3 Smoothed Particle Hydrodynamics

Where as latticed based methods allow the communication of discrete elements within the grid, Smoothed Particle Hydrodynamics (SPH) advances communication further by considering interactions between continuous valued particles in a continuous space environment. SPH were originally introduced by Ginhold and Monaghan [GM77] for the modelling of dynamic incompressible fluids, using normalised particle forces to enforce the NS equations by preserving momentum. SPH works by using a smoothing function (or kernel) over a particles variables, for a fixed smoothing length (typically referred to as $h$). This ensures that each of the particle variables can be calculated by using a weighted sum of its neighbouring particles. This weight can be determined by either a Gaussian function or a cubic spline function, for which particles weights are only significant for particles within a distance of $2h$.

Since its introduction, SPH has been adapted by the graphics community to simulate real time fluids [KC05, MCG03, KW06]. Aside from complex implementations using parallel sorting (Section 2.3.5) SPH methods implemented on the GPU involve partitioning 3D space into a number of discrete elements to avoid the explicit determination of the neighbourhood for each particle in the system. Kolb et al [KC05] describes a method which renders particles as point sprites to an off-screen 3D array (actually comprising of a stack of 2D arrays), where, for each pixel of the

point sprite rendered, the position of the particle in 3D space and slice space are stored. Sampling the 3D array is then implemented using linear interpolation both within and across slices, yielding a tri-linear interpolation scheme. Multiple Render Targets (MRTs) are used to improve performance and the 2D slices are expressed as sub regions in a larger 2D grid. Although this method is reasonably successful, rendering 2400 particles at 12 FPS (on a NVIDIA GeForce 6800 GTX), its performance is dependant on three factors; the smoothing kernel radius, the distribution of particles and the resolution of the discrete space grid. The discrete space grid is the most important of these restrictions, and as its granularity decreases memory consumption increases exponentially. In fact, to render 2400 particles requires 32768 spatial partitions (a 3D grid of 323), which, although is a resolution which is required to work correctly for particles in close proximity, wastes significant space where particles are sparse. A similar but alternative method presented by Müller et al [MCG03], uses an optimal space partition of $h$ (the SPH radii) to store particles' locations, ensuring only a fixed number of particles per partition. This in effect reduces the neighbourhood lookup to a O(nm) (m being the average number of particles per partition) by limiting the lookup to the 26 directly neighbouring partitions. Further to this, an additional 10x speedup is achieved by exploiting the efficiency of the GPU texture cache by storing particles objects directly in the partition grid, rather than storing references to their locations.

### 2.3.4 Continuum Methods for Pedestrian Modelling on the GPU

Pedestrian modelling techniques for the GPU have favoured a similar technique to that used for SPH. Treuille et al [TRE06] presents Continuum Crowds, a typical example of such a pedestrian modelling simulation, which uses a multi layer platform to perform collision avoidance, and more advanced agent behaviour. The multilayer platform comprises of an inter-agent collision detection layer, an environment (i.e. building) collision detection layer, a behaviour layer for more complex agent behaviour, such as waiting or turning, and a call back layer, which simulates agent to environment interactions. Agents then use the discrete multiplatform space to perform updates, by checking the cell values of their current occupied space and the cell they intend to move to (for collision detection). More precisely, updates are achieved

through the assertion of a number of hypotheses which state: each person is trying to reach a geographic goal, people move at the maximum speed possible and a discomfort field exists, which specifies that an object would like to move from point '*a*' to '*b*'. The third hypothesis is most important as it implies that agent movement and collision avoidance are the result of a using discrete valued lookup, in this case a it restricts the implementation scale by first introducing a high memory requirement with a large degree of redundancy, a problem shared by similar work [COU05] which used dynamic potential fields for modelling emergency situations. Additionally, any technique which uses discrete space to encode environmental and movement behaviours only approximates agent based dynamics, hence limiting the individual cognitive model of pedestrians.

### 2.3.5 Spatial Partitioning and GPU Sorting

In order to avoid the massive memory overhead of the storing particles, pedestrians or agents within a large finely grained discrete grid, a dynamic data structure is required which does not store particle data directly. Work by Kipfer et al [KSW04] implements such a system by discretely partitioning particles into spatially partitioned areas, called *bins*. Each spatial bin is associated a unique id which is then used as a sorting key to re-order the particle in O(log n) steps. With particles sorted in this way, each particle can then calculate interactions with neighbouring particles in the same spatial bin, by sequentially looking them up from surrounding positions in texture space. In order to improve further upon this, Kipfer uses a number of techniques which, although improve efficiency, pose a number of restrictions on particle interaction. Firstly, a suitably small spatial partition is used to reduce the possible number of particles to a maximum fixed number. Secondly, a separate collision detection and response module is implemented, which calculates only the single nearest particle and a response as a result of collision with this particle. Thirdly, as the described method sorts particles only into a single dimension, a number of staggered grids (one for each dimension) are used to detect any collisions with particles across neighbouring bins. Whilst this guarantees that the nearest partner is located, it does require that a separate sorted list for each staggered grid is maintained.

Whilst Kipfer's technique is successful at rendering up to a quarter of a million fully interacting particles at 31 FPS, the key to the algorithm lies within keeping the search stage entirely on the GPU. Not only does this exploit the nature of the GPU for fast computation, but it avoids the heavy overhead associated with performing a CPU read back in order to offload the search stage. The parallel sorting algorithm used within Kipfer's implementation was first introduced by Batcher [Bat68] and previously implemented on the GPU by Purcell et al [PDC+05] for the purposes of GPU based photon mapping. It essentially uses a similar technique to a parallel merge sort with *O(log n)* rendering passes to merge bitonic sequences into a final sorted list with an overall *O(n log n)* complexity. Whilst the implementation described by Kipfer does improve upon Purcell's method, by minimising the fragment based texture lookups and instruction count, there have since been a number of further improvements to bitonic parallel sorting.

GPUSort [GRH+05] and GPUTeraSort [GGKM06] use an improved bitonic network, i.e. an improved parallel comparison network for each rendering pass. This improved network offers two significant advantages despite the same *O(n log n)* overall complexity. The first of these is that the sorting network enhances the GPU cache memory hit rate, by increasing the number of lookups in close proximity. The second improvement is the use of GPU blending functionality to perform the sorting steps. Overall these changes balance the GPU much more than a pure fragment processor implementation. This is extended further by work from Greß et al [GZ06], whose GPU-ABiSort implementation improves further on cache efficiency for addition speed gains. Whilst GPU-AbiSort offers a near optimal solution for stream architectures, GPUTeraSort offers the significant advantage of a GPU and CPU balanced alternative. The result, which uses the CPU to perform key generation and a re-order stage, is described as a hybrid bitonic-radix sort and is able to sort billion record wide key databases by efficiently paging data in and out of the GPU.

More recently, the CUDA architecture has allowed the performance of GPU sorting to be improved through the introduction of parallel radix sorting [SHG09, LG07]. Radix sorting proceeds using multi passes, which consider each digit of the sort values in turn. CUDA makes this form of sorting possible through its ability to perform scattered writes. Likewise, the efficient use of shared memory and coherence of global scatters, allows for implementations which perform roughly 4 times faster than GPU sort. With respect to interacting particle systems, fast GPU sorting allows

inter particle interactions to computed efficiently, by considering the neighbours in each neighbouring spatial bin [Gre07]. Since its conception, this efficient CUDA technique has been adopted in many areas of application, including collision detection [LG07] and molecular dynamics interactions [ALT08].

### 2.3.6 N-Forces Interactions

Unlike inter particle collision techniques for systems with limited neighbourhood influence, N-forces represent an all pairs scenario where each individual continuously interacts with one another. Such all-pairs behaviour is common in astrophysical simulation, where gravitational influence of a star is a function of the distance between two objects (Equation 1).

$$F = G\frac{m_1 m_2}{r^2}$$

Where;

- $F$ is the magnitude of the gravitational force between the two point masses,
- $G$ is the gravitational constant,
- $m_1$ is the mass of the first point mass,
- $m_2$ is the mass of the second point mass, and
- $r$ is the distance between the two point masses.

**Equation 1 - Newton's Universal Law of Gravitation**

Although the very nature of all-pairs interactions provides a complexity of $O(N^2)$, GPU parallelism provides an excellent mechanism for computing pair wise interactions efficiently. Nyland et al [NHP04] provides an insight into N-body simulation using traditional GPGPU, which suggests that by using a grid of $N \times N$ to compute force interactions in parallel, parallel reduction could then be used to sum a total influential force which can be used in a final rendering pass to update the position of the body. The limitation of this technique is that graphics card hardware supports a maximum texture size of only 2048, meaning that if the red, green, blue alpha (rgba) components of the texture were each used to store a bodies mass, a maximum number of 8192 bodies could be simulated. Although this problem was addressed in work from Chinchilla et al [CHI04], which used a communicating network of N-body simulations, the constraint of $O(N^2)$ memory usage and sub

optimal Giga Floating Point Operations Per Second (GFLOPS) performance has encouraged a number of improved implementations.

Both Hamada et al [HI07] and Zwart et al [ZBG07, BZ08] present methods for simulating N-body simulations using CUDA to mimic special purpose gravity pipe hardware (GRAPE) [MT98]. Both implementations make use of CUDAs fast shared memory functionality and use a multiple time step scheme that evaluates longer range interactions less frequently, due to weak force interactions. When directly compared to the GRAPE-6 hardware, which they emulate, the performances are similar, with the GPU slightly outperforming GRAPE-6 and performing roughly order of magnitude faster than specialised CPU versions. For small $N$ (less than 512), Zwart reports that the GPU performance is worse than GRAPE-6, which is attributed to the expense of uploading across the GPU bus, however up to 9 million particles can be rendered using the GPU (obviously not in real time) over the maximum of 256k on the specialist GRAPE hardware.

Unlike both Hamada et al [HI07] and Zwart et al [ZBG07, BZ08] whose multiple time step scheme simulates the effect of long-range forces in a similar way to Barnes Hut far field approximation, Nyland [NHP07] presents an efficient method for brute force for all-pairs interactions using CUDA. Unlike his traditional GPGPU implementation [NHP04], limited considerably by the maximum texture size, CUDA's shared memory alleviates the requirement for N × N size communication space. The technique for implementing this instead uses tiling as a method of passing $2p$ (where $p$ is the tile size and $p^2$ is the total interactions per tile) body descriptions into shared memory registers, where serial operations with heavy data reuse push the processors to near peak performance. Nyland uses a number of additional optimisations to improve the performance, which include loop unrolling (the replacement of inner loop calls with an increased number out of loop calls) to reduce loop overhead and optimised tile sizing, which increases the size of $p$ to reduce memory traffic to the maximum possible value that does not cause idling of any of the 16 GPU multiprocessors. The result of the brute force technique, although not directly comparable to either of the multiple time step methods, demonstrates a two times greater performance with respect to pair wise force interactions per second. The overall comparison to highly tuned serial implementations suggests a 50 times speed up, with a staggering 250 times speed up over a standard C implementation.

### 2.3.7 Real Time Agent Based Models on the GPU

Although the previously described methods do not directly describe the implementation of ABM on the GPU, the communication between individual elements (either particles or gravitational bodies) is analogous with that of an AB system. Erra et al [EDCST04, CES06] describes an implementation of GPU ABM, which influenced by real time SPH, partitions space and uses a sorting algorithm to assign individuals to spatial cells. As with most other implementations of an ABM using the GPU the authors have favoured a mixed CPU and GPU approach, using the CPU for the calculations of local perception. The reasons for this are mainly due to the advantage of being able to serially store a variable sized list of agents, making local interactions a case of simply looking up the agents within its own cell and local cells. Whilst the lack of serial random access memory prevents this technique from being directly translatable to the GPU, it does remain the weakness of this method due mainly to the expensive CPU read back (and further data transfer to the GPU). In order to combat this Erra proposes an approximation method, in which each individual writes to a 'scatter matrix' (during the CPU stage) with a value depending on its likelihood to change spatial cell. This matrix is then used to determine how uniform the flock is and as a result avoid calculation of local perception for each frame. Using the approximation method this technique allows 8000 agents to be modelled at up to 20 FPS.

Work by Reynolds [Rey06] demonstrates how the Play Station 3's (PS3s) Cell Processor [PHA05] can be used to efficiently render crowds of fish. The PS3 architecture is somewhat different to that of traditional GPU or CPU, and although a NVIDIA RSX5 card is available for graphics processing, the PS3 contains an additional IBM Cell Microprocessor capable not only of running an entire OS but is also able to provide parallel processing functionality. The Cell Processor consists of a single Power Processing Element/Unit capable of scheduling eight Synergistic Processing Elements or Units (SPU) with a high bandwidth (25.6 GBytes/sec) 'Element Interconnect Bus' (EIB) connecting the two. Reynolds uses the architecture to batch a number of spatial buckets of fixed array size to the SPUs, which in turn calculate the nearest N neighbours for each individual in the bucket, by considering

[5] The RSX architecture is comparable to the NVIDIA 7900 series

neighbouring buckets through communication across the EIB. Although direct comparison with GPU methods is difficult, the technique which is similar to that of Quinn [QMHz03], which uses the Message Passing Interface (MPI) on a multi processor machine, is able to render up to 10,000 individuals at 60 FPS. It is also suggested that further work to reduce the fixed bucket size and support for networked Cell processors would greatly improve this.

In a much simpler scenario than ABM on the GPU, Rudomín et al [RMH05] use texture space as a discrete agent map, where each pixel is capable of holding a single agent. Whilst technically this approach is similar to lattice based methods, Rudomín extends earlier work using image maps with Extensible Markup Language (XML) scripting [RMH+04] to introduce fragment shaders as a mechanism for implementing state machines. More specifically this involves representing FSMs as a table, representing state verses input with a mapping to the next state (and position). Although it is suggested that hierarchical or layered FSMs could be implemented, a simple predator prey example is described demonstrating interactive frame rates (34 FPS) for up to a million agents. Although algorithmically this is inferior to much of the work described in the report it demonstrates a novel attempt to separate modelling logic from GPU programming.

D'Souza et al. [DLR07, LD08] presents the most complete example of ABM on the GPU, describing a system which utilises discrete partitioning, with agents directly scattered into discrete spatial partitions. The discrete partitioning nature of the algorithms is memory intensive and limits the agent environments to fine grained 2D or course grained 3D. As the discrete partitions are increased in size, the memory requirement is reduced. However the likelihood of collisions (multiple agents scattered within the same partition) increases. Collisions are addressed through the implementation of a multi pass priority scheme, but this is only guaranteed to succeed if a separate render pass is used for each possible collision (with a 2D agent range of 9x9 there are in total 81 possible collisions to resolve). Whilst it is suggested that much fewer collision passes can be used, the reliability of this, as with the convergent random iterative scheme used for birth allocation, is at worst unpredictable. Additionally, little consideration is given towards agent specification or more general agent systems, such as those that exist within spatially distributed, continuous 3D environments.

## 2.4 Summary

Whilst there has been some limited research into the use of the GPU for ABM, these are limited to either, fixed implementations of swarms or systems, or systems dependant upon a mapping to discrete space to resolve interactions, such as collision avoidance. For an implementation to allow the modelling of continuous agent based dynamics, particle based methods [KSW04, Gre07], utilising parallel sorting and all-pairs N-Body methods [NHP07], offer a more appropriate solution. In addition to interaction techniques, there is an obvious absence of frameworks which allow non graphics specialists to take advantage of the performance of GPU hardware. Although some basic attempts have been made to resolve this [RMH05, RMH+04], these are extremely limited and unsuitable for the specification of complex interacting agents.

# Chapter 3
# GPGPU Swarm Modelling

The previous chapter reviewed the use of the GPU for ABM, where it was concluded that only limited ABM on the GPU has been demonstrated through specific examples. This chapter describes the implementation of an AB framework for swarm modelling on the GPU (ABGPU). It represents the first step towards the development of an ABM on the GPU and addresses two issues, which were raised by reviewing literature;

- A flexible approach to ABM on the GPU, which allows non GPU specialists to take advantage of the GPU's parallelism through a simple agent specification technique; and
- The use of the GPU to build a grid based data structure through parallel sorting, which allows limited range agent interactions without the reading back to the CPU. A similar technique for particle collision detection within

CUDA [Gre07] was reviewed in the previous chapter and was developed independently to the GPGPU alternative which is presented.

ABGPU uses a translational technique to map both agents and behaviours to the GPU, using OpenGL and the Cg shading language. A 2D grid is used to store both agent data and process agents within the fragment processor. Flexibility is achieved through an API interface for model specification and simulation control, with an agent behaviour script to determine an agent's behaviour. Simulations consist of a single agent type and a homogenous population, each with a defined set of internal memory variables and a single agent update script. Agents communicate by iterating through a list of agents within a specified interaction radius during the update stage.

## *3.1 Implementing Agent Based GPU*

This section discusses the implementation of the ABGPU API and demonstrates its use.

### 3.1.1 Agents and Data Mapping

The mapping of agent data to the GPU is a process which it is important to abstract as it hides the complexities of the underlying data storage. This is achieved through a translation function 'F' which provides a mapping for agent variables in agent function scripts. It also facilitates the getting, and setting, of initial agent data (through the API) into a number of 2D stacked 32bit floating point textures (agent space). Similar to previous work on particle systems [Lat04], a 1D list of variables is easily translated into 2D texture space with a 2D position *(i, j)* in each stacked texture, '*t*' representing an individual set of data. As texture access is read/write only, '*2t*' textures are required in total, with data being stored in up to all four of the red, green, blue and alpha colour channels respectfully (Figure 4). Within OpenGL the Frame Buffer Object (FBO) extension allows simultaneous writing to MRTs, on the implementation hardware (a NVIDIA GeForce 8800 card) up to 8 targets are supported, giving a total of 32 agent variables. For a non communicating system of agents, simulation is achievable by performing $N$ parallel operations by rendering a

single quad primitive, as described in Section 2.2.4. Assuming the quad primitive is the same dimension as agent space (which due to the texture format used, is limited to power of 2 dimensions) and rendered from an orthogonal perspective, rasterisation will invoke a parallel operation for each of the *(i, j)* agent positions.

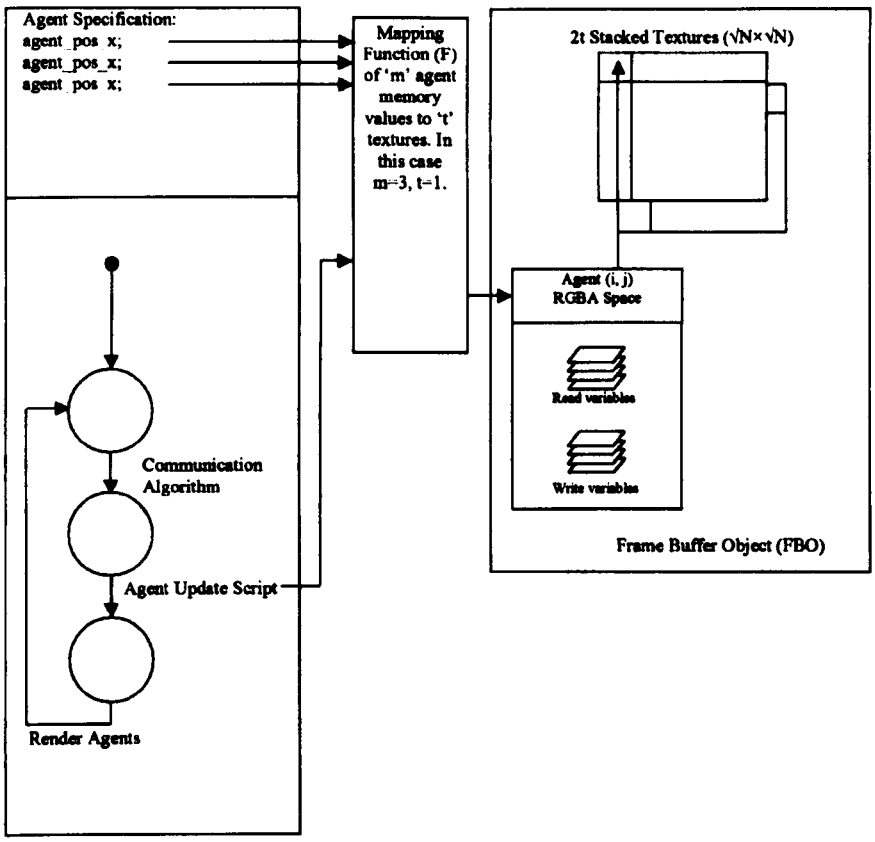



**Figure 4 – The mapping of an agent specification into agent space at position 'i, j'.**

### 3.1.2 Specifying a Simulation using ABGPU API

Specifying a swarm simulation requires a number of C++ classes, which handle the allocation and uploading of data to the GPU as well as iterating the simulation. The basic functionality of these classes is described below, where examples are provided for clarity. More advanced use of ABGPU for visualisation, encoding global map data and real time feedback is addressed separately within this chapter.

**`AgentSpecification`** - An agent specification object is used to specify the internal memory of an agent by passing through a list of agent variable names.

The object determines the number of textures required to store each of the agent variables and provides the mapping between variable names and texture space. Agent specifications currently support up to 32 internal agent values on a DirectX 10 series card with 8 MRTs or alternatively 16 values through 4 MRTs on older DirectX 9 cards. The example below shows an `AgentSpecification` object being used to specify an agent with three variables.

```
AgentSpecification* as = new AgentSpecification();
char* variables[] = {"x", "y", "z"};
as->setAgentVariables(3, variables);
```

**`Agent`** – An agent object is used to get and set individual agent variables through a get and set method respectively. When an agent is first created an agent specification must be supplied which is used to provide the agent variable to texture space mapping described in the previous section. The example below shows the creation of N agents using the `setAgentVariable` method to set the agent's data.

```
Agent* a[N];

for(int i=0; i<N; i++){
  a[i] = new Agent(as);
  a[i]->setAgentVariable("x", randUniform());
  a[i]->setAgentVariable("y", randUniform());
  a[i]->setAgentVariable("z", randUniform());
}
```

**`AgentPopulation`** – The `AgentPopulation` class is responsible for the specification of global variables, uploading and translation of the agent update script into compliable Cg code and the processing of simulation steps. The names of global variables are set by passing an array of variable names to an `AgentPopulation` object. Once the names have been set the global variables themselves can be specified as in the example below where `N`

represents the agent count, `ENV_size` represents the upper bound and lower bound (`-ENV_size`) size of the environment and `COMM_radius` indicates the range which agents communicate over.

```
AgentPopulation* population =
        new AgentPopulation(as, N, ENV_size, COMM_radius);
//set global variables
char* global_variables[] = {"STEER_SCALE"};
population->setGlobalVariables(1, global_variables);

population->setGlobalVariable("STEER_SCALE", steer_scale_value);
```

Agent data is uploaded by passing an array of agent data to an `AgentPopulation` object. The below example shows this, in addition to the setting of a plain text agent update script and the stepping of the simulation 'S' times. Alternatively, the simulation can be stepped a single time by using the `step` method.

```
population->uploadAgentData(a);
population->setAgentUpdateScript("update.abgpu");
population->stepN(S);
```

### 3.1.3 Scripting Agent Behaviour

During the agent update stage, behaviour is specified though the use of an agent function, which uses a C-like scripting syntax. The agent function, which is always called `agentMain`, represents a function which is applied to every agent in parallel and therefore accepts a single agent instance as its first argument and returns a single agent instance representing the updated agent. This agent instance is specified using a C structure, which contains a variable for each agent memory variable, specified in the API's `AgentSpecification` object. Within the agent function, agent variables can be accessed directly from this agent structure as each reference to a variable is later mapped to a texture space using the `AgentSpecification's` mapping function. In addition to a single agent instance, the main agent function also accepts a `GLOBALS` structure containing each of the global variables specified in the

API's `AgentPopulation` object. As with agent data these can be accessed directly and are later translated into GPU memory space. With respect to communication, agent scripts use two placeholders `FOR_EACH_AGENT_A`, and `END_FOR_EACH` as place holders for the communication algorithm (discussed in the next section), which efficiently iterates agent data within the interaction range. Between these two placeholders the agent function may make reference to an agent structure 'a' that represents an agent within the population (excluding the current agent instance) which is being iterated. Figure 5 shows an example of an agent update script which corresponds with the agent specification in the previous section.

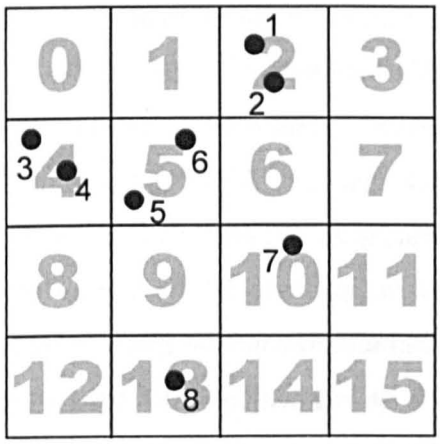
```
struct agent{
 float x;
 float y;
 float z;
};
struct globals{
 float STEER_SCALE;
};

agent agentMain(agent IN, globals GLOBALS)
{
 float average_position_x = 0.0f;
 float average_position_y = 0.0f;
 float average_position_z = 0.0f;
 float count = 0;
 //Iterate agents
 FOR_EACH_AGENT_A
 {
  average_position_x += a.x;
  average_position_y += a.y;
  average_position_z += a.z;
  count += 1;
 }
 END_FOR_EACH
 //Calculate average
 if (count > 0){
  average_position_x /= count;
  average_position_y /= count;
  average_position_z /= count;
 }
 //Move agent
 IN.x += average_position_x * STEER_SCALE;
 IN.y += average_position_y * STEER_SCALE;
 IN.z += average_position_z * STEER_SCALE;

 return IN;
}
```

**Figure 5 – A simple agent script using ABGPU scripting.**

### 3.1.4 Agent Communication

Agent communication over a limited range requires a spatially partitioned data structure, which keeps track of all agents located within it. Agent communication can

then be achieved by consulting all agents within an agent's own and neighbouring partitions and using a radial test to see if the agent is within the interaction radius. On the CPU this task is trivial and each agent in a partition can be stored as a linked list. On the GPU this is significantly more difficult, as dynamic storage sizes are unsuitable for parallel computation. The spatial partitioning implementation in ABGPU is achieved by first generating a unique sort identifier (based on the partition an agent is located within) for each agent, along with a pointer to the agent's position in 2D (agent) texture space [KSW04]. This identifier is then used to sort the pointers, which are mapped to a 2D texture, and reorder the agent data in order to increase the cache hit rate during later stages. In early implementations of ABGPU a simple bitonic sorting algorithm based on Purcell's [PDC+05] sort routine was used. A modified version of GPUSort [GRH+05] has since been adopted and improves upon the performance of the sorting stage significantly. This has been adapted for use within ABGPU by altering the comparison routines which were previously only suitable for unique key identifiers.




| Partition | First agent | Last agent |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 1 | 2 |
| 3 | | |
| 4 | 3 | 4 |
| 5 | 5 | 6 |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | |
| 10 | 7 | 7 |
| 11 | | |
| 12 | | |
| 13 | 8 | 8 |
| 14 | | |
| 15 | | |

**Figure 6 - An example of a 2D partitioned space (gray) containing sorted agents (black). The matrix to the right holds the first and last message in the partition which is used to iterate messages. For example, an agent in partition space 10 iterates 9 partitions (5,6,7,9,10,11,13,14,15) to ensure all agents (5,6,7,8) within the potential interaction range are examined.**

Agents are able to perform a linear search between agents in the same partition by iterating to the left and right of their virtual (all data is physically stored in 2D textures) 1D linear position in the sorted agent list, until the sort identifier changes. In order to communicate with agents within the interaction range, but contained within a

neighbouring partition, the start and end position of agents within the sorted list must be calculated and stored in a matrix (which is referred to as a partition boundary matrix). This then allows the same linear search to take place in all neighbouring partitions (26 in a 3D environment and 9 in a 2D environment), guaranteeing an agent communicates with every other agent within its interaction range (a simple 2D case is presented in Figure 6 above).

The method used for dynamically generating the start and end position of each spatial partition requires a scatter operation. As scattered write support is unavailable within Cg shaders, this is achieved by rendering N (where N is the agent population size) point primitives positioned in a regular grid into a viewpoint with an output size that matches the partition boundary matrix. Vertex texture fetching then allows each point primitive to lookup an agent sort value, which is compared to the previous sort value to find the start of a spatial boundary. The same vertex program is also used to perform the linear search to find the end of each spatial boundary, which avoids the process being repeated for every agent who examines the partition. Once found, both the start and end position of agents within a partition boundary are scattered into the partition boundary matrix by changing the point primitives location and assigning the agent's position to a multi-texture semantic. Although the output size must be equal to the partition matrix size (which ensures a fragment operation is invoked for each), the storage mechanism for the matrix does not necessarily require a 3D texture for 3D simulations. As 3D textures have limited hardware support with no significant performance advantage, a 3D position is instead mapped into 2D space and stored in a large 2D texture. For an interaction radius '`i`', in an environment space '`e`', between `0` and `1` all partition boundary values can be stored within a 2D texture of size `ceil(`$\sqrt{(1/i^3)}$`)`. This provides over 4 million total partitions when the maximum 2D texture size of 2048 is used, and considerably more on newer GPU devices. Figure 7 demonstrates the complete steps of the algorithm including the agent update. Texture space inputs and outputs are indicated between each stage.

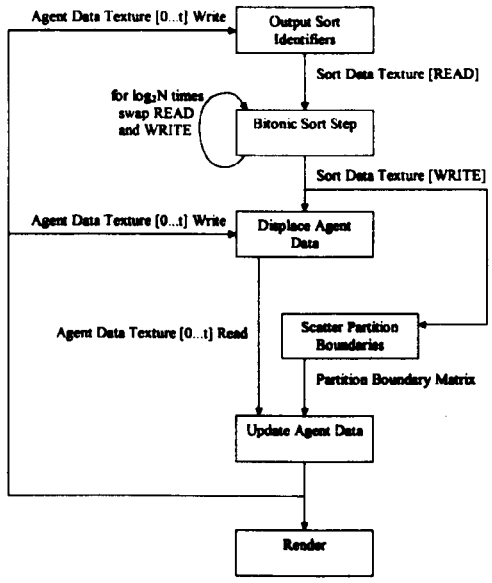



**Figure 7 - Render passes and data bindings for a single update step of ABGPU.**

### 3.1.5 Visualising Agents

ABGPU provides inbuilt visualisation, which allows agents to be rendered as simple OpenGL points using the cross platform Glut library[6]. This can be used to provide visualisation by including the `PointVisualiser.h` header file and by invoking the initialisation function (`initPointVisualiser`) and entering the main glut rendering loop (`glutMainLoop`). In addition to this, two techniques extend the basic rendering to provide more advanced representation of agents. The first of these (Figure 8) uses the same technique as the inbuilt primitive point rendering, where instead of a point, a low polygon count model for each agent can be rendered into a single display list. In this case each triangle primitive is rendered at the origin with multi-texture coordinates reflecting an agent position in (i, j) agent space. A vertex shader then looks up agent positional and velocity memory variables, which are used to translate and rotate the object accordingly. As an entire population represented as simple objects contains relatively few OpenGL draw calls in total, the entire population can be stored in a single display list. For more advanced agent representations, where individual model sizes become much larger, this technique quickly becomes unsuitable. In such cases it is necessary to store only a single model

[6] http://www.opengl.org/resources/libraries/glut/

representation in a display list. The single display list is then called for each agent with the multi texture coordinate value set before the display list is called, allowing each instance of the model to be translated by a differing set of agent values.

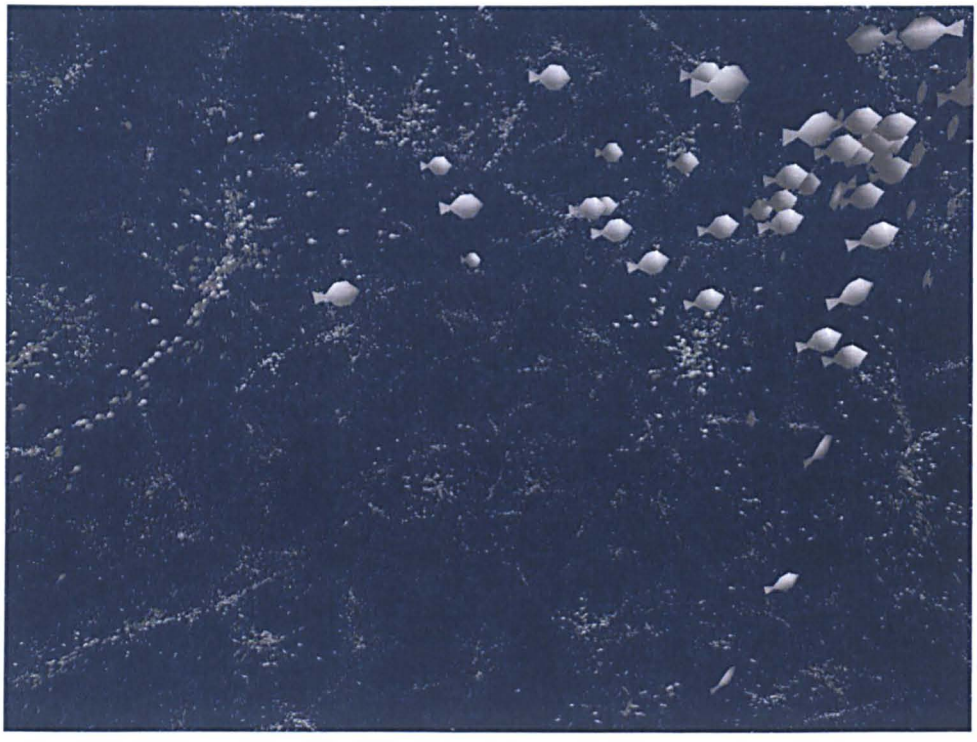


**Figure 8 - 65,536 Interacting fish at 30 Frames Per Second.**
**Agents are rendered using a single large display list and a simple model containing 66 faces.**

With the previous technique every individual of the population is rendered with the same detail level. A more suitable technique is to therefore apply a Level of Detail (LOD) rendering system which varies the agent's fidelity, depending on distance to the viewer. This is achieved through the use of a generalised feedback system, available for retrieving data about the agent population without CPU read-back from the graphics card. Parallel reduction is used to reduce values in agent space to singular values for a number of common reduction functions such as minimum, maximum, sum and count. For the purposes of a LOD system is it required that the total number of each detail level is known. An agent variable therefore is used to hold a LOD level and is calculated during the agent update stage. A reduction function for each detail level then uses a filtered count function, counting only the number of occurrences of the specified detail level. This is demonstrated below, where three

`FeedbackVariable` object instances are used to count the specific number of occurrences of the value specified by the `feedbackCountN` variable.

```
//Feedback
FeedbackVariable lod1, lod2, lod3;
lod1.feedbackType = FEEDBACK_COUNTN;
lod1.feedbackVariable = "lod";
lod1.feedbackCountN = 1;

lod2.feedbackType = FEEDBACK_COUNTN;
lod2.feedbackVariable = "lod";
lod2.feedbackCountN = 2;

lod3.feedbackType = FEEDBACK_COUNTN;
lod3.feedbackVariable = "lod";
lod3.feedbackCountN = 3;



FeedbackVariable feedback_variables [] = {lod1, lod2, lod3};
AgentFeedback* feedback =
        new AgentFeedback(3, feedback_variables, population);

float3 lod_counts;
feedback->getFeedback(lod_counts);

population->setDistanceSortVariable("lod");
```

After the parallel reduction is complete, the agent data must then be sorted according to the LOD levels (shown above). This ensures that when calling a display list for each detail level, the number of times reported by the feedback step matches the detail levels to the agent data. Whilst this technique is computationally more expensive due to the secondary sort, it allows a massive reduction in rendering overheads when high resolution models are required. This technique is demonstrated in Figure 9 and Figure 10 and shows fish and pedestrians respectively, coloured by their corresponding LOD. Additionally, the same technique can be applied to achieve variance in agent representation. In this case, varying agent models, each with an associated value, are used within the population with the value used as feedback and the sorting key. This

also allows simultaneous LOD rendering, as long as each LOD and unique representation combination has an associated identifier and display list containing the draw calls.

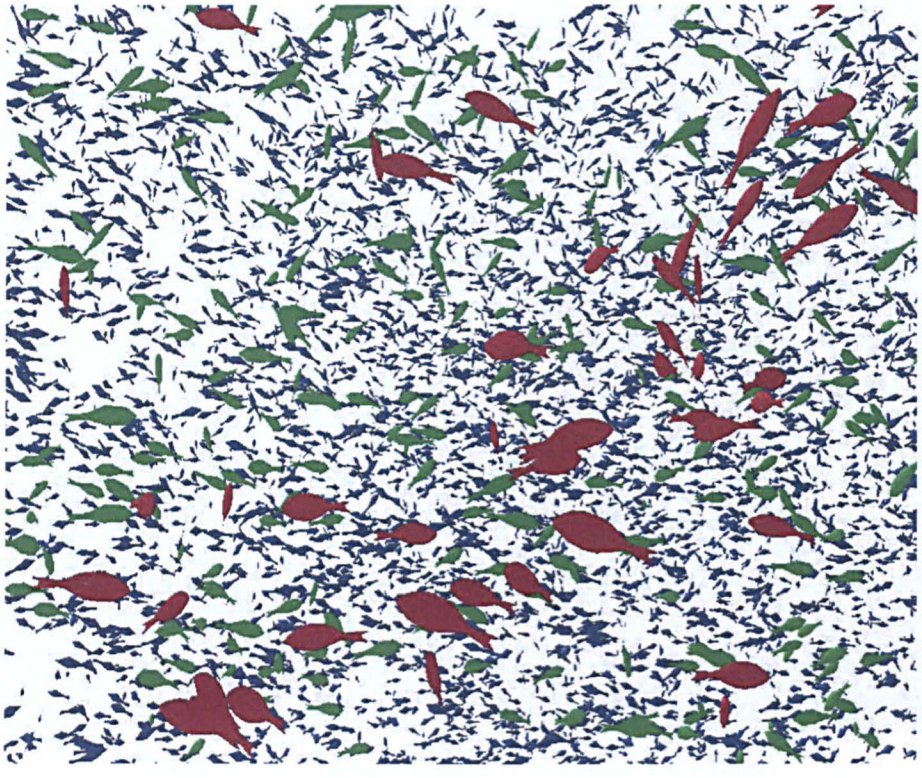


**Figure 9 - A population of 16,384 fish agents rendered with the LOD system.**

**Figure 10 – 65,000 fully interacting agent based pedestrians rendered by LOD level.**

## 3.2 Case Studies

In order to evaluate ABGPU with respect to performance and flexibility of modelling swarm systems, both a Boids flocking simulation and pedestrian dynamics model have been implemented. In both cases the results are obtained from a single PC, with an AMD Athlon 2.51 GHz Dual Core Processor, 3GB of RAM and a GeForce 8800 GT. As the performance of any swarm model 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 described. A more general comparison with more diverse approaches for swarm modelling is considered later in the discussion section of this chapter.

### 3.2.1 Implementing the Boids Model in ABGPU

The Boids model, used to demonstrate the functionality of ABGPU, is adapted from Reynolds' original model [Rey87] with the introduction of a *goal rule,* implemented using global variables. The agent specification consists of seven variables, an 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, also bound to a maximum threshold to preserve a maximum speed within the simulation. Animation of the fish agents 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. Finally, a

simple menu system is used to control the rule weights, which in turn affects the behaviour in real time.

### 3.2.2 Evaluation of the Boids Model

Figure 11 shows the recorded performance of the Boids model in ABGPU using a number of communication radii, in an environment clamped between the range of 0 and 1 and no visualisation. The FPS readings were obtained after the simulation had run for some time, allowing any local groups to form and to avoid the influence of the Boids initial random positions and velocities. The performance results are capped at 200 FPS maximum and each communication radius is only demonstrated to the largest population able to sustain over 5 FPS. With an interaction radius of between 0.03125 and 0.0078125, it is possible to simulate 65,536 Boid agents at 60 FPS. Erra et al. [EDCST04] simulated the same number of agents at less than 5 FPS. The results shown indicate that ABGPU is able to render up to a million agents at 5 FPS without visualisation. It does, however, have 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. Likewise, Erra [EDCST04] included interaction with 5 static scene objects and one dynamic one, which currently are not included in ABGPU performance tests. Such behaviour could however be incorporated through the use of global variables.

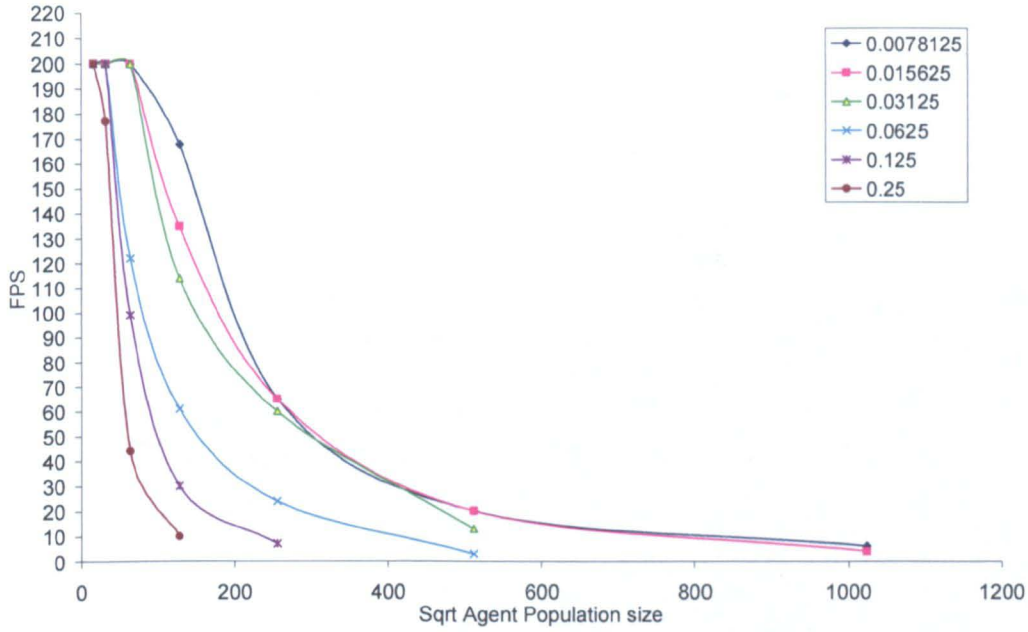



**Figure 11 - Recorded performance of various communication radii. FPS were recorded over multiple frames.**

Adding point primitive and low polygon count visualisation to the simulations described above has very little effect on the performance of most population sizes. In most cases a maximum of 5-10% drop in frame rate is observed, however, as population sizes begin to exceed 16,384 a more substantial performance penalty is reported. In fact, with a communication radius of 0.03125, 65,536 agents can be rendered as simple fish sustaining 30 FPS (Figure 9) and as point sprites at 50 FPS. In contrast, this is substantially more than the reported performance of previous hardware assisted Boid's models, the most impressive being Reynolds [Rey06] reporting 10,000 agents at 60 FPS. When rendering using higher resolution agents with a LOD system, the performance is obviously reduced due to the additional feedback and sorting stages performed before rendering (discussed in more detail in the evaluation of the pedestrian model). Despite this, 16,384 agents with a maximum detail level of 1,500 polygons can be rendered at over 30 FPS (Figure 12). The majority of the performance slowdown is attributed to the secondary sort, rather than the LOD feedback which makes little overall performance difference.

**Figure 12 – 16,384 agents with a maximum detail level of 1,500 polygons rendered at over 30 FPS.**

### 3.2.3 Agent Based Pedestrian Dynamics in ABGPU

In order to evaluate the performance of ABGPU within a 2D environment, a pedestrian dynamics simulation has been implemented. The decision to do so is influenced by the requirement of massive scale pedestrian models required in large scale serious games. The pedestrian behaviour itself is influenced by both Reynolds' work [Rey99] and Helbing's social forces model [HM97], with the exact rules being somewhat of a hybrid. The following equation (Equation 2) is able to describe the force exerted on each pedestrian during the update stage, for all examples within this chapter.

**Equation 2 - Helbings social forces model.**

$$F_i = R_i + Cr_i + G_i + M_i$$

The total force (expressed as a two component x and y vector) exerted on each agent $F_i$, is the result of a social repulsion force $R_i$, a close range interaction force $Cr_i$, a short term goal force $G_i$ and an environmental force $M_i$. This force is then used as a

directional steering force to make some change to the pedestrian internal velocity. This altered velocity is then checked to ensure it does not exceed the pedestrian's maximum velocity, and if required the velocity is normalised accordingly. The social repulsion force gives preference to events in the direct line of sight (as in Helbing's [HM97] model). In Equation 3, the symbol λ demonstrates this preference by representing a scalar value indicating the size of angle between the pedestrian i's line of sight and pedestrian j's position. As in Reynolds' work [Rey99] agents are given a limited vision, which filters agents outside their field of view. Equation 3 describes the force $R_i$ where the letter $j$ represents only agents from within the limited vision filter.

**Equation 3 - Equation describing the force applied to pedestrian agents with limited vision**

$$R_i = S\sum_{j}^{0}\left(\frac{D\lambda_{ij}}{(|P_i - P_j|)^2}\right)$$

The static value S indicates a scalar value controlling the global influence of the social repulsive force. The positions $P_i$ and $P_j$ represent the vector positions of agent $i$ and $j$ respectfully, and the distance between them represents the directional force vector between the two agents. This repulsive force is scaled by the inverse square of the distance between the two pedestrian agents. The value $D$ is used to scale the effect of the inverse distance effect, and within the examples presented in this chapter, has been adjusted depending on the interaction radius between agents. Unlike the social repulsive force, the force $Cr_i$ is independent of the direction between agents. Its influence is over a far smaller radius and rarely has affect, unless there is a high concentration of agents, in which it acts mainly as collision avoidance. The force $G_i$ acts as an influence towards a specific goal point in the environment. In the case of a random walk the goal position is directly in front of the pedestrian encouraging them to follow their current path. It is however possible to integrate longer range navigation by using a global variable array to hold information about the environment.

A global variable array (or map) is defined as a standard global variable (Section 3.1.2), however its value is set to an instance of a `GlobalVaribleMap` object as demonstrated below.

```
GlobalVariableMap* map = new GlobalVariableMap(array_size);
```

```
for (int i=0; i<array_size; i++){
 float4 map_data = {r, g, b, a};
 map->setMapDataValue(i, map_data);
}

p->setGlobalVariableMap("global_map_variable", map);
```

Using this technique it is possible to encode an environmental force field [CM05], which can be used to direct agents away from static obstacles. This avoids the expensive computational process of comparing each agent and obstacle combination as described in Helbing's original model [HM97]. Global array maps are stored on the GPU in 2D textures and as a result up to 4 float values (r, g, b and a above) can be stored in each data map value (shown above). As the granularity of this environmental force field texture is independent from agent interaction, large grained force fields can be used, providing they have sufficient detail to capture the smallest static obstacle. For the purposes of encoding a force field, the first two components of the array (red and green) are enough to hold a directional velocity. The remaining components in the experiments presented have been used to store either a greyscale image value or an identifier used to zone the environment into unique areas. Figure 13 demonstrates a complex force map representing the Peace Gardens area of Sheffield city centre. The results of a pedestrian simulation compared to satellite imagery are also show in Figure 14.

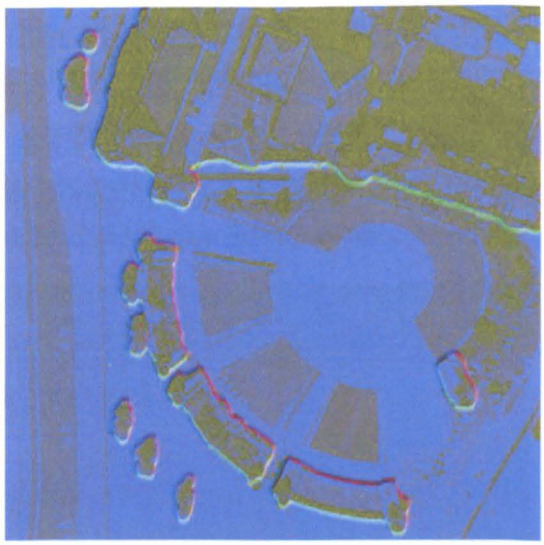


**Figure 13 - A Force Map encoded into the red and green channel of an image, representational of Sheffield Peace Gardens.**

**The satellite image is stored as a greyscale value in the blue channel.**

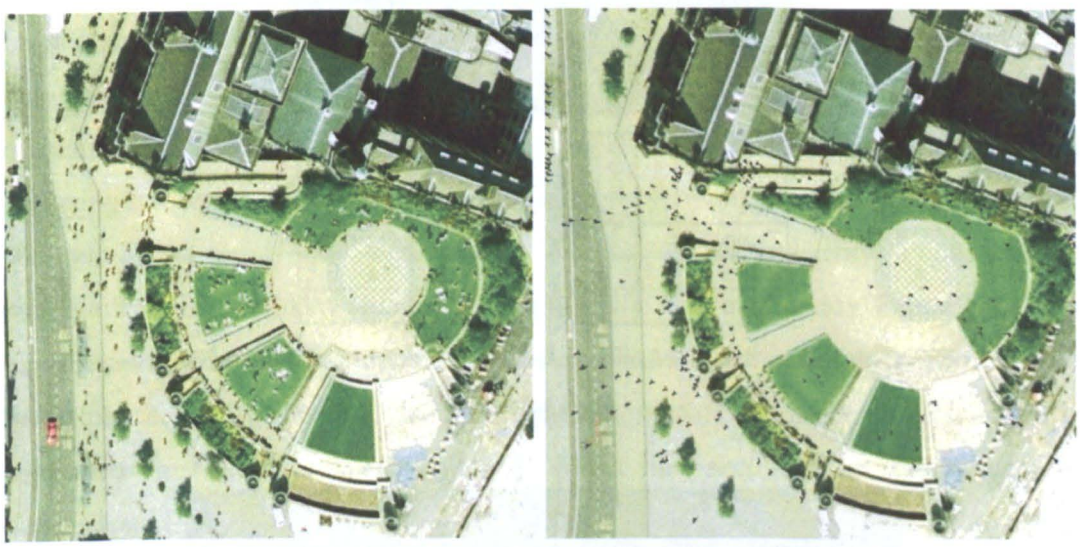
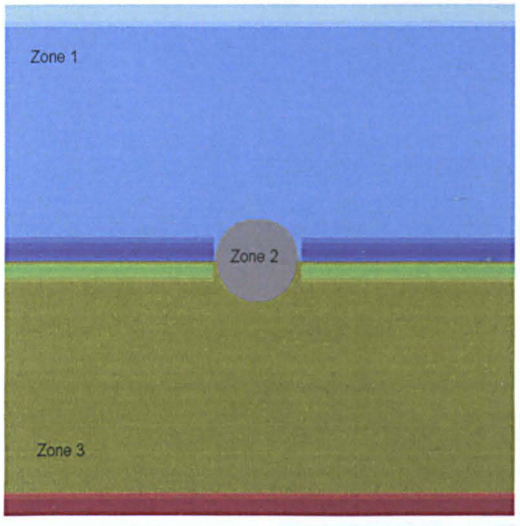


**Figure 14 - Satellite imagery of Sheffield Peace Gardens (Left) and the ABGPU simulation (Right).**

Zoning the environment into unique areas offers the possibility of simulating long range path planning. This is achieved through the use of three global variable maps as show in Figure 15. Within Figure 15, the top image represents an environment, which contains a narrow opening between two zones (1 and 3). The global variable map shows the force field values (red and green channels) as well as the zone value (blue). The two additional 2D data sets encode, firstly, a navigation lookup grid, which indicates the next zone to move into in order to reach a particular long range zone (horizontal). Secondly, a set of data is required to store an x and y (goal) point value

for each of the zones. These are used to determine a point of interest which the agent will move towards. The second zone in this environment ensures that pedestrians intelligently pass through gateways avoiding a situation where there desired path is blocked. This is shown in Figure 16, which contains roughly 2,000 pedestrian agents.




| | Zone 1 | Zone 2 | Zone 3 |
|---|---|---|---|
| Zone 1 | - | 2 | 2 |
| Zone 2 | 1 | - | 3 |
| Zone 3 | 2 | 2 | - |

| | Goal X | Goal Y |
|---|---|---|
| Zone 1 | 0.5 | 0.0 |
| Zone 2 | 0.5 | 0.5 |
| Zone 3 | 0.5 | 1.0 |

**Figure 15 - A simple illustrative zoned environment encoded into an image, with corresponding data tables.**

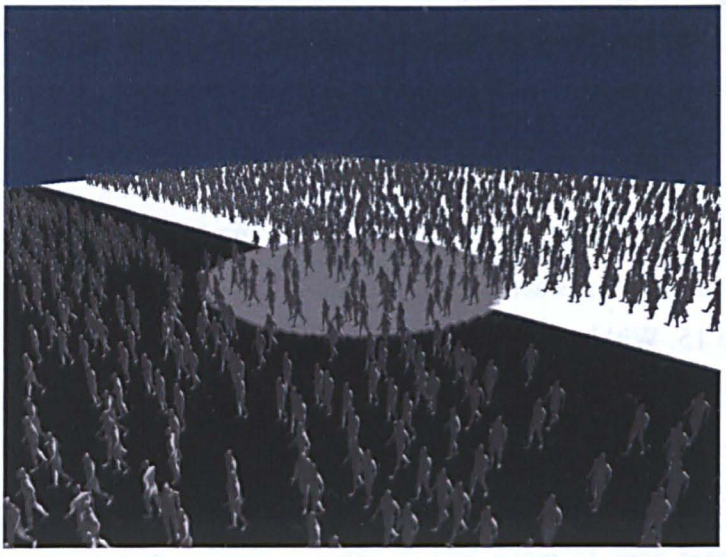


**Figure 16 - Zoning around a congestion point where Black to White boundaries represent walls.**

As with Boids modelling, animation has been implemented for pedestrians as they move around the environment. Key-framing provides an ideal animation technique as it is computationally cheap and is easily capable of representing simple human locomotion. Through experimentation it is evident that reasonable walking behaviour can be achieved through interpolation between only two key frames. For improved fidelity multiple key frames can be used, however as all draw calls are stored in a single display list, it is necessary to store the positional and normal information for each key frame model in the list. Whilst this has a visually improved effect on animation of close pedestrians, the overall performance degradation makes interpolation between two key frames the preferred option.

### 3.2.4 Evaluation of ABGPU for Pedestrian Dynamics

Simulation performance of the pedestrian model is evaluated by increasing the population size whilst maintaining a roughly static population density. Figure 17 demonstrates the population density by showing the number of pedestrians considered for communication (lookups) and the number actually inside the pedestrians' communication radius (communications). As the population density is constant, the communications to lookup ratio remains at roughly 35% in all cases. Figure 18 shows a performance chart which demonstrates the effect of increased population size on performance. Two pedestrian vision (or interaction) radii are used, the first of 4m is suggested in Helbing's work [HM97], whilst the second 32m radius acts to demonstrate the performance in cases of longer range social force planning or higher congestion population densities. In both cases an environment force map is used to simply direct pedestrians away from the outer edges of the environment. From the results it is clear that the simulation performance which includes rendering (with a low polygon count model) is suitable for large population sizes. More specifically, interactive population sizes of 262,144 pedestrians can be maintained at 13 FPS, or 65,536 at 42 FPS for a 4m pedestrian vision.

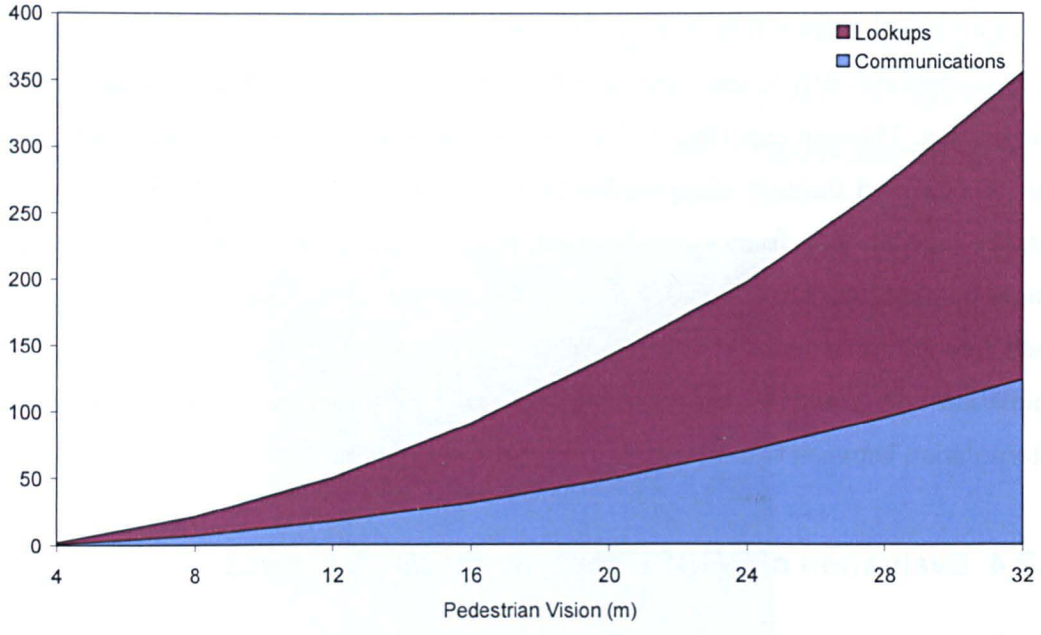



**Figure 17 - Pedestrian vision compared to pedestrian lookups and inter-agent communications.**

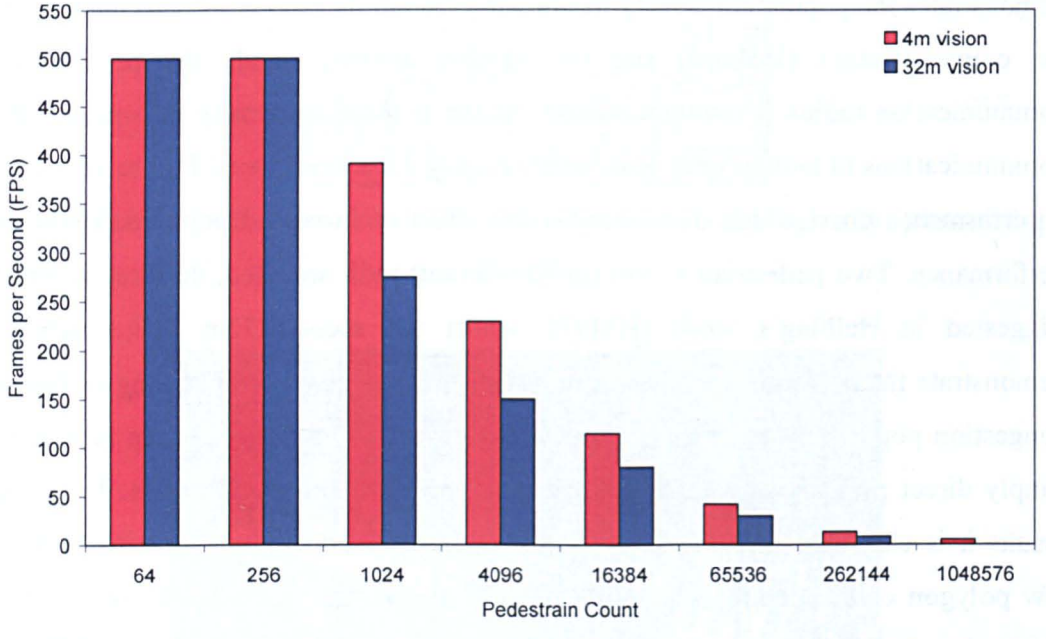



**Figure 18 - Simulation and rendering performance for primitive agents with 4m and 32m vision.**

As with the rendering of Boids, the pedestrian model has been simulated using the LOD rendering technique. Figure 19 demonstrates the performance of the pedestrian simulation with various geometric representations, ranging from a simple billboard through to a model containing over 1,000 polygons per pedestrian (detail level 2). Using only the highest detail model 16,384 pedestrians can be simulated and rendered

at over 40 FPS, where as the more modest pedestrian representation of 400 polygons (level 1) achieves a performance of 67 FPS. Despite the additional cost of calculating the LODs and reordering the agent data, the more intelligent dynamic LOD rendering is able to achieve 50 FPS for the same population size. The results indicate that the dynamic LOD approach outperforms the use of the highest resolution model for all population sizes above 1,024. Below this number the cost of the additional LOD calculations is outweighed by the ability to render the entire population using the highest resolution model at over 200 FPS. With respect to previous pedestrian modelling work the performance of the simulations in this chapter are beyond that of existing social forces implementations [CM04, CM05]. Whilst improvements in GPU hardware have some part to play in this, the decision to maintain data storage and simulation entirely on the GPU plays an important role.

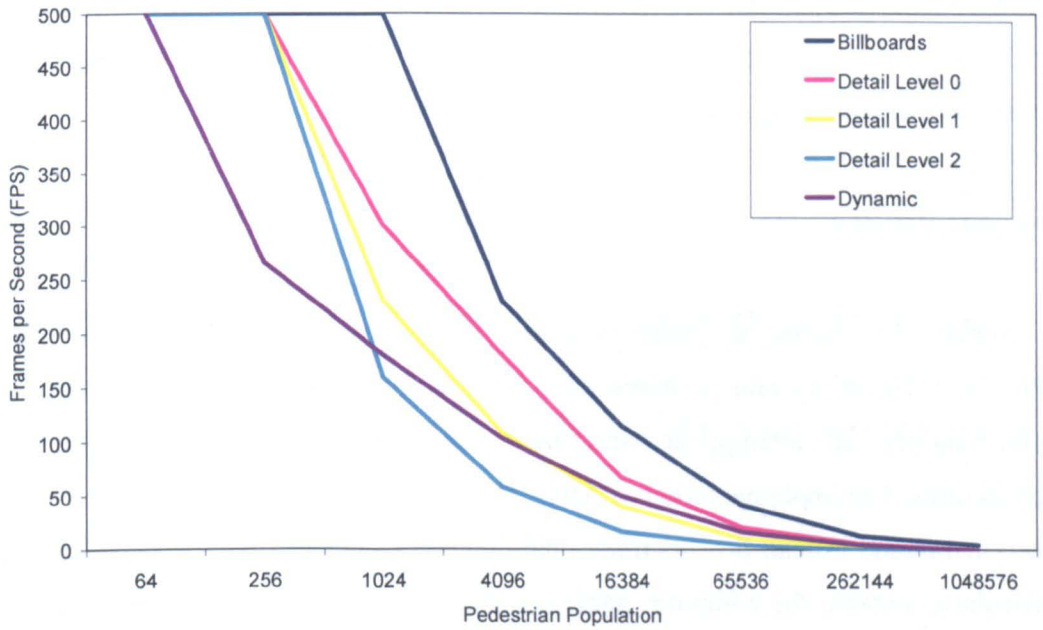



**Figure 19 - Simulation and rendering performance for advanced pedestrian rendering at various detail levels.**

## 3.3 Discussion

Arguably the most significant difference between ABGPU and other hardware assisted agent based swarm systems [Rey06, EDCST04], is the neighbourhood heuristic and consequently the number of agents considered for communication. Both

Erra's [EDCST04] and Reynolds' [Rey06] implementations favour an N-Nearest neighbour solution in opposition to the communication radius technique used within this thesis and that of the original Boids paper [Rey87]. Additionally ABM GPU work by D'Souza [LD08], continuum based pedestrian dynamics [TCP06], and the collision detection systems by Green [Gre07] perform significantly fewer communications than the agents described in this thesis. In the case of D'Souza [LD08] a real time performance of 2 million agents is reported, however agents are placed within a 2D discrete environment with only a 9x9 vision filter. Continuum approaches [TCP06] use significantly less lookups as pedestrians are not agent based and instead are averaged to produce a dynamic discrete force grid. The performance of Green's [Gre07] physically based particle demonstration, which uses the same spatial partitioning technique as ABGPU, is slightly higher than the results presented within this chapter. This is most likely as a result of utilising the CUDA radix sort and scattered write support for the generation of partition boundaries. The use of this CUDA technique is adopted within later chapters which favour CUDA's flexibility and simple programming interface for more generalised ABM.

## 3.4 Summary

In summary this chapter has presented a novel framework for ABM of swarms on the GPU. Both flexibility and performance have been addressed with case studies out performing previous attempts at swarm modelling on the GPU and similar hardware architectures. The implementation of ABGPU does not use vendor specific libraries or functionality and instead favours traditional GPGPU techniques. This makes ABGPU particularly suitable for computer game environments. Future chapters sacrifice this portability for flexibility and consider the use of more robust GPU programming techniques, which allow the simulation of AB systems which extend beyond that of simple swarms.

# Chapter 4

# Flexible ABM for the GPU

The previous chapter looked at a swarm implementation of ABM on GPU hardware. It demonstrated that the GPU is able to simulate swarm systems with very high performance. This chapter builds upon this to describe a framework for more general and flexible ABM. More specifically, this chapter addresses the limitations of ABGPU, which can be summarised as follows;

- All agents are homogeneous and require slow conditional branching to achieve heterogeneity;
- Only a single agent type is supported;
- Only a single update step is supported; and
- Only static agent populations are supported, where the size of the population is restricted by power of 2 texture limitations.

Rather than address these issues through continuing the development of ABGPU a more flexible approach has been adopted. The FLAME framework [CSH06, Coa07, ACKM08] is used as a starting point for this work as its focus on parallel agent based modelling makes it an ideal candidate for GPU implementation (FLAME GPU). Technically little remains of the original FLAME implementation, other than the concept of template generated code and the use of the X-Machine for formal model

specification. The decision to tag this work with the FLAME name, rather than producing an entirely new X-Machine framework, is due to the decision to maintain, as closely as possible, the FLAME style XMML syntax and code template format. The advantage of this is that FLAMEs existing user base are able to transfer models from standard FLAME to FLAME GPU relatively easily (this chapter provides a guide in Section 4.3.2). Likewise, the extension of an open framework and specification technique, promotes the notion of a unified modelling environment.

This chapter first reviews the FLAME framework and highlights a number of limitations. The revised extendable specification technique and templating system are then presented as is the process of generating CUDA compatible code. The decision to use CUDA and the implementation on the GPU is discussed in the following chapter.

## 4.1 The FLAME Framework

Technically the FLAME framework is not an ABM application it is instead a template based simulation environment that maps X-Machine models into simulation code (Figure 20). In order to specify X-Machine agent models, a variant of the X-Machine Description Language (XMDL) [KK01], called X-Machine Mark-up Language is used (XMML). This uses XML syntax to fully specify both agent and message structure and function order and dependencies. Within this syntax agents are described as X-Machines (or X-Agents, with messages forming inputs or outputs for each state transition function. Models are converted to compliable simulation code through the use of a custom built parser (the XParser), which inputs template files and model files to produce C code. This can then be compiled using any standard C compiler and executed by specifying a fixed number of simulation steps. The simulation code produced also contains IO functions for reading in the initial states of agents and outputting simulation steps to XML.

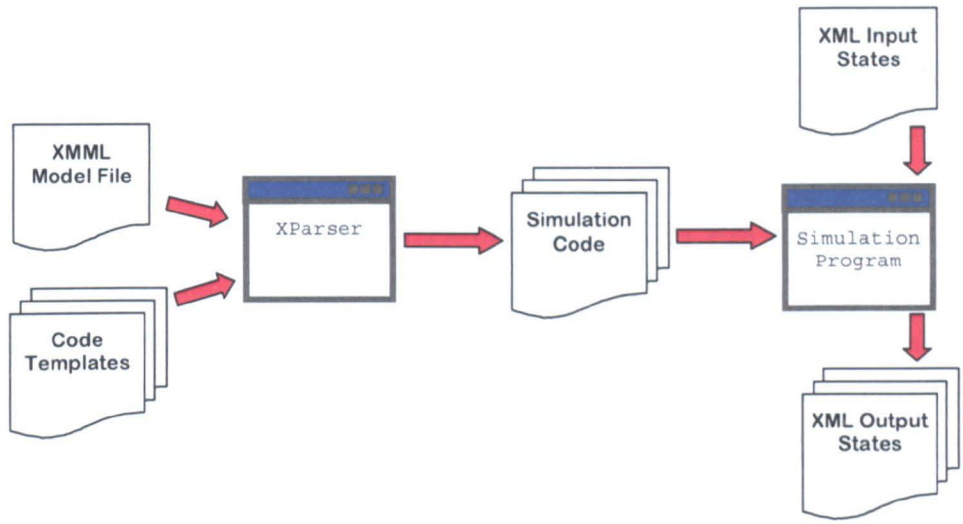



**Figure 20 - The FLAME simulation process**

FLAME simulations are processed by stepping through agent functions in an order determined by dependencies on communication or internal memory. Function dependencies are expressed within the XMML function definitions and processed by the X-Parser. The X-Parser is then able to determine function order by separating functions into layers which act as global synchronisation points. For communication dependencies between functions a separate layer is required, which ensure message lists are fully populated and avoids race conditions. For internal dependencies no global synchronisation is required, as the ordered functions are processed linearly until the next synchronisation point (level) is reached.

### 4.1.1 High Performance Computing and FLAME

Compute clusters represent the most common of todays HPC architectures (according to the top 500 supercomputer list, www.top500.org) and are built from a large number of independent processors communicating over dedicated fast network hardware. Communication is achieved through message passing libraries, which distribute messages efficiently between processors. Vendor specific message libraries include IBM's Message Passing Library (MPL). However, the more generic Message Passing Interface (MPI) library is far more common. Cost effective scalability of clusters is easily achieved, as nodes (processors) may simply be added to the network. Likewise home-built or Beowolf clusters are also easily achievable, with reasonable sums of

money using standard computer parts and free Unix operating systems [Bro09]. FLAME has been extensively developed for use with task parallel cluster architectures by the Science and Technology Facilities Council (STFC) as part of the Eurace project (www.eurace.org), which focuses on the modelling of economic market models. As part of this work, FLAME has been substantially tested on numerous high performance architectures and common C compilers. Numerous improvements have been made to the FLAME library including bug fixes, optimisations of simulation code and improvements to the clarity of the code templates [CGW07, WG06]. In order to distribute agents across processing nodes, agents are split according to spatial partition, which distributes them evenly at the beginning of the simulation. Each node examines messages to determine if their range extends into neighbouring partitions. If it does, the message is communicated through the use of a message board library, which communicates messages in batches rather than independently. This ensures maximum performance minimising the start up costs associated with communication. Aside from the distribution of messages that overlap spatial boundaries the processing of agent functions is much the same on a single processor as for serially processed model. A single processor executes the simulation by traversing the agent function layers, serially processing each agent in turn. Figure 21 represents the performance results of testing a benchmark model (referred to as the Circles model) used for "force resolution" [Coa07]. The model itself consists of a simple iterative solver, where each agent is represented by a fixed radius sphere which exerts a repulsive force on its neighbours. The HPC architectures used for benchmarking consist of the following;

**SCARF** – A cluster of 360 2.2GhzAMD Opteron cores connected via a gigabit network and Myrinet low latency interconnect.

**HAPU** – A HP cluster with 128 2.4 GHz Opteron cores with a Voltaire InfiniBand interconnect.

**NW-GRID** – A cluster of 32 Sun x4100 server nodes with two Dual core 2.4 GHz Opteron processors.

**HPCx** – An IBM cluster of consisting of 160 clusters of 16 1.5 GHz POWER5 processors connected via an IBM high performance switch.

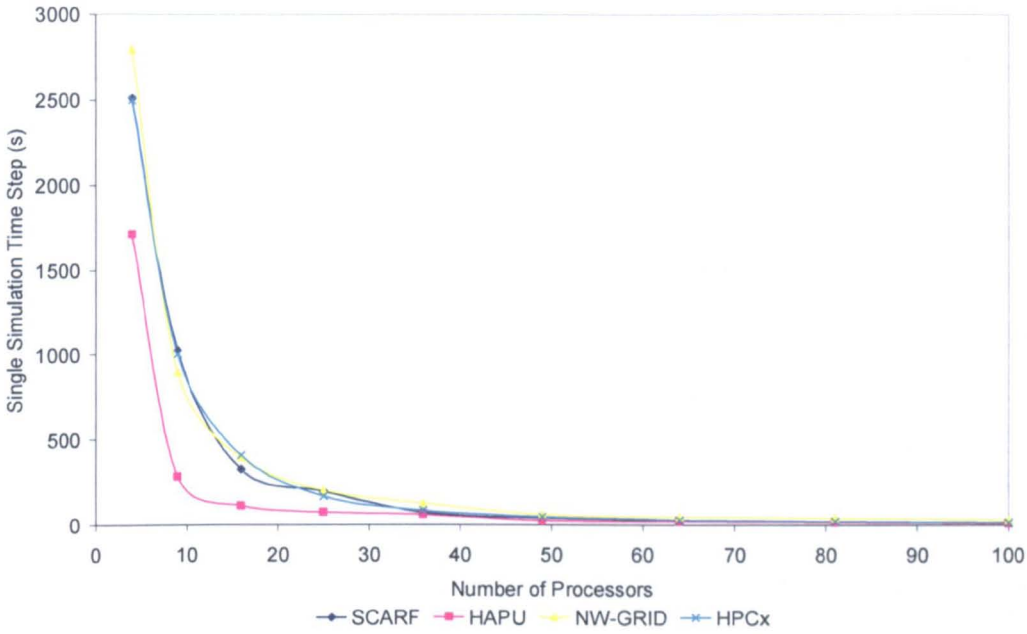



**Figure 21: The performance of the Circles force resolution model on a number of computing clusters used for benchmarking.**

### 4.1.2 The Limitations of FLAME

The FLAME framework is used by numerous researchers from various disciplines mainly at the University of Sheffield. It has been included in major research projects such as the Epitheliome Project and EURACE[7] and is continually under development and use. As a result of the multi-user base there have been a number of additions and deviations from the original FLAME specification which have been tailored towards specific projects or models. In addition to this, the widespread use and adaptation of FLAME have highlighted a number of issues, or areas of improvement, which can be summarised as follows.

1. FLAME's parallel implementation for cluster architectures is only suitable for researchers with access to the specialist hardware;
2. The templating system (XParser) is hard coded and unable to cope with changes to the XMML model syntax; and
3. Visualisation of small models is only available offline by reading each simulation output.

[7] http://eurace.org/

In addition to the availability issue associated to compute clusters, they are largely cost prohibitive and difficult to maintain. FLAME does not currently have any multi core support and as a result many FLAME users resort to the serialised single PC implementation, negating the benefits of using a parallel inspired framework. Likewise, models designed and tested using the single PC implementation often integrate external non parallel simulation code which can not be directly ported to HPC architecture at a later date, should they become available. One example of this is the use of physical solvers for inter-cellular force resolution. This is discussed and used as a case study in Chapter 6 .

The inflexibility of the XParser makes extending the XMML syntax particularly difficult. Each time additional XML elements are added to the XMML specification the XParser itself must be modified to recognise the extensions. During the development of FLAME for the GPU this proved particularly unproductive, as relatively simple changes required considerable development time and additional testing. This process leads to many different versions of the XParser, which may become highly specialised by users with particular modelling requirements or extensions. In each case a new XMML schema or Document Type Definition (DTD) may be defined along with a corresponding set of template files. As versions of the XParser diverge it is impossible to be sure if old or modified templates will result in error free code, a problem that is exasperated by the lack of XML validation support in the XParser. This is addressed within this chapter through the use of an extendable XMML specification format and templating system.

Finally, the relatively poor performance of FLAME on a single machine and the distributed nature of HPC implementations limit the suitability of FLAME for real time visualisation. Essentially the FLAME application runs on either a parallel grid or single machine (at an obviously slower speed) and for each discrete time step outputs an XML file representing the state of all agents in the system. Whilst OpenGL visualisations are possible using the 'X-visualiser', it is severely bandwidth limited, due to the slow transfer throughput of both reading from the hard drive to main memory and uploading each frames data to the GPU. With even the CPU memory to GPU memory bandwidth considered slow for real time applications, the bottleneck of hard drive access [sat04] highlights a clear bandwidth limitation to this approach of real time visualisation.

## 4.2 An Extendible X-Machine Agent Specification

Whilst not explicitly required in previous versions of FLAME, XML syntax checking (or XML validation[8]) is vitally important as it acts as a firewall against diversity [vdV02] ensuring model files contain only expected content. This guarantees that FLAME simulation templates can be processed without the possibility of unexpected XML content or absence of important model information. Within FLAME DTDs have previously been used for this role, however this thesis proposes the use of an extendable XML Schema approach which is inspired by OO extension mechanisms. Extendibility is important, particularly as FLAME is continually under development with multiple projects each adding their own degree of configurability to the XMML syntax. In addition to offering the potential to manage extensions to XMML model format, XML Schemas are themselves XML based and can hence be validated in the same way that documents using the schema can be.

This section describes the OO properties of XML Schema and the methods which are used to allow flexible extensions. It goes on to describe the implementation of a GPU XMML Schema (GPUXMML), which allows polymorphic extension of a well defined base XMML Schema. The content of the base XMML Schema is itself based upon the XMML syntax used in recent versions of the standard FLAME framework. It has been designed through deliberation with other FLAME users to ensure maximum compatibility and minimal changes.

### 4.2.1 Object Orientation within XML Schema

XML Schema represents a powerful means by which to express XML data structure, which draws strong parallels with OO Design. This is made possible through the use of reusable type definitions, which are analogous with classes in the same way that XML elements assimilate objects. A number of predefined type definitions such as `string`, `int`, `double` are available in order to specify the content or legal value of XML elements or attributes. Likewise, user defined simple types can extend these 'primitive' types through derivation. Such derivations include the following;

[8] Validation with respect to XML refers to legal syntax rather than the modelling and simulation term, which refers to testing if the correct model has been built.

- Restriction – Types may be restricted by maximum or minimum length or by use of regular expressions;
- List – Types may be limited to a list of legal values. Such types are similar to enumerations; and
- Union – A number of primitive or user defined type definitions can be used as legal values of the type. User defined types can be either referenced or embedded within the simple type definition.

In contrast complex types provide a mechanism to specify the format of the mark-up itself. Data structure can be encapsulated within complex types, which allow simple reuse of common components. Inheritance of complex types is made possible through derivation of a base class by either extension or restriction. Extension allows additional content to be added, whilst restriction allows a subset of the original base type to be created. As with OO Design abstract types are supported and may not be used to define concrete elements within further type definitions. Finally, polymorphism is supported through the use of substitution groups, which allow a number of elements within the group to be used in place of an element at the head of the group. The group head may only be global (i.e. outside of the local scope of some type definition) and as with types may be abstract to prevent its use directly.

### 4.2.2 XML Schema Design for Extendible Schemas

Various techniques can be used to design XML Schemas which range in their simplicity and flexibility (Table 2). The simplest of these is often referred to as a Russian Doll design and comprises of a highly nested set of elements where only the root element has a global scope. Whilst very compact, this design technique is highly self contained and changes made to types within the narrow local scope are not propagated to other Schemas or global definitions. The ridged structure offers similar functionality as DTD validation with no way to take advantage of the OO Principles discussed in the last section. In contrast, the Flat model (or sometime known as the Salami Slice design) is highly reusable and consists of globally defined elements which may be referenced by any other inside or outside of the Schema. In addition to

this, the Flat model is the only technique which offers the ability to exploit polymorphism through substitution groups. Finally, the Complex Type model (or Venetian Blind design) consists of a number of globally defined complex types, which may exploit inheritance by defining extensions or restriction elsewhere inside or outside of the Schema.

**Table 2 – Contrasting XML Schema design methodologies**

| | Encapsulation | Inheritance | Polymorphism |
|---|---|---|---|
| **Russian Doll** | No | No | No |
| **Flat** | By Element | No | Yes |
| **Complex Type** | By Type | Yes | No |

### 4.2.3 Extending the XMML Schema

The major design concern for a flexible XMML Schema is extendibility and as a result, a hybrid of both the Flat and Complex Type schema designs have been used, to exploit both inheritance and polymorphism. From an implementation perspective this requires a reusable complex type definition for each aspect of the base XMML model. Within the Schema, concrete element definitions for each complex type are defined at the global scope. These are referenced within other complex types and form the head of potential substitution groups where extensions may be used. The following code shows an example of how this technique is used. It describes the global definition for the *xagent* type in the base XMML Schema (available in Appendix A.1).

```
<complexType name="xagent_type">
  <sequence>
    <element name="name" type="string" maxOccurs="1"
             minOccurs="1" />
    <element name="description" type="string" maxOccurs="1"
             minOccurs="0" />
    <element minOccurs="1" maxOccurs="1" ref="memory" />
    <element maxOccurs="1" minOccurs="1" ref="functions"/>
    <element maxOccurs="1" minOccurs="1" ref="states" />
  </sequence>
</complexType>
```

Essential non-changeable properties of the model (e.g. name and description) are defined as elements within the complex type definition. Element definitions, which use references rather than simple types, or nested definitions, must reference a global concrete element of the same name. For the *xagent* example this is demonstrated below;

```
<element name="xagent" type="xagent_type"></element>
```

Once a global element has been defined for the abstract type this may then be referenced with other element sequences. In the case of the *xagent* definition this is referenced by the *xagents* element which contains a list of *xagent* elements (with a minimum occurrence of at least a single *xagent*).

```
<element name="xagents">
  <complexType>
    <sequence>
 <!– reference to the concrete xagent element -->
      <element maxOccurs="unbounded" minOccurs="1" ref="xagent" />
    </sequence>
  </complexType>
</element>
```

Within instances of XMML documents, any reference to a global element is free to use either the concrete base XMML Schema definition (as above) or alternatively any other element from within a substitution group where the base case forms the head (from Schema which extend the base). This is demonstrated below through the redefinition by derivation through extension of the *xagent_type* and *xagent* element. The use of polymorphism allows the new *xagent* definition (which contains two new elements type and buffer size which are explained later in this chapter) to replace the base definition in any XMML document instance.

```
<complexType name="xagent_type">
  <complexContent>
    <extension base="xmml:xagent_type">
      <sequence>
        <element name="type" type="xagent_type_options" />
```

```
        <element name="bufferSize" type="int" />
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element substitutionGroup="xmml:xagent" name="xagent"
         type="xagent_type" />
```

In the above example (taken from the GPUXMML Schema in Appendix A.2) ambiguity caused by using the same element and type names is avoided through the use of namespaces, which are evident through the use of the 'xmml' prefix. Technically namespaces allowing a simple method for qualifying element and attribute names through association with a Uniform Resource Identifier (URI) reference. This technique allows elements and attributes to be grouped together into an independent collection, often referred to as a vocabulary. In the case of the XMML Schemas this offers the important advantage of separating the static base XMML Schema from the GPUXMML Schema which extends it. The GPUXMML example above can therefore be contained within a separate document which references the base XMML Schema through the following code;

```
<xs:schema id="GPUXMML"
     targetNamespace="http://www.flamegpu.com/GPUXMML"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">
     xmlns:xmml="http://www.flamegpu.com/XMML">


<xs:import namespace="http://www.flamegpu.com/XMML"/>
```

In the above example the target Namespace attribute refers to the Uniform Resource Identifier (URI) in which the schema definition is bound to. The prefix of 'xs' used within the schema and import element is a result of the XML Schema binding. Both the XMML base schema and GPUXMML Schema are defined fully within Appendix A. Examples of two GPUXMML document instances, used as case studies in Chapter 6 are also provided in Appendix B.1 and Appendix C.1.

### 4.2.4 Extensions within GPUXMML

The majority of the extensions which the GPUXMML Schema adds over the base XMML Schema are a result of the parameterisation of agent functions[9] within the FLAME GPU simulation API. Message and agent, inputs and outputs, must now be explicitly specified and are additionally limited to a single input and output type (some versions of FLAME already use full input/output specification in the XMML syntax and so this has been included within the base XMML Schema). As message reading is performed by three differing communication patterns (described in Section 5.2) any message specifications must define the partitioning method to be used. The GPUXMML schema includes a number of elements which are used to define interaction ranges and environment bounds. Within the FLAME GPU implementation, memory is pre-allocated, and as a result messages and agents require a buffer size representing an upper bound on their population size. FLAME GPU adds the ability to simulate discrete spaced agents (Section 5.2.3) and therefore an XML element is required for specifying an agent type. XML Elements are included for random number generation (RNG) within agent functions and indicate that some RNG structure must be passed to the agent function (Section 5.1.7). A reallocation element can optionally be specified as false if an agent function has no requirement to process agent deaths. Initialisation functions can be defined for the model. These are functions that exist within one of the agent function files which must be run before the simulation starts. Such functions are demonstrated later in Section 5.1.5 and are essential for setting up of global constant variables used within the simulation. Finally, global conditions add additional functionality which is described in Section 5.1.6. Essentially global function conditions are similar to normal function conditions, with the exception that every agent must meet the condition for the function to be processed. Within GPUXMML there are two elements used to limit the maximum number of times a global condition may be evaluated as either true or false. Later within this chapter Table 4 gives a summary of the extensions made by GPUXMML (as well as any additional base XMML Schema elements not present in the original FLAME DTD)

[9] The original FLAME framework used global methods rather than function parameters/arguments

Dependencies, which are used within the original FLAME implementation to determine function and communication order, have been excluded from the XMML Schemas described within this thesis. As an alternative, the base XMML Schema includes function layer XML elements to directly define the function layers, and hence processing order. Whilst it would be possible to process dependencies during the template processing stage, the lack of complex dependencies in the case studies used throughout this thesis is insufficient to justify rewriting of the dependency graph generator. This is however entirely feasible and is left as future work.

The final addition to the GPUXMML Schema is the use of data relationships through keys and key references. This is highly advantageous, as it allows a mechanism to ensure that message and agent name references, used within the model definition, are defined elsewhere in the instance of the XMML model file.

**Table 3 - Keys and relationships within the GPUXMML Schema**

| Key name | Description of Key | Relationships (references) |
|---|---|---|
| xagent_func_name_key | X-agent function name | Transition Function Name within Function Layers |
| xagent_state_name_key | X-agent state name | Transition Function currentState<br>Transition Function nextState<br>X-Agent Initial State<br>Transition Function X-Agent Output State |
| message_name_key | Message name | Transition Function Input Message Name<br>Transition Function Output Message Name |

Table 3 highlights the relationships within the GPUXMML Schema. Whilst it would be desirable if the relationships could be defined within the base XMML Schema and inferred for polymorphic substitution, this is not supported in XML Schema and as a result relationships must be redefined each time an element containing a key is extended.

## 4.3 FLAME GPU Code Generation

The simulation code generation within the original FLAME framework is heavily reliant on template processing through its own template processor (the XParser). The problems with the XParser have previously been discussed and can be summarised as;

1. The requirement to rewrite IO code to recognise new XML elements dependant on changes to the XMML specification;
2. Multiple versions of the XParser existing with no consistency between versions; and
3. A Lack of XMML syntax validation support within the XParser itself.

In order to combat these problems this thesis proposes the use of Extensible Stylesheet Transformations (XSLT) in order to generate simulation code. XSLT is a flexible functional language based on XML and, as with other XML technologies, allows documents to be validated against a W3C Schema[10]. XSLT documents are translated through the use of any compliant processor which in the case of FLAME GPU can convert a GPUXMML model file to compliable code using the predefined templates. Not only does this remove the dependency on a specific template parser and avoid versioning issues, but this ensures inclusion of the XMML Schemas guaranteeing that templates and Schema versions remain compatible. Failure to match the correct XMML Schema with an XSLT template simply produces a validation error. In contrast with XQuery[11], the other prominent XML based template engine, there is a significant overlap in functionality with both including XPath functionality for XML tree traversal. The choice to use XSLT over XQuery is justified by the stronger focus of XSLT towards document translation, rather than XQuerys focus of SQL type database querying.

### 4.3.1 XSLT Templates

XSLT is more prominently used in the translation of XML documents into other HTML or other XML document formats on the web. Despite this there is no limitation on the type of file that may be generated from an XSLT template and it is hence suitable for the generation of source code. As XSLT standards are well defined there exists a number of compliant processors any of which can be used to generate the same output (Saxon[12], Xalan[13], Visual Studio, and even common Web Browsers such as Mozilla Firefox). In all cases an XSLT processor works by recursively

[10] http://www.w3.org/TR/xslt
[11] http://www.w3.org/TR/xquery/
[12] http://saxon.sourceforge.net/
[13] http://xalan.apache.org/

matching XML nodes and applying a template to it. In the case of generating code, each required source code file uses a single template matched to the root `<xmodel>` element of the GPUXMML document. Code generation is then heavily dependant on the branching and control elements of XSLT which are used to iterate agents, messages, states and variables. The code sample below (from `header.xslt`) demonstrates how the iterative `for-each` control is used to generate a C structure for each `xagent` within an GPUXMML model document. The select attribute uses an XPath expression to match nodes in the document. Likewise, XPath expressions are used to match nodes within the `value-of` attributes and any other XSLT elements which require XML document querying.

```
<xsl:for-each select="gpuxmml:xmodel/xmml:xagents/gpuxmml:xagent">
struct __align__(16) xmachine_memory_<xsl:value-of
select="xmml:name"/>
{ <xsl:for-each select="xmml:memory/gpu:variable">
  <xsl:value-of select="xmml:type"/><xsl:text> </xsl:text>
  <xsl:value-of select="xmml:name"/>;</xsl:for-each>
};
</xsl:for-each>
```

Within the XMML specification there are specific elements which are recursive. One of these is the use of function conditions which determine if an agent should use a particular function dependant on some condition. The recursive ability to have a condition within a condition allows complex conditions to be specified, which include reference to any number of the agents internal memory variables. The example below demonstrates the use of recursive templates to generate a conditional statement in compliable C code.

```
<!--Recursive template for function condititions-->
<xsl:template match="xmml:condition">(
<xsl:choose>
  <xsl:when test="xmml:lhs/xmml:value">
    <xsl:value-of select="xmml:lhs/xmml:value"/>
  </xsl:when>
  <xsl:when test="xmml:lhs/xmml:agentVariable">
    currentState-><xsl:value-of
```

```
                    select="xmml:lhs/xmml:agentVariable"/>[index]
  </xsl:when>
  <xsl:otherwise>
    <xsl:apply-templates select="xmml:lhs/xmml:condition"/>
  </xsl:otherwise>
</xsl:choose>
<xsl:value-of select="xmml:operator"/>
<xsl:choose>
  <xsl:when test="xmml:rhs/xmml:value">
    <xsl:value-of select="xmml:rhs/xmml:value"/>
  </xsl:when>
  <xsl:when test="xmml:rhs/xmml:agentVariable">
    currentState-><xsl:value-of
                    select="xmml:rhs/xmml:agentVariable"/>[index]
  </xsl:when>
  <xsl:otherwise>
    <xsl:apply-templates select="xmml:rhs/xmml:condition"/>
  </xsl:otherwise>
</xsl:choose>
)</xsl:template>
```

The recursive template is called within the main template using the following syntax.

```
<xsl:apply-templates select="."/>
```

As each top level function condition is iterated using an XPath expression, the above selects statement simply selects the current node element. The template then uses a conditional choose XSLT element to distinguish between a value, agent variable or nested condition for each side of the conditional expression. In the case of a nested condition the template is recalled passing the nested expression as the top level element.

### 4.3.2 Conversion of old FLAME models to the GPUXMML Schema

The major component of converting original FLAME models to the FLAME GPU format is the adoption of the new XSLT templates and GPUXMML schema. In the case of an existing model an XMML document validating against the original

FLAME DTD can adopt the new schemes by simply importing the relevant namespaces (for both the base XMML schema and the GPUXMML schema), applying the appropriate namespace prefixes to the XMML elements and adding the additional XMML tags (described in Section 4.2.4 and summarised alongside any other changes to the original XMML DTD in Table 4). Alternatively, using an XML editor, such as Visual Studio or Eclipse, to rewrite the XMML model will provide auto-completion to ensure syntax validation during design. Where an XML editor is not used a GPUXMML model instance can be validated using any XML Schema compliant validator such as Xerces [XERCES]. It is important that validation is checked as the processing of each XSLT template file using the GPUXMML model instance will be unsuccessful if validation errors are present.

Each function file containing scripted C code agent functions must use a parameterised version of the original FLAME API functions. In order to ensure these are correct it is possible to generate function prototypes and empty agent functions using an additional XSLT template. Function code can then be copied from the original FLAME function files, adding in parameters specified by prototypes. As with all agent functions (excluding initialisation function which use a `__FLAME_GPU_INIT_FUNC__` prefix) any function referenced by one of the agent functions must contain the `__FLAME_GPU_FUNC__` prefix. This is used to ensure all functions are in-lined, which is necessary within CUDA.

**Table 4 – Base XMML and GPUXMML element additions over the original FLAME XMML DTD.**

| XML Element | Parent Element | Description |
|---|---|---|
| `<gpu:initFunctions>` | `<gpu:environment>` | Holds any number of `<gpu:initFunction>` elements. |
| `<gpu:initFunction>` | `<gpu:initFunctions>` | Defines the `<gpu:name>` of an initialisation function prefixed within the simulation code by the `__FLAME_GPU_INIT_FUNC__` macro. |
| `<outputs>` | `<gpu:function>` | Holds a single agent function `<gpu:output>` element. |
| `<gpu:output>` | `<outputs>` | Defines an agent function output by specifying a `<messageName>` and `<gpu:type>` which may indicate a single message (every agent outputs a message) or optional message output (some agents do not output a message). |
| `<xagentOutputs>` | `<gpu:function>` | Holds a single agent function `<gpu:xagentOutput>` element. |

| | | |
|---|---|---|
| <gpu:xagentOutput> | <xagentOutputs> | Defines an agent function X-machine agent output (or agent birth) by specifying the <xagentName> and <state>. |
| <inputs> | <gpu:function> | Holds a single agent function <gpu:input> element. |
| <gpu:input> | <inputs> | Defines an agent function input by specifying a <messageName>. |
| <gpu:reallocate> | <gpu:function> | Specifies if an agent function may result in an agent death. |
| <gpu:RNG> | <gpu:function> | Specifies if an agent function requires the use of random number generation. |
| <gpu:type> | <gpu:xagent> | The agent type which may be either continuous or discrete |
| <gpu:bufferSize> | <gpu:xagent> and <gpu:message> | The maximum size of either an agent or message list within the simulation. |
| <layers> | <gpu:xmodel> | The explicit definition of at least a single <layers> of agent functions. |
| <layer> | <layers> | The definition of a single agent function layer containing at least a single <gpu:layerFunction>. |
| <gpu:layerFunction> | <layer> | The definition of a layer function which specifies a function through a <name> element. |
| <gpu:globalCondition> | <gpu:function> | Specifies a global condition on an agent function. |
| <gpu:maxItterations> | <gpu:globalCondition> | Limits the number of times a global condition can be evaluated as false. An explanation of global conditions is given in section 5.1.6. |
| <gpu:mustEvaluateTo> | <gpu:globalCondition> | Allows the global condition to be successfully evaluated as either true or false. |
| <gpu:partitioningNone> <gpu:partitioningSpatial> <gpu:partitioningDiscrete/> | <gpu:message> | Specifies the partitioning technique used for inputting the message within agent functions. |
| <gpu:radius> | <gpu:partitioningSpatial> <gpu:partitioningDiscrete/> | Sets the interaction radius of a message. |
| <gpu:xmin> <gpu:xmax> <gpu:ymin> <gpu:ymax> <gpu:zmin> <gpu:zmax> | <gpu:partitioningSpatial> | Used to set the environment bounds of a spatially partitioned message. |

### 4.3.3 Building Simulation Code

In order to simplify the processes of generating and building simulation code a number of optional tools are provided. The first of these is a Visual Studio project file, which includes all the necessary default library paths for the CUDA and CUDA Parallel Primitives (CUDPP) libraries, which are required to build the simulation

code. In addition to this, a number of custom build rules are provided, which automatically processes each FLAME GPU template during the build stage of the project. This has options to turn off the regeneration of individual templates if the source code is modified for some reason after the simulation code has been generated. In order to process XSLT documents an XSLT processor has been designed using the .NET library. This is a standalone application which is called by the custom build rule and will report any validation or template errors within the output window of visual studio during the build stage. Figure 22 demonstrates the complete simulation process of generating and building simulation code. The visual studio project file simply automates each of these stages.

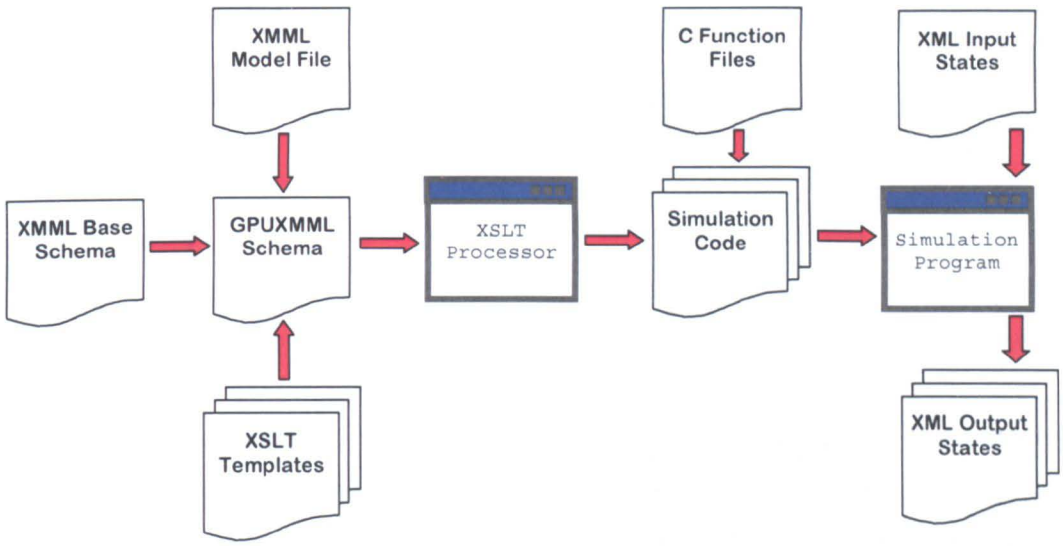



**Figure 22 – The FLAME GPU Simulation Process**

## 4.4 Summary

In this chapter the limitations of ABGPU have been addressed through the introduction of a technique for flexible agent specification inspired by the FLAME framework. In contrast with FLAME's previous use of DTD validation and use of the XParser, the technique described within this chapter is considerably more robust and extendible. The use of standardised schema and translation languages lends itself to the notion of an open specification system. This also reduces dependency on external tools to generate functional simulation code. It should be noted that whilst this technique no longer relies on the XParser, the XSLT templates do not generate

function dependencies and function order must be explicitly specified in the XMML description. The automatic generation of function order is however possible using XSLT and is left as future work.

# Chapter 5

# Implementing FLAME GPU

In the previous chapter the FLAME framework was described along with techniques that allow XMML model files to be extended. This allowed the specification of the GPUXMML schema which extended a basic XMML Schema model to include the necessary details required to map FLAME to GPU hardware. An XSLT template system was also described, and it was demonstrated how a GPUXMML model can be used to create compliable code from a number of code templates.

This chapter discusses the implementation of FLAME GPU by considering the techniques and algorithms encoded into the XSLT templates. The templates allow a mechanism for abstracting the GPU programming process from modellers, in the same way that ABGPU used an API. In contrast with the original FLAME framework, the implementation of FLAME GPU described within this section targets data parallelism, rather than task parallelism. As a result, a completely new set of templates have been implemented which use the XSLT technique described in the previous chapter. This has been essential, as much of the original FLAME template code relied heavily on the use of dynamic linked lists, which do not map well to the

GPU architecture. Instead, alternative techniques are described within this chapter which allow birth and death allocation and dynamic sized agent lists.

In previous chapters traditional GPGPU techniques were used to program the GPU. The decision to use these techniques was inspired by the requirement for flexible algorithms on the GPU, which utilised many of the low level features. Since the development of this earlier work both the Stream SDK [ATI09] and CUDA programming languages [NVI07] have emerged offering substantial advantages over GPU programming using a graphics API. Both technologies offer an increased flexibility and familiar C syntax, which significantly reduces the development time. Of the two technologies, the CUDA language has been chosen for the implementation of FLAME GPU. Whilst this decision restricts the work to NVIDIA (G80 or later) graphics cards, the early support for CUDA and large user base make it the most suitable choice for advanced GPU programming.

## 5.1 Implementing FLAMEGPU with CUDA

Conceptually, the level of parallelism between the original FLAME for HPC and FLAME GPU is very different. The GPU's parallelism offers a very fine grained data parallel architecture in comparison with the coarse, task level parallel architecture for which FLAME was originally designed. As a result, each agent is conceptually represented by a thread of execution, with each GPU multiprocessor simultaneously processing agent functions in fixed sized blocks of agents. Individual agent functions are wrapped in unique CUDA kernels, which are processed one after another. This ensures global synchronisation of all agents and message communications between each agent function.

This section looks at the underlying techniques and algorithms used to map FLAME GPU to CUDA and the GPU. It describes all aspects of the implementation, excluding the techniques used for message communication, which are discussed later in this chapter.

### 5.1.1 Efficient Agent Data Storage and Access

As with the original version of FLAME, it is important that agents and messages are stored in variable length lists. During simulation these lists remain persistent in GPU memory, with equivalent lists on the CPU used only as intermediate placeholders during uploading or downloading of data. As allocation and uploading is an expensive operation, compared to actually processing data on the GPU, all allocation is done in advance using the maximum buffer sizes, as specified in the GPUXMML model. Whilst this may appear restrictive, there is no performance disadvantage to specifying significantly larger buffer sizes, assuming enough GPU device memory is available.

In order to ensure agent data is transferred from the GPU device optimally, agent functions are wrapped in a kernel which transfers the agent data from global GPU memory to the multiprocessors' register space. For simplicity, this kernel loads an individual agent's data into a C structure which contains a member variable for each agent memory variable. This C structure can then be passed to the agent function, where it can be updated directly, before the wrapper kernel efficiently writes it back to global memory. Although it would be intuitive to therefore store the agent population data in global memory with an Array of Structures (AoS), this has serious memory access performance implications. Instead, agent population data is stored as a single Structure of Arrays (SoA). This allows a more efficient memory access pattern for both reading and writing data in global GPU memory. The reason for this is GPU *memory coalescing*, which allows data accessed by consecutive threads to issue fewer wide memory requests, making more efficient use of the memory bus [How07]. The conditions of coalescing are that data variables within consecutive threads are accessed with the same linear consecutive order. The exact performance advantage of this technique is evaluated later in Section 6.1.1.

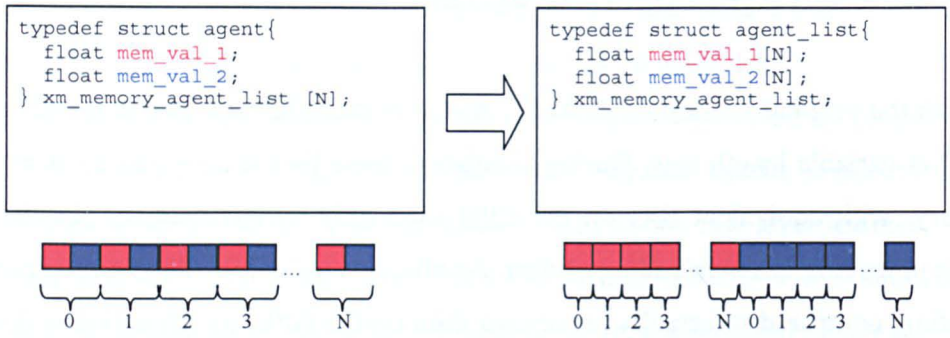



**Figure 23 – Array of Structure (AoS) vs. Structure of Array (SoA) data storage of an agent structure.**

Figure 23 demonstrates the difference between data storage in the SoA and AoS form. In the case of SoA storage each unique variable is stored consecutively, rather than storing each agent consecutively, which interlaces the variables' positions in memory. Message data is read using the same technique, with messages being stored in global memory using the SoA format. The individual message functions, described later in section 5.2 are responsible for loading messages into the AoS format (within the per multiprocessor shared memory) which allows more logical memory variable access within the agent functions.

### 5.1.2 Birth and Death Allocation

As memory allocation during simulation is avoided in FLAME GPU, the addition of new agents requires pre-allocated memory space. In the case of agent function, xagent outputs (or *agent births*), any agent function may, or may not, produce a new agent. Therefore, the entire agent population must be double buffered (i.e. make use of an additional agent list of equal size, using the same SoA format) to provide sufficient storage space for any new agents. The process of agent births can then be achieved using a linear mapping where each agent in the current agent list outputs to the same global position within the new agent buffer. As not all agent functions which require agent birth functionality will result in every agent giving birth, the linear mapping will most likely result in the new agent list containing empty gaps (or being sparse). To remove these gaps (or compact the list) requires the use of a technique to reorder agents into a compacted form. This is possible using an inclusive parallel prefix sum algorithm [SHZO07]. In early implementations of FLAME GPU this was achieved

using simple prefix sum kernels, however the CUDPP library [CUDPP] provides more optimal kernels for this purpose which are work efficient and heavily optimised. Within the sparse new agent list a simple flag variable is used to indicate the presence of a new agent. This flag is then used to perform the inclusive prefix sum, which determines the position of the new agent in the sparse list. The total number of new agents can easily be determined by considering the position of the last new agent. This is used to ensure the current agent population size will not exceed the specified buffer size when the new agents are added. A final (post agent function) scatter kernel is then run over the new agent memory list, using the agent's new position value to append flagged data to the end of the current agent list. The agent count can then be updated, both on the host and device, and the new agent list is ready to be used in subsequent agent functions or simulation steps.

Similarly to agent births, *agent deaths* require additional buffered data storage (of the same length and SoA format as the current agent list) which is referred to as the temporary swap list. In order for an agent to die an agent function (which specifies a *true* reallocation element with the GPUXMML specification) simply returns any value other than 1. The return value is used as a flag to indicate an active or dead agent and is compacted using the same technique as described above. The swap list is used for the final scattering of the compacted agent list, which is finally swapped with the original agent list. Agent deaths are always handled before birth allocation as this ensures a compacted agent list before any new agents are appended. The following example shows the specification of a simple GPUXMML agent function which requires both *agent birth* and *agent death* functionality.

```
<...>
<gpu:function>
  <name>function</name>
  <description></description>
  <currentState>default</currentState>
  <nextState>default</nextState>
   <xagentOutputs>
    <gpu:xagentOutput>
      <xagentName>agentName</xagentName>
      <state>default</state>
    </gpu:xagentOutput>
   </xagentOutputs>
```

```
    <gpu:reallocate>true</gpu:reallocate>
</gpu:function>
<...>
```

The corresponding function code for the above example is shown below. The use of the <gpu:xagentOutput> forces the function definition to accept the new agent list buffer (agentName_agents), which is used for the linear mapping within the add_agentName_agent API function. The use of the reallocate element allows agents which are flagged to be killed (by returning 0), to be removed after the agent function has been processed.

```
//Simple agent function demonstrating agent output
__FLAME_GPU_FUNC__ int function(xmachine_memory_agentName* agent,
                   xmachine_memory_agentName_list* agentName_agents)
{
    //New agents id variable using the current agent count
    int id = agent->id + d_xmachine_memory_agentName_count;

    //Add the agent to the agent list
    add_agentName_agent(agentName_agents, int id);

    //If some condition is true kill the agent
    if (somecondition == true)
        return 0;
    else
        //Don't kill the agent
        return 1;
}
```

Alternative attempts at birth and death allocation for ABM on the GPU [DLR07] have used randomised iterative schemas to allow agent births. This works by randomly distributing agents within an agent list and then using multiple kernels to output agents to a position, based on some random offset of the agent's position. In contrast, with using a prefix sum, a randomised scheme only converges towards success and is not guaranteed to either successfully add all new agents, or fail if the agent list is full.

### 5.1.3 Agent States

With respect to the X-Machine methodology of which the FLAME and FLAME GPU libraries are based upon, agent functions represent a transition function from one state to another, which modifies agent memory. Within the task parallel original implementation of FLAME, all agents may reside within a single agent list, with the concept of an agent's state represented by a simple variable held in agent memory. In this scenario, agent functions can be evaluated by iterating the agent list and applying the function only to agents which are in the correct state. For FLAME GPU, the use of an agent state variable is far from optimal. In order to apply an agent function to the agents of a particular state, every agent in the mixed state agent list must be launched in its own thread, with only the agents in the correct state evaluating the full function code. As groups of threads are processed in warps (of 32 threads), there is a high likelihood in this case that not all threads (agents) will follow the same instruction path (i.e. those agent threads which are in the wrong state will perform no further computation and remain idle). Agents are not stored in any particular order and therefore the degree of divergence is difficult to predict. Figure 24 gives an example of a half warp (for clarity only, as threads are always evaluated in full warps) being processed, where only half the agents meet the function condition. In the worst possible case each warp would contain only a single agent in the function's initial state condition and hence only a single vector processor will be utilised during the processing of the agent function. Simulations containing large numbers of states, or well mixed agents, will suffer from this problem to a larger extent and as a result mixed state agent lists are avoided. This is achieved by maintaining a separate agent list for each possible agent state, removing the requirement to hold a state variable and avoiding any state based divergence. This will result in the number of threads launched per agent function to be drastically reduced which in most cases provides a positive performance benefit. It must however be considered that in order to fully occupy the GPU's multiprocessors a relatively large number of agents in each state are required. Where there are relatively few agents in any one state there is a chance that underutilising the GPU will demonstrate far from optimal performance. The possibility of this is easily outweighed by the massive performance benefit gained over the mixed state alternative.

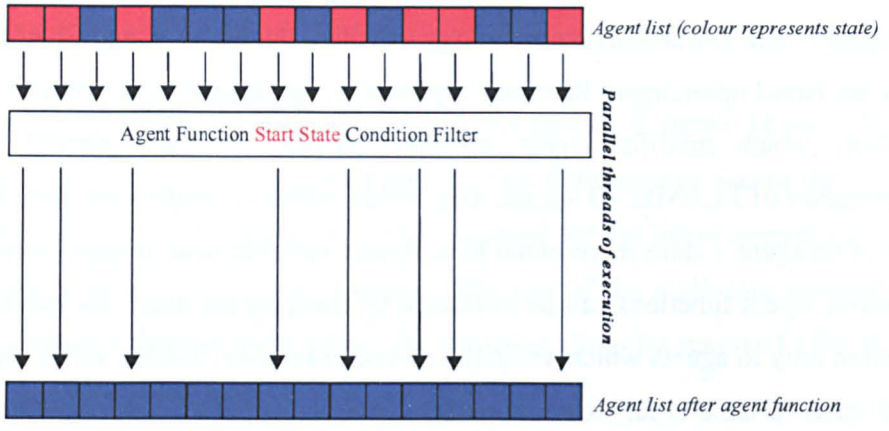



**Figure 24 – A mixed state agent list executing an agent function.**
**Agents which meet the functions start state are indicated in red which move into a blue state after processing the agent function.**

### 5.1.4 Conditional Functions

Function conditions present a similar scenario to agent states, as on the CPU an agent list can simply be iterated to see if the function should be applied to the agent. As with agent states, the fact that some agents may, and some may not, meet the function condition, introduces a high likelihood for divergence. To overcome this every agent in the agent functions start state agent list is filtered, using a function condition kernel (Figure 25). This is used to set a flag value to distinguish between agents which meet the function condition. The flag is then used with the parallel prefix sum compaction technique which was described for agent birth and death functionality. Agents which meet the function condition are then removed from their current agent state list (which is compacted) and scattered into a non sparse working list for the agent function. After the agent function has been applied to the agents within the working list, the agents move into the agent functions end state by being appended to the appropriate agent state list. As with the original FLAME, agent functions do not provide any guarantee of deterministic behaviour. If an agent in a given state meets one or more function conditions it is processed, using the agent function which appears first in the function layers.

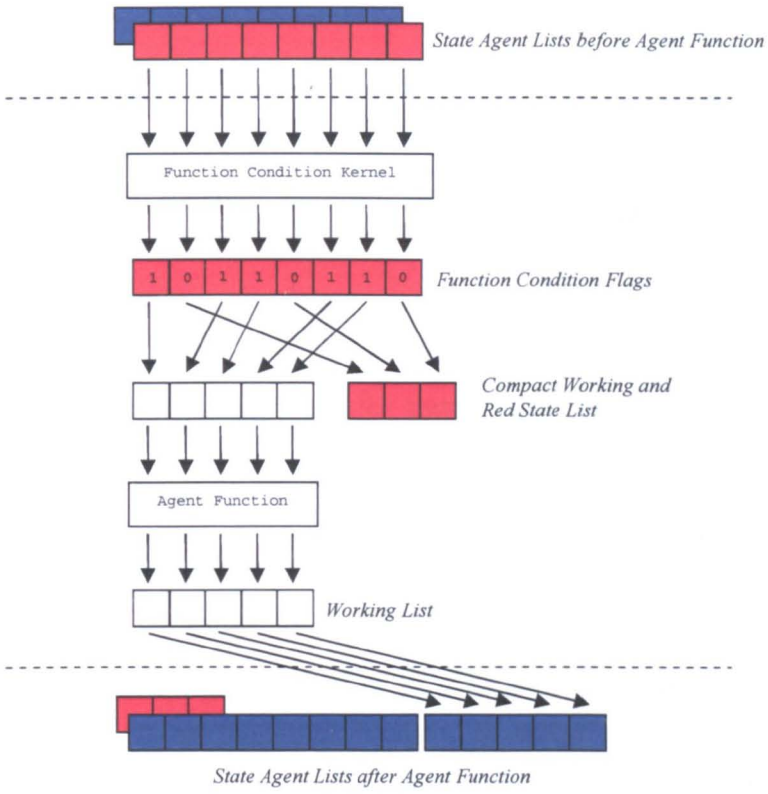



**Figure 25 – Evaluation of an agent function using unique state lists and a working list. A function condition is used to separate agents into a working list (white) which is appended to the functions end state list (blue) after the function has been evaluated.**

### 5.1.5 Global Variables and Initialisation Functions

Global (constant) variables are read only variables which are accessible to all agents. Within CUDA, a small area of device memory (64Kb) is allocated as constant memory, accessible via a per-multiprocessor cache. This constant memory is used to store all FLAME GPU global variables as well as any hidden agent function constants such as agent counts. When multiple threads request the same data from constant memory, the cache hides a large amount of memory latency and if all threads in the same warp read the same value, the memory access is as fast as reading from registers. CUDA constant memory must be set using CUDA API methods which are abstracted from FLAME GPU users through the use of a dynamic API function for each global variable. In order to use the API functions to set a global variable, an initialisation function can be defined with the XMML syntax. An example of this is shown below which defines both a global variable and a global variable array;

```
<gpu:environment>
   <gpu:constants>
      <gpu:variable>
         <type>float</type>
         <name>global_var</name>
      </gpu:variable>
      <gpu:variable>
         <type>int</type>
         <name>global_var_array</name>
         <arrayLength>5</arrayLength>
      </gpu:variable>
   </gpu:constants>
   <gpu:initFunctions>
      <gpu:initFunction>
         <gpu:name>setConstants</gpu:name>
      </gpu:initFunction>
   </gpu:initFunctions>
</gpu:environment>
```

Use of the both initialisation and global constant API functions is demonstrated in the example below. The initialisation function is prefixed with the __FLAME_GPU_INIT_FUNC__ macro which defines it as CPU host code, rather than a GPU kernel. Before the simulation is processed all initialisation functions defined within the GPUXMML model definition are called in the order they appear within the GPUXMML specification.

```
__FLAME_GPU_INIT_FUNC__ void setConstants(){
   float temp_global_variable = 3.14;
   int temp_global_variabe_array[5] = {0, 1, 2, 3, 4};

   set_global_var(&temp_global_variable);
   set_global_var_array(&temp_global_variable_array);
}
```

### 5.1.6 Global Conditions and Non Linear Modelling

Global function conditions add functionality to FLAME which allows some global control over an agent population. A global function condition is expressed in much the same way as a standard function condition, however for the function to be evaluated on any agent, every agent must meet the condition. Technically it allows multiple paths through a simulation, depending on the global state of the agent population. This is highly effective for force resolution and is demonstrated for continuous agents in section 6.2 and for simulating mobile discrete agents in section 6.3. In order to evaluate the result of a global condition for an agent population, the same technique as that for standard agent functions is used to flag agents within a global function condition. The number of agents which meet the condition can then be determined and be compared to the agent count to establish if the agent function should be executed. In either case, a counter can be set to keep track of the number of times the global function condition evaluates to either true or false. In conjunction with the `<gpu:mustEvaluateTo>` element this can then be used to place a limit on the number of times the global condition can be evaluated one way or the other. In practise, this is useful for limiting the number of iterations of particular force resolution steps, which when evaluating physical movement between agents are not guaranteed to ever reduce this enough for a global condition to be evaluated as false.

### 5.1.7 Random Number Generation

RNG on the GPU is achieved through the use of a GPU implementation of the GNU rand48 algorithm [vMAF+07]. This uses independent streams of pseudo-random numbers, which are initialised before a simulation by using a serial RNG rule on the CPU. Each stream uses the previously stored random number in order to generate the next random number and hence each stream is independent of the other. Agent functions which require the use of RNG must set the `<gpu:RNG>` element within the functions GPUXMML specification to true. This indicates the agent function arguments contain an `RNG_rand48` structure which contains the independent streams of random numbers. This structure can then be passed to FLAME GPUs

`rand48` function, which will return a new random number and update the state of the stream for the current thread.

### 5.1.8 Agent Visualisation

In addition to improving the performance of simulation, modelling on the GPU provides the obvious benefit of maintaining agent information directly where it is required for visualisation. Alternatively, CPU simulations incur a large performance cost when transferring large amounts of data to the GPU, which significantly affects the population sizes that can be viewed in real time. In the case of FLAME this is further hindered by the fact that each simulation step must be read from the hard disk, which creates a significant bottleneck even when using compressed binary data storage (rather than XML).

Agent data is stored in CUDA global memory and so the first step to rendering is to make this data available in the rendering pipeline. This can be achieved through the use of OpenGL Buffer Objects, which are able to share the same memory space as CUDA global memory. As with ABGPU, simple agent representations can be rendered with a single draw call, rendering the positions as either OpenGL points or axis aligned point sprites, giving the appearance of more complex geometry. More specifically, FLAME GPU provides an additional visualisation template which maps an agent's positional data into a Vertex Buffer Object (VBO) that is used to displace the simple agent representations using a GLSL shader. Rendering of agents using more complex geometry is also available as a configurable FLAME GPU, XSLT template, and is used to displace sets of vertices which specify 3D geometry. This is achieved by using a CUDA kernel to pass agent data to a Texture Buffer Object (TBO). All vertices of a model are then rendered with a vertex attribute, which corresponds to the agent's position in the TBO texture data. The vertex shader uses this attribute to offset the vertex using the agent's position, with a further fragment shader used to perform per pixel lighting. As it is possible to store model data within a VBO, rendering a population of agents is achieved by setting a unique vertex attribute and drawing the vertex data once per agent. Alternatively, for simple agent models (few vertices) a large VBO, containing a model instance for each agent, can be used with an accompanying VBO holding a vertex attribute array. In this case, the entire

population can be drawn using a single draw call. This is obviously unsuitable for complex agent models, due to the exponential scale of the vertex/attribute data sets. The advantage of either of these instancing based methods is that arbitrary models can be used (of greater complexity than can be represented by point spites), whilst maintaining significantly high performance, by minimising draw calls and GPU data transfer.

## 5.2 Agent Communication

Agent communication is achieved through the use of messages lists that, as with agent memory data, use efficient structured access to ensure memory coalescing. The original FLAME syntax for message iteration has been maintained and requires two specialised message retrieval functions shown in red below.

```
__FLAME_GPU_FUNC__ int function(
      xmachine_memory_agentName* agent,
      xmachine_message_messageName_list* messageName_messages,
      RNG_rand48* rand48)
{
    //Get the first message from the message list
    xmachine_message_messageName_list current_message =
         get_first_messageName_message(messageName_messages);

    while (current_message)
    {
        /* Process the message data here */

        //Get the next message from the message list
        current_message = get_next_messageName_message(
                        current_message,
    messageName_messages);
    }

}
```

Within the original FLAME for the CPU only brute force message iteration was supported. FLAME GPU improves the performance of message iteration by providing

a range of specialised message retrieval functions, which are suitable for various agent distributions and communication patterns. The first of these is a brute force message communication technique which is described in Section 5.2.1. This is directly comparable with the technique used in the original FLAME framework for message iteration on single CPU host. A spatially partitioned technique is described in Section 5.2.2, using Euclidian spatial partitions to minimise the number of messages each agent is required to process. This is conceptually similar to how FLAME is able to distribute agents between multiple processors. Finally, a discrete message technique is discussed in Section 5.2.3. This is most suitable for cellular automaton and assumes that homogeneous agents are placed in a regular grid.

### 5.2.1 Brute Force Message Communication

Brute force message communication is required in systems where there is a high degree of communication between agents, or where spatial partitioning is unsuitable. In early implementations of ABGPU, brute force messaging was implemented by treating the processing of each agent to agent interaction as a unique thread of execution. Whilst this is easily parallelised in a shader based implementation, its performance can be greatly improved in CUDA by making use of shared memory to reduce global memory reads, and hence increase the arithmetic intensity. To achieve this FLAME GPU implements a tiling based technique inspired by Nyland et al [NHP07]. This serialises message access across agent threads, by loading groups of messages into the GPU's shared memory. Technically, this requires messages to be split into groups, with the first message group being loaded into shared memory by the `get_first_message` function (Figure 26). Following this, each thread within the same thread block uses a broadcast access pattern to sequentially read the same message from shared memory using the `get_next_message` function. After each thread has exhausted the messages within the message group (or tile) the `get_next_message` function synchronises threads in the block and loads the next group of messages into shared memory (Figure 26).

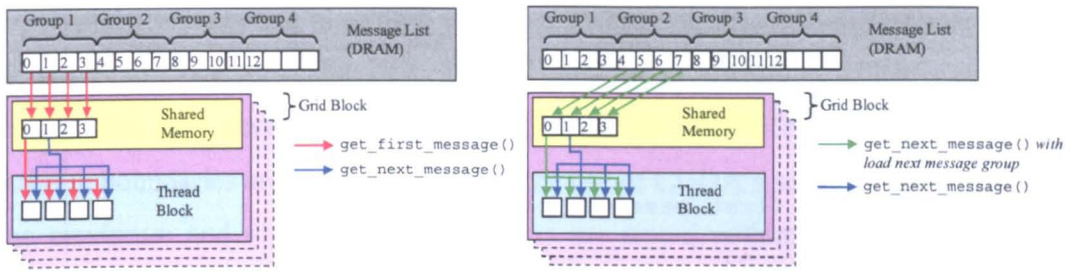



**Figure 26 - Message group loading when requesting the first and next message.**

Message group size and thread block size are the same, so individual threads are responsible for reading a single message into shared memory concurrently. To avoid all thread blocks reading the same groups, the first group load of any block (issued by the `get_first_message` function) starts by loading data into shared memory at offset locations in global memory. Any access to shared memory is preceded and immediately followed by a call to the CUDA `__syncthreads` function (shown in Figure 27 in green), which ensures all data threads in the same block, reading from the group of messages, are at the same stage with no race conditions. The `get_next_message` function is shown in pseudocode in Figure 27. Each message holds a message position value indicating its position in the message list. This is then use to check if the end of the message list is reached (line 4), or if the message position is equal to the first message position (line 7). If the end of the message list is reached this does not necessarily indicate that all messages have been iterated, due to the initial offsets of message groups used in the `get_first_message` function. In this case, the message position is set to zero with message iteration only ending when the first message position is reached, indicating the full list has been iterated. The message position is also used to calculate the current tile, and then the message position within the current tile. If the message position within the current tile reaches the `blocksize` then this indicates that a new message group must be loaded (line 12).

```
1.  INCREMENT message position
2.  CALL wrap WITH message position
3.   RETURN i
4.  IF (i > message count) THEN
5.   i = 0
6.  ENDIF
7.  IF (i == first message position) THEN
8.   RETURN FALSE
9.  ENDIF
10. SET tile TO floor(i/blockSize)
11. SET i TO i MOD blockSize
12. IF (i == 0) THEN
13.  CALL __syncthreads
14.  SET index TO (tile*blockSize) + threadID
15.       CALL readMessageSoA WITH index
16.       RETURN temp_message
17.       SET sharedmessages[threadID] TO temp_message
18.       CALL __syncthreads
19. ENDIF
20. RETURN sharedmessages[i]
```

**Figure 27 – Pseudocode of the message iteration algorithm used for loading the next available message.**

As agent, and hence message list, sizes are liable to change through out the simulation process, it is important to consider thread path divergence to avoid any deadlock problems. Unused threads are likely and are a result of the total number of agents not being a multiple of the thread block size. Rather than leave these threads idle, it is essential for this messaging iterating technique that they follow the same path as occupied threads within the block. Whilst this results in agent data beyond the last agent in the list being processed with the agent function, the path these threads follow ensures that full message groups are loaded into shared memory. Likewise, it is vitally important that there are no conditional dependencies on message iteration, or breaks from the message loop. If any agent becomes excluded from the message loop the agent's thread will fail to load shared message data (causing a thread synchronisation deadlock in the `get_next_message` function) and will result in the simulation ending.

### 5.2.2 Limited Range Communication

Limited range communication is based on the same technique as that which is used within ABGPU, the obvious difference being that messages, rather than agents, are iterated using the dynamic data structure. Green's [Gre07] implementation of the

same spatial partitioning algorithm uses fast radix sort. FLAME GPU incorporates the same radix sort algorithm described by Satish et al. [SHG09], which is available through the most recent (v1.1) CUDPP library. As with Green's implementation, partition boundaries are generated through the use of CUDA's scattered write support. The maximum and minimum boundary lists are stored within a partition boundary matrix structure. This must be passed as an argument (as well as the agent's position) to message retrieval functions as demonstrated in the code sample below.

```
__FLAME_GPU_FUNC__ int function(
      xmachine_memory_agentName* agent,
      xmachine_message_messageName_list* messageName_messages,
      RNG_rand48* rand48,
      xmachine_message_messageName_PBM* partition_matrix)
{
     //Get the first message from the message list
     xmachine_message_messageName_list current_message =
          get_first_messageName_message(
                         messageName_messages,
                         partition_matrix,
                         agent->x, agent->y, agent->z);

     while (current_message)
     {
         //Process the message data

         current_message = get_next_messageName_message(
                         current_message,
     messageName_messages,
                         partition_matrix);
     }

}
```

As the processing of messages is unique to an agent function and is specified within the message loop, it is not possible to specify a simple kernel which computes interaction between agents directly (as with [Gre07, ALT08]). Instead, message iteration is provided through an algorithm that, given an existing message (stored in shared memory), returns the next message (shown in Figure 28). More specifically, it

loops through the neighbouring partitions looking for a partition that is not empty and hence contains messages (Figure 28, line 21). When a partition containing messages is found, or if there are more messages in the previous messages partition (Figure 28, line 10), then a message is returned. The variable `relative_cell` holds a vector of integers in the range $-1 \leq x \geq 1$, which identifies the relative position of the current message to the `agent_grid_cell`. The function `nextCell` therefore determines if relative position can be incremented (i.e. when all 27 unique values have been exhausted the function returns false) and the function `cellPosition` calculates the cellular partition position of a continuous valued point, within the partition space. Partition bounds are specified for each message within the GPUXMML model file. If the point (x) in any dimension lays outside of the bounds dimensions (m) but within the cell position range of;

```
2m > x > -m
```

Then the position is wrapped using the following macro which avoids the use of an expensive modulus operation.

```
#define WRAP(x,m)  (((x)<m)?(((x)<0)?(m+(x)):(x)):(m-(x)))
```

The function `hashCellPosition` performs a hash function mapping the partition position to a unique integer [KSW04]. The algorithm does not perform an additional radial check on messages and roughly 1/3 of the messages returned will be outside the message range of the agent. It is therefore important that messages are filtered, using a user defined distance check within the agent function, as they would be when using the brute force technique.

```
1.  IF first message THEN
2.    SET relative_cell TO null
3.    SET cell_index TO 0
4.    SET cell_index_max TO 0
5.    CALL cellPosition WITH agent position
6.         RETURN agent_grid_cell
7.  ENDIF
8.  SET move_cell TO true
9.  INCREMENT cell_index
10. IF (cell_index < cell_index_max) THEN
11.   SET move_cell TO false
12. ENDIF
13. WHILE(move_cell)
14.   IF(CALL nextCell WITH relative_cell RETURN bool) THEN
15.     INCREMENT next_cell
16.     SET next_cell TO agent_grid_cell + relative_cell
17.     CALL hashCellPosition WITH next_cell
18.          RETURN next_cell_hash
19.     SET cell_index_min
20.         TO cell_start_boundaries[next_cell_hash]
21.     IF (cell_index_min != null) THEN
22.       SET cell_index_max
23.           TO cell_end_boundaries[next_cell_hash]
24.       SET cell_index TO cell_index_min
25.       SET move_cell TO false
26.     ENDIF
27.   ELSE
28.     RETURN NULL
29.   ENDIF
30. ENDWHILE
31. RETURN message from message list AT cell_index
```

**Figure 28 - Pseudocode algorithm for spatial message loading**

### 5.2.3 Non Mobile Discrete Agents

Both brute force and limited range communication strategies are ideal for use with continuous valued mobile agents. In the case of regularly spaced, non-mobile agents in 2D discrete space (CA), a different communication pattern can be used to provide efficient communication between agents. This pattern makes the assumption that agents are ordered within a regular grid which does not contain gaps (or missing agents). Within this grid, agents are able to output a single message into a message grid structure of equal dimension to the agent grid. This message grid can then be read by both discrete and continuous agents, using either a shared memory or textured based technique respectively. Messages are wrapped in both techniques both horizontally and vertically.

The shared memory method of discrete message communication is only available for discrete agents and is currently only available for 2D populations of agents. As

discrete agents are conceptually arranged within a grid, kernels are launched using a 2D grid of threads blocks rather than the previously used linear block structure. Despite this, all discrete agent and message data is stored and accessed on the GPU using a linear SoA format, as was used for continuous agents. Whilst this requires a computation to convert a threads 2D block position into a linear memory address, it ensures messages remain in the message list format and can therefore be read by continuous agents. As agents are structured regularly and communicate over a limited range, the number of messages required by agents is relatively small and therefore can all be stored within shared memory, reducing expensive global memory reads. This requires that during the `load_first_message` function every agent loads at least a single message, with agents on the edge of the thread block performing additional work to ensure message's output from neighbouring blocks are also loaded into shared memory (Figure 29, Figure 30).

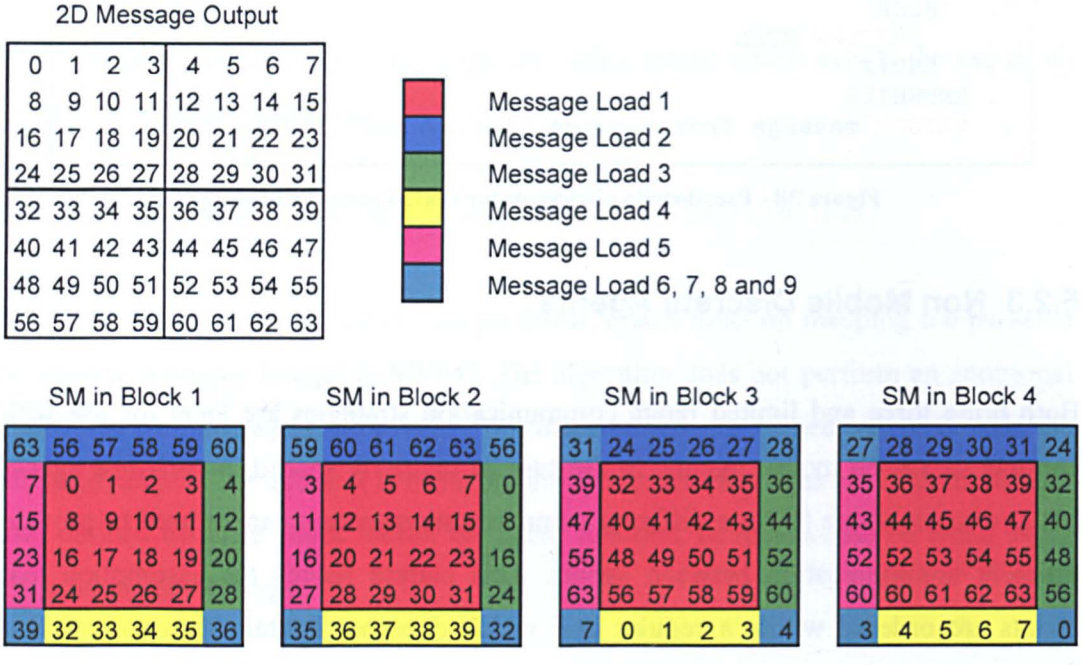



**Figure 29 – The loading of messages into shared memory for non mobile discrete agents. The grid top left represents the output of messages into a 2D grid. The 4 coloured blocks show the configuration of shared memory for each of the 4 thread blocks launched.**

This technique is not optimal, as some threads will be idle during the extra message loads, but it is a trade off between the alternative technique, that requires an oversized thread block. In this case all messages can be loaded into shared memory with only a

single load per thread, however the additional border threads remain idle over the rest of the agent function. This creates a scenario where increasing the message range degrades performance, as an increased number of threads are required that remain idle. In contrast, the multiple message load technique handles increased message ranges well, using the previously idle threads to perform the additional work. In order to explain this, consider a case where a message range of only 1 cell is used. In this case only single agents on the very bounds block are require to load additional data with agents on two bounds (i.e. corners of the thread block) loading a total of 4 messages.

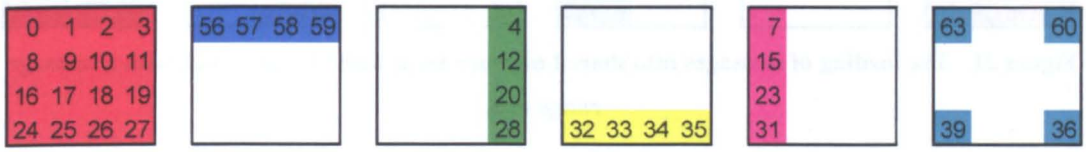



**Figure 30 – The Message load steps and corresponding message loads for each thread corresponding to Figure 29.**

When the range is then increased to 2 (Figure 31) the additional message loads are performed by the next nearest agents during the time that they were previously idle. This therefore scales well assuming that message ranges do not exceed the block size that agents are processed in. In order to iterate messages the `load_next_message` function iterates through the messages within range, returning a pointer to the message in shared memory. A value stored within each message is used to indicate the current messages position and determine the next message to return. This value starts at (-range, -range) and ends at (range, range) ignoring the message (0, 0) which is the current agent's message output.

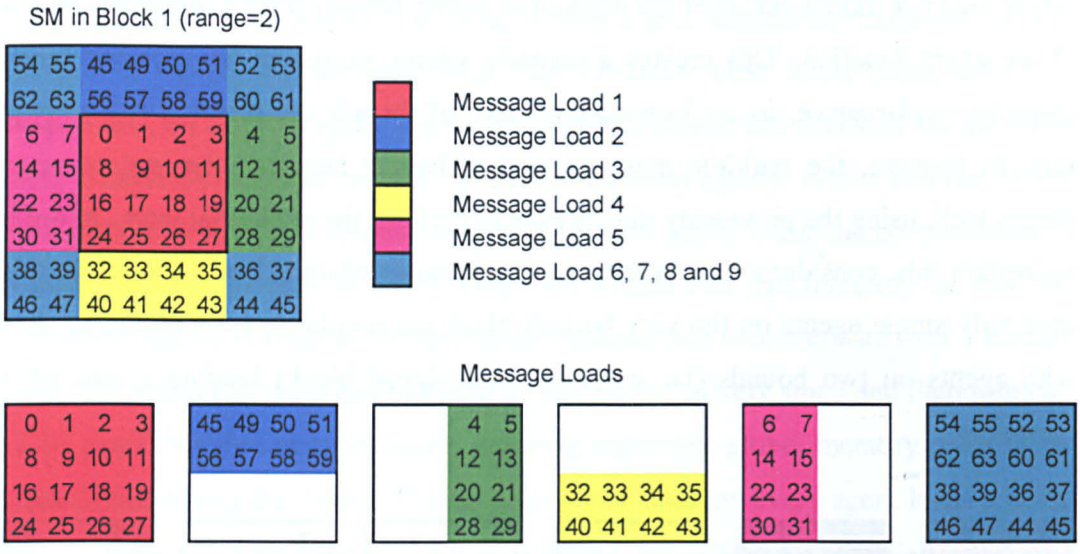



**Figure 31 - The loading of messages into shared memory for a single thread block with a message range of 2.**

Continuous agent functions are unable to use the shared memory method for loading discrete messages as their non regular physical location does not guarantee that agents will read from the same regular grid of messages. Even if this were strictly enforced, the variable number of agents in continuous systems could not guarantee that all messages would be loaded. Instead of using shared memory, a textured based method can be used taking advantage of the GPU's built in texture cache. This method is used by binding each 'message variable data array' to a linear texture. Each thread then uses a 1D texture lookup and stores the message data in shared memory. The message is only read by the thread which placed it in shared memory and there is no communication between threads, other than through the texture cache. This can also be forced for discrete agents by using the `CONTINUOUS` agent type value for the templated message functions as follows.

```
enum AGENT_TYPE{
 CONTINUOUS,
 DISCRETE_2D
};

get_first_messagename_message<enum AGENT_TYPE>(...)
get_next_messagename_message<enum AGENT_TYPE>(...)
```

## 5.3 Summary

This chapter has demonstrated the techniques contained within the XSLT templates which make up FLAME GPU. The use of the CUDA architecture has allowed an extremely powerful and flexible technique to be described which address the limitations of swarm systems described in previous chapters. In the next chapter, the performance of FLAME GPU is considered through careful benchmarking of the various communication techniques and through the use of a number of case studies, which test both the flexibility and performance.

# Chapter 6

# Evaluating FLAME GPU

The previous chapter described the implementation of the FLAME GPU framework. This chapter focuses on performance, through the implementation of simple benchmarking models. Flexibility of the framework is then considered by describing the implementation of both a cellular biology model and a mobile discrete artificial society. In both cases the use of FLAME GPU's new global function conditions provides a mechanism for non linear simulation steps (simulation steps which do not represent integration in the usual discrete time unit), allowing recursive functionality within a single time step. This is used to efficiently resolve both intercellular forces and transactional movement between discrete mobile agents respectively.

All results within this section were obtained on a single Windows XP PC with an AMD Athlon 2.51 GHz Dual Core Processor with 3GB of RAM and a GeForce 9800 GX2. Whilst the GX2 card consists of two independent GPU cores, only a single core has been used for CUDA processing with the other handling the active display. This technique allows the circumvention of the windows watchdog timer[14], which halts GPU kernels exceeding five seconds in execution time. Where original FLAME

[14] A security feature which protects the Windows XP OS from graphics card crashes. The watchdog can be disabled within Windows Vista/7 using a registry change (Linux does not limit GPU kernel execution time in this way).

models have been implemented they have been updated to match the FLAME GPU models as closely as possible (including single precision agent and message variables). All original FLAME models have been compiled using GCC with MingGW and full compiler optimisations.

## 6.1 Benchmarking

This section describes the benchmarking of FLAME GPU. It is divided into three sections which evaluate the three message communication techniques described in the previous chapter.

### 6.1.1 Evaluation of Brute Force Messaging

In order to evaluate the performance of brute force message communication, a simple force resolution model (referred to as the Circles model), has been implemented. The same model was previously used for benchmarking of FLAME on HPC architectures (reviewed in Section 4.1.1) where the initial states were generated using the same state generation code as the results presented here. It consists of only a single agent and message type with three agent functions, which output and input a location message and move the agent accordingly. Speedup is calculated by considering the iteration time of a single time step of each model, on both the GPU and CPU implementations in single floating point precision. In order to understand where the performance increase of a message access is achieved, Figure 32 shows the effect of the various implementation techniques described Chapter 5 .

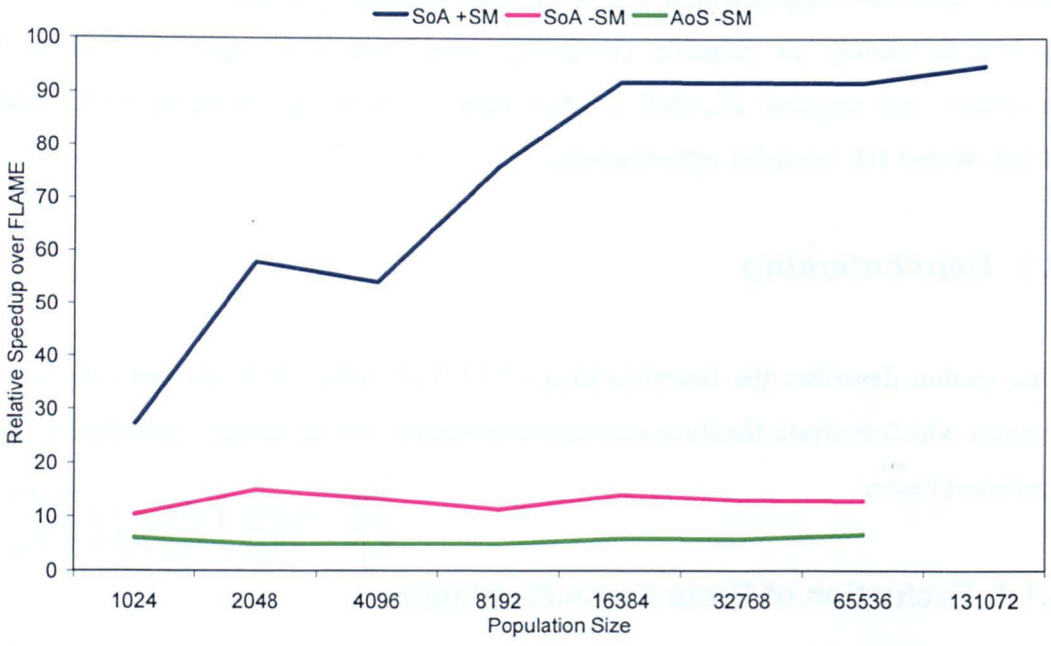



**Figure 32 – Performance of the Circles model using brute force message iteration with various optimisations enabled.**

Within Figure 32, +SM represents the use of the message tiling technique utilising shared memory, or alternatively in the case of –SM, messages are loaded directly from global memory. AoS and SoA represent the uncoalesced and coalesced global data storage pattern used to access global memory (described in Section 5.1.1). Figure 32 clearly shows that whilst coalesced memory access makes a considerable difference to the model, tiling messages into shared memory within message loops is where the majority of performance can be attributed. The CUDA visual profiler has been used to examine the main message processing function. This drew attention to the large number of warp serialisations, which are the result of shared memory bank conflicts (which occur in half warps during shared memory access requests) when tiling the message data into shared memory. To avoid bank conflicts in the message functions, message structures in shared memory are padded to occupy an odd number of 32 bit sizes. As shared memory addresses are mapped into 16 (the size of a half warp) cyclic, 32 bit word banks, using an odd number of 32 bit words results in a stride pattern where banks are not accessed by multiple threads. Table 5 shows this by considering two cases where a number of consecutive threads read a 32byte message from an array in shared memory. Shared memory positions are represented as byte offsets into the array and bank positions are calculated by converting the shared

memory position into bits, dividing by the size of the banks and cycling the 16 banks available. With no padding the access pattern has only 4 unique addresses and a 4 way bank conflict which results in accesses being serialised. Using an offset of 32 bits (4bytes) ensures that each consecutive message access is mapped to a unique memory bank, which guarantees a one way conflict (or single access per bank). To ensure the same 32 bit padding avoids conflicts for any message size, all messages in shared memory are aligned to an even 16 byte size.

**Table 5 – Memory bank conflicts when accessing message data from an array in Shared Memory.**

| Message or Thread Index | SM Position no Padding | SM Bank no padding | SM Position 4byte padding | SM Bank 4byte padding |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 32 | 8 | 36 | 9 |
| 2 | 64 | 0 | 72 | 2 |
| 3 | 96 | 8 | 108 | 11 |
| 4 | 128 | 0 | 144 | 4 |
| 5 | 160 | 8 | 180 | 13 |
| 6 | 192 | 0 | 216 | 6 |
| 7 | 224 | 8 | 252 | 15 |
| 8 | 256 | 0 | 288 | 8 |
| 9 | 288 | 8 | 324 | 1 |
| 10 | 320 | 0 | 360 | 10 |
| 11 | 352 | 8 | 396 | 3 |
| 12 | 384 | 0 | 432 | 12 |
| 13 | 416 | 8 | 468 | 5 |
| 14 | 448 | 0 | 504 | 14 |
| 15 | 480 | 8 | 540 | 7 |
| Unique addresses | | 2 | | 16 |
| Bank conflicts | | 8 | | 1 |

Avoiding multiple requests to the same bank avoids requests having to be serialised, allowing maximum memory access performance from shared memory. The reason this is not shown in Figure 32 is that this optimisation makes only a minute difference to the performance (typically about 0.5x additional speedup). This is most likely a result of bank conflicts being hidden by the global memory latency when reading data to be placed in shared memory. Additionally when agent functions read message data from within message loops, there are no bank conflicts to avoid as message's

variables are read by simultaneous threads using a broadcast access pattern. Despite this the same padding technique has been applied to all three messaging techniques.

The initial fluctuation in Figure 32 can be attributed to the fact that, at this relatively low agent count, the multiprocessors are under utilised resulting in unpredictable amounts of idle time and global memory access latency coverage. In fact, for population sizes up to, and including 4096 (and a thread block size of 128), the maximum number of thread blocks per multiprocessor is limited by the register count (of 8,192 for compute capability 1.1 cards), to 2 (Table 6). This suggests that 4,096 agents (or 32 blocks of 128 agent threads) are the minimum number required to fill all 16 of the multiprocessors.

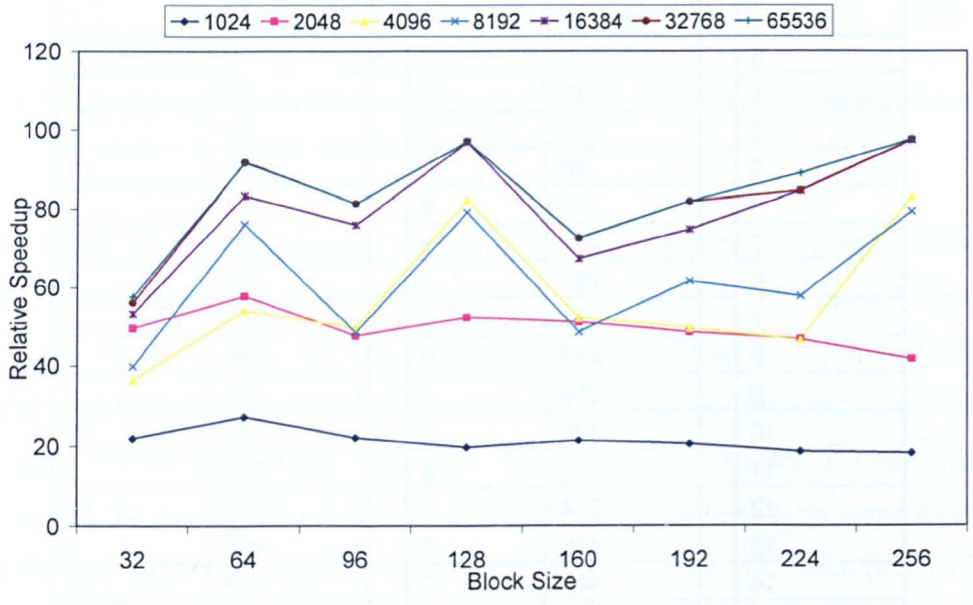



**Figure 33 – Performance of the Circles model at various thread block and population sizes.**

In order to consider the effect of varying the thread block size for brute force message communication, Figure 33 shows the results of the Circles model at a range of population size and block size combinations. Block sizes are incremented by 32 threads, as this is the size of a single warp (which is the smallest unit of computation and cannot be broken down any further). The maximum thread block size of 256 is used, as a value any larger than this exceeds the maximum register count for the available hardware. The results show a relatively chaotic fluctuation over the varying blocks sizes for each population size. Having said this, block sizes with particularly poor performance are relatively consistent over the varying populations. For example,

block sizes of 96 and 160 have a relatively low performance in contrast with the nearest block size of 128. The explanation of this can be accounted for by considering the occupancy of the multiprocessors. Occupancy is defined as the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps [NVI07]. A higher occupancy indicates that a multiprocessor is able to be kept busy with processing, effectively hiding any global memory read, or register dependency latency. Occupancy can be limited by a number of factors, including the maximum number of warps per multiprocessor, the available registers per multiprocessor and the available shared memory per multiprocessor.

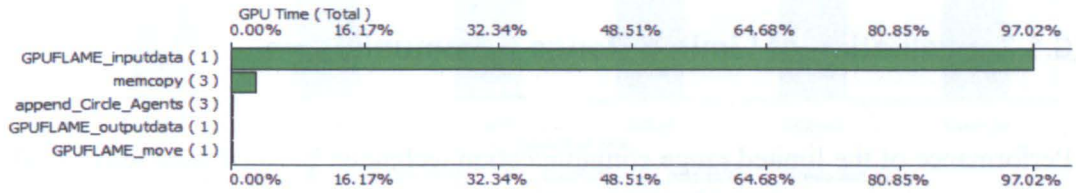



**Figure 34 – Distribution of processing time for the Circles model using brute force message iteration.**

Figure 34 (obtained using the CUDA visual profiler with a population size of 16,384 and a block size of 128) clearly indicates that the bottleneck stage of the Circles model is the agent function which deals with message iteration. This is hardly surprising due to the $O(n^2)$ message reading operation which is used. Table 6 provides a detailed breakdown of `inputdata` function for varying block sizes with a population of 16,384 agents. It shows that the block sizes of 64, 128 and 256 have the highest occupancy accounting for the performance reported in Figure 33. It also shows that the occupancy of the `inputdata` function is generally limited by the register usage. Whilst it is possible that the use of registers could be reduced through careful optimisation, occupancy does not guarantee higher performance. As long as the number of active warps per multiprocessor is enough to hide any global memory latency, then attempts to increase the occupancy may in fact have negative effects. Additionally, the next generation of hardware contains double the number of registers per multiprocessor (16,384), and will instead be limited by the availability of shared memory (with occupancy of 0.44 and a block size of 224).

**Table 6 – Occupancy of various thread block sizes for the Circles `inputdata` function using brute force message iteration.**

| | | Active / Multiprocessor (MP) | | | Max Blocks / MP (Limited By) | | |
|---|---|---|---|---|---|---|---|
| *Block Size* | *Occupancy* | *Threads* | *Warps* | *Blocks* | *Warps* | *Registers* | *SM* |
| 32 | 0.167 | 128 | 4 | 4 | 8 | 4 | 10 |
| 64 | 0.333 | 256 | 8 | 4 | 8 | 4 | 6 |
| 96 | 0.250 | 192 | 6 | 2 | 8 | 2 | 4 |
| 128 | 0.333 | 256 | 8 | 2 | 6 | 2 | 3 |
| 160 | 0.208 | 160 | 5 | 1 | 4 | 1 | 2 |
| 192 | 0.250 | 192 | 6 | 1 | 4 | 1 | 2 |
| 224 | 0.292 | 224 | 7 | 1 | 3 | 1 | 2 |
| 256 | 0.333 | 256 | 8 | 1 | 3 | 1 | 1 |

### 6.1.2 Evaluation of Limited Range Communication

Performance of the limited range communication technique has also been tested using the Circles benchmarking model. This has only been changed from the previous section by adding the partition boundary matrix structure argument to the message retrieval functions. As FLAME GPU uses a brute force messaging technique per node, it does not make sense to evaluate the performance by using a relative speedup as before (as it will obviously be exponential). Instead, the performance of a single simulation stage for various thread block sizes is shown in Table 7.

**Table 7 – Performance times for the Circles model using limited range message iteration at various thread block sizes.**

**N represents the population size. Times are displayed in milliseconds.**

| N | 32 | 64 | 96 | 128 | 160 | 192 | 224 | 256 |
|---|---|---|---|---|---|---|---|---|
| 1024 | 0.94 | 1.05 | 0.90 | 0.86 | 0.93 | 0.89 | 0.95 | 0.88 |
| 4096 | 1.24 | 1.25 | 1.30 | 1.22 | 1.39 | 1.22 | 1.24 | 1.25 |
| 16384 | 2.45 | 2.48 | 2.62 | 2.53 | 2.76 | 2.81 | 2.77 | 2.60 |
| 65536 | 9.09 | 9.34 | 9.47 | 9.23 | 9.22 | 9.31 | 9.45 | 9.42 |
| 262144 | 33.74 | 37.99 | 36.88 | 37.39 | 36.61 | 36.83 | 37.81 | 38.12 |
| 1048576 | 136.28 | 169.73 | 147.39 | 172.98 | 145.21 | 165.34 | 151.26 | 177.06 |

Figure 35 shows an additional breakdown of the percentage of GPU time spent on each kernel function of the Circles model. It also gives an indication of the CPU to GPU transfer speed required to upload the data from the host to the GPU device (memcopyHtoD). Both Figure 35 and Figure 36 make reference to other FLAME GPU kernels. These include kernels for the hashing of location messages, reordering of location messages and appending of Circle agents into the default state from the working list.

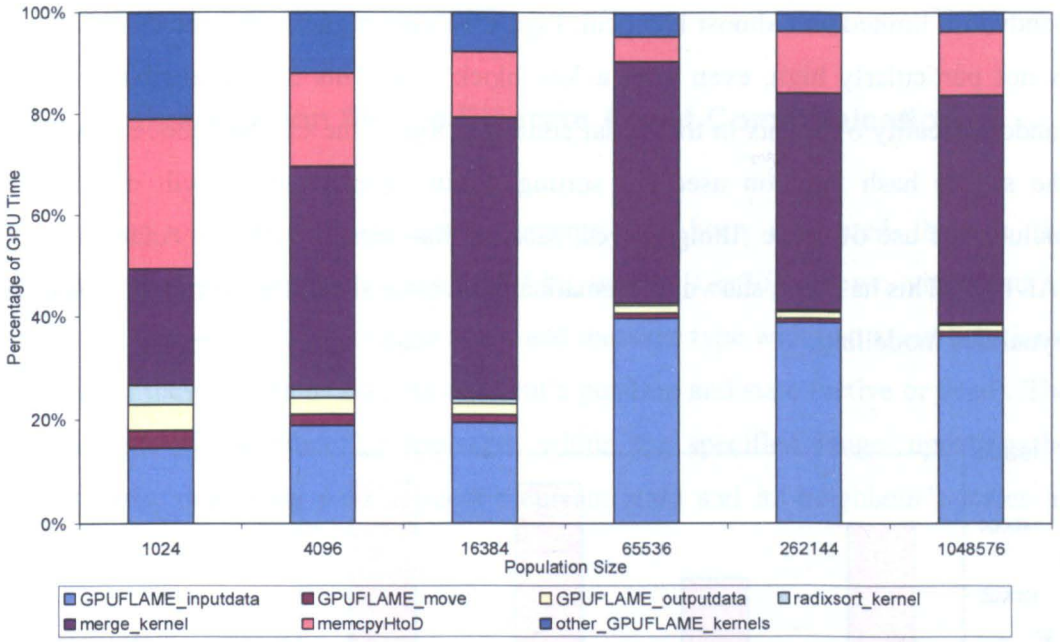



**Figure 35 – Breakdown of where GPU time is spent during simulation of the Circles model using limited range message iteration at various population sizes.**

From the results obtained it is clear that the thread block size of 32 performs consistently well, particularly for larger population sizes, where it demonstrates the highest performance. Figure 36 evaluates this for a population size of 1,048,576 agents by highlighting a breakdown of GPU time during a single simulation step (for thread block sizes of both 32 and 128). It is clear from this that the major contributing factor for the difference between thread block sizes is the `inputdata` function which takes 30% longer for the larger thread block size. Unfortunately, the most obvious kernel metric of the occupancy gives no explanation of this performance difference. For a thread block size of 32 the occupancy is limited by the register usage to 16%, where as for a thread block size of 128 the occupancy is limited by both registers and shared memory to 33%, suggesting that, if anything, the larger block size should perform better. Likewise, other performance measurements for the kernel show no difference in the number of branches, memory reads or instructions executed. The performance difference is therefore attributed to the kernels' use of the texture cache, which for smaller thread block size has an increased locality between threads. This leads to a higher cache hit rate reducing the number of global memory reads. To confirm this, Figure 36 also shows the simulation performed without the use of

texture caching (using global memory reads). In this case, the performance is bandwidth limited and almost identical. Figure 36 also suggests that the cache hit rate is not particularly high, even with a low block size. This can be attributed to the random locality of agents in the initial configuration of the Circles model, as well as the simple hash function used for sorting. Future improvements will most likely include the use of space filling curves, such as that demonstrated by Anderson et al [ALT08]. This has been shown to dramatically increase the cache hit rate in molecular dynamics modelling.

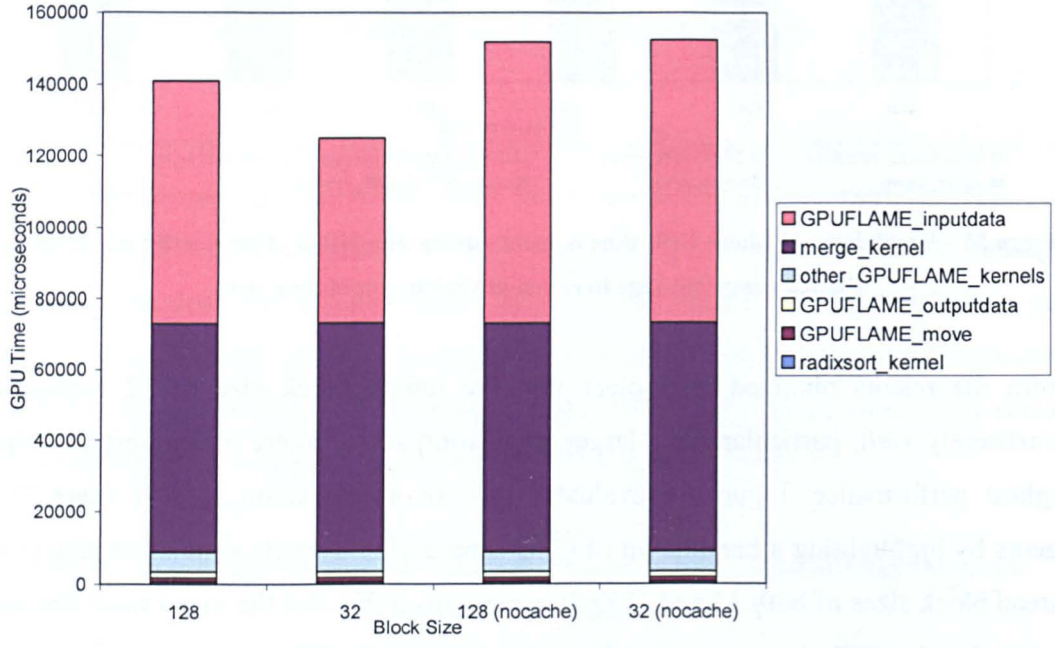



**Figure 36 – Performance times demonstrating the effect of the texture cache for the `inputdata` function of the circles model using limited range message iteration.**

In contrast with the benchmark performance of the Circles model on HPC architectures, FLAME GPU performs considerably well. Both the SCARF and HAPU architectures were able to perform a single simulation step in double precision in just over 6 seconds, using a total of 100 processing cores. The results in Table 7 suggest that even the worst performing thread block size took 0.2 seconds to complete a simulation step. Whilst it would be interesting to consider the performance of FLAME GPU in full double precision, the necessary hardware has only recently been made available. A rough indication of the performance can still be made by considering the ratio of 8 to 1 double to single precession units offered by the

hardware. Even order of magnitude in performance degradation allows FLAME GPU to easily compete with HPC architectures.

### 6.1.3 Evaluating Non Mobile Discrete Agent Communication

Performance of non mobile discrete agents has been evaluated through the implementation of Conway's Game of Life model [Gar70] (shown visualised in Figure 37). This consists of a single agent and message type with two agent functions. The first of these functions outputs an agent's position and state (active or dead). The second reads all neighbouring messages within the specified range, updating the agent's state, depending on the agent's current state and its neighbour's states as follows.

- If the agent is active:
  - If there are no active neighbours the agent dies of loneliness;
  - If there are four or more active neighbours the agent dies of overpopulation.; or
  - If there are either two or three active neighbours the agents survives.
- If the agent is dead:
  - If there are exactly three active neighbours the agent becomes active.

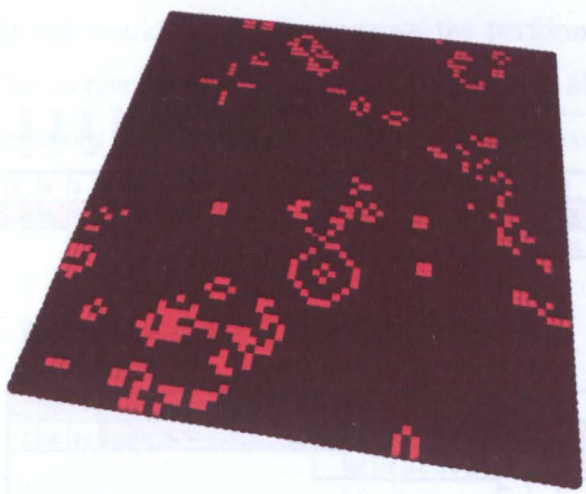


**Figure 37 – A visualisation of the Game of Life Model implemented using non mobile discrete agents.**

**Table 8 –Performance of the Game of Life model at two block sizes using both the shared memory and texture based methods.**
**SMC refers to shared memory with bank conflicts.**

| | SMC (64) | SM (256) | TEX (256) | TEX (64) |
|---|---|---|---|---|
| **256** | 0.305 | 0.267 | 0.361 | 0.352 |
| **4096** | 0.356 | 0.321 | 0.379 | 0.418 |
| **65536** | 1.586 | 0.592 | 0.982 | 1.450 |
| **1048576** | 19.650 | 4.946 | 9.203 | 18.140 |

Table 8 shows the results of performing the simulation, using both the discrete shared memory (SM and SMC) and continuous texture based (TEX) methods for a sample of population sizes, at two thread block sizes. The thread block sizes are limited to sizes that fit into a 2D grid (i.e. their square root value must be a whole number), and therefore only sizes of 64 and 256 are suitable. For a block size of 256 the shared memory technique is roughly twice as fast as using the texture based alternative. However, for the smaller block size the texture based method outperforms the shared memory technique. The reason for this is due to shared memory bank conflicts, which occur only in block sizes where the width is less than the size of a half warp (16). Figure 38 demonstrates that for a block size of 256 (and a block width of 16) the first half warp of the thread (in red) is able to read or write 16 consecutive messages. Using the conflict free technique described in section 6.1.1, this avoids any conflicts (shown on the right). For a block size of 64 (and a width of only 8) the first half warp does not read consecutive messages and as a result bank conflicts occur, in this case in bank 12 and 13.

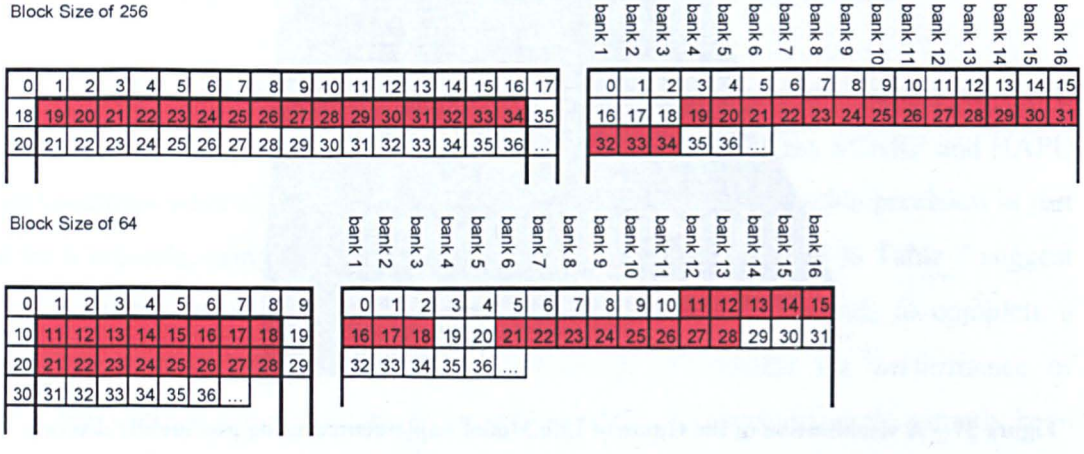



**Figure 38 – Reading or writing of messages from shared memory with block sizes of 256 and 64.**

**The numbers indicates the index position of the message within an array in shared memory. The LHS shows SM, where the red represents the threads of the first half of a warp. RHS shows the banks in which each thread is accessing.**

To avoid this it would be possible to apply padding to each row of the shared memory grid of 16, less twice the message range (in the case of the example, 14). This would shift the second half of the half warp to a conflict free position (Figure 39), but would waste a large amount of shared memory drastically, reducing the occupancy of the multiprocessors.

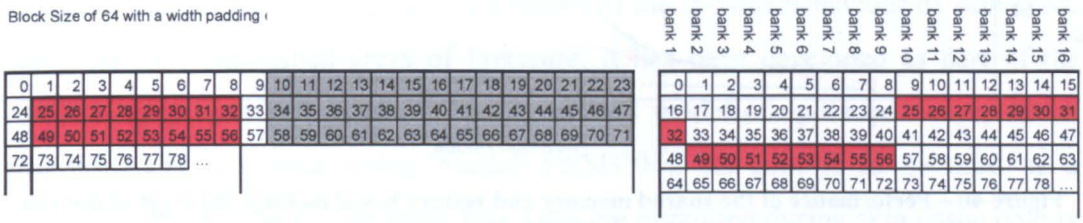



**Figure 39 – Proposed padding to avoid shared memory bank conflicts for a 2D thread block of 64 threads.**

For the same reason that bank conflicts occur in the smaller block size, memory coalescing does not occur for message reads (or writes) due to the non consecutive location of messages in global memory. The effect of this is visible in both the shared memory and texture based results of Table 8, as both sets of results use a 2D thread block to store agents. It is expected that the use of more recent hardware (compute capability 1.2 or above) would drastically improve the performance of the smaller block size. This is due to relaxed coalescing rules which would allow coalesced reads and writes, despite memory requests being non-consecutive across threads.

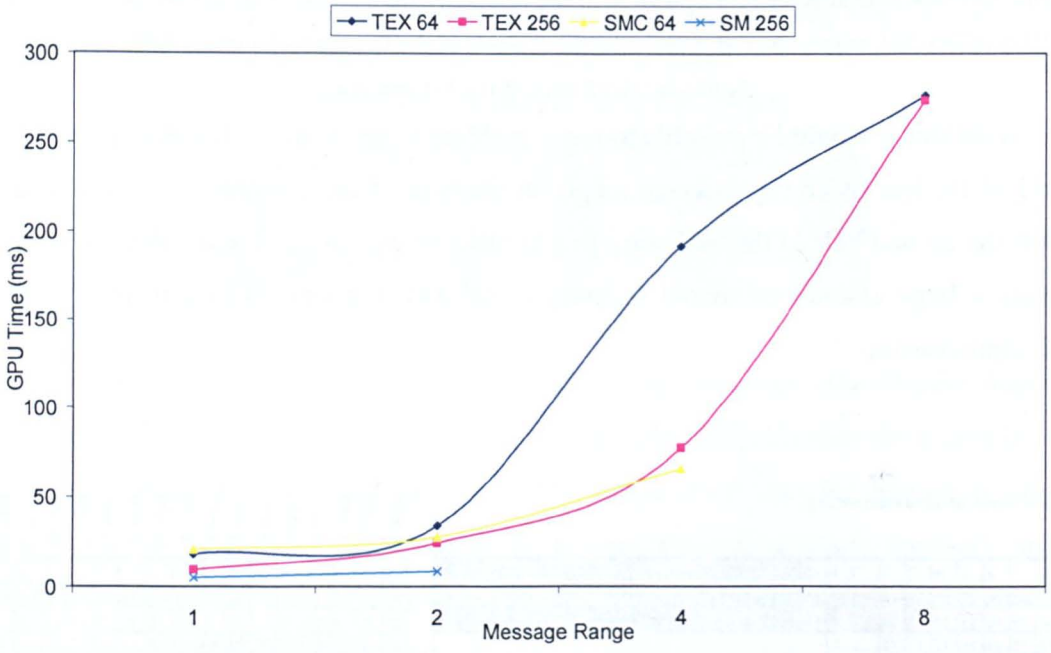



**Figure 40 – Performance of the shared memory and texture based message iteration of discrete agent messages at a block size of 64 and 256.**

Figure 40 shows the effect of varying the message range for both the shared memory and textured based techniques for a population size of 1,048,576. The shared memory technique has a small number of divergent branches, that are the result of threads loading the border messages into shared memory as described in Section 6.1.1. Increasing the message size confirms that, despite the additional loads, the number of divergent branches remains, allowing the technique to perform well for a message range of up to 4. Beyond this range (and beyond a range of 2 for block sizes of 256) the shared memory technique is unfortunately limited by the shared memory availability per multiprocessor.

## 6.2 Cellular Level Tissue Modelling

Driven by the availability of experimental data and ability to simulate a biological scale, which is of immediate interest, the cellular scale is fast emerging as an ideal candidate for middle-out modelling. This section describes the implementation of a cellular level, Keratinocyte Colony model [SMC+07] which has been implemented in FLAME GPU. Whilst the functionality of this model remains the same as that described by Sun et al., the model has been modified to limit agent functions to only a

single message input and output (according to the GPUXMML specification). All agent functions have also been rewritten to avoid any conditional dependency on message iteration or breaking from the message loops. The model specification has also been updated for compatibility with the GPUXMML schema and the function code has been parameterised as described in section 4.3.2. The complete GPUXMML model specification and function code can be found in Appendix B.

### 6.2.1 The Keratinocyte Model

The Keratinocyte model [SMC+07] is a model of the in-vitro behaviour of skin cells, based on well established areas of literature. It has been developed as part of the Epitheliome project and extends a computational model of epithelial tissue, which was originally developed using Matlab [WSH+04]. Its goal is to be used as a predictive model to help understand how cells are organised during skin tissue colony formation. This promotes the development of methods to artificially produce reconstructed skin for patients who have suffered skin loss. Of particular interest is the use of the model to predict the success of scratch wound healing. This is examined by simulating a virtual scratch under differing calcium conditions. The model is able to predict if the wound is able to heal, depending on the location of colony epicentres. Likewise various hypothesis of the importance of specific biological rules have been tested with in-virtuo experimentation using the model which has later been validated in-vitro [SMC+07, TPM+08].

Within the Keratinocyte model a reasonable level of abstraction has been used, which includes a simple spherical cell representation of cells (with constant size and shape) with no complex cell signalling mechanisms. Within the model there are 4 distinct cell types which are as follows.

1. Stem Cells – Are found at cell colonies and divide to produce two daughter stem cells, providing there is space to do so. They remain fairly static throughout the simulation.
2. Transit Amplifying (TA) Cells – Like stem cells, will divide to produce two TA cells, if there is sufficient space
3. Committed Cells (comm) – Both stem and TA cells become committed cells after a differentiation processes. This occurs in stem cells by first becoming

TA cells when they are located on the edge of a cluster which reaches a certain size. TA cells differentiate into committed cells when they become a certain distance away from a stem cell epicentre.

4. Corneocyte Cells (corn) – When a cell in any of the three above states dies it becomes a corneocyte cell. Corneocyte cells do not divide, migrate or differentiate and are generally found in the top layer of the epidermis.

Agent functions are used to simulate the biological processes and encode such behaviour as cell-cell and cell substrate adhesion, division, migration and differentiation. More specifically, Table 9 indicated the specific agent functions and their role within the model.

**Table 9 – Agent functions used within the Keratinocyte colony model.**
**Blue indicates standard agent functionality, red indicates functions used for force resolution.**

| Function Name | Description |
|---|---|
| output_location | Outputs the location of the cell. |
| cycle | Simulates a cell cycle where stem and TA cells divide after a predetermined period of time. |
| differentiate | Simulates the differentiation process where cells change from one state to another. |
| death_signal | Determines if a cell should become a Corneocyte cell. |
| migrate | Simulates cell migration (or movement of cells) |
| force_resolution_output | Outputs the location of a cell after the normal cell simulation process (blue). |
| resolve_forces | Resolves forces between cells to ensure there is no overlap. |

### 6.2.2 Parallel Force Resolution

In the case of force resolution, standard agent conditions do not provide enough flexibility to ensure agents are able to reach a stable state. Unstable states are the consequence of repulsive and attractive (bond) forces, resulting in overlapping cells. This occurs due to the discrete time nature of agent functions which determine force strengths during phenomenon, such as migration and cell division. A single force resolution step requires a minimum of two agent functions. The first of these is

required to output a positional message, the second to process neighbours' positions and update the agent's position. With a cellular model it is highly unlikely that a single resolution step (output and update) will result in a stable state. To avoid this multiple force resolution, steps have previously been used. Careful review has however suggested that to ensure a stable condition has been met, a large amount of resolution steps are required (typically over 200). Multiple force resolution functions also introduced a large amount of code repetition, as each resolution step requires a separate agent function.

A more suitable technique is to apply force resolution recursively until the population has reached some stable condition. In the past non parallel physical solvers have been used for this purpose. These operate by reading all agent data output by a simulation step, recursively applying a force resolution algorithm and then writing the resolved agent states back to a file, which is used for the next simulation step. The use of non parallel simulation codes becomes an instant performance bottleneck, making models unsuitable for parallel execution. In order to avoid any bottlenecks, it is essential that recursive behaviour can be encoded directly within agent functions allowing the entire simulation to remain on the GPU. To achieve this, the constraint that a single simulation step represents a single fixed length of time is simply removed. Technically, this implies that each simulation step may follow either a regular path through each agent function (Figure 41 blue) or, if the population is unresolved, perform only a force resolution step (Figure 41 red). In order for this to be possible a global function condition is required. Rather than filtering agents into separate states and paths through the simulation, a global condition ensures that all agents follow the same path, providing every agent meets the condition (Figure 41). In the case of inter cellular force resolution this global condition is applied to the first agent function (`output_location`) and checks to ensure agents have reached equilibrium, by moving less than some small amount. If all agents meet this condition it suggests that the physical forces between them have reached a stable physical state, with a minimal probability of overlaps. Using this technique has the added benefit of avoiding unnecessary resolution steps, which occurs when a large fixed number is instead used. As there is a possibility that a physical model may reach an oscillating physical state, the global conditions maximum iterations value is set to 300. This places an upper bound on the number of times the global equilibrium test can be

evaluated as false. If the global condition is not evaluated as true after this many iterations then the condition result is overwritten as true.

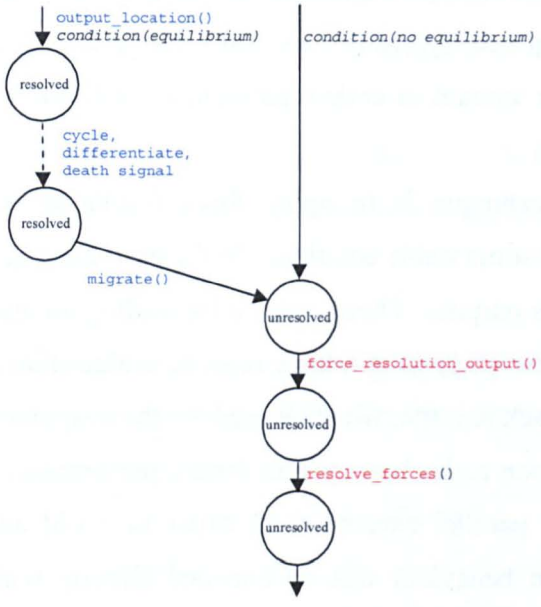



**Figure 41 - Non linear simulation in the Keratinocyte model showing a separate force resolution path**

### 6.2.3 Simulation Performance

Figure 42 demonstrates the relative speedup of the Keratinocyte model achieved using the two message communication techniques. Each simulation consists of an initial configuration state containing randomly distributed stem cells at a constant density. The speedup is calculated by considering the relative speed increase of the FLAME GPU iteration time in comparison with the original FLAMEs CPU iteration time. As FLAME message processing on the CPU uses only an $O(n^2)$ algorithm, the result of the brute force algorithm gives the most direct comparison. The exponential speedup of the spatially partitioned message communication is not surprising and would be better suited to comparison with a grid based implementation. Unfortunately, as the original FLAME framework is unable to perform force resolution, no such data exists. Likewise, the measurements in this experiment are performance orientated and use only a single resolution step to give an indication of processing time. Even with this

simplification, the final simulation run of 131,072 agents took almost 8 hours to complete on the CPU. With brute force messaging on the GPU the simulation time is reduced to just less than two minutes, whilst the spatially partitioned alternative took little over a second.

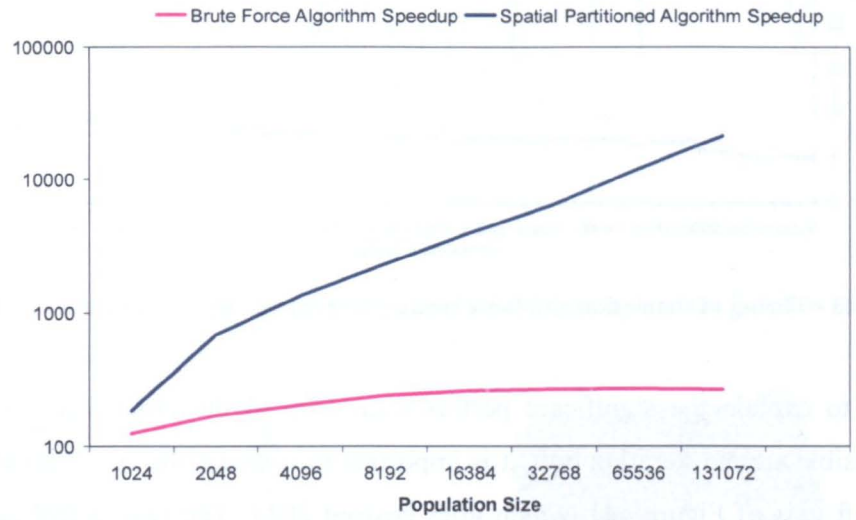



**Figure 42 - Relative performance of the Keratinocyte model (logarithmic scale).**

In order to evaluate a more realistic scenario than randomly distributed agents, the performance over an entire simulation has been measured using an initial configuration representing a scratch wound (300μm wide). Force resolution was done by evaluating agents' movement to ensure they had moved less than 0.25 μm with a maximum of 200 resolution steps. Figure 43 shows the performance of this simulation, which took roughly 1500 iterations (not including force resolution iterations) to reach a stable state (shown in Figure 45). The timing of the force resolution step is shown separately from the timing of regular agent behaviour and is measured in centiseconds ($10^{-2}$) for clarity. Whilst it is possible to visualise only the linear time steps at 2-3 FPS, inclusion of the force resolution steps ensures simulation remains interactive at over 60 FPS throughout. The erratic performance of force resolution is explained by the random movement of agents and the varying resolution steps required reaching a stable state. The slight trend towards increased performance of force resolution throughout the simulation is attributed to the reduced number of cell divisions. Fewer new agents require less force resolution steps, as fewer agents within the densely packed population need to move in order to accommodate them.

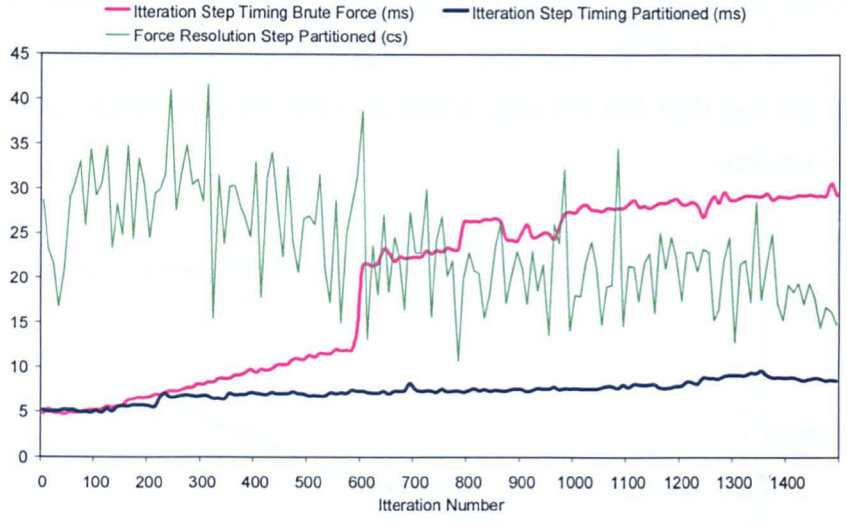



**Figure 43 - Timing of simulation and force resolution steps during scratch would simulation**

In order to explain the significant performance drop of the brute force simulation timing visible around iteration 600, it is important to consider the agent count (shown on the left axis of Figure 44), which goes beyond 2048. The reason this number is significant is that at this agent count the number of blocks (32) is equally split amongst the multiprocessors, which restricted by register use, are able to hold 2 blocks each. For agent counts above 2048 and below the next optimal agent population size of 4096 there are an uneven number of blocks to distribute per multiprocessors. The result of this uneven number is that once the first 2048 agent have been processed the left over thread blocks must be scheduled to multiprocessors, leaving many of them idle. In Figure 44 potential idle blocks per iteration (right axis) represents the number of potential block spaces available on all 16 multi processors as a result of the odd block number. This effect is not visible in the previous results, as population sizes are increased by factors which provide equal mapping of blocks to multiprocessors. The spatially partitioned communication pattern does not suffer in the same way.

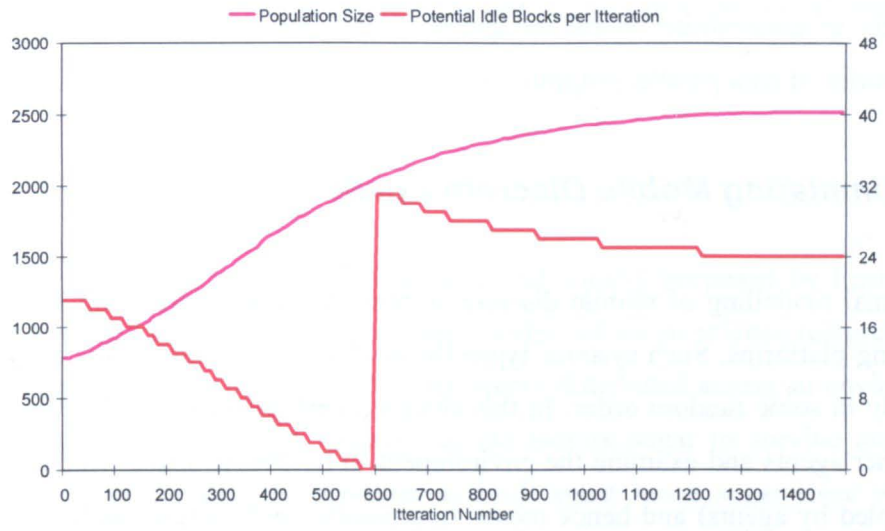



**Figure 44 -Population size and potential idle blocks with respect to the iteration number**

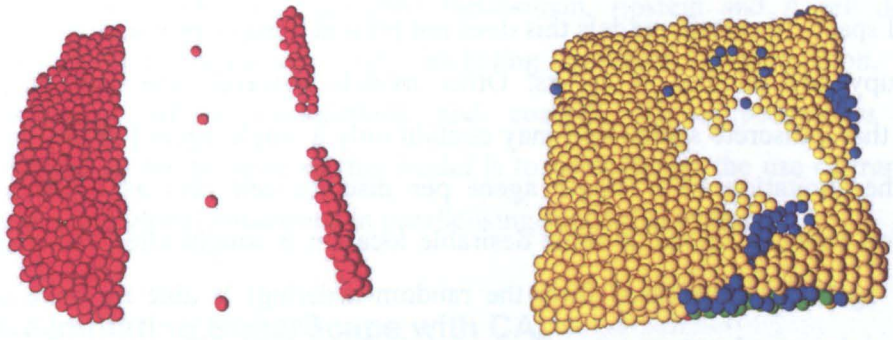


**Figure 45 - Keratinocyte scratch wound model at iteration 0 and 1500 rendered as spheres. Red spheres represent stem cells, green represent TA cells, blue represent committed cells and yellow represents corneocyte cells.**

### 6.2.4 Discussion

Cellular level modelling has been shown to achieve massive performance benefits through simulation with FLAME GPU. Such performance cannot be ignored and makes a significant leap forward with respect to the fast development and visual validation of such models. Likewise, the use of the GPU has allowed real time visualisation to be coupled with simulation steps, which offers the potential for real time interaction (or steering) during the simulation process. Aside from performance related advantages, FLAME GPU has clearly demonstrated a step forward in the use of formal specification techniques for ABM on the GPU. The ability for biologists to specify models which can be automatically mapped to GPU hardware allows

modellers to concentrate on model specification, without an understanding of the complexities of data parallel programming.

## 6.3 Simulating Mobile Discrete Agents

Traditional modelling of mobile discrete systems has been done using highly serial modelling platforms. Such systems typically work by processing agents sequentially, generally in some random order. In this situation each agent is free to communicate with other agents and examine the environment to determine space which is free (or unoccupied by agents) and hence move. In a parallel architecture, such as the GPU, this behaviour is less straightforward. If each agent makes an informed decision to move to some space there is a potential risk that multiple agents will move to the same grid space. In certain models this does not present a major problem, as grid cells may occupy any number of agents. Other models however, are built upon the principle that a discrete spatial cell may contain only a single agent [EA96]. In such models the limitation of a single agent per discrete cell acts as a method of competition between agents. If some desirable location is sought after, the strongest agent (or agent first processed from the random ordering) is able to move to the position gaining an advantage over other agents in the system.

In section 2.3.7 it was shown that previous work has already implemented discrete mobile systems, by using an explicit collision map to resolve collisions as a result of movement [DLR07]. This section however presents a parallel implementation of a mobile discrete system which is based on cellular automaton. Overall, the technique presented is advantageous for the following reasons;

1. The technique used to evaluate movement and resolve collisions can be applied more generally to any transactional event between agents;
2. The technique is agent based, with emphasis on individuals rather than global collision evaluation. This allows it to be integrated into an agent modelling environment rather than as a one off model; and
3. Collisions are guaranteed to succeed in the minimal number of resolution steps. This is not the case with existing work [DLR07] which only converges towards success.

The complete model specification and code for the example presented within this section can be found in Appendix C.

### 6.3.1 The Sugarscape Model

The Sugarscape model is a model of an artificial society proposed by Epstein and Axtell [EA96] and forms the basis for many models of social science research. In its simplest form it consists of a population of agents distributed across an environment with a renewable resource (i.e. sugar). Agents require sugar to survive and move sequentially to empty cells to consume it. Each agent has a sugar store which is incremented by accumulating sugar from the environment. During each simulation step an agent is required to use up part of its sugar store to survive at a rate determined by its randomly assigned metabolism. Epstein and Axtell describe a number of more advanced rules including pollution, reproduction, seasonal environments, cultural connections and combat. These behaviours are not implemented as the purpose of this model is to demonstrate the use of transactional techniques and global conditions in parallelising sequential systems.

### 6.3.2 Simulating SugarScape with CA

Sugarscape agents are mapped to FLAME GPU by using a discrete agent specification and discrete message communication. As this implies the simulation is based on CA, both agents and the environment must be embodied within a single agent type referred to as a cell. To distinguish between cells which contain active agents and those that are empty, a state variable is used. Movement is only achievable through the use of messages, which can be used to send agents between cells. When the decision to move an agent is made, it is output to a message with its desired target location (where it will be recreated) and destroyed from its original cell. Using this technique does not however excuse the requirement to resolve collisions, it instead stops collisions occurring through the simple manipulation of a positional agent variable. In order to avoid collisions during movement, a cell therefore uses the following sequence of events which are defined as individual agent functions to ensure global synchronisation between each step.

1. Cells containing active agents read in environment messages to determine the best place to move to. Once a target location is identified, they output a request to move there within a message containing the targets location identifier and the agents information.
2. Unoccupied cells read all request messages to determine if any neighbouring agents would like to move to the location. If an agent is found which would like to move to the cell then it is given a random priority and saved to the location (this is equivalent to determining a random order in a sequential environment). If multiple agents request to move to the same cell then the cell uses the agent's priority to determine which agent should move. After all requests have been considered the cell outputs a confirmation response, which includes the agent's originating cellular location identifier.
3. Cells containing agents which previously requested to move check the confirmation response messages. If a confirmation is found the agent knows it has been relocated and changes cell to an unoccupied state. If no confirmation is found then the agent does not move.

Whilst the above steps ensure that collisions do not occur, they do not guarantee that all agents wanting to relocate will actually do so. To overcome this problem, it is essential that stages 1 to 3 are repeated until every agent has moved. To guarantee that all movements are evaluated could potentially require the process to be repeated for each possible location that the agent could move into (with a simple vision radius of 1 this results in 8 in total). To reduce the number of movement iterations the same technique is applied as that which was used for intercellular force resolution. A global function condition is used to evaluate the state of all agents. If during stage 3 of the above technique, an agent is unable to move, its state is changed to indicate that it is unresolved. A global function condition on the first agent function ensures that the function is only processed if there are no agents within the unresolved state. If the global function is evaluated as true then the first function, is evaluated. This function then performs environment grow back, removal of sugar from the environment, feeding (according to the metabolic rate) and finally the moving of all agent cells into unresolved state. If the global condition is not met, then the first function is not evaluated. In this case, all cells will immediately perform the second agent function

which begins a movement evaluation step. During this simulation step only cells containing agents which were previously unresolved will be able to send movement requests (Figure 46).

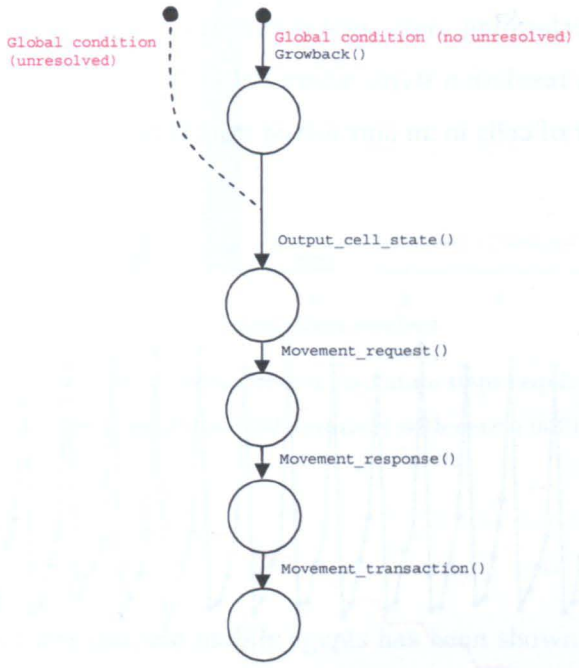



**Figure 46 – Function order of the Sugarscape model demonstrating a global function condition used to bypass the first agent function if the population contains any unresolved agents.**

### 6.3.3 Simulation Performance

Simulation performance of the Sugarscape model has been evaluated by considering the simulation time, of a single simulation step, at various population sizes. The results are shown in Table 10. For a grid size of $1{,}024^2$, with a total of over a million cells, the simulation can be run at over 50 simulation steps per second.

**Table 10 - Performance of the Sugarscape model at various grid/population sizes**

| Grid Size | Simulation Time | Updates/sec |
|---|---|---|
| 16x16 | 0.577 | 1733 |
| 64x64 | 0.68 | 1462 |
| 256x256 | 1.78 | 562 |
| 1024x1024 | 18.43 | 54 |

Figure 47 shows the performance of the model running over a period of 100 iterations with over a million cells. During this time the performance fluctuates between simulation steps in a fairly repeatable trend. The explanation for this is that a normal simulation step (including the first agent function) takes longer to process than a simulation step performing only movement resolution. This is observable even between movement resolution steps, where the following steps are required to do less work as the number of cells in an unresolved state is reduced.

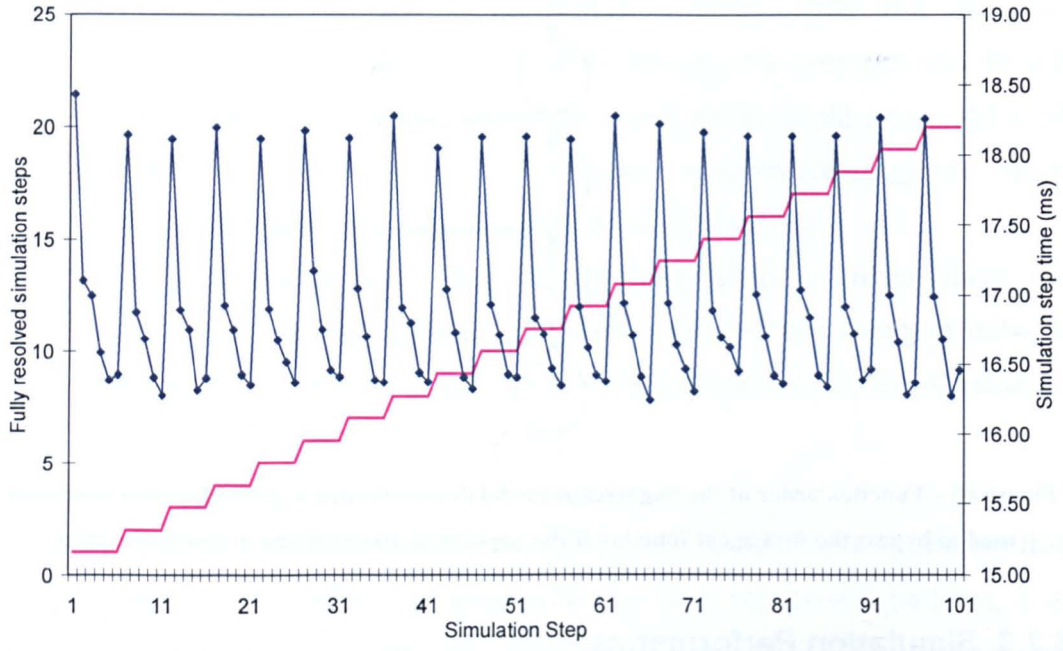



**Figure 47 – Simulation step performance (blue) and number of fully resolved simulation steps (pink) over a 100 iteration simulation.**

Figure 48 considers the number of movement resolution steps which are required over a period of 500 normal simulation steps. On average, 4 resolution steps are required to resolve the population size of over a million agents. By the fourth resolution step it is highly unusual that more than 2 or 3 agents will be unresolved. In smaller population sizes the number of movement resolution steps is reduced. A population size of 4,096 requires on average only 3 resolution steps to evaluate all agent movements. This is due to the reduced probability of collisions in smaller population sizes.

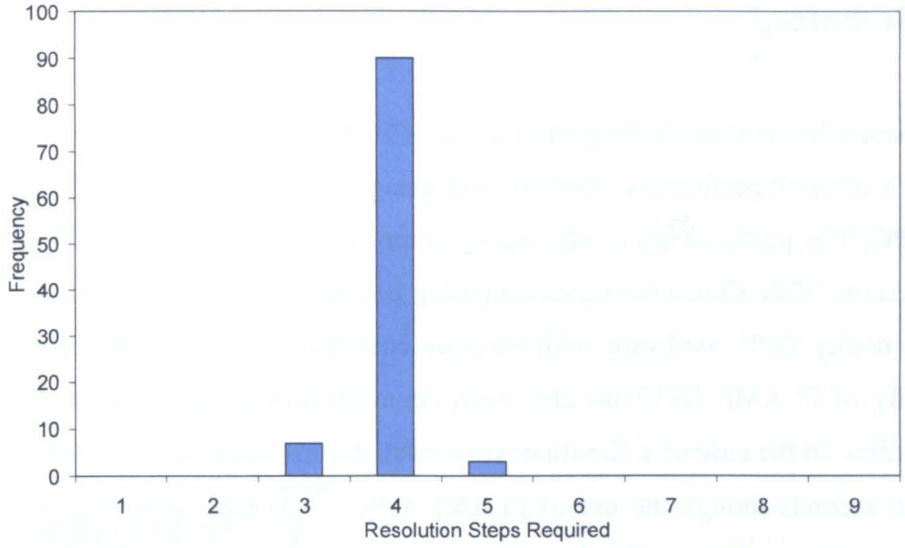



**Figure 48 – Frequency of the number of movement resolution steps required per fully resolved simulation step measured over 500 iterations with over a million cells.**

### 6.3.4 Discussion

The use of CA to simulate discrete mobile agents has been shown to be effective and suitable for integration within an agent based framework. The example presented is not only suitable for any other mobile discrete agent model requiring movement, but is also suitable for evaluating any transactional events between agents. In contrast, with existing work the implementation of the Sugarscape model roughly shows the same massive performance benefits over CPU based alternative as other GPU implementations. D'Souza [DLR07] reported a performance of 74 iterations per second for a grid size of $1024^2$. This is more efficient than the implementation described here however, this can be attributed to the fact that his implementation is a highly optimised implication of a specific model, which does not consider integration within a more formal framework. There is little doubt that the requirement to communicate via messages within FLAME GPU causes a significant slowdown over more direct memory access methods. Whilst detrimental to performance, in this case the use of messages is however essential within a flexible framework and acts as a safeguard to ensure modellers are unable to directly access or modify agent variables.

## 6.4 Summary

This chapter has evaluated the performance of FLAME GPU which has been shown to offer a massive performance benefit over using the traditional FLAME library on a single PC. The performance is also easily compatible with that of FLAME on HPC architectures. This allows for supercomputing performance of ABM through the use of commodity GPU hardware with obvious cost incentives. The performance and flexibility of FLAME GPU has also been demonstrated through two very different case studies. In the case of a Keratinocyte model, performance has been reduced from hours to seconds though the use of FLAME GPU's efficient messaging and use of global function conditions. This has allowed real time visualisation of a model, which was previously inconceivable. In the case of simulating mobile discrete agents with CA, discrete message communication and global function conditions have been shown to allow parallelisation of transactional events suitable for resolving agent movement. The success of these models acts as validation that FLAME GPU is suitable for high performance simulation of a wide range of ABM.

# Chapter 7

# Conclusion

This thesis presented a novel agent-based framework addressing the performance limitations of previous ABM toolkits by providing a flexible approach to modelling agents on the GPU. Traditional GPGPU techniques were first used to map the behaviour of simple swarm systems to the GPU within the newly created ABGPU library. This demonstrated the performance potential, which was evaluated through the implementation of both a flocking and pedestrian dynamics model. The FLAME framework was then used as inspiration to create FLAME GPU, a new framework using the X-Machine notation and scripting syntax of FLAME to perform simulation on the GPU. A flexible and extendible agent specification technique was proposed, based on XMML Schema, and a new robust templating mechanism for producing simulation code was also described. The flexible Schema technique was used to define GPUXMML, which added the additional necessary information to an XMML model to allow the translation of models to CUDA code. The necessary algorithms for ABM, within the data parallel constraints of FLAME GPU, were presented and the performance of the system was evaluated through careful benchmarking against the original FLAME framework. Functionality to achieve non linear simulation steps were integrated and exploited within two case studies. The first of these demonstrated the acceleration of a Keratinocyte cell colony model which incorporated parallel force

resolution between cells. The second of these case studies demonstrated the use of CA to simulate the behaviour of mobile discrete systems within a parallel environment. Finally, ABGPU and FLAME GPU demonstrated real time visualisation of both massive swarm systems and the Keratinocyte colony model respectively, at scales and speeds which were previously unachievable.

## 7.1 Limitations and Future Work

The techniques described within this thesis have been demonstrated and tested for both performance and flexibility. Despite this, it is important to consider the limitations of this work and highlight potential areas of future work.

The major limitation of ABGPU, FLAME GPU and ABM on the GPU in general, is that the GPUs resources are significantly more limited than those of a general PC architecture. The amount of physical GPU memory, for example, restricts the scale of models to a finite limit, which in turn may constrain either the population sizes or the size of agent memory. Likewise, the use of limited shared memory and multiprocessor registers places limitations on both the size of messages and the complexity of agent behaviour. The limitations of physical memory (which depend greatly on the hardware used, with a maximum of 4 GB being supported in Tesla cards) can be addressed in future work, by considering the use of a multi GPU approach to modelling. This could use the original FLAME communication techniques to distribute messages between GPU devices (either distributed between hosts, or within the same machine). Limitations on register usage could be alleviated through the splitting of agent functions into multiple functions, either manually or automatically. An automatic method would likely require the agent behaviour to take a more abstract form (such as XML), which could be mapped to code rather than directly scripting the behaviour. Addressing the hardware limitations of shared memory size is somewhat more difficult. The fixed 16Kb size is unlikely to increase drastically in future hardware (mainly due to cost considerations), and limits messages to at most 125, 32 bit variables[15].

In order to gain the maximum performance benefits of the GPU, it is important to ensure the GPU is fully utilised. The results presented have confirmed that, although

[15] The value of 125 is determined by dividing the size of shared memory (16Kb) by a single 4byte variable 32 times which is the smallest recommended thread block size.

small scale models have shown performance increases, the GPU is suited to systems where there are a large number of agents. For this reason, agent systems with a large number of very different agents (in differing states, which are processed separately), or agent types with very low agent counts are unlikely to gain the same performance benefits as systems containing large numbers of similar agents. In future work it may be possible to isolate these cases and perform simulation simultaneously on the CPU. This will allow the GPU to be more available for parts of the model requiring large amounts of agent processing.

The decision to sacrifice portability for flexibility and performance drove the implementation of FLAME GPU towards the CUDA architecture. This restricts models to recent NVIDIA hardware, alienating a large number of GPU owners. Since the development of the work within this thesis the OpenCL specification has been introduced, which allows a more portable solution for data parallel programming. OpenCL is heavily influenced by the CUDA architecture, with the majority of differences being purely down to syntax rather than the underlying concepts which remain the same. This suggests that a future port of FLAME GPUs CUDA templates to OpenCL can be achieved with relative simplicity.

Finally there are a number of minor improvements that could be made to the ABM techniques presented in this thesis. The first of these is the implementation of FLAME GPUs dependency specification and automatic layering of functions. This could be achieved in the future using standard XSLT functionality. The use of space filling curves would also increase the performance of limited range messaging, as would a method that could exploit shared memory rather than relying on the texture cache. The performance achievements gained allow the possibility for a more advanced representation of cells within the Keratinocyte model. Likewise, the flexibility of the work presented will allow pedestrian models to contain significantly advanced intelligence, allowing reactive agents with improved navigational and reasoning skills.

## 7.2 Last Words

The work within this thesis has presented a flexible technique for ABM on the GPU with massive performance and cost benefits over alterative architectures. The ability

to abstract the process of mapping models to the GPU allows non-graphics specialists to harness the GPUs performance potential through simple specification syntax. The use of the GPU for simulation has also allowed large scale models to be simulated in real time. This is beneficial for visual model validation, and has implications towards reducing the development time of the modelling process itself.

# Appendix A.

# FLAME GPU XMML Schemas

## *A.1 XMML Base Schema*

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="XMML"
targetNamespace="http://www.dcs.shef.ac.uk/~paul/XMML"
elementFormDefault="qualified"
xmlns="http://www.dcs.shef.ac.uk/~paul/XMML"
xmlns:mstns="http://www.dcs.shef.ac.uk/~paul/XMML"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="xmodel_type" abstract="true">
  <xs:sequence>
   <xs:element name="name" type="xs:string" maxOccurs="1"
               minOccurs="1" nillable="false" />
   <xs:element name="version" type="xs:string" maxOccurs="unbounded"
               minOccurs="0" />
   <xs:element name="description" type="xs:string" maxOccurs="1"
               minOccurs="0" />
   <xs:element minOccurs="1" maxOccurs="1" ref="environment">
   </xs:element>
   <xs:element minOccurs="1" maxOccurs="1" ref="xagents">
   </xs:element>
   <xs:element minOccurs="1" maxOccurs="1" ref="messages">
   </xs:element>
   <xs:element minOccurs="1" maxOccurs="1" ref="layers">
   </xs:element>
  </xs:sequence>
 </xs:complexType>
```

```
<xs:simpleType name="type_type">
 <xs:restriction base="xs:string">
  <xs:enumeration value="int" />
  <xs:enumeration value="float" />
  <xs:enumeration value="double" />
 </xs:restriction>
</xs:simpleType>
<xs:complexType name="variable_type" abstract="true">
 <xs:sequence>
  <xs:element name="type" type="xs:string" maxOccurs="1"
              minOccurs="1" nillable="false" />
  <xs:element name="name" type="xs:string" maxOccurs="1"
              minOccurs="1" nillable="false" />
  <xs:element name="description" type="xs:string" minOccurs="0"
              maxOccurs="1" />
  <xs:element name="arrayLength" type="xs:int" minOccurs="0"
              maxOccurs="1" />
  <xs:element name="defaultValue" type="xs:double" minOccurs="0"
              maxOccurs="1" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="environment_type" abstract="true">
 <xs:sequence>
  <xs:element ref="constants" minOccurs="0" maxOccurs="1" />
  <xs:element ref="functionFiles" minOccurs="1" maxOccurs="1" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="constants_type" abstract="true">
 <xs:sequence>
  <xs:element ref="variable" minOccurs="1" maxOccurs="unbounded" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="functionFiles_type" abstract="true">
 <xs:sequence>
  <xs:element name="file" type="xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:element name="xagents" abstract="false">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="xagent" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="messages" abstract="false">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="message" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="layers" abstract="false">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="layer" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:complexType name="xagent_type" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="xs:string" maxOccurs="1"
```

```
                minOccurs="1" />
  <xs:element name="description" type="xs:string" minOccurs="0"
                maxOccurs="1" />
  <xs:element minOccurs="1" maxOccurs="1" ref="memory">
  </xs:element>
  <xs:element maxOccurs="1" minOccurs="1" ref="functions">
  </xs:element>
  <xs:element minOccurs="1" maxOccurs="1" ref="states">
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:complexType name="function_type" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="xs:string" maxOccurs="1"
                minOccurs="1" />
  <xs:element name="description" type="xs:string" minOccurs="0"
                maxOccurs="1" />
  <xs:element name="currentState" type="xs:string" minOccurs="1"
                maxOccurs="1" />
  <xs:element name="nextState" type="xs:string" minOccurs="1"
                maxOccurs="1" />
  <xs:element minOccurs="0" maxOccurs="1" ref="inputs">
  </xs:element>
  <xs:element minOccurs="0" maxOccurs="1" ref="outputs">
  </xs:element>
  <xs:element minOccurs="0" maxOccurs="1" ref="xagentOutputs">
  </xs:element>
  <xs:element ref="condition" minOccurs="0" maxOccurs="1" />
 </xs:sequence>
</xs:complexType>
<xs:element name="functions">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="function" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="states">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="state" minOccurs="0" maxOccurs="unbounded" />
   <xs:element name="initialState" type="xs:string" maxOccurs="1"
                minOccurs="1" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="outputs">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="output" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="inputs">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="input" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="xagentOutputs">
```

```
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="xagentOutput" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:complexType name="input_type" abstract="true">
 <xs:sequence>
  <xs:element name="messageName" type="xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="output_type" abstract="true">
 <xs:sequence>
  <xs:element name="messageName" type="xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="xagentOutput_type" abstract="true">
 <xs:sequence>
  <xs:element name="xagentName" type="xs:string" />
  <xs:element name="state" type="xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="message_type" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="xs:string" maxOccurs="1"
              minOccurs="1" />
  <xs:element name="description" type="xs:string" minOccurs="0"
              maxOccurs="1" />
  <xs:element minOccurs="1" maxOccurs="1" ref="variables">
  </xs:element>
 </xs:sequence>
</xs:complexType>
<xs:element name="variables">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="variable" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="layer">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="layerFunction" minOccurs="1"
               maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="message" type="message_type" abstract="true">
</xs:element>
<xs:element name="xagentOutput" type="xagentOutput_type"
            abstract="true">
</xs:element>
<xs:element name="output" type="output_type" abstract="true">
</xs:element>
<xs:element name="input" type="input_type" abstract="true">
</xs:element>
<xs:complexType name="state_type" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="xs:string" minOccurs="1"
              maxOccurs="1" />
  <xs:element name="description" type="xs:string" minOccurs="0"
```

```
            maxOccurs="1" />
 </xs:sequence>
</xs:complexType>
<xs:element name="state" type="state_type" abstract="true">
</xs:element>
<xs:element name="function" type="function_type" abstract="true">
</xs:element>
<xs:element name="xagent" type="xagent_type" abstract="true">
</xs:element>
<xs:element name="variable" type="variable_type" abstract="true">
</xs:element>
<xs:element name="environment" type="environment_type"
            abstract="true">
</xs:element>
<xs:element name="constants" type="constants_type" abstract="true">
</xs:element>
<xs:element name="functionFiles" type="functionFiles_type"
            abstract="true">
</xs:element>
<xs:element name="memory">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="variable" minOccurs="1" maxOccurs="unbounded" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="xmodel" type="xmodel_type" abstract="true">
</xs:element>
<xs:complexType name="layer_function_type" abstract="true">
 <xs:sequence>
  <xs:element name="name" type="xs:string" />
 </xs:sequence>
</xs:complexType>
<xs:element name="layerFunction" type="layer_function_type"
            abstract="true" />
<xs:complexType name="condition_type">
 <xs:sequence>
  <xs:element name="lhs">
   <xs:complexType>
    <xs:sequence>
     <xs:choice>
      <xs:element ref="condition" />
      <xs:element name="value" type="xs:string">
      </xs:element>
      <xs:element name="agentVariable" type="xs:string" />
     </xs:choice>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="operator" type="xs:string" />
  <xs:element name="rhs">
   <xs:complexType>
    <xs:sequence>
     <xs:choice>
      <xs:element ref="condition" />
      <xs:element name="value" type="xs:string" />
      <xs:element name="agentVariable" type="xs:string" />
     </xs:choice>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
```

```
  </xs:sequence>
 </xs:complexType>
 <xs:element name="condition" type="condition_type">
 </xs:element>
</xs:schema>
```

## A.2 GPUXMML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="GPUXMML"
           targetNamespace="http://www.dcs.shef.ac.uk/~paul/GPUXMML"
           elementFormDefault="qualified"
           xmlns="http://www.dcs.shef.ac.uk/~paul/GPUXMML"
           xmlns:mstns="http://www.dcs.shef.ac.uk/~paul/GPUXMML"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xmml="http://www.dcs.shef.ac.uk/~paul/XMML">
 <xs:import namespace="http://www.dcs.shef.ac.uk/~paul/XMML">
 </xs:import>
 <xs:complexType name="xmodel_type" abstract="false">
  <xs:complexContent>
   <xs:extension base="xmml:xmodel_type">
    <xs:sequence>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="variable_type">
  <xs:complexContent>
   <xs:extension base="xmml:variable_type">
    <xs:sequence>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="environment_type">
  <xs:complexContent>
   <xs:extension base="xmml:environment_type">
    <xs:sequence>
     <xs:element ref="initFunctions" maxOccurs="1" minOccurs="0" />
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="constants_type">
  <xs:complexContent>
   <xs:extension base="xmml:constants_type">
    <xs:sequence>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="functionFiles_type">
  <xs:complexContent>
   <xs:extension base="xmml:functionFiles_type">
    <xs:sequence>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
```

```
<xs:complexType name="xagent_type">
 <xs:complexContent>
  <xs:extension base="xmml:xagent_type">
   <xs:sequence>
    <xs:element name="type" type="xagent_type_options" />
    <xs:element name="bufferSize" type="xs:int" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="function_type">
 <xs:complexContent>
  <xs:extension base="xmml:function_type">
   <xs:sequence>
    <xs:element name="reallocate" type="xs:boolean" />
    <xs:element name="RNG" type="xs:boolean" minOccurs="0"
                maxOccurs="1" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="input_type">
 <xs:complexContent>
  <xs:extension base="xmml:input_type">
   <xs:sequence>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="output_type">
 <xs:complexContent>
  <xs:extension base="xmml:output_type">
   <xs:sequence>
    <xs:element name="type" type="output_type_option" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="xagentOutput_type">
 <xs:complexContent>
  <xs:extension base="xmml:xagentOutput_type">
   <xs:sequence>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:complexType name="message_type">
 <xs:complexContent>
  <xs:extension base="xmml:message_type">
   <xs:sequence>
    <xs:element ref="partitioningNone" />
    <xs:element name="bufferSize" type="xs:int" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:element substitutionGroup="xmml:message" name="message"
            type="message_type">
</xs:element>
<xs:element substitutionGroup="xmml:xagentOutput"
            name="xagentOutput" type="xagentOutput_type">
```

```
</xs:element>
<xs:element substitutionGroup="xmml:output" name="output"
            type="output_type">
</xs:element>
<xs:element substitutionGroup="xmml:input" name="input"
            type="input_type">
</xs:element>
<xs:complexType name="state_type">
 <xs:complexContent>
  <xs:extension base="xmml:state_type">
   <xs:sequence>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:element substitutionGroup="xmml:state" name="state"
            type="state_type">
</xs:element>
<xs:element substitutionGroup="xmml:function" name="function"
            type="function_type">
</xs:element>
<xs:element substitutionGroup="xmml:xagent" name="xagent"
            type="xagent_type">
</xs:element>
<xs:element substitutionGroup="xmml:variable" name="variable"
            type="variable_type">
</xs:element>
<xs:element substitutionGroup="xmml:environment" name="environment"
            type="environment_type">
</xs:element>
<xs:element substitutionGroup="xmml:constants" name="constants"
            type="constants_type">
</xs:element>
<xs:element substitutionGroup="xmml:functionFiles"
            name="functionFiles" type="functionFiles_type">
</xs:element>
<xs:element name="xmodel" type="xmodel_type">
 <xs:key name="xagent_func_name_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                          mstns:function" />
  <xs:field xpath="xmml:name" />
 </xs:key>
 <xs:keyref name="layer_functions" refer="xagent_func_name_key">
  <xs:selector xpath=".//xmml:layers/xmml:layer/
                          mstns:layerFunction" />
  <xs:field xpath="xmml:name" />
 </xs:keyref>
 <xs:key name="xagent_state_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:states/
                          mstns:state" />
  <xs:field xpath="xmml:name" />
 </xs:key>
 <xs:keyref name="xagent_function_currentState"
            refer="xagent_state_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                          mstns:function" />
  <xs:field xpath="xmml:currentState" />
 </xs:keyref>
 <xs:keyref name="initial_state" refer="xagent_state_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:states" />
  <xs:field xpath="xmml:initialState" />
```

```
 </xs:keyref>
 <xs:keyref name="xagentOutput_state" refer="xagent_state_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                        mstns:function/xmml:xagentOutputs/
                        mstns:xagentOutput" />
  <xs:field xpath="xmml:state" />
 </xs:keyref>
 <xs:keyref name="xagent_function_nextState"
            refer="xagent_state_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                        mstns:function" />
  <xs:field xpath="xmml:nextState" />
 </xs:keyref>
 <xs:key name="message_name_key">
  <xs:selector xpath=".//xmml:messages/mstns:message" />
  <xs:field xpath="xmml:name" />
 </xs:key>
 <xs:keyref name="xagent_func_input" refer="message_name_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                        mstns:function/xmml:inputs/mstns:input" />
  <xs:field xpath="xmml:messageName" />
 </xs:keyref>
 <xs:keyref name="xagent_func_output" refer="message_name_key">
  <xs:selector xpath=".//xmml:xagents/mstns:xagent/xmml:functions/
                        mstns:function/xmml:outputs/
                        mstns:output" />
  <xs:field xpath="xmml:messageName" />
 </xs:keyref>
</xs:element>
<xs:simpleType name="output_type_option">
 <xs:restriction base="xs:string">
  <xs:enumeration value="single_message" />
  <xs:enumeration value="optional_message" />
 </xs:restriction>
</xs:simpleType>
<xs:simpleType name="xagent_type_options">
 <xs:restriction base="xs:string">
  <xs:enumeration value="continuous" />
  <xs:enumeration value="discrete" />
 </xs:restriction>
</xs:simpleType>
<xs:complexType name="partitioning_type">
 <xs:sequence>
 </xs:sequence>
</xs:complexType>
<xs:element name="partitioningNone" type="partitioning_type" />
<xs:complexType name="partitioning_discrete_type">
 <xs:complexContent>
  <xs:extension base="partitioning_type">
   <xs:sequence>
    <xs:element name="radius" type="xs:int" />
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:element substitutionGroup="partitioningNone"
            name="partitioningDiscrete"
            type="partitioning_discrete_type">
</xs:element>
<xs:complexType name="partitioning_spatial_type">
 <xs:complexContent>
```

```
    <xs:extension base="partitioning_type">
     <xs:sequence>
      <xs:element name="radius" type="xs:decimal" />
      <xs:element name="xmin" type="xs:decimal" />
      <xs:element name="xmax" type="xs:decimal" />
      <xs:element name="ymin" type="xs:decimal" />
      <xs:element name="ymax" type="xs:decimal" />
      <xs:element name="zmin" type="xs:decimal" />
      <xs:element name="zmax" type="xs:decimal" />
     </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
  <xs:element substitutionGroup="partitioningNone"
              name="partitioningSpatial"
              type="partitioning_spatial_type" />
  <xs:element name="initFunction">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="name" type="xs:string" />
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:complexType name="initFunctions_type">
   <xs:sequence>
    <xs:element ref="initFunction" minOccurs="1"
                maxOccurs="unbounded" />
   </xs:sequence>
  </xs:complexType>
  <xs:element name="initFunctions" type="initFunctions_type">
  </xs:element>
  <xs:complexType name="layer_function_type">
   <xs:complexContent>
    <xs:extension base="xmml:layer_function_type">
     <xs:sequence />
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
  <xs:element substitutionGroup="xmml:layerFunction"
              name="layerFunction" type="layer_function_type" />
  <xs:complexType name="globalCondition_type">
   <xs:complexContent>
    <xs:extension base="xmml:condition_type">
     <xs:sequence>
      <xs:element name="maxItterations" type="xs:int" />
      <xs:element name="mustEvaluateTo" type="xs:boolean" />
     </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
  <xs:element substitutionGroup="xmml:condition"
              name="globalCondition" type="globalCondition_type">
  </xs:element>
 </xs:schema>
```

# Appendix B. Keratinocyte Case Study Model

## *B.1 GPUXMML Model Specification*

```
<?xml version="1.0" encoding="utf-8"?>
<gpu:xmodel xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/GPUXMML"
        xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
 <name>Keratinocyte</name>
 <gpu:environment>
  <gpu:constants>
   <gpu:variable>
    <type>float</type>
    <name>calcium_level</name>
   </gpu:variable>
   <gpu:variable>
    <type>int</type>
    <name>CYCLE_LENGTH</name>
    <arrayLength>5</arrayLength>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>SUBSTRATE_FORCE</name>
    <arrayLength>5</arrayLength>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>DOWNWARD_FORCE</name>
    <arrayLength>5</arrayLength>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>FORCE_MATRIX</name>
    <arrayLength>25</arrayLength>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>FORCE_REP</name>
```

```
  </gpu:variable>
  <gpu:variable>
   <type>float</type>
   <name>FORCE_DAMPENER</name>
  </gpu:variable>
  <gpu:variable>
   <type>int</type>
   <name>BASEMENT_MAX_Z</name>
  </gpu:variable>
 </gpu:constants>
 <gpu:functionFiles>
  <file>functions.c</file>
 </gpu:functionFiles>
 <gpu:initFunctions>
  <gpu:initFunction>
   <gpu:name>setConstants</gpu:name>
  </gpu:initFunction>
 </gpu:initFunctions>
</gpu:environment>

<xagents>
 <gpu:xagent>
  <name>keratinocyte</name>
  <memory>
   <gpu:variable>
    <type>int</type>
    <name>id</name>
   </gpu:variable>
   <gpu:variable>
    <type>int</type>
    <name>type</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>x</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>y</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>z</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>force_x</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>force_y</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>force_z</name>
   </gpu:variable>
   <gpu:variable>
    <type>int</type>
    <name>num_xy_bonds</name>
   </gpu:variable>
   <gpu:variable>
```

```
      <type>int</type>
      <name>num_z_bonds</name>
     </gpu:variable>
     <gpu:variable>
      <type>int</type>
      <name>num_stem_bonds</name>
     </gpu:variable>
     <gpu:variable>
      <type>int</type>
      <name>cycle</name>
     </gpu:variable>
     <gpu:variable>
      <type>float</type>
      <name>diff_noise_factor</name>
     </gpu:variable>
     <gpu:variable>
      <type>int</type>
      <name>dead_ticks</name>
     </gpu:variable>
     <gpu:variable>
      <type>int</type>
      <name>contact_inhibited_ticks</name>
     </gpu:variable>
     <gpu:variable>
      <type>float</type>
      <name>motility</name>
     </gpu:variable>
     <gpu:variable>
      <type>float</type>
      <name>dir</name>
     </gpu:variable>
     <gpu:variable>
      <type>float</type>
      <name>movement</name>
     </gpu:variable>
    </memory>
    <functions>

     <gpu:function>
      <name>output_location</name>
      <currentState>resolve</currentState>
      <nextState>default</nextState>
      <outputs>
       <gpu:output>
        <messageName>location</messageName>
        <gpu:type>single_message</gpu:type>
       </gpu:output>
      </outputs>

      <!-- If this condition is met by all agents then they will
perform the default agent functions. If not they will resolve forces
and return back to this first function -->
      <gpu:globalCondition>
       <lhs>
        <agentVariable>movement</agentVariable>
       </lhs>
       <operator>&lt;</operator>
       <rhs>
        <value>0.25</value>
       </rhs>
       <gpu:maxItterations>200</gpu:maxItterations>
```

```
    <gpu:mustEvaluateTo>true</gpu:mustEvaluateTo>
   </gpu:globalCondition>

   <gpu:reallocate>false</gpu:reallocate>

  </gpu:function>

  <gpu:function>
   <name>cycle</name>
   <currentState>default</currentState>
   <nextState>default</nextState>
   <xagentOutputs>
    <gpu:xagentOutput>
     <xagentName>keratinocyte</xagentName>
     <state>default</state>
    </gpu:xagentOutput>
   </xagentOutputs>
   <gpu:reallocate>false</gpu:reallocate>
   <gpu:RNG>true</gpu:RNG>
  </gpu:function>

  <gpu:function>
   <name>differentiate</name>
   <currentState>default</currentState>
   <nextState>default</nextState>
   <inputs>
    <gpu:input>
     <messageName>location</messageName>
    </gpu:input>
   </inputs>
   <gpu:reallocate>false</gpu:reallocate>
  </gpu:function>

  <gpu:function>
   <name>death_signal</name>
   <currentState>default</currentState>
   <nextState>default</nextState>
   <inputs>
    <gpu:input>
     <messageName>location</messageName>
    </gpu:input>
   </inputs>
   <gpu:reallocate>false</gpu:reallocate>
   <gpu:RNG>true</gpu:RNG>
  </gpu:function>

  <gpu:function>
   <name>migrate</name>
   <currentState>default</currentState>
   <nextState>resolve</nextState>
   <inputs>
    <gpu:input>
     <messageName>location</messageName>
    </gpu:input>
   </inputs>
   <gpu:reallocate>false</gpu:reallocate>
   <gpu:RNG>true</gpu:RNG>
  </gpu:function>

  <gpu:function>
   <name>force_resolution_output</name>
```

```
    <currentState>resolve</currentState>
    <nextState>resolve</nextState>
    <outputs>
     <gpu:output>
      <messageName>force</messageName>
      <gpu:type>single_message</gpu:type>
     </gpu:output>
    </outputs>
    <gpu:reallocate>false</gpu:reallocate>
   </gpu:function>

   <gpu:function>
    <name>resolve_forces</name>
    <currentState>resolve</currentState>
    <nextState>resolve</nextState>
    <inputs>
     <gpu:input>
      <messageName>force</messageName>
     </gpu:input>
    </inputs>
    <gpu:reallocate>false</gpu:reallocate>
   </gpu:function>

  </functions>

  <states>
   <gpu:state>
    <name>default</name>
    <description>represents a normal cell which is
                 resolved</description>
   </gpu:state>
   <gpu:state>
    <name>resolve</name>
    <description>when in this state the agent needs to be
                 resolved</description>
   </gpu:state>
   <initialState>resolve</initialState>
  </states>

  <gpu:type>continuous</gpu:type>
  <gpu:bufferSize>8192</gpu:bufferSize>

 </gpu:xagent>
</xagents>

<messages>
 <gpu:message>
  <name>location</name>
  <variables>
   <gpu:variable>
    <type>int</type>
    <name>id</name>
   </gpu:variable>
   <gpu:variable>
    <type>int</type>
    <name>type</name>
   </gpu:variable>
   <gpu:variable>
    <type>float</type>
    <name>x</name>
   </gpu:variable>
```

```
    <gpu:variable>
     <type>float</type>
     <name>y</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>z</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>dir</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>motility</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>range</name>
    </gpu:variable>
    <gpu:variable>
     <type>int</type>
     <name>iteration</name>
    </gpu:variable>
   </variables>
   <gpu:partitioningSpatial>
    <gpu:radius>250</gpu:radius>
    <gpu:xmin>0.0</gpu:xmin>
    <gpu:xmax>500</gpu:xmax>
    <gpu:ymin>0.0</gpu:ymin>
    <gpu:ymax>500</gpu:ymax>
    <gpu:zmin>0.0</gpu:zmin>
    <gpu:zmax>500</gpu:zmax>
   </gpu:partitioningSpatial>
   <gpu:bufferSize>8192</gpu:bufferSize>
  </gpu:message>

  <gpu:message>
   <name>force</name>
   <variables>
    <gpu:variable>
     <type>int</type>
     <name>type</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>x</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>y</name>
    </gpu:variable>
    <gpu:variable>
     <type>float</type>
     <name>z</name>
    </gpu:variable>
    <gpu:variable>
     <type>int</type>
     <name>id</name>
    </gpu:variable>
   </variables>
```

```
    <gpu:partitioningSpatial>
     <gpu:radius>20</gpu:radius>
     <gpu:xmin>0.0</gpu:xmin>
     <gpu:xmax>500</gpu:xmax>
     <gpu:ymin>0.0</gpu:ymin>
     <gpu:ymax>500</gpu:ymax>
     <gpu:zmin>0.0</gpu:zmin>
     <gpu:zmax>500</gpu:zmax>
    </gpu:partitioningSpatial>
    <gpu:bufferSize>8192</gpu:bufferSize>
   </gpu:message>
  </messages>

  <layers>
   <layer>
    <gpu:layerFunction>
     <name>output_location</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>cycle</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>differentiate</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>death_signal</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>migrate</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>force_resolution_output</name>
    </gpu:layerFunction>
   </layer>

   <layer>
    <gpu:layerFunction>
     <name>resolve_forces</name>
    </gpu:layerFunction>
   </layer>
  </layers>

</gpu:xmodel>
```

## B.2 Agent Function Simulation Code

```
#ifndef _FUNCTIONS_H_
#define _FUNCTIONS_H_

#include "header.h"


/*********************************************
** Definitions
*********************************************/
struct distance_result
{
 double nearest_distance;
 double nearest_xy;
 double nearest_z;
};

#define FALSE 0
#define TRUE  1

/* general constants*/
#ifndef PI
#define PI      3.142857143
#endif

#define SURFACE_WIDTH    500.0
#define K_WIDTH      20.0

/* keratinocyte cell types*/
#define K_TYPE_STEM        0
#define K_TYPE_TA          1
#define K_TYPE_COMM        2
#define K_TYPE_CORN        3
#define K_TYPE_HACAT       4

/* forces act within this radius of a cell*/
#define FORCE_IRADIUS     10

/* control G0 and cornicyte phases*/
#define MAX_TO_G0_CONTACT_INHIBITED_TICKS  300
#define MAX_DEAD_TICKS                     600

/* bond number constants*/
#define MAX_NUM_LATERAL_BONDS 4


__constant__ float calcium_level;
__constant__ int CYCLE_LENGTH[5];
__constant__ float SUBSTRATE_FORCE[5];
__constant__ float DOWNWARD_FORCE[5];
__constant__ float FORCE_MATRIX[5*5];
__constant__ float FORCE_REP;
__constant__ float FORCE_DAMPENER;
__constant__ int BASEMENT_MAX_Z;


__FLAME_GPU_INIT_FUNC__ void setConstants(){
```

```
    float h_calcium_level = 1.300000;
    int h_CYCLE_LENGTH[5] = {120, 60, 0, 0, 120};
    float h_SUBSTRATE_FORCE[5] = {0.3, 0.1, 0.2, 0.1, 0.3};
    float h_DOWNWARD_FORCE[5] = {0.1, 0.6, 0.3, 0.6, 0.1};
    float h_FORCE_MATRIX[5*5] = {0.06, 0.01, 0.01, 0.01, 0.0,
          0.01, 0.01, 0.01, 0.01, 0.0,
          0.01, 0.01, 0.06, 0.01, 0.0,
          0.01, 0.01, 0.01, 0.08, 0.0,
          0.01, 0.01, 0.01, 0.08, 0.0};
    float h_FORCE_REP      = 0.5;
    float h_FORCE_DAMPENER = 0.4;
    int h_BASEMENT_MAX_Z = 5;

    set_calcium_level(&h_calcium_level);
    set_CYCLE_LENGTH(h_CYCLE_LENGTH);
    set_SUBSTRATE_FORCE(h_SUBSTRATE_FORCE);
    set_DOWNWARD_FORCE(h_DOWNWARD_FORCE);
    set_FORCE_MATRIX(h_FORCE_MATRIX);
    set_FORCE_REP(&h_FORCE_REP);
    set_FORCE_DAMPENER(&h_FORCE_DAMPENER);
    set_BASEMENT_MAX_Z(&h_BASEMENT_MAX_Z);
}


/* tests if cell is deemed to be on the substrate surface */
/* used by resolve forces, differentiate and mitigate*/
__FLAME_GPU_FUNC__ int on_substrate_surface(float z)
{
    return (z < (float) BASEMENT_MAX_Z);
}

/* used by differentiate*/
__FLAME_GPU_FUNC__ float get_ta_to_comm_diff_minor_axis(
                                        float calcium_level)
{
    return K_WIDTH * 1.5;
}

/* used by differetiate */
__FLAME_GPU_FUNC__ float get_ta_to_comm_diff_major_axis(
                                        float calcium_level)
{
    return K_WIDTH * 5;
}

/* used in cycle */
__FLAME_GPU_FUNC__ int get_max_num_bonds(float calcium_level)
{
    return 6;
}

/* used in cycle*/
__FLAME_GPU_FUNC__ int can_stratify(int cell_type,
                                    float calcium_level)
{
    if (cell_type == K_TYPE_HACAT) {
      return FALSE;
    } else {
      return TRUE;
    }
}
```

```
/* used in differentiate */
__FLAME_GPU_FUNC__ float get_max_stem_colony_size(
                                        float calcium_level)
{
    return 20;
}

/* used in cycle */
__FLAME_GPU_FUNC__ float get_new_motility(int cell_type,
                                          float calcium_level)
{
    if (cell_type == K_TYPE_TA) {
      return 0.5;
    }
    else
    {
      return 0;
    }
}

/* checks if a cell can divide */
/* used in cycle*/
__FLAME_GPU_FUNC__ int divide(int type, int cycle)
{
    return (type == K_TYPE_STEM || type == K_TYPE_TA ||
            type == K_TYPE_HACAT) && cycle > CYCLE_LENGTH[type];
}

/* returns a new coordinate based on the old one, but deviated
slightly */
/* used in cycle*/
__FLAME_GPU_FUNC__ float get_new_coord(float old_coord, int pos_only,
                                       RNG_rand48* rand48)
{
    float coord = 0;

    while (coord == 0) {
      coord = rnd(rand48) * K_WIDTH / 10;
    }

    if (!pos_only && coord > 0.5) {
      coord = -coord;
    }
    return old_coord + coord;
}

/* generate a new starting position in the cell's cycle */
/* used in cycle*/
__FLAME_GPU_FUNC__ int start_new_cycle_postion(int type,
                                               RNG_rand48* rand48)
{
    float cycle_fraction = CYCLE_LENGTH[type] / 4;
    float pos = rnd(rand48) * cycle_fraction;

    return (int) round(pos);
}


/* get the radius of an ellipse given its radii and an angle theta */
__FLAME_GPU_FUNC__ float ellipse_radius(float major_radius,
```

```
                                        float minor_radius,
                                        float theta)
{
    float a_squ = major_radius * major_radius;
    float b_squ = minor_radius * minor_radius;
    float sin_theta = sin(theta);
    float sin_theta_squ = sin_theta * sin_theta;
    float cos_theta = cos(theta);
    float cos_theta_squ = cos_theta * cos_theta;
    float r_squ = (a_squ * b_squ) /
                  (a_squ * sin_theta_squ  + b_squ * cos_theta_squ);
    float r = sqrt(r_squ);
    return r;
}

/* is the nearest stem cell in range */
/* used in differentiate*/
__FLAME_GPU_FUNC__ int check_distance(struct distance_result nearest,
                                      float major_radius,
                                      float minor_radius,
                                      float thres)
{
    if (nearest.nearest_distance == -1.0)
    {
      return 1;
    }else
    {
      float theta = tan(nearest.nearest_z/nearest.nearest_xy);
      float er = ellipse_radius(major_radius, minor_radius, theta);
      return (thres * er < nearest.nearest_distance);
    }
}

/* test if on the edge of the colony */
/* used in differentiate */
__FLAME_GPU_FUNC__ int on_colony_edge(int num_bonds)
{
    return num_bonds <= 2;
}

/*********************************************
** Keratinocyte Agent Functions
*********************************************/


//Input :
//Output: location
//Agent Output:
__FLAME_GPU_FUNC__ int output_location(
                  xmachine_memory_keratinocyte* xmemory,
                  xmachine_message_location_list* location_messages)
{
    add_location_message(location_messages,
                         xmemory->id,
                         xmemory->type,
                         xmemory->x,
                         xmemory->y,
                         xmemory->z,
                         xmemory->dir,
                         xmemory->motility,
                         SURFACE_WIDTH,
```

```
                        0);

    return 0;
}

//Input :
//Output:
//Agent Output: keratinocyte
__FLAME_GPU_FUNC__ int cycle(
            xmachine_memory_keratinocyte* xmemory,
            xmachine_memory_keratinocyte_list* keratinocyte_agents,
            RNG_rand48* rand48)
{

    /* find number of contacts/bonds*/
    int contacts = xmemory->num_xy_bonds + xmemory->num_z_bonds;

    /* touching a wall counts as two contacts*/
    if (xmemory->x == 0 || xmemory->x == SURFACE_WIDTH) {
      contacts += 2;
    }
    if (xmemory->y == 0 || xmemory->y == SURFACE_WIDTH) {
      contacts += 2;
    }

    if (contacts <= get_max_num_bonds(calcium_level)) {
      /* cell comes out of G0*/
      xmemory->cycle = xmemory->cycle+1;
      xmemory->contact_inhibited_ticks = 0;
    } else {
      /* cell enters G0*/
      xmemory->contact_inhibited_ticks =
                xmemory->contact_inhibited_ticks+1;
    }

    /* check to see if enough time has elapsed as to whether cell can
    divide*/
    if (divide(xmemory->type, xmemory->cycle)) {
      int new_cycle = start_new_cycle_postion(xmemory->type, rand48);
      float new_x = get_new_coord(xmemory->x, FALSE, rand48);
      float new_y = get_new_coord(xmemory->y, FALSE, rand48);
      float new_z = xmemory->z;
      float new_diff_noise_factor = 0.9 + (rnd(rand48)*0.2);
      float new_dir = rnd(rand48) * 2 * PI;
      float new_motility = (0.5 + (rnd(rand48) * 0.5)) *
                get_new_motility(xmemory->type, calcium_level);

      if (can_stratify(xmemory->type, calcium_level) &&
                xmemory->num_xy_bonds >= MAX_NUM_LATERAL_BONDS)
      {
        new_z = get_new_coord(xmemory->z, TRUE, rand48);
      }
      xmemory->cycle =
                start_new_cycle_postion(xmemory->type, rand48);


      add_keratinocyte_agent(
                      keratinocyte_agents,
                      xmemory->id+1,
                      xmemory->type,
                      new_x,
```

```
                        new_y,
                        new_z,
                        0,                  /* force_x*/
                        0,                  /* force_y*/
                        0,                  /* force_z*/
                        0,                  /* num_xy_bonds*/
                        0,                  /* num_z_bonds*/
                        0,                  /* num_stem_bonds*/
                        new_cycle,
                        new_diff_noise_factor,
                        0,                  /* dead_ticks*/
                        0,                  /*contact_inhibited_ticks*/
                        new_motility,
                        new_dir,
                        SURFACE_WIDTH);

    }

    return 0;
}

//Input : location
//Output:
//Agent Output:
__FLAME_GPU_FUNC__ int differentiate(
                    xmachine_memory_keratinocyte* xmemory,
                    xmachine_message_location_list* location_messages,
                    xmachine_message_location_PBM* partition_matrix)
{
    float x1 = xmemory->x;
    float y1 = xmemory->y;
    float z1 = xmemory->z;

    struct distance_result nearest_stem;

    nearest_stem.nearest_distance = -1.0;
    nearest_stem.nearest_xy = -1.0;
    nearest_stem.nearest_z  = -1.0;

    xmachine_message_location* location_message =
                    get_first_location_message(location_messages,
                                               partition_matrix,
                                               x1, y1, z1);
    int num_stem_neighbours = 0;

    while(location_message){

      if (xmemory->type == K_TYPE_STEM) {
        if (on_colony_edge(xmemory->num_stem_bonds)) {
          float x2 = location_message->x;
          float y2 = location_message->y;
          float z2 = location_message->z;

          if (location_message->type == K_TYPE_STEM) {
            float distance_check = sqrt((x1-x2)*(x1-x2) +
                                        (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2));

            float max_distance = K_WIDTH *
                        (get_max_stem_colony_size(calcium_level) / 2);

            if (distance_check < max_distance) {
```

```
            num_stem_neighbours ++;
          }
        }
      }
    }
    /* If the TA cell is too far from the stem cell centre, it
    differentiates into in a committed cell.*/
    else if (xmemory->type  == K_TYPE_TA) {
      float x2 = location_message->x;
      float y2 = location_message->y;
      float z2 = location_message->z;

      if (location_message->type == K_TYPE_STEM) {
        float distance_check = sqrt((x1-x2)*(x1-x2)+
                                    (y1-y2)*(y1-y2)+
                                    (z1-z2)*(z1-z2));
        if (nearest_stem.nearest_distance == -1.0 ||
              distance_check < nearest_stem.nearest_distance)
        {
          nearest_stem.nearest_distance = distance_check;
          nearest_stem.nearest_xy = sqrt((x1-x2)*(x1-x2) +
                                    (y1-y2)*(y1-y2));
          nearest_stem.nearest_z  = fabs(z1-z2);
        }
      }
    }

    //load next message
    location_message = get_next_location_message(
                                    location_message,
                                    location_messages,
                                    partition_matrix);
}

/* For stem cells, we check if the colony is too big and if they
are on the edge.*/
/* If so, they differentiate into TA cells.*/
if (xmemory->type == K_TYPE_STEM) {
  if (on_colony_edge(xmemory->num_stem_bonds)) {
    if (num_stem_neighbours >
                      get_max_stem_colony_size(calcium_level))
    {
      xmemory->type = K_TYPE_TA;
    }
  }

  /* If the cell stratifies, it also differentiates into a TA
  cell*/
  if (!on_substrate_surface(xmemory->z)) {
    xmemory->type = K_TYPE_TA;
  }
}
else if (xmemory->type  == K_TYPE_TA) {
  int do_diff = check_distance(
              nearest_stem,
              get_ta_to_comm_diff_major_axis(calcium_level),
              get_ta_to_comm_diff_minor_axis(calcium_level),
              xmemory->diff_noise_factor);


  if (do_diff) {
```

```
        xmemory->type = K_TYPE_COMM;

      }
      /* If it has been in G0 for a certain period, it also
      differentiates.*/
      else if (xmemory->contact_inhibited_ticks >=
                                  MAX_TO_G0_CONTACT_INHIBITED_TICKS)
      {
        xmemory->type = K_TYPE_COMM;
      }
    }
    /* after a period as a committed cell, it dies for good -
    differentiation into a corneocyte*/
    else if (xmemory->type == K_TYPE_COMM) {
      xmemory->dead_ticks = xmemory->dead_ticks+1;
      if (xmemory->dead_ticks > MAX_DEAD_TICKS) {
        xmemory->type = K_TYPE_CORN;
      }
    } else if (xmemory->type == K_TYPE_HACAT) {
      if (xmemory->contact_inhibited_ticks >=
                                  MAX_TO_G0_CONTACT_INHIBITED_TICKS)
      {
        xmemory->type = K_TYPE_COMM;
      }
    }

    return 0;
}

//Input : location
//Output:
//Agent Output:
__FLAME_GPU_FUNC__ int death_signal(
                   xmachine_memory_keratinocyte* xmemory,
                   xmachine_message_location_list* location_messages,
                   xmachine_message_location_PBM* partition_matrix,
                   RNG_rand48* rand48)
{
    float x1 = xmemory->x;
    float y1 = xmemory->y;
    float z1 = xmemory->z;

    int num_corn_neighbours = 0;

    xmachine_message_location* location_message =
           get_first_location_message(location_messages,
                                      partition_matrix,
                                      x1, y1, z1);
    while(location_message)
    {
      if (xmemory->type != K_TYPE_CORN) {
        float x2 = location_message->x;
        float y2 = location_message->y;
        float z2 = location_message->z;

        if (location_message->type == K_TYPE_CORN) {
          float distance_check = sqrt((x1-x2)*(x1-x2) +
                                 (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2));
          if (distance_check != 0) {
            if (distance_check - K_WIDTH <= FORCE_IRADIUS) {
              num_corn_neighbours ++;
```

```
            }
          }
        }
      }

      location_message = get_next_location_message(
                                          location_message,
                                          location_messages,
                                          partition_matrix);
    }

    float prob = num_corn_neighbours * num_corn_neighbours * 0.01;

    if (rnd(rand48) < prob) {
      /* jump through another hoop*/
      if (rnd(rand48) < 0.01) {
        xmemory->type = K_TYPE_CORN;
      }
    }

    return 0;
}

//Input : location
//Output:
//Agent Output:
__FLAME_GPU_FUNC__ int migrate(
                   xmachine_memory_keratinocyte* xmemory,
                   xmachine_message_location_list* location_messages,
                   xmachine_message_location_PBM* partition_matrix,
                   RNG_rand48* rand48)
{
    /* these are the 'current' parameters*/
    float x1 = xmemory->x;
    float y1 = xmemory->y;
    float dir1 = xmemory->dir;
    float motility1 = xmemory->motility;

    /* if rnd less than 0.1, then changed direction within +/- 45
    degrees*/
    if (rnd(rand48) < 0.1) {
      dir1 += PI * rnd(rand48)/4.0;
    }
    x1 += motility1 * cos(dir1);
    y1 += motility1 * sin(dir1);


    // check if we're about to bump into a stationary cell
    xmachine_message_location* location_message =
                  get_first_location_message(location_messages,
                                             partition_matrix,
                                             x1, y1, 0.0);
    while(location_message)
    {
      // only TAs and HACATs can move
      if (xmemory->type == K_TYPE_TA ||
          xmemory->type == K_TYPE_HACAT)
      {
        float x2        = location_message->x;
        float y2        = location_message->y;
        float z2        = location_message->z;
```

```
        float motility2 = location_message->motility;

        // check if we're on the base of the dish and other cell is
        // stationary
        if (on_substrate_surface(z2) && motility2 == 0)
        {
          // find distance
          float distance_check = sqrt((x1-x2)*(x1-x2) +
                                 (y1-y2)*(y1-y2));
          if (distance_check != 0 && distance_check < K_WIDTH) {
            dir1 -= PI;
            // reverse direction
            if (dir1 > 2 * PI) {
              dir1 -= 2 * PI;
            }

            x1 = xmemory->x + motility1 * cos(dir1);
            y1 = xmemory->y + motility1 * sin(dir1);
          }
        }
      }

      location_message = get_next_location_message(location_message,
                                                   location_messages,
                                                   partition_matrix);
    }

    /* update memory with new parameters*/
    xmemory->x = x1;
    xmemory->y = y1;
    xmemory->dir = dir1;
    xmemory->motility = motility1;

    /* check we've not gone over the edge of the dish!*/
    /* if so, reverse direction*/
    if (xmemory->x > SURFACE_WIDTH) {
      xmemory->x   = SURFACE_WIDTH - xmemory->motility *rnd(rand48);
      xmemory->dir = PI + PI * (rnd(rand48)-0.5)/4.0;
    }
    if (xmemory->x < 0) {
      xmemory->x   = xmemory->motility *rnd(rand48);
      xmemory->dir = PI * (rnd(rand48)-0.5)/4.0;
    }
    if (xmemory->y > SURFACE_WIDTH) {
      xmemory->y   = SURFACE_WIDTH -xmemory->motility *rnd(rand48);
      xmemory->dir = (3.0 * PI/2.0) + PI * (rnd(rand48)-0.5)/4.0;
    }
    if (xmemory->y < 0) {
      xmemory->y   = xmemory->motility * rnd(rand48);
      xmemory->dir = (PI/2.0) + PI * (rnd(rand48)-0.5)/4.0;
    }

    return 0;
}

//Input :
//Output: force
//Agent Output:
__FLAME_GPU_FUNC__ int force_resolution_output(
                    xmachine_memory_keratinocyte* xmemory,
                    xmachine_message_force_list* force_messages)
```

```
{
    add_force_message( force_messages,
                       xmemory->type,
                       xmemory->x,
                       xmemory->y,
                       xmemory->z,
                       xmemory->id);

    return 0;
}

//Input : force
//Output:
//Agent Output:
__FLAME_GPU_FUNC__ int resolve_forces(
                        xmachine_memory_keratinocyte* xmemory,
                        xmachine_message_force_list* force_messages,
                        xmachine_message_force_PBM* partition_matrix)
{

    float x1 = xmemory->x;
    float y1 = xmemory->y;
    float z1 = xmemory->z;
    int type1 = xmemory->type;

    int num_xy_bonds   = 0;
    int num_z_bonds    = 0;
    int num_stem_bonds = 0;

    xmemory->force_x = 0.0;
    xmemory->force_y = 0.0;
    xmemory->force_z = 0.0;

    xmachine_message_force* force_message = get_first_force_message(
                                            force_messages,
                                            partition_matrix,
                                            x1, y1, z1);

    while (force_message)
    {
      float x2 = force_message->x;
      float y2 = force_message->y;
      float z2 = force_message->z;
      int type2 = force_message->type;
      float distance_check = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) +
                                  (z1-z2)*(z1-z2));

      if (distance_check != 0.0){
        float force;
        float separation_distance = (distance_check - K_WIDTH);
        if (separation_distance <= FORCE_IRADIUS) {
          if (z2 >= z1) {
            if (z2 - z1 > (K_WIDTH/2)) {
              num_z_bonds ++;
            } else {
              num_xy_bonds ++;
            }
          }
          if (force_message->type == K_TYPE_STEM) {
            num_stem_bonds ++;
          }
```

```
        if (separation_distance > 0.0) {
          force = FORCE_MATRIX[type1+ (type2*5)];
        } else {
          force = FORCE_REP;
        }
        if (on_substrate_surface(z1)) {
          force *= DOWNWARD_FORCE[xmemory->type];
        }
        force *= FORCE_DAMPENER;

        xmemory->force_x = (xmemory->force_x +
            force * (separation_distance)*((x2-x1)/distance_check));
        xmemory->force_y = (xmemory->force_y +
            force * (separation_distance)*((y2-y1)/distance_check));
        xmemory->force_z = (xmemory->force_z +
            force * (separation_distance)*((z2-z1)/distance_check));

      }
    }
    force_message = get_next_force_message(force_message,
                                           force_messages,
                                           partition_matrix);
}

/* attraction force to substrate*/
if (z1 <= (K_WIDTH * 1.5)) {
  xmemory->force_z = (xmemory->force_z - SUBSTRATE_FORCE[type1]);
}

xmemory->num_xy_bonds = num_xy_bonds;
xmemory->num_z_bonds = num_z_bonds;
xmemory->num_stem_bonds = num_stem_bonds;

x1 += xmemory->force_x;
y1 += xmemory->force_y;
z1 += xmemory->force_z;

if (x1 < 0) {
  x1 = 0;
}

if (y1 < 0) {
  y1 = 0;
}

if (z1 < 0) {
  z1 = 0;
}

if (x1 > SURFACE_WIDTH) {
  x1 = SURFACE_WIDTH;
}

if (y1 > SURFACE_WIDTH) {
  y1 = SURFACE_WIDTH;
}

xmemory->movement = sqrt((x1-xmemory->x)*(x1-xmemory->x) +
                         (y1-xmemory->y)*(y1-xmemory->y) +
                         (z1-xmemory->z)*(z1-xmemory->z));
```

```
    xmemory->x = x1;
    xmemory->y = y1;
    xmemory->z = z1;

    return 0;
}

#endif // #ifndef _FUNCTIONS_H_
```

# Appendix C. Mobile Discrete Case Study Model

## *C.1 GPUXMML Model Specification*

```
<?xml version="1.0" encoding="utf-8"?>
<gpu:xmodel xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/GPUXMML"
            xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
  <name>Mobile Discrete Agents </name>

  <gpu:environment>
    <gpu:functionFiles>
      <file>functions.c</file>
    </gpu:functionFiles>
  </gpu:environment>

  <xagents>
    <gpu:xagent>
      <name>agent</name>
      <memory>
        <gpu:variable>
          <type>int</type>
          <name>location_id</name>
        </gpu:variable>
        <gpu:variable>
          <type>int</type>
          <name>agent_id</name>
        </gpu:variable>
        <gpu:variable>
          <type>int</type>
          <name>state</name>
        </gpu:variable>
        <!-- agent specific variables-->
        <gpu:variable>
          <type>int</type>
          <name>sugar_level</name>
        </gpu:variable>
        <gpu:variable>
```

```
      <type>int</type>
      <name>metabolism</name>
    </gpu:variable>
    <!-- environment specicic var -->
    <gpu:variable>
      <type>int</type>
      <name>env_sugar_level</name>
    </gpu:variable>
  </memory>
  <functions>

    <gpu:function>
      <name>metabolise_and_growback</name>
      <description>decreases the sugar level and increases and
                   agents sugar store</description>
      <currentState>default</currentState>
      <nextState>default</nextState>
      <gpu:globalCondition>
        <lhs><agentVariable>state</agentVariable></lhs>
        <operator>!=</operator>
        <rhs><value>AGENT_STATE_MOVEMENT_UNRESOLVED</value></rhs>
        <gpu:maxItterations>9</gpu:maxItterations>
        <gpu:mustEvaluateTo>true</gpu:mustEvaluateTo>
      </gpu:globalCondition>
      <gpu:reallocate>false</gpu:reallocate>
      <gpu:RNG>false</gpu:RNG>
    </gpu:function>

    <gpu:function>
      <name>output_cell_state</name>
      <description>outputs the state of the cell</description>
      <currentState>default</currentState>
      <nextState>default</nextState>
      <outputs>
        <gpu:output>
          <messageName>cell_state</messageName>
          <gpu:type>single_message</gpu:type>
        </gpu:output>
      </outputs>
      <gpu:reallocate>false</gpu:reallocate>
      <gpu:RNG>false</gpu:RNG>
    </gpu:function>

    <gpu:function>
      <name>movement_request</name>
      <description>an agents requests to move to a new
                   location</description>
      <currentState>default</currentState>
      <nextState>default</nextState>
      <inputs>
        <gpu:input>
          <messageName>cell_state</messageName>
        </gpu:input>
      </inputs>
      <outputs>
        <gpu:output>
          <messageName>movement_request</messageName>
          <gpu:type>single_message</gpu:type>
        </gpu:output>
      </outputs>
      <gpu:reallocate>false</gpu:reallocate>
```

```
        <gpu:RNG>false</gpu:RNG>
      </gpu:function>

      <gpu:function>
        <name>movement_response</name>
        <description>an agents responds to requests flagging itself
                     for a transaction</description>
        <currentState>default</currentState>
        <nextState>default</nextState>
        <inputs>
          <gpu:input>
            <messageName>movement_request</messageName>
          </gpu:input>
        </inputs>
        <outputs>
          <gpu:output>
            <messageName>movement_response</messageName>
            <gpu:type>single_message</gpu:type>
          </gpu:output>
        </outputs>
        <gpu:reallocate>false</gpu:reallocate>
        <gpu:RNG>true</gpu:RNG>
      </gpu:function>

      <gpu:function>
        <name>movement_transaction</name>
        <description>completes the transaction by removing the
                     agent from its old position</description>
        <currentState>default</currentState>
        <nextState>default</nextState>
        <inputs>
          <gpu:input>
            <messageName>movement_response</messageName>
          </gpu:input>
        </inputs>
        <gpu:reallocate>false</gpu:reallocate>
        <gpu:RNG>false</gpu:RNG>
      </gpu:function>
    </functions>

    <states>
      <gpu:state><name>default</name></gpu:state>
      <initialState>default</initialState>
    </states>

    <gpu:type>discrete</gpu:type>
    <gpu:bufferSize>4096</gpu:bufferSize>
  </gpu:xagent>
</xagents>

<messages>
  <gpu:message>
    <name>cell_state</name>
    <variables>
      <gpu:variable>
        <type>int</type>
        <name>location_id</name>
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>state</name>
```

```
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>env_sugar_level</name>
      </gpu:variable>
    </variables>
    <gpu:partitioningDiscrete>
      <gpu:radius>1</gpu:radius>
    </gpu:partitioningDiscrete>
    <gpu:bufferSize>4096</gpu:bufferSize>
  </gpu:message>

  <gpu:message>
    <name>movement_request</name>
    <variables>
      <gpu:variable>
        <type>int</type>
        <name>agent_id</name>
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>location_id</name>
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>sugar_level</name>
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>metabolism</name>
      </gpu:variable>
    </variables>
    <gpu:partitioningDiscrete>
      <gpu:radius>1</gpu:radius>
    </gpu:partitioningDiscrete>
    <gpu:bufferSize>4096</gpu:bufferSize>
  </gpu:message>

  <gpu:message>
    <name>movement_response</name>
    <variables>
      <gpu:variable>
        <type>int</type>
        <name>location_id</name>
      </gpu:variable>
      <gpu:variable>
        <type>int</type>
        <name>agent_id</name>
      </gpu:variable>
    </variables>
    <gpu:partitioningDiscrete>
      <gpu:radius>1</gpu:radius>
    </gpu:partitioningDiscrete>
    <gpu:bufferSize>4096</gpu:bufferSize>
  </gpu:message>
</messages>

<layers>
  <layer>
    <gpu:layerFunction>
      <name>metabolise_and_growback</name>
```

```
      </gpu:layerFunction>
    </layer>
    <layer>
      <gpu:layerFunction>
        <name>output_cell_state</name>
      </gpu:layerFunction>
    </layer>
    <layer>
      <gpu:layerFunction>
        <name>movement_request</name>
      </gpu:layerFunction>
    </layer>
    <layer>
      <gpu:layerFunction>
        <name>movement_response</name>
      </gpu:layerFunction>
    </layer>
    <layer>
      <gpu:layerFunction>
        <name>movement_transaction</name>
      </gpu:layerFunction>
    </layer>
  </layers>
</gpu:xmodel>
```

## C.2 Agent Function Simulation Code

```
#ifndef _FUNCTIONS_H_
#define _FUNCTIONS_H_

#include "header.h"

//Agent state variables
#define AGENT_STATE_UNOCCUPIED 0
#define AGENT_STATE_OCCUPIED 1
#define AGENT_STATE_MOVEMENT_REQUESTED 2
#define AGENT_STATE_MOVEMENT_UNRESOLVED 3


//Growback variables
#define SUGAR_GROWBACK_RATE 1
#define SUGAR_MAX_CAPACITY 4


__FLAME_GPU_FUNC__ int metabolise_and_growback(
                              xmachine_memory_agent* agent){

    //metabolise if occupied
    if (agent->state == AGENT_STATE_OCCUPIED)
    {
      //store sugar
      agent->sugar_level += agent->env_sugar_level;
      agent->env_sugar_level = 0;

      //metabolise
      agent->sugar_level -= agent->metabolism;

      //check if agent dies
      if (agent->sugar_level == 0)
```

```
    {
      agent->state = AGENT_STATE_UNOCCUPIED;
      agent->agent_id = -1;
      agent->sugar_level = 0;
      agent->metabolism = 0;
    }
  }

  //growback if unoccupied
  if (agent->state == AGENT_STATE_UNOCCUPIED)
  {
    if (agent->env_sugar_level < SUGAR_MAX_CAPACITY)
    {
      agent->env_sugar_level += SUGAR_GROWBACK_RATE;
    }
  }

  //set all active agents to unresolved as they may now want to move
  if (agent->state == AGENT_STATE_OCCUPIED)
  {
    agent->state = AGENT_STATE_MOVEMENT_UNRESOLVED;
  }

  return 0;
}


__FLAME_GPU_FUNC__ int output_cell_state(
              xmachine_memory_agent* agent,
              xmachine_message_cell_state_list* cell_state_messages){


    add_cell_state_message<DISCRETE_2D>(cell_state_messages,
                                        agent->location_id,
                                        agent->state,
                                        agent->env_sugar_level);

    return 0;
}


__FLAME_GPU_FUNC__ int movement_request(
                           xmachine_memory_agent* agent,
                           xmachine_message_cell_state_list*
                                        cell_state_messages,
                           xmachine_message_movement_request_list*
                                        movement_request_messages)
{

    int best_sugar_level = -1;
    int best_location_id = -1;

    //find the best location to move to
    xmachine_message_cell_state* current_message =
                     get_first_cell_state_message<DISCRETE_2D>(
                                          cell_state_messages,
                                          get_agent_x(),
                                          get_agent_y());
    while (current_message)
    {
      //if occupied then look for empty cells
```

```
        if (agent->state == AGENT_STATE_MOVEMENT_UNRESOLVED)
        {
          //if location is unoccupied then check for empty locations
          if (current_message->state == AGENT_STATE_UNOCCUPIED)
          {
            //if the sugar level at current location is better than
            //currently stored then update
            if (current_message->env_sugar_level > best_sugar_level)
            {
              best_sugar_level = current_message->env_sugar_level;
              best_location_id = current_message->location_id;
            }
          }
        }

        current_message = get_next_cell_state_message<DISCRETE_2D>(
                                              current_message,
                                              cell_state_messages);
    }

    //if the agent has found a better location to move to then update
    //its state
    if ((agent->state == AGENT_STATE_MOVEMENT_UNRESOLVED))
    {
      //if there is a better location to move to then state indicates
      //a movement request
      if (best_location_id > 0)
      {
        agent->state = AGENT_STATE_MOVEMENT_REQUESTED;
      }
      else
      {
        agent->state = AGENT_STATE_OCCUPIED;
      }
    }

    //add a movement request
    add_movement_request_message<DISCRETE_2D>(
                                  movement_request_messages,
                                  agent->agent_id,
                                  best_location_id,
                                  agent->sugar_level,
                                  agent->metabolism);


    return 0;
}


__FLAME_GPU_FUNC__ int movement_response(
                      xmachine_memory_agent* agent,
                      xmachine_message_movement_request_list*
                                  movement_request_messages,
                      xmachine_message_movement_response_list*
                                  movement_response_messages,
 RNG_rand48* rand){

    int best_request_id = -1;
    int best_request_priority = -1;
    int best_request_sugar_level = -1;
    int best_request_metabolism = -1;
```

```
    xmachine_message_movement_request* current_message =
                get_first_movement_request_message<DISCRETE_2D>(
                                        movement_request_messages,
                                        get_agent_x(),
                                        get_agent_y());
    while (current_message)
    {
    //if the location is unoccupied then check for agents requesting
    //to move here
      if (agent->state == AGENT_STATE_UNOCCUPIED)
      {
        //check if request is to move to this location
        if (current_message->location_id == agent->location_id)
        {
          //check the priority and maintain the best ranked agent
          int message_priority = 0; //rand48(rand);
          if (message_priority > best_request_priority)
          {
            best_request_id = current_message->agent_id;
            best_request_priority = message_priority;
          }
        }
      }

      current_message =
                  get_next_movement_request_message<DISCRETE_2D>(
                                        current_message,
                                        movement_request_messages);
    }
    //if the location is unoccupied and an agent wants to move here
    //then do so and send a response
    if ((agent->state == AGENT_STATE_UNOCCUPIED)&&
        (best_request_id > 0))
    {
      agent->state = AGENT_STATE_OCCUPIED;
      //move the agent to here
      agent->agent_id = best_request_id;
      agent->sugar_level = best_request_sugar_level;
      agent->metabolism = best_request_metabolism;
    }

    //add a movement response
    add_movement_response_message<DISCRETE_2D>(
                                        movement_response_messages,
                                        agent->location_id,
                                        best_request_id);

    return 0;
}

__FLAME_GPU_FUNC__ int movement_transaction(
                xmachine_memory_agent* agent,
                xmachine_message_movement_response_list*
                                    movement_response_messages){

    xmachine_message_movement_response* current_message =
                get_first_movement_response_message<DISCRETE_2D>(
                                        movement_response_messages,
                                        get_agent_x(),
                                        get_agent_y());
```

```
    while (current_message)
    {
      //if location contains an agent wanting to move then look for
      //responses allowing relocation
      if (agent->state == AGENT_STATE_MOVEMENT_REQUESTED)
      {
        //if the movement response request came from this location
        if (current_message->agent_id == agent->agent_id)
        {
          //remove the agent and reset agent specific variables as
          //it has now moved
          agent->state = AGENT_STATE_UNOCCUPIED;
          agent->agent_id = -1;
          agent->sugar_level = 0;
          agent->metabolism = 0;
        }
      }

      current_message =
                get_next_movement_response_message<DISCRETE_2D>(
                                        current_message,
                                        movement_response_messages);
    }

    //if request has not been responded to then agent is unresolved
    if (agent->state == AGENT_STATE_MOVEMENT_REQUESTED)
    {
      agent->state = AGENT_STATE_MOVEMENT_UNRESOLVED;
    }

    return 0;
}

#endif // #ifndef _FUNCTIONS_H_
```

# References

[ACKM08] Salem F. Adra, Simon Coakley, Mariam Kiran, and Phil McMinn. An agent-based software platform for modelling systems biology. Epitheliome Project Technical report, University of Sheffield, 2008.

[ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227(10):5342–5359, 2008.

[ASM80] Jean-Raymond Abrial, Stephen Schuman, and Bertrand Meyer. *A Specification Language, in On the Construction of Programs.* Cambridge University Press, 1980.

[ATI09] ATI. Ati stream computing overview. Technical report, Advanced Micro Devices, 2009.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[BBPP98] Francois Bousquet, Innocent Bakam, Hubert Proton, and Christophe Le Page. Cormas: Common-pool resources and multi-agent systems. In *IEA/AIE '98: Proceedings of the 11th International Conference on Industrial and Engineering Applications of Artificial In telligence and Expert Systems*, pages 826–837, London, UK, 1998. Springer-Verlag.

[BCG+99] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan. Communicating stream x-machines systems are no more than x-machines. *j-jucs*, 5(9):494–507, 1999. |http://www.jucs.org/jucs_5_9/communicating_stream_x_machines|.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[BMOO01] Bernhard Bauer, Jörg P. Müller, James Odell, and James Odell. Agent uml: A formalism for specifying multiagent interaction. In *in: Ciancarini, P.; Wooldridge, M. [Eds.], Agent-Oriented Software Engineering*, pages 91–103. Springer, 2001.

[Bro09] Robert G. Brown. *Engineering a Beowulf-Style Compute Cluster*. Duke University Physics Department, Durham, NC 27708-0305 http://www.phy.duke.edu/ rgb/Beowulf/beowulf_book.php, 2009.

[Buc06] Ian Buck. *Stream Computing on Graphics Card Hardware*. PhD thesis, Stanford University, 2006.

[BZ08] R.G. Belleman and J. Bédorfand S.F. Portegies Zwart. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, 13(2):103–112, 2008.

[CES06] R. De Chiara, U. Erra, and V. Scarano. An architecture for distributed behavioural models with gpus. *Proceedings of the 4th Eurographics Italian Chapter (EGITA 2006) Eurographics press*, pages 197–203, 2006.

[CGW07] LS Chin, C Greenough, and DJ Worth. Optimising communication routines in parallel x-agents. Technical report, Software Engineering Group Note SEG-N-004, 2007.

[CM04] N. Courty and S. Musse. Fastcrowd: Real-time simulation and interaction with large crowds based on graphics hardware. *Eurographics/ACM Siggraph Symp. on Computer Animation, SCA 04, Grenoble, France*, 2004.

[CM05] N. Courty and S. R. Musse. Simulation of large crowds in emergency situations including gaseous phenomena. In *CGI '05: Proceedings of the Computer Graphics International 2005*, pages 206–212, Washington, DC, USA, 2005. IEEE Computer Society.

[Coa07] Simon Coakley. *A Formal Software Architecture for Agent Based Modelling in Biology*. PhD thesis, Department of Computer Science, University of Sheffield, 2007.

[CSH06] Simon Coakley, Rod Smallwood, and Mike Holcombe. Using x-machines as a formal basis for describing agents in agent-based modelling. In *Proceedings of 2006 Spring Simulation Multiconference*, pages 33–40, April 2006.

[CUDPP] http://gpgpu.org/developer/cudpp.

[Den91] D.C. Dennett. *Consciousness Explained.* Boston: Back Bay Books. ISBN 0316180661, 1991.

[dL00] Mark d'Inverno and Michael Luck. Formal agent development: Framework to system. In *In Formal Approaches to Agent-Based Systems: First International Workshop, FAABS 2000*, pages 133–147. Springer-Verlag, 2000.

[DLR07] R. M. D'Souza, M. Lysenko, and K. Rahmani. Sugarscape on steroids: simulating over a million agents at interactive rates. In *Proceedings of Agent2007*, 2007.

[Dor01] Jim Doran. Agent-based modelling of ecosystems for sustainable resource managment. pages 383–403, 2001.

[DSR98] Van Dyke, Robert Savit, and Rick L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. pages 10–25, 1998.

[EA96] Joshua M. Epstein and Robert L. Axtell. *Growing Artificial Societies: Social Science from the Bottom Up.* The MIT Press, November 1996.

[EDCST04] Ugo Erra, Rosario De Chiara, Vittorio Scarano, and Maurizio Tatafiore. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. November 2004.

[Eil74] Samuel Eilenberg. *Automata, Languages, and Machines.* Academic Press, Inc., Orlando, FL, USA, 1974.

[FK03] R. Fernando and M. Kilgard, editors. *The Cg Tutorial: The Definative Guide to Programmable Real-Time Graphics.* Addison Wesley, ISBN 0-031-19496-9, 2003.

[FQK08] Zhe Fan, Feng Qiu, and Arie Kaufman. Zippygpu: Programming toolkit for general-purpose computation on gpu clusters. *Supercomputing 2006 Workshop on Gneral Purpose GPU Computing*, 2008.

[Gar70] M Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game life. *Scientific American 223*, pages 120–123, 1970.

[gde] http://www.gremedy.com.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GpuTerasort: high performance graphics co-processor sorting for large database management. pages 325–336, 2006.

[Ghe05] Marian Gheorghe. Molecular x-machines. Technical report, Department of Computer Science, Univeristy of Sheffield, 2005.

[gls08] OpenGL shading language v 1.20. Technical report, Khronos Group, 2008.

[GM77] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Mon. Not. Roy. Astron. Soc.*, 181:375–389, November 1977.

[GMST99] E. A. Gaffney, P. K. Maini, J. A. Sherratt, and S. Tuft. The mathematical modelling of cell kinetics in corneal epithelial wound healing. *Journal of Theoretical Biology*, 197:15–40, 1999.

[Gre05] Simon G. Green. GPU-accelerated iterated function systems. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 15, New York, NY, USA, 2005. ACM.

[Gre07] Simon Green. Cuda particles. Technical report, NVIDIA SDK White Paper, 2007.

[GRH+05] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[GZ06] A. Greβ and G. Zachmann. Gpu-abisort: optimal parallel sorting on stream architectures. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10 pp.–, April 2006.

[HCSL02] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[HI07] Tsuyoshi Hamada and Toshiaki Iitaka. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units. 2007.

[HM97] Dirk Helbing and Peter Molnar. *Self-Organization Phenomena in Pedestrian Crowds*, page 569577. Gordon and Breach, 1997.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol88] M. Holcombe. X-machines as a basis for dynamic system specification. *Softw. Eng. J.*, 3(2):69–76, 1988.

[How07] Lee Howes. Loading structured data efficiently using cuda. Technical report, NVIDIA, 2007.

[HRRT05] M. G. Hinchey, C. A. Rouff, J. L. Rash, and W. F. Truszkowski. Requirements of an integrated formal method for intelligent swarms. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 125–133, New York, NY, USA, 2005. ACM.

[HSH09] Liang Hu, Pedro V. Sander, and Hugues Hoppe. Parallel view-dependent refinement of progressive meshes. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 169–176, New York, NY, USA, 2009. ACM.

[IGCG99] Carlos Argel Iglesias, Mercedes Garijo, and José Centeno-González. A survey of agent-oriented methodologies. In *ATAL '98: Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, pages 317–330, London, UK, 1999. Springer-Verlag.

[JC06] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 39, New York, NY, USA, 2006. ACM.

[JEB+06] MD Johnston, CM Edwards, WF Bodmer, PK Maini, and SJ Chapman. Mathematical modeling of cell population dynamics in the colonic crypt and in colorectal cancer. *Proc Natl Acad Sci U S A.*, 104(10):4008–4013, 2006.

[KC05] Andreas Kolb and Nicolas Cuntz:. Dynamic particle coupling for gpu-based fluid simulation. *Proceedings of 18th Symp. on Simulation Technique*, page 722727, 2005.

[KHEG03] P. Kefalas, M. Holcombe, G. Eleftherakis, and M. Gheorghe. A formal method for the development of agent based systems. *In Intelligent Agent Software Engineering, V.Plekhanova (eds), Idea Group Publishing Co*, pages 68–98, 2003.

[KK01] E. Kapeti and P. Kefalas. A design language and tool for x-machine specification. in: Fotadis, d.i., nikolopoulos, s.d. (eds.), advances in informatics, world scientific, athens. pp. 134-145. *In: Fotadis, D.I., Nikolopoulos, S.D. (Eds.), Advances in Informatics, World Scientific, Athens.*, pages 134–145, 2001.

[KKKW05] J. Kruger, P. Kipfer, P. Kondratieva, and R. Westermann. A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualisation and Computer Graphics*, 11, 2005.

[KLRS04] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. pages 123–131, 2004.

[Kru06] Jens Kruger. A gpu framework for interactive simulation and rendering of fluid effects. PhD Thesis, Technische Universität München, 2006.

[KSG05] Petros Kefalas, Ioanna Stamatopoulou, and Marian Gheorghe. *Multi-Agent Systems and Applications IV*, chapter A Formal Modelling Framework for Developing Multi-agent Systems with Dynamic Structure and Behaviour, pages 122–131. Springer Berlin / Heidelberg, 2005.

[KSW04] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[KW06] Peter Kipfer and Rüdiger Westermann. Realistic and interactive simulation of rivers. In *GI '06: Proceedings of Graphics Interface 2006*, pages 41–48, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[Lat04] Lutz Latta. Building a million particle system. *Game Developers Conference (GDC)*, 2004.

[LCRP+05] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[LD08] M. Lysenko and R.M. D'Souza. A framework for megascale agent-based model simulations on the gpu. *Journal of Artificial Societies and Social Simulation. (JASSS)*, 11(4):10, 2008.

[LG07] Scott Le Grand. Broad-phase collision detection with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 32. Addison Wesley Professional, August 2007.

[LL93] Gilbert Thomas Laycock and Gilbert Thomas Laycock. The theory and practice of specification based software testing. Technical report, Department of Computer Science, University of Sheffield, 1993.

[LWK03] Wei Li, Xiaoming Wei, and Arie Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8):444–456, 2003.

[MBLA96] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Working Paper 96-06-042, Santa Fe Institute, Santa Fe., 1996.

[MCG03] Matthias Muller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[Mer07] Bruce Merry. Bugle user manual. Technical report, OpenGL SDK, 2007.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[MT98] J. Makimo and Makoto Taiji. *Scientific Simulations with Special Purpose Computers: The Grade Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[NHCV05] M.J. North, T.R. Howe, N.T. Collier, and J.R. Vos. Repast simphony runtime system. In *in Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms, ANL/DIS-06-1, co- sponsored by Argonne National Laboratory and The University of Chicago*, 2005.

[NHP04] Lars Nyland, Mark Harris, and Jan Prins. "The rapid evaluation of potential fields using programmable graphics hardware.". *Poster*

*presentation at GP2, the ACM Workshop on General Purpose Computing on Graphics Hardware.*, 2004.

[NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with cuda. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[NVI07] NVIDIA. NVIDIA cuda compute unified device architecture programming guide. Technical report, NVIDIA, 2007.

[Ode02] James J. Odell. Objects and agents compared. *Journal of Object Technology*, 1:41–53, 2002.

[OJL+07] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[OPB00] J. Odell, H. Parunak, and B. Bauer. Extending uml for agents, 2000.

[ope09] The opencl 1.0 specification. Technical report, Khronos Group, 2009.

[PDC+05] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 258, New York, NY, USA, 2005. ACM.

[PKK07] Bernd Page, Nicolas Knaaka, and Sven Kruse. A discrete event simulation framework for agent-based modelling of logistic systems. In *Informatik 2007 Informatik trifft Logistik, Proc. 37. Jahrestagung der Gesellschaftr Informatik. Bremen*, pages 397–404, September 2007.

[PM04] Craig Peeper and Jason L. Mitchell. Introduction to the directxÂ® 9 high level shading language. *Shader X2 - Shader Tips and Tricks*, 2004.

[QMHz03] Michael J. Quinn, Ronald A. Metoyer, and Katharine Hunter-zaworski. Parallel implementation of the social forces model. 2003.

[Ree83] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[Rey87] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics, (SIGGRAPH '87 Conference Proceedings)*, 21(4):25–34, 1987.

[Rey99] Craig Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, 1999.

[Rey06] Craig Reynolds. Big fast crowds on ps3. *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, 2006.

[RMH+04] Isaac Rudomin, Erik Millán, Benjamin Hernández, Marissa Diaz, and Daniel Rivera. Art applications for crowds. *The Knowledge Engineering Review*, 23(04):399–412, 2004.

[RMH05] Isaac Rudomin, Erik Millán, and Benjamin Hernández. Fragment shaders for agent animation using finite state machines. *Simulation Modelling Practice and Theory*, 13(8):741–751, 2005.

[RTRH05] C. Rouff, W. Truszkowski, J. Rash, and M. Hinchey. A survey of formal methods for intelligent swarms. Technical Report 2005-0156631, NASA Goddard Space Flight Center, Greenbelt, MD 2005., 2005.

[sat04] Serial ata ii, extension to serial ata 1.0, revision 1.2, http://www.sata-io.org/docs/s2ext_1_2_gold.pdf, 2004.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[SHG09] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. 2009.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[SKKW08] Jens Schneider, Polina Kondratieva, Jens Kruger, and Rudiger Westermann. Visualization contest 2005- all you need is...- particles! 2008.

[SMC+07] Tao Sun, Phil McMinn, Simon Coakley, Mike Holcombe, Rod Smallwood, and Sheila MacNeil. An integrated systems biology approach to understanding the rules of keratinocyte colony formation. *Journal of the Royal Society*, 4:1077–1092, 2007.

[SOHTG99] Thorsten Schelhorn, David O'Sullivan, Mordechay Haklay, and Mark Thurstain-Goodwin. Streets: an agent-based pedestrian model. casa

working papers (9). Technical report, Centre for Advanced Spatial Analysis UCL, London, UK, 1999.

[TCP06] Adrien Treuille, Seth Cooper, and Zoran Popovic. Continuum crowds. *ÁCM Trans. Graph.*, 25(3):1160–1168, 2006.

[TJL04] John Tran, Don Jordan, and David Luebke. New challenges for cellular automata simulation on the gpu. *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[TPM+08] Sun T, McMinn P, Holcombe M, Smallwood R, and MacNeil S. Agent based modelling helps in understanding the rules by which fibroblasts support keratinocyte colony formation. *PLoS ONE*, 3(5):e2129, 2008.

[vdV02] Eric van der Vlist. *XML Schema.* O'Reiley, 2002.

[vMAF+07]J. A. van Meel, A. Arnold, D. Frenkel, S. F. Portegies Zwart, and R. G. Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34:259–266, 2007.

[WG06] DJ Worth and C Greenough. Optimising x-agent models in computational biology. Technical report, Software Engineering Group Note SEG-N-001, 2006.

[Wil99] U. Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL., 1999.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[WLMK04] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004.

[WS09] Dawn C. Walker and Jennifer Southgate. The virtual cell–a candidate co-ordinator for 'middle-out' modelling of biological systems. *Brief Bioinform*, pages bbp010+, March 2009.

[WSH+04] D. C. Walker, J. Southgate, G. Hill, M. Holcombe, D. R. Hose, S. M. Wood, Mac, and R. H. Smallwood. The epitheliome: agent-based modelling of the social behaviour of cells. *Bio Systems*, 76(1-3):89–100, October 2004.

[XERCES] http://xerces.apache.org.

[ZBG07] Simon Portegies Zwart, Robert Belleman, and Peter Geldof. High performance direct gravitational n-body simulations on graphics processing unit i: An implementation in cg. 2007.