BRIEFINGS IN BIOINFORMATICS. VOL 11. NO 3. 334–347
Advance Access published on 1 February 2010 doi:10.1093/bib/bbp073

High performance cellular level agent-based simulation with FLAME for the GPU

Paul Richmond, Dawn Walker, Simon Coakley and Daniela Romano

Submitted: 2nd September 2009; Received (in revised form): 4th December 2009

Abstract

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. As with 'bottom-up' simulation approaches, cellular level simulations demand a high degree of computational power, which in large-scale simulations can only be achieved through parallel computing. The flexible large-scale agent modelling environment (FLAME) is a template driven framework for agent-based modelling (ABM) on parallel architectures ideally suited to the simulation of cellular systems. It is available for both high performance computing clusters (www.flame.ac.uk) and GPU hardware (www.flamegpu.com) and uses a formal specification technique that acts as a universal modelling format. This not only creates an abstraction from the underlying hardware architectures, but avoids the steep learning curve associated with programming them. In benchmarking tests and simulations of advanced cellular systems, FLAME GPU has reported massive improvement in performance over more traditional ABM frameworks. This allows the time spent in the development and testing stages of modelling to be drastically reduced and creates the possibility of real-time visualisation for simple visual face-validation.

Keywords: *agent-based modelling; high-performance computing; GPU; cellular modelling; computational modelling; parallel simulation*

INTRODUCTION

High-performance parallel computing offers enormous potential in analysing data and accelerating the simulation performance of large-scale complex system biology models. Whilst bottom-up [1], high-volume molecular level systems are often easier to parallelise and more computationally demanding than low volume top-down approaches [2, 3], the middle-out alternative centred on the cellular level is arguably a more natural starting point [4]. By modelling cells, the basic unit of biological function, predictive models based on the large amounts of data available at the cellular level, are able to provide insight into larger biological systems. Of the techniques used to model cellular systems, agent-based modelling (ABM), takes an individual-based approach, which unlike equation-based alternatives [5, 6], 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*/*in vivo* and *in silico* experiments for simple visual model validation.

Traditional ABM methods are highly serialised and are often based on mobile discrete spaced agents [7, 8]. Continuous spaced agent simulation

Corresponding author. Paul Richmond, The Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK. Tel: +44-7793712845; Fax: +44 (0) 114 222 1810; E-mail: p.richmond@sheffield.ac.uk
Paul Richmond is a Royal Academy of Engineering Student Development Fellowship Awardee at the University of Sheffield, where he has recently been awarded a Doctoral Prize Fellowship.
Dr Dawn Walker is an RCUK Fellow in the Computational Systems Biology group, in the Department of Computer Science, at the University of Sheffield.
Simon Coakley has a doctorate in Computational Systems Biology. His current research involves adapting a parallel agent based framework for economic modelling. He currently resides at the University of Sheffield.
Daniela Romano, lecturer in Computer Science, conducts research in virtual reality (VR) and is the team leader in the Kroto Research Institute for Multidisciplinary research at the University of Sheffield.

approaches tend to extend discrete simulation frameworks and with the exception of swarm systems [9], remain largely un-parallelised. The lack of parallel computing software for ABM places stringent limitations on the scale of models that may be simulated. Likewise, the trend toward multi-core processors [10] suggests that serial simulations are unlikely to be benefit from new processor architectures. Whilst multi-core processors allow Moore's law [11] to continue to predict increasing overall processing speed, ABM software must be able to sufficiently take advantage of the independent cores through new parallel algorithms and approaches. In order to scale ABM to massive simulation sizes, architectures and software able to exploit high-performance computing (HPC) are required. Specialist distributed processing remains a popular choice for such computation [12–14], however the GPU has received a vast amount of interest from the biological and physical sciences [15–19]. This can be attributed to cost effective performance gains achieved through exploiting its high throughput design (Figure 1). Technically, the GPU not only exceeds the transistor count of modern CPUs, with a significantly higher portion of transistors available for data processing, but memory bandwidth outperforms system memory by a factor of 10 [20]. Emerging architectures such as the Cell [21] and Larrabee [22] share similar design principles with the GPU and likewise offer enormous potential for ABM, providing software is available to exploit it.

This article describes the application of the flexible large-scale agent modelling environment framework for the GPU (FLAME GPU) [23, 24] to cellular level systems biology simulation. FLAME is an ABM environment designed around a formal modelling technique, orientated towards highly efficient parallel simulation. Previously it has been extensively developed for use with task parallel

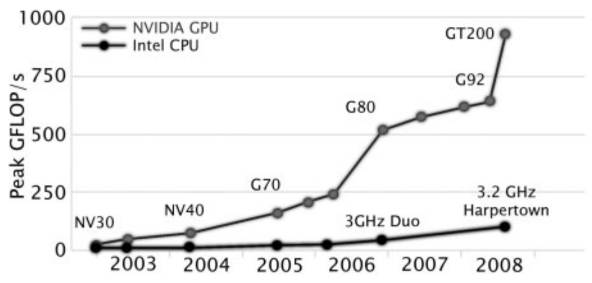


Figure 1: Peak Performance of NVIDIA GPU Hardware (grey) versus Intel CPU Hardware (black).

cluster architectures [25, 26]. This article describes the implementation on GPU hardware which offers the following advantages:

(i) Simulation of cellular models can be massively accelerated using the parallel GPU architecture to achieve performance beyond that of CPU-based frameworks and similar to that of execution on expensive clusters or grids.
(ii) The use of a continuous space environment allows a realistic simulation of cellular tissue growth and resulting tissue formation.
(iii) Model data can be saved and analysed post simulation, or as data is persistent within GPU memory can be used for real-time visual simulation aiding face-validation through comparison with *in-vitro* experiments.

Within this article, a review of ABM is first presented where it is contrasted with equation-based modelling approaches. Agent specification and formal modelling techniques for ABM are then discussed and compared with more common Object Orientated (OO) techniques. The FLAME GPU modelling and simulation process are then described, as are the details of implementation on GPU hardware. The limitations of FLAME GPU and the more general limitations of the GPU are discussed, as well as the implications for cellular simulation. A tissue growth modelling example from literature [27] is introduced as is a discussion of techniques for solving intercellular 'force' resolution. Using the exemplar skin tissue growth model, the performance of FLAME GPU is evaluated and finally, our plans for future work are presented.

AGENT- VERSUS EQUATION-BASED MODELLING

Equation-based models (EBMs) are extensively used to model system-level behaviour in biology. They often represent observable time varying quantities, such as population size or concentrations of a particular entity. Although some can predict systems level behaviour based on assumed lower level activities (e.g. change in population of a species over time, based on fixed birth and death rates), some are simply descriptive in that they are designed to match real-world observations. Despite some advantages of this type of simulation (e.g. faster run time and the availability of established libraries dedicated to the numerical integration of equations), EBM offers little insight into the micro-level behaviour representing the interactions of the individuals 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 [1, 28]. As with multi-agent systems (MAS), ABMs can be described as a 'system of interacting parts' (the notable difference being that agents (or individuals) within ABM are simulated as autonomous individuals where as MAS may use a more generic agent representation). Typically an ABM consists of a number of agents, an environment and a set of rules governing agent behaviour. Agents themselves are self-contained entities consisting of states and a set of behavioural rules. Agents may represent discrete spatial entities such as molecules or cells, in which case they may reside within a continuous or discrete spatial environment. In either case, agents may interact directly or through an environment where they compete for, or generate, 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 in many systems agents cannot be expressed as simple uniform entities. This is particularly true in cellular level systems, where cells in differing states (e.g. different cell-cycle phases) or subject to different micro-environments (e.g. local stress of biochemical gradients), may perform entirely different behaviours or exhibit differing phenotypes.

AGENT MODEL SPECIFICATION

Object Orientated Programming (OOP) is widely adopted as the most common paradigm for ABM frameworks [7, 8, 29, 30]. 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 [31–33]. Of these frameworks, most are accessible through an application programming interface (or API) and application layer. The API provides a tool for building and describing models, whilst the application layer implements common routines such as agent communication and scheduling of agent behaviour and interactions.

By contrast, FLAME GPU adopts the use of a formal technique for model specification called the X-machine [34, 35]. Formal techniques are advantageous as they provide a technique not only for specification but also validation using state machine analysis and testing algorithms [36]. Whilst formal specification is useful in the generation of system implementations, automatic validation (and verification) [37] is invaluable as it allows testing and error checking of systems. In the case of high integrity or mission critical systems [38], the guarantee of reliability and correct behaviour is not only advantageous, but essential. More specifically the advantageous of formal specification techniques can be described as [25]:

- The use of a unified and open format promotes better collaboration and understanding.
- The use of formal techniques for validation and verification of models.
- Modellers describe agents and behaviour using a simple formal concept which avoids having to map algorithms to complex parallel architectures.

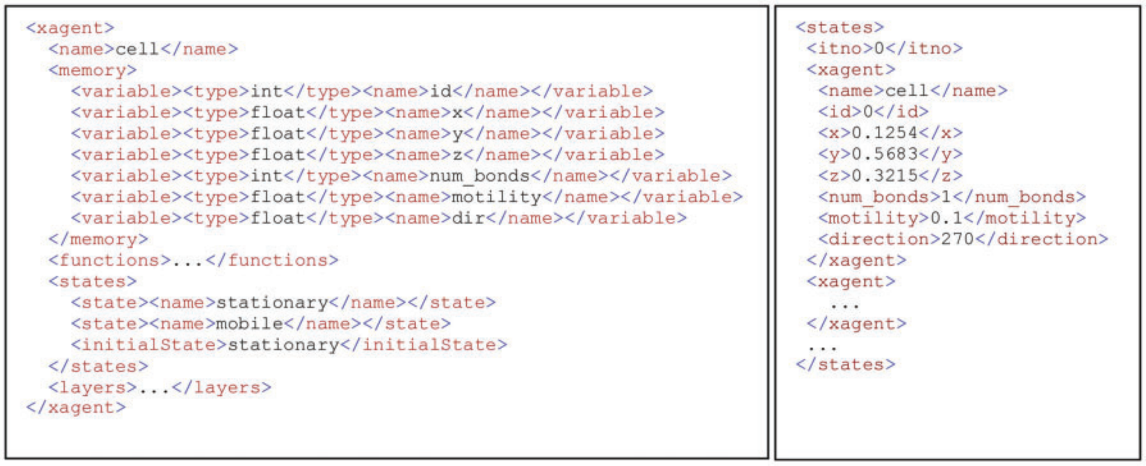
Many formal techniques appropriate for multi-agent systems are based around automaton- and state-based representation of agents. For example, cellular automaton (CA) can be described as a grid of interacting finite-state machines (FSMs). This provides a powerful technique for simple models, as states can be specified and rules defined as transition functions which describe the agents control flow from one state to another. 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 [39], π-calculus [40] and communicating sequential processes [41] have also been used for specification of concurrent systems and sequential processes. These algebraic techniques are particularly useful at modelling subcellular behaviour, particularly intracellular pathway signalling [42]. However, they tend to become extremely complex when applied to either continuous or discrete time ABM [43], with the generation of simulation code being far from straightforward [44].

The X-Machine is an extension to FSMs that is based around automaton and state based representation of agents. The inclusion of internal memory overcomes the exponential growth of FSM systems by allowing the number of states to be greatly reduced. This provides a powerful technique for computational modelling, as states can be specified and rules defined as transition functions controlling flow from one state to another. As a result the X-machine has been successfully applied to the formal verification of swarms [38], and as a technique to computationally model biological systems [45–47]. With respect to biological cell modelling, an X-machine can be used to directly model a cellular agent as a black box system. 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 as a 8-tuple $(\sum, \Gamma, Q, M, \Phi, F, q_0, m_0)$ [48], where $\sum$ and Γ is the input and output finite alphabet, respectively; Q is the finite set of states; M is the (possibly) infinite set called memory; Φ is the type of the machine SXM, a finite set of partial functions ø that map an input and a memory state to an output and a new memory state, ø: $\sum \times M \rightarrow \Gamma \times M$; F is the next state partial function that given a state and a function from the type Φ, provides the next state, F: $Q \times \Phi \rightarrow Q$ (F is often described as a transition state diagram or transition funtion); q_0 and m_0 are the initial state and memory, respectively.

Within FLAME GPU, the communicating stream X-machine (CSXM) [49], a variant of the X-machine which adds a communication mechanism between agents, is used to allow inter-agent interactions. Agents communicate by iterating input messages ($\sum$), output from neighbouring agents (Γ) during the transition function (F) which specifies movement between states. Rather than adhere strictly to the exact formal definition of a CSXM, FLAME GPU replaces fixed sized communication matrices with variable length message lists. This allows the possibility of dynamic population sizes during the simulation process (an essential feature in models that represent an expanding cell population).

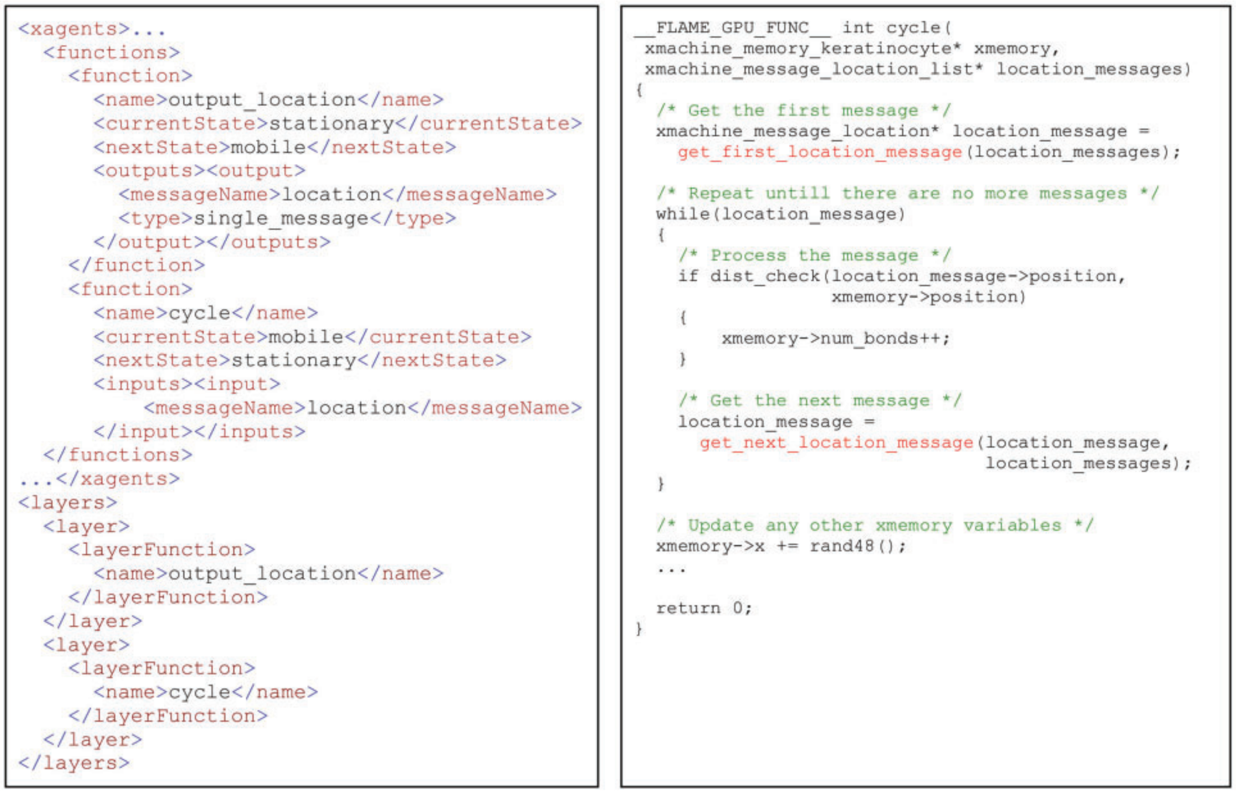
Specifying an X-machine agent in FLAME GPU can be achieved through the use of extensible mark-up language (XML) model files. An extendible XML schema [23] is used within FLAME to ensure the correct syntax of describing an X-machine model. The resulting XML syntax is known as the X-machine mark-up language, or XMML. Figure 2 demonstrates a simplified example of a cell specified using an XMML model. The memory XML element contains variables representing the agent's internal memory (M). States (Q) are listed within the states XML element, with an initialState element being used to define $q_{0.}$ The initial set of memory (m_0), or XML input state (Figure 3, left), is described



```
<xagent>
  <name>cell</name>
  <memory>
    <variable><type>int</type><name>id</name></variable>
    <variable><type>float</type><name>x</name></variable>
    <variable><type>float</type><name>y</name></variable>
    <variable><type>float</type><name>z</name></variable>
    <variable><type>int</type><name>num_bonds</name></variable>
    <variable><type>float</type><name>motility</name></variable>
    <variable><type>float</type><name>dir</name></variable>
  </memory>
  <functions>...</functions>
  <states>
    <state><name>stationary</name></state>
    <state><name>mobile</name></state>
    <initialState>stationary</initialState>
  </states>
  <layers>...</layers>
</xagent>
```

```
<states>
 <itno>0</itno>
 <xagent>
  <name>cell</name>
  <id>0</id>
  <x>0.1254</x>
  <y>0.5683</y>
  <z>0.3215</z>
  <num_bonds>1</num_bonds>
  <motility>0.1</motility>
  <direction>270</direction>
 </xagent>
 <xagent>
   ...
 </xagent>
 ...
</states>
```

Figure 2: An example of cell expressed as an X-machine agent. Left shows the XMML model definition and right shows an example of an initial memory configuration file.



```xml
<xagents>...
  <functions>
    <function>
      <name>output_location</name>
      <currentState>stationary</currentState>
      <nextState>mobile</nextState>
      <outputs><output>
        <messageName>location</messageName>
        <type>single_message</type>
      </output></outputs>
    </function>
    <function>
      <name>cycle</name>
      <currentState>mobile</currentState>
      <nextState>stationary</nextState>
      <inputs><input>
          <messageName>location</messageName>
      </input></inputs>
  </functions>
...</xagents>
<layers>
  <layer>
    <layerFunction>
      <name>output_location</name>
    </layerFunction>
  </layer>
  <layer>
    <layerFunction>
      <name>cycle</name>
    </layerFunction>
  </layer>
</layers>
```

```c
__FLAME_GPU_FUNC__ int cycle(
 xmachine_memory_keratinocyte* xmemory,
 xmachine_message_location_list* location_messages)
{
  /* Get the first message */
  xmachine_message_location* location_message =
    get_first_location_message(location_messages);

  /* Repeat untill there are no more messages */
  while(location_message)
  {
    /* Process the message */
    if dist_check(location_message->position,
                  xmemory->position)
    {
        xmemory->num_bonds++;
    }

    /* Get the next message */
    location_message =
      get_next_location_message(location_message,
                                location_messages);
  }

  /* Update any other xmemory variables */
  xmemory->x += rand48();
  ...

  return 0;
}
```

Figure 3: An example of the definition of XMML agent function definitions with corresponding function code script. Left shows the XMML function definition including the processing order layer definition. Right shows an example of a simple function script performing message iteration using the FLAME GPU message functions.

in a corresponding XML file which contains a list of agents and with values for each agent memory variable defined within the model specification. The definition of transition functions and the layering of function order (layers) are also included within the XMML definition however these are discussed in the following section.

AGENT BEHAVIOUR AND SIMULATION

Function specification in common OO frameworks and APIs [7, 8, 29, 30] is achieved through the use of functions or actions within an agent object. Similarly within FLAME GPU, the transition between states of the X-machine agent (*F* in the formal definition) is used to specify agent behaviour. The transition function (or agent function) is able to modify an agent's internal memory, as well as process input and output in the form of messages. Figure 3 (right) demonstrates the definition of two agent functions. The first of these is used to output a location message which is defined within the XMML model file as a simple list of memory variables. The second of these defines a function (migrate) which inputs the list of location messages. The function code is defined as scripted C code (Figure 3, right) with a simple example of the cell-cycle function defined below.

Although state-based formalisms have been previously been used for discrete event simulation [13], most agent based systems (including the FLAME GPU framework) require the ability to perform numerical integration, which by definition, requires execution of agent functions over a number of discrete time steps. Within FLAME GPU, the order of which agent functions are executed is determined through the specification of a number of function layers (Figure 3, within the <layers> element). Each function layer can contain any number of agent functions and functions within the same layer are assumed to have no dependencies as they may be processed simultaneously. Current GPU hardware supports only Single Program Multiple Data execution which forces functions in the same layer to be processed serially, however the next generation of NVIDIA hardware (Fermi) will allow function in the same layer to be processed simultaneously through Multiple Program Multiple Data execution support [50]. Functions which have dependencies are specified in additional layers. For example, the two functions in Figure 3 (left) have a dependency on the location message. It is important that the first function completes before the second attempts to read the list of location messages, and therefore the functions are defined using two layers.

Unlike OO methods, FLAME GPU's C based API is dynamically generated for each XMML model. In order to achieve this, templates are used to generate compliable simulation code, including agent and message specific API functions and data structures. Figure 4 shows the process of generating a FLAME GPU simulation. XMML model files can either be processed though a compliant extensible stylesheet language transformation (XSLT) processor. Two XMML schemas are used to ensure correct syntax of the model file. The first of these schemas (the base schema) validates the syntax of a base X-machine model, the second uses OO style extension of the base model to add any GPU specific model parameters. After generating simulation code, the simulation program must be compiled. If the visualisation template is used, then the resulting simulation program allows the model to be visualised in real-time. Alternatively the simulation can generate XML output for any number of simulation steps.

ACCELERATING FLAME WITH THE GPU

Modelling paradigms based on the interaction of discrete spaced entities are relatively trivial to simulate on GPU hardware [51–53]. Homogeneous agents, tightly packed into regular discrete environments can easily be stored in 2D data structures (such as textures) and simulated using a regular grid of parallel processes (or threads) for each agent. Communication between the discrete partitions is easily achieved using a gather operation over a fixed radius, spatially aware caching, such as that offered by the built in texture cache can provide extremely high performance.

Mobile discrete simulations consisting of agents navigating a grid like environment are less well-suited to a GPU implementation as they are traditionally implemented in sequential environments. Parallel implementations of discrete mobile systems on the GPU must explicitly handle collision conditions which result from agents simultaneously moving to the same discrete location. D'Souza [54] presents a novel technique for achieving this that uses a multi-pass priority scheme to iteratively solve all potential discrete movement collision possibilities. Within the continuous-based environment of FLAME GPU, discrete collision evaluation is not required as agents are not bound to a grid like environment. Likewise, collision detection between agents in continuous space is not explicitly required. Technically, messages allow any form of

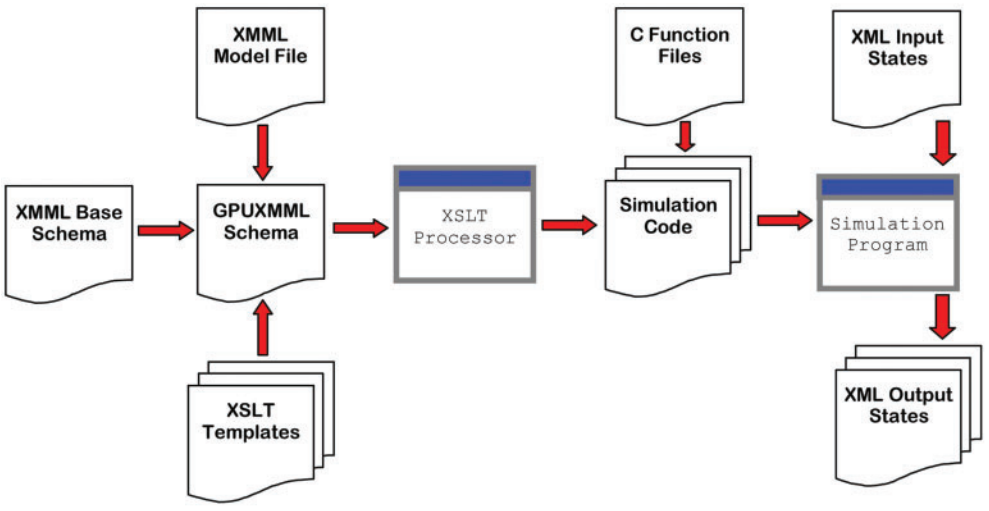
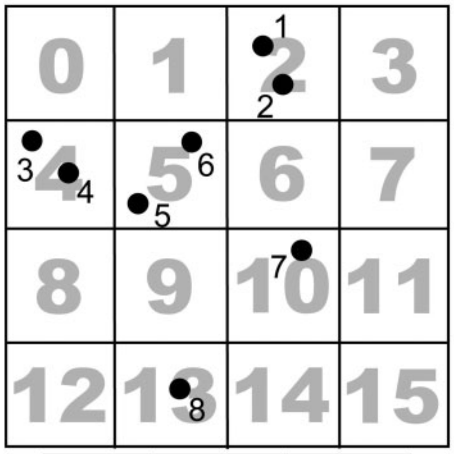
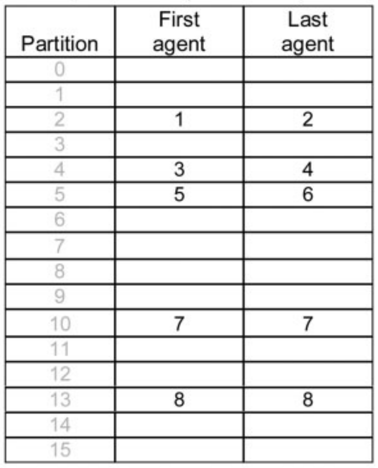


Figure 4: The FLAME GPU modelling process. Models are translated from an XMML file using a template processor which produces simulation code. The simulation program includes routines for real-time visualisation if the appropriate XSLT template is included.

communication to take place between agents and it is the agent functions which determine behaviour through the processing messages information. For example, within the cell modelling example, used later in this paper, agents migrate independently of potential physical agent collisions and use a subsequent force resolution process to solve physical cell overlaps.

The processing of messages (input) within agent functions is achieved through iteration using the optimised message iteration functions shown in the scripting example in Figure 3 (right). These can be customised for each message type within the XMML model for either brute force or spatially partitioned local message iteration. Brute-force message communication is implemented in FLAME for the GPU by using a tiling-based-technique inspired by N-body gravitational field summation [55]. A block of threads, each representing an agent, loads the batches of tiled messages into per multiprocessor shared memory. Agents are then able to serialise some message reads with minimal communication overhead achieving near optimal performance of the GPU hardware. Whilst $O(n^2)$ communication is extremely efficient, many agent systems including, those representing cellular systems, communicate using a small interaction radius (a restricted distance within which agents are able to communicate or mutually influence one anothers' behaviour). In this case, a common spatial partitioning technique is used, which is inspired by interacting particle systems [56]. The algorithm has been implemented in a number of interacting systems including swarm-based systems on both the GPU [57, 58] and the PS3 [59]. It consists of partitioning message space into a regular grid based on the interaction range of the particle or message. Messages are sorted using a parallel sort algorithm [60] and a matrix containing the start and end index position of any messages within the sorted list is built by using a scatter kernel (Figure 5). This in turn is used to iterate through messages in all 27 neighbouring partitions to that in which an agent is located. By examining all messages within neighbouring partitions, every message within the defined radius is considered for communication. As agents in different positions load a different number of messages from differing locations, shared memory cannot be used to accelerate the message reading. Instead the texture cache is used to exploit the spatial coherence of the partitioning algorithm. The efficiency of the texture cache depends on the ordering of the agents and hence the spatial partitioning algorithm used. Within FLAME GPU a relatively simple location hash is used, however, it has been suggested [19] that the Hilbert space filling curve [61] is able to increase the cache hit rate and hence performance.





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 5: An example of a 2D partitioned message space containing sorted messages. The matrix below 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 message partitions (5–7, 9–11, 13–15) to ensure all messages (5–8) within the potential message range are examined.

ADDRESSING THE LIMITATIONS OF GPU SIMULATION

The major weakness of performing simulation on the GPU resides within the general difficulty of data parallel programming. In the past, GPGPU programming was only available through the mapping of algorithms into textures which could be processed within the numerous programmable stages of the graphics pipeline through graphics APIs. Translational techniques partly solve this issue by mapping the concept of a Single Instruction Multiple Data stream kernel, to a high level GPU shading language [62–64]. More recently, however, hardware vendors have embraced GPGPU by providing low-level driver supported instruction sets with high-level language support. NVIDIA compute unified device architecture (CUDA) [20] has emerged as the most prominent of these, with the concept of data parallel execution within a grid of thread blocks forming the foundations for OpenCL [65], the recent open standard for parallel programming of heterogeneous systems.

Despite the drive towards higher level language support for GPU computing, achieving high performance still requires detailed knowledge of the underlying hardware and memory access patterns. Within FLAME GPU specialist knowledge is abstracted through the use of the template system allowing all GPU specific (CUDA) code to be automatically generated. The automatically generated code includes efficient data access and storage techniques [66], which maximise memory coalescing, reducing the number of memory read operations which must be issued to load agent and message data within agent functions. Likewise, efficient techniques are used to ensure lists of data within FLAME remain dense, which enables simple mapping to the underlying hardware. There are many occasions in FLAME simulations where dense lists of agent data become sparse, either through agent deaths, the introduction of new agents into empty lists, or the movement of agents from one state list to another. In order for the GPU to process sparse lists of data efficiently, they must be compacted. This is achieved through the use of a highly optimised parallel prefix sum operation [67] in which every element in the resulting lists is obtained from the sum of all previous elements. A scatter kernel can then move each active agent into a compacted list by changing its location to the position specified by the result of the scan.

The more general limitations of the GPU for simulating cellular systems, such as difficulty in including stochasticity and complex behaviour, are also addressed with FLAME GPU. A GPU implementation of GNUs random 48 algorithm [68] is included and can be used within agent function code to provide real-time random number generation. Likewise, FLAME GPU stores agents within state lists to ensure that agent functions are only applied to the correct subset of the agent population in the correct state. This minimises divergence across the agent population allowing greater performance even when agent functions become complex. Despite this, cellular simulation on the GPU in FLAME or otherwise does have specific limitations resulting from the hardware itself. GPU memory is a limited resource—even the GPGPU specific Tesla GPUs are limited to 4 GB. Behaviour complexity is also limited by the number of registers per multiprocessor. Once registers are filled, storage spills over to local memory, causing massive performance degradation. GPU hardware in the past has scaled well, suggesting that hardware specific limitations may be less of a problem in the future, however, for now extremely large or complex simulations may not be suitable for execution on a single piece of GPU hardware.

INTERCELLULAR CELLULAR FORCE RESOLUTION IN A KERTINOCYTE EXAMPLE

The Keratinocyte (cell) model [27] is a model of the *in vitro* behaviour of skin epithelial cells, based on published behaviour, or direct observation of this cell type *in vitro*. It has been developed as part of the Epitheliome project which aims to develop predictive ABMs of both skin and bladder [69] epithelial tissue. Ultimately, the goal is to use these models to help understand how cells interact during normal tissue growth, and to understand the potential sources of dysregulation in pathology (e.g. malignant development, compromised wound healing). The model includes behaviour representing traverse of the cell cycle, including cell growth and division (proliferative behaviour), leading to the addition of agents to the model. Simulation of differentiation describes how cells change type, from keratinocyte stem cells capable of infinite divisions, through a number of intermediate stages to fully differentiated superficial cells that cannot divide and are ultimately lost from the tissue surface. The removal of dead agents (apoptosis) from the simulation occurs and simulation of migration allows cells to actively move within a continuous space environment (Figure 6). Various hypotheses relating to the importance of specific biological rules have been tested

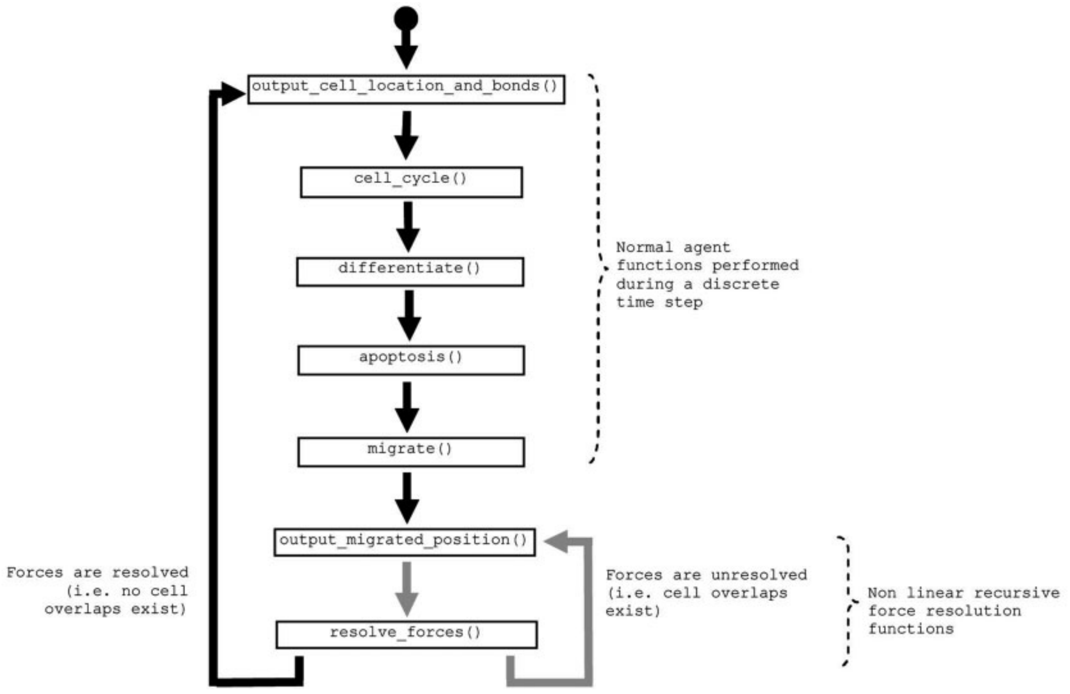

output_cell_location_and_bonds()
cell_cycle()
differentiate()
apoptosis()
migrate()
output_migrated_position()
resolve_forces()
Normal agent functions performed during a discrete time step
Forces are resolved (i.e. no cell overlaps exist)
Forces are unresolved (i.e. cell overlaps exist)
Non linear recursive force resolution functions

Figure 6: An example of the simulation flow of the keratinocyte modelling example. Arrows represent the resolution state of forces within the simulation (resolved or unresolved). After the processing of the normal cell functions, agents may perform any number of recursive force resolution steps.

with in-virtuo experimentation using the model, which have later also been tested *in vitro* [27, 69, 70].

Within FLAME GPU, the physical form of a cell within the keratinocyte model can be easily represented by incompressible spheres. As the simulation proceeds in step-wise discrete time, there is a high likelihood that the simulation will introduce some physical overlap of the cell objects. This is particularly true when simulating tissue cell colonies which form tight bonds and demonstrate a large amount of cell division. In order to correct any physical overlaps, a repulsive force can be applied to separate overlapping cells, therefore maintaining the physical integrity of the simulation. This repulsive force can be applied to cells in the same way as any other inter cellular forces, which may be the result of cell growth, motility, inter cell and substrate bonds or forces generated by the internal cytoskeleton.

Resolving intercellular forces of any kind using discrete time steps is extremely difficult to achieve within a single simulation step or agent function. However, there are a number of techniques which can be applied to agent simulations which aim to iteratively perform force resolution in order to achieve a stable state of cells (i.e. to reach a state where cells have trivial movement for additional force resolution steps). The first of these is to use a sufficiently large number of agent functions for force resolution within a single simulation step. For example, within FLAME GPU a large number of similar functions can be expressed for repeatedly resolving forces. Whilst this technique is suitable for perhaps a small fixed number of force resolution steps, larger numbers of resolution steps introduce large amounts of code repetition. Additionally, as the number of resolution steps required is variable between simulation steps (as different steps introduce greater amounts of force depending on the behaviour performed) using a fixed number of steps to ensure accurate resolution will in most cases perform more computation than is required. To avoid this, a recursive force resolution process can be used which iterates the force resolution behaviour until the population reaches a stable state and then stops. Within previous cell-based agent simulations, this has typically been achieved using external, sequential physical force solvers [27, 69, 70]. Using this technique, an agent program, such as FLAME, performs a simulation step and then outputs all agent information into some common format. This allows the external solver to read the agent information and iteratively perform force resolution before outputting the agent data back into the agent simulation environment ready for the next simulation step. The disadvantage of this technique is that external solvers are often defined only for single CPU architectures introducing a large bottleneck into simulation performance.

In order to ensure parallel force resolution, without a fixed number of resolution steps, the concept of a global agent function has been introduced to FLAME GPU to achieve recursive behaviour. A global function simply performs some conditional evaluation on each agent at the beginning of an agent function which can be used to determine if the agent has moved less than some small amount (indicating it has reached a stable state). A global parallel reduction algorithm [71] can then be used to count the number of agents which meet the global condition. If all agents evaluate the global condition as true, then the population is ascertained to be in a stable state and a normal simulation step (i.e. a step including any defined cell functions such as cycling, division and migration) can take place, incrementing the simulation by the defined discrete time step. If the global condition is not evaluated as true by all agents, then a non-linear time step is processed. This non-linear step performs only the force resolution agent functions and as a result does not advance the simulation by the discrete time interval. A non-linear simulation step can be repeated any number of times, up to a specified fixed limit, ensuring that any un-resolvable oscillating movement does not prevent the simulation from continuing. Figure 6 demonstrates this technique for the keratinocyte tissue colony simulation.

PERFORMANCE ENHANCEMENT OF GPU SIMULATIONS

The performance benefit of mapping simulations to the GPU is well documented [15–19]. In the case of simulating CA and discrete agent systems, speedups of over 100 times that of a sequential simulation environment are fairly common, however, the exact performance is dependant on the model specification itself. In order to evaluate FLAME GPU for the purposes of cellular simulation, the keratinocyte tissue model gives a good indication of the expected performance benefit. Figure 7 shows the speedup of performing the simulation using both brute force and limited range message iteration within the agent functions. Each test simulation is based upon an initial configuration state consisting of randomly distributed cells at a particular density. The speedup is calculated by considering the relative speed increase of the FLAME GPU iteration time in comparison with FLAME iteration time on the CPU (for the same initial configuration). Simulation steps utilising brute force computation were accelerated by an average of 250 times that of a comparable FLAME simulation running on a single CPU. Where spatial partitioning on the GPU was used to reduce the computation overhead of agent communication,

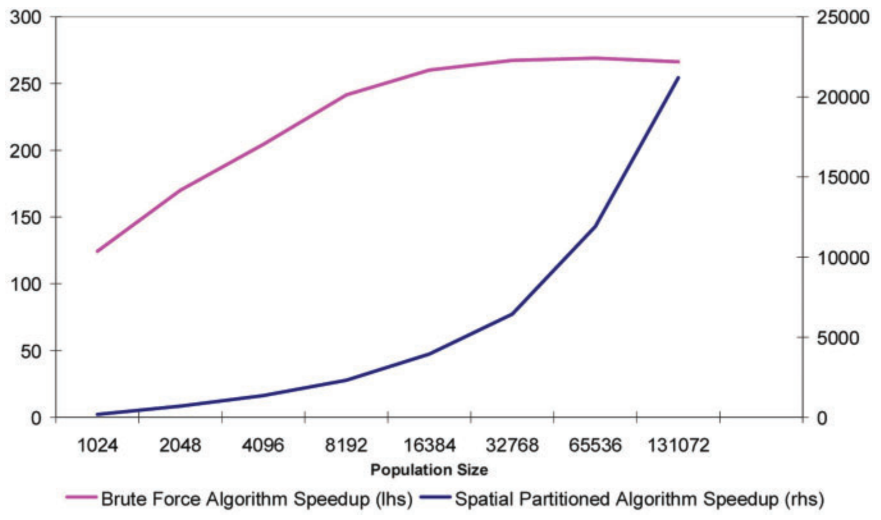


Figure 7: Relative performance of the keratinocyte model using both brute force and the spatially partitioned message iteration functions within FLAME GPU.

simulation time was reduced significantly. A single simulation step with over 130 000 agents which was previously simulated on the CPU for hours took only seconds.

To contrast the performance of FLAME GPUs limited range interactions with the performance of FLAME running on a HPC grid architecture, a second benchmarking model of a million simple agents, performing only movement functionality, has been compared. The model consists of two agent functions: the first outputs agent locations and the second iterates these to determine a migration movement for each agent. On the grid, agents are spatially distributed across nodes with messages spanning numerous nodes communicated using the message passing interface (MPI). The simulation performance of the model on a HPC grid scales well as the number of processors is increased, however, any more than 100 processing cores are unable to decrease performance time below 6 s due to memory bandwidth restrictions. By contrast, the same simulation running in FLAME GPU is able to perform the same simulation in less than a quarter of a second.

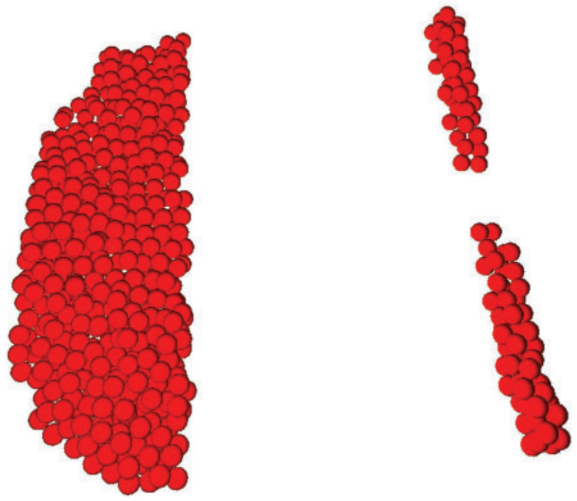


Figure 8: An example of an initial configuration of a keratinocyte scratch wound in FLAME GPU.

In the case of a tissue wound model (where a pre-existing high density agent population is subject to 'wounding' by removal of agents representing a 300 μm wide scratch, as shown in Figure 8) which reached a total stable population of over 2500 agents, simulation took no longer than half a second for both simulation and intercellular force resolution [23]. Force resolution was achieved by testing an agent's movement to ensure that it had moved <0.25 μm in a maximum of 200 resolution steps. Figure 9 shows the performance of this simulation, which took roughly 1500 iterations. The timing of the non-linear recursive force resolution steps is shown separately from the timing of normal 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 frames per second (FPS), inclusion of the force resolution steps ensures that the simulation remains real-time at over 60FPS throughout. In total, the simulation which previously took several hours using a single CPU core, could easily be

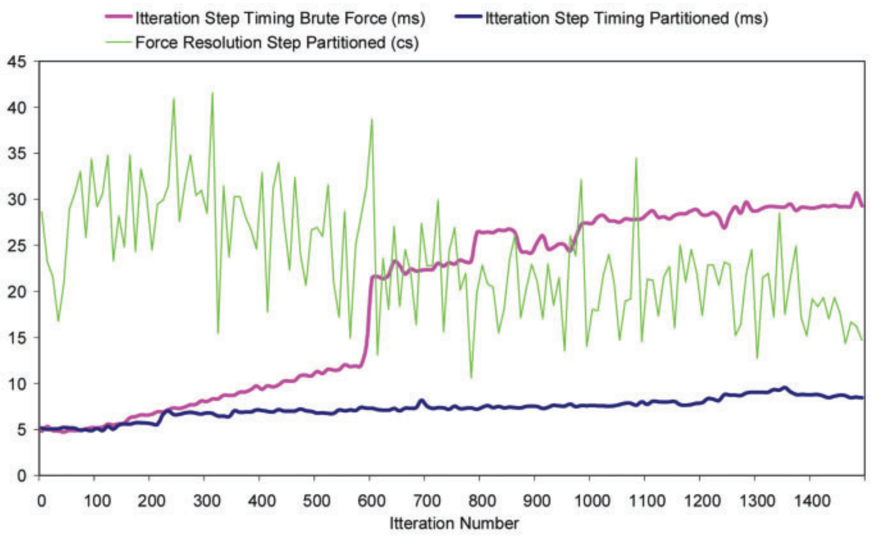


Figure 9: Progressive timing of normal agent functions and the recursive force resolution steps during a scratch would simulation of keratinocyte cells in FLAME GPU.

processed on the GPU in <2 min (for additional evaluation of the performance of this specific scratch wound simulation within FLAME GPU the reader is directed to [23]). Simple face-validation using the real-time visualisation technique can be used to ensure that the specified agent behaviour does not result in any obvious abnormalities in tissue formation. Comparison of the simulation with the non-parallel CPU simulation can also be visually validated before a more in-depth direct agent variable comparison can be performed between the two simulations offline.

DISCUSSION AND FUTURE WORK

Improvement in performance offered by cell simulation within FLAME GPU makes a significant leap forward with respect to the fast development and checking 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 techniques for ABM. The ability to specify models which can be automatically mapped to a complex architecture allows modellers to concentrate on model specification without having to understand the underlying hardware architecture.

In the future, it is expected that FLAME GPU will be extended in a number of ways. The first area to be improved will be the detail of cell models through the use of hierarchical modelling. The scaling up and down of the current middle-out cellular focus will be achievable through future hierarchical X-machine models. This is achievable by considering any agent function to be represented by another X-machine model. Such an approach will allow modelling of lower level physical properties of the cell, including more realistic cell communication processes (e.g. cell contact mediated or 'juxtacrine' signalling), as well as mechanotransduction (i.e. how cells translate mechanical forces into biochemical signals that may results in gene transcription and alter cell phenotype).

Distributing agents has already proved highly successful within cluster based HPC environments. However, future work will likely consider the use of low cost GRID technology. Middleware applications such as BOINC [72] offer excellent process scheduling functionality which could be exploited within low cost existing local networks. The use of messaging and formal description of models places no limit on the types of nodes which may be used. This can lead to a truly heterogeneous ABM platform with a mix of nodes using CPU and GPU processors.

Key Points

- FLAME GPU is a modelling environment allowing high-performance agent-based modelling on computer graphics card hardware.
- Discrete time steps are used to advance the simulation.
- Modellers do not require specialist knowledge of the underlying architecture used for simulation, as models are designed using formal specification techniques.
- Efficient algorithms for inter-agent communication and birth and death allocation ensure high simulation performance.
- The system is based on simple XML syntax which is easily extendable.
- Performance of a complex cellular tissue simulation has been increased drastically.

FUNDING

Paul Richmond is funded through an EPSRC Doctoral Prize Fellowship and Dawn Walker is funded through an RCUK Fellowship.

References

1. Noble D. The rise of computational biology. *Nat Rev Mol Cell Biol* 2002;**3(6)**:459–63.
2. Sanbonmatsu KY, Tung CS. High performance computing in biology: multimillion atom simulations of nanoscale systems. *J Struct Biol* 2007;**157**:470–80.
3. Burrage K, Hood L, Ragan M. Advanced computing for systems biology. *Brief Bioinform* 2006;**7**:390–8.
4. Walker DC, Southgate J. The virtual cell – a candidate co-ordinator for 'middle-out' modelling of biological systems. *Brief Bioinform* 2009;**10(4)**:450–61.
5. Gaffney EA, Maini PK, Sherratt JA, *et al*. The mathematical modelling of cell kinetics in corneal epithelial wound healing. *J Theoret Biol* 1999;**197**:15–40.
6. Johnston M, Edwards C, Bodmer W, *et al*. Mathematical modeling of cell population dynamics in the colonic crypt and in colorectal cancer. *Proc Natl Acad Sci USA* 2006; **104(10)**:4008–13.
7. Luke S, Cioffi-Revilla C, Panait L, *et al*. MASON: a multiagent simulation environment. *Simulation* 2005;**81(7)**: 517–27.
8. Minar N, Burkhart R, Langton C, *et al*. The Swarm simulation system: a toolkit for building multi-agent simulations. Working Paper 96-06-042. Santa Fe Institute, Santa Fe, 1996.
9. Quinn MJ, Metoyer RA, Hunter-zaworski K. *Parallel implementation of the Social Forces Model,* 2003.

10. Geer D. Industry trends: chip makers turn to multicore processors. *Computer* 2005;**38(5)**:11–3.
11. Moore GE. Cramming more components onto integrated circuits. *Electronics* 1965;**38(8)**:114–7.
12. Theodoropoulos G, Logan B. Distributed simulation of agent-based systems. In: Topping BHV (ed). In: *Developments in Computational Mechanics with High Performance Computing: Proceedings of the Third Euro-conference on Parallel and Distributed Computing for Computational Mechanics*. Weimar: Civil-Comp Press, 1999;147–54.
13. Lees M, Logan B, Theodoropoulos G. Distributed simulation of agent-based systems with HLA. *ACM Trans Model Comput Simul* 2007;**17(3)**:11.
14. Uhrmacher AM, Gugler K. Distributed, parallel simulation of multiple, deliberative agents. In: *PADS '00: Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*. Washington, DC: IEEE Computer Society, 2000;101–8.
15. Ackermann J, Baecher P, Franzel T, *et al*. Massively-parallel simulation of biochemical systems. In: *Proceedings of Massively Parallel Computational Biology on GPUs, Jahrestagung der Gesellschaft für Informatik e.V,*. Workshop proceedings. Lübeck, Germany, 2009.
16. Phillips JC, Zheng G, Kumar S, *et al*. NAMD: biomolecular simulation on thousands of processors. In: *Supercomputing '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Los Alamitos, CA: IEEE Computer Society Press, 2002; 1–18.
17. Kupka S. Molecular dynamics on graphics accelerators. University of Vienna: Web Proceedings of CESCG, 2006.
18. Liua W, Schmidt B, Vossa G, *et al*. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. *Comp Phys Commun* 2008;**179**:634–41.
19. Anderson JA, Lorenz CD, Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J Comput Phys* 2008;**227(10)**:5342–59.
20. NVIDIA. NVIDIA CUDA compute unified device architecture programming guide. *NVIDIA* 2007.
21. Williams S, Shalf J, Oliker L, *et al*. The potential of the cell processor for scientific computing. In: *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*. New York, NY: ACM, 2006;9–20.
22. Seiler L, Carmean D, Sprangle E, *et al*. Larrabee: A many-core x86 Architecture for visual computing. *IEEE Micro* 2009;**29(1)**:10–21.
23. Richmond P, Coakley S, Romano D. Cellular level agent based modelling on the graphics processing unit. In: *Proceedings of the Workshop on High Performance Systems Biology (HiBi09)*. CoSBi, Trento, Italy: IEEE Computer Society, 2009;43–50.
24. Richmond P, Coakley S, Romano D. A high performance agent based modelling framework on graphics card hardware with CUDA. In: *Proceedings of Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* 2009, Budapest, Hungary.
25. Coakley S, Smallwood R, Holcombe M. Using X-machines as a formal basis for describing agents in agent-based modelling. In: *Proceedings of 2006 Spring Simulation Multiconference*. Huntsville, AL: Workshop proceedings, 2006;33–40.
26. Adra SF, Coakley S, Kiran M, *et al*. *An Agent-based Software Platform for Modelling Systems Biology*. University of Sheffield Epitheliome Project: Technical Report: University of Sheffield, 2008.
27. Sun T, McMinn P, Coakley S, *et al*. An integrated systems biology approach to understanding the rules of Keratinocyte colony formation. *J Royal Soc* 2007;**4**:1077–92.
28. Dyke V, Savit R, Riolo RL. *Agent-Based Modeling vs Equation-Based Modeling: A Case Study and Users' Guide* 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.1164 (4 December 2009, date last accessed).
29. Iglesias CA, Garijo M, Centeno-González J. A survey of agent-oriented methodologies. In: *ATAL '98: Proceedings of the Fifth International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*. London, UK: Springer-Verlag, 1999;317–30.
30. North MJ, Howe TR, Collier NT, *et al*. Repast Simphony Runtime System. 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.
31. Odell J, Parunak H, Bauer B. *Extending UML for Agents,* 2000. http://citeseer.comp.nus.edu.sg/odell00extending.html (4 December 2009, date last accessed).
32. Bauer B, Müller JP, Odell J. Agent UML: A formalism for specifying multiagent interaction. In: Ciancarini P, Wooldridge M (eds). *Agent-oriented Software Engineering*. Berlin: Springer, 2001;91–103.
33. Page B, Knaaka N, Kruse S. A discrete event simulation framework for agent-based modelling of logistic systems. In: *Informatik 2007 Informatik trifft Logistik, Proc. 37*. Jahrestagung der Gesellschaft für Informatik. Bremen, 2007;397–404.
34. Eilenberg S. *Automata, Languages, and Machines*. Orlando, FL: Academic Press Inc, 1974.
35. Holcombe M. X-machines as a basis for dynamic system specification. *Software Eng J* 1988;**3(2)**:69–76.
36. Kiran M, Coakley S, Walkinshaw N, *et al*. Validation and discovery from computational biology models. *Biosystems* 2008;**93(1–2)**:141–50.
37. Balci O. Verification validation and accreditation of simulation models. In: *WSC '97: Proceedings of the 29th Conference on Winter Simulation*. Washington, DC: IEEE Computer Society, 1997;135–41.
38. Hinchey MG, Rouff CA, Rash JL, *et al*. 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*. New York, NY: ACM, 2005;125–33.
39. Abrial JR, Schuman S, Meyer B. A specification language. In: Macnaghten AM, McKeag RM (eds). *On the Construction of Programs*. Cambridge University Press, 1980;343–410.
40. Milner R, Parrow J, Walker D. *A Calculus of Mobile Processes, Parts I and II; 1989. -86*. http://citeseerx.ist.psu.edu/viewdoc/summary?doi = 10.1.1.35.8193 (4 December 2009, date last accessed).
41. Hoare CAR. Communicating sequential processes. *Commun ACM* 1978;**21(8)**:666–77.
42. Regev A, Silverman W, Shapiro E. Representation and simulation of biochemical processes using the pi-calculus process algebra. *Pac Symp Biocomput* 2001;459–70. Conference Proceedings of Pacific Symposium of Biocomputing, Hawaii.

43. d'Inverno M, Luck M. Formal agent development: framework to system. In: *In Formal Approaches to Agent-based Systems: First International Workshop, FAABS 2000*. London: Springer-Verlag, 2000;133–47.
44. Rouff C, Truszkowski W, Rash J, *et al*. *A Survey of Formal Methods for Intelligent Swarms*. Greenbelt, MD: NASA Goddard Space Flight Center, 2005.
45. Eleftherakis G, Kefalas P, Sotiriadou A, *et al*. Modeling biology inspired reactive agents using X-machines. In: *Proceedings of the International Conference on Computational Intelligence (ICCI04), Istanbul*. Istanbul: Workshop proceedings, 2004.
46. Gheorghe M. *Molecular X-machines*. Department of Computer Science, University of Sheffield, 2005.
47. Kefalas P, Stamatopoulou I, Gheorghe M. A formal modelling framework for developing multi-agent systems with dynamic structure and behaviour. In: *Multi-Agent Systems and Applications IV*. Springer: Berlin/Heidelberg, 2005;122–31.
48. Laycock GT. *The Theory and Practice of Specification Based Software Testing*. Department of Computer Science, University of Sheffield, 1993.
49. Balanescu T, Cowling AJ, Georgescu H, *et al*. Communicating stream X-machines systems are no more than X-machines. *j-jucs* 1999;**5(9)**:494–507.
50. NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture*. NVIDIA Technical Report: NVIDIA, 2009.
51. Tapia JJ, D'Souza R. Data parallel algorithms for large-scale real-time simulation of the cellular potts model on graphics processing units. In: *Proceedings of SMC2009*. San Antonio, Texas: IEEE, SMC, 2009.
52. Singler J. Implementation of cellular automata using a graphics processing unit. In: *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*. Los Angeles, CA: Workshop Proceedings, 2004.
53. Tran J, Jordan D, Luebke D. New challenges for cellular automata simulation on the GPU. *ACM Workshop on General Purpose Computing on Graphics Processors*. Los Angeles, CA: Workshop Proceedings, 2004.
54. Lysenko M, D'Souza RM. A framework for megascale agent-based model simulations on the GPU. *J Artificial Soc & Social Simul* 2008;**11(4)**:10.
55. Nyland L, Harris M, Prins J. Fast N-Body Simulation with CUDA. In: Nguyen H (ed). *GPU Gems 3*. Addison Wesley Professional, 2007.
56. Green S. *CUDA Particles*. NVIDIA SDK White Paper, 2007.
57. Richmond P, Romano D. Agent Based GPU, a Real-time 3D Simulation and interactive visualisation framework for massive agent based modelling on the GPU. In: *Proceedings of International Workshop on Supervisualisation 2008 (IWSV08), ICS08, Kos Island, Greece,* 2008.
58. Erra U, De Chiara R, Scarano V, *et al*. *Massive Simulation using GPU of a Distributed Behavioral Model of a Flock with Obstacle Avoidance,* 2004. http://wonderland.dia.unisa.it/projects/gebs/ (4 December 2009, date last accessed).
59. Reynolds C. Big fast crowds on PS3. In: *Sandbox'06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*. Boston, MA: New York, ACM, 2006; 113–21.
60. Le Grand S. Broad-Phase Collision Detection with CUDA. In: Nguyen H (ed). *GPU Gems 3*. Addison Wesley Professional, 2007.
61. Moon B, Jagadish Hv, Faloutsos C, *et al*. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Trans on Knowl and Data Eng* 2001;**13(1)**:124–41.
62. Buck I, Foley T, Horn D, *et al*. Brook for GPUs: stream computing on graphics hardware. *ACM Trans Graph* 2004; **23(3)**:777–86.
63. Fan Z, Qiu F, Kaufman A. ZippyGPU: programming toolkit for general-purpose computation on GPU clusters. *Supercomputing 2006 Workshop on Gneral Purpose GPU Computing* 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.127.5390 (4 December 2009, date last accessed).
64. McCool MD, Qin Z, Popa TS. Shader metaprogramming. In: *HWWS'02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002;57–68.
65. The OpenCL 1.0 Specification. Khronos Group, 2009. Khronos Group Technical Report.
66. Howes L. *Loading Structured Data Efficiently using CUDA*. NVIDIA, 2007.
67. Sengupta S, Harris M, Zhang Y, *et al*. Scan primitives for GPU computing. In: *GH'07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007;97–106.
68. van Meel JA, Arnold A, Frenkel D, *et al*. Harvesting graphics power for MD simulations. *Mol Simul* 2007;**34**: 259–66.
69. Walker DC, Southgate J, Hill G, *et al*. The epitheliome: agent-based modelling of the social behaviour of cells. *Bio Systems* 2004;**76(1–3)**:89–100.
70. Sun T, McMinn P, Holcombe M, *et al*. Agent based modelling helps in understanding the rules by which fibroblasts support keratinocyte colony formation. *PLoS ONE*, 2008; **3(5)**:e2129.
71. Harris M.*Optimizing Parallel Reduction in CUDA Whitepaper*. NVIDIA Developer Technology Group, 2008.
72. Anderson DP. *BOINC: A System for Public-Resource Computing and Storage* 2004;4–10. http://dx.doi.org/10.1109/GRID.2004.14.