Cellular Level Agent Based Modelling on the Graphics Processing Unit

Richmond, Paul
University of Sheffield, UK
P.Richmond@sheffield.ac.uk

Coakley, Simon
University of Sheffield, UK
S.Coakley@sheffield.ac.uk

Romano, Daniela, M.
University of Sheffield, UK
D.Romano@sheffiled.ac.uk

Abstract

Cellular level agent based modelling is currently reliant on either sequential processing environments or expensive and largely unavailable PC grids. The GPU offers an alternative architecture for such systems, however the steep learning curve associated with the data parallel architecture has previously limited the uptake of this emerging technology. This paper presents a template driven agent architecture (FLAME GPU) which provides a mapping of XML model specifications and C language scripting to optimised Compute Unified Device Architecture (CUDA) for the GPU. Our work has been validated though the implementation of a Keratinocyte model which uses limited range message communication with non linear time simulation steps to resolve inter cellular forces. The performance gain achieved over existing modelling techniques reduces simulation times from hours to seconds. Furthermore the improvement of simulation performance allows us to present a real time visualisation technique which was previously unobtainable.

1. Introduction

Agent Based Modelling (ABM) is a powerful technique which is used to simulate group behaviour from a number of simple interacting rules between communicating autonomous agents. Within systems biology ABM offers an alternative to using higher level differential equations which fail to capture important lower level interactions. Such interactions are important to improving understanding of biological models as they result in the observable systems level emergent behaviour. An obvious consequence of this bottom up modelling technique is that simulating many individuals is computationally expensive especially considering the scale of models which are required to produce observable results.

Traditional ABM toolkits such as Repast [1], Mason [2] and Swarm[3] are primarily aimed at a single CPU architecture, and whilst they offer simple agent specification techniques, their inherent lack of parallelism seriously affects the scalability of models. Alternatively frameworks exploiting parallelism [4] have concentrated on coarse task level parallelism targeted at processing clusters or grids. Whilst this technique offers the potential to improve simulation performance, such systems are expensive and unavailable for the majority of AB modellers. In addition to this, cellular models of tissue growth [22], including the Keratinocyte model [19] which we use to evaluate our own work, requires non parallel external tools to solve inter cellular forces between simulation steps. Whilst computationally expensive, physical force resolution is essential in maintaining a physical integrity. Without this stage critical biochemical reactions during cell contact are likely to produce abnormal tissue growth.

The trend towards the use of emerging hardware architectures has been advantageous to many fields of biological and physical sciences [1, 18, 21]. The introduction of the Compute Unified Device Architecture (CUDA) has simplified the previously tedious process of mapping General Purpose computation on the GPU (GPGPU) [8, 14]. Despite this examples of ABM on the GPU are limited in quantity and are typically examples of specific models [3, 5, 16]. The most prominent of these includes the work of D'Souza [5] which implements a discrete Sugarscape model using a densely partitioned grid to hold agent data. Whilst efficient, the grid based implementation is however memory intensive and suitable only for fine grained 2D or coarse grained 3D environments.

---

[1] http://repast.sourceforge.net/
[2] http://cs.gmu.edu/~eclab/projects/mason/
[3] http://www.swarm.org/

Although GPU programming has certainly become more accessible, achieving optimal performance still requires sound knowledge of the underlying hardware architecture. Flexibility is an aspect of ABM which is often overlooked specifically within GPU examples. As with other specific examples such as molecular dynamics [1, 18, 21], ABM requires tools for using the GPU which are not only efficient but concentrate on reusability and flexibility. The focus of this paper is therefore to describe a data parallel framework for the ABM on the GPU. Our work aims to overcome the limitations of previous techniques including our own previous work, [15, 16] to be suitable for parallel cellular tissue modelling. More specifically our work makes the following contributions.

1. An ABM framework: FLAME GPU is presented, which uses formal agent specification techniques with model validation and a simple extension mechanism.
2. A technique for non linear time step modelling is presented, which allows the implementation of parallel physical solvers required for tissue modelling.
3. The performance of existing tissue models is massively accelerated reducing simulation time to the point that real time 3D visualisation of results is possible.

2. A Review of GPU Hardware

The speed of GPU hardware is attributed to the architectural design. Unlike more generic and flexible CPUs the GPUs architecture is task specific making it highly optimised for stream 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 [12]. In addition to this the GPUs memory bandwidth exceeds that of system memory bandwidth by roughly a factor of 10. Figure 1 demonstrates the computational power of the GPU in direct comparison with Intel processors.

The class of GPU hardware targeted by the work in this paper is specifically limited to CUDA enabled graphics cards. Whilst it is desirable to support a wider range of GPUs the CUDA API allows access to a hardware functionality not supported by older generation cards and competing GPU manufactures. More specifically the availability of local (on chip) shared memory offers extremely fast parallel memory access operations for threads within the same multiprocessor. In addition to this local synchronisation provides thread cooperation allowing data caching through shared memory access.

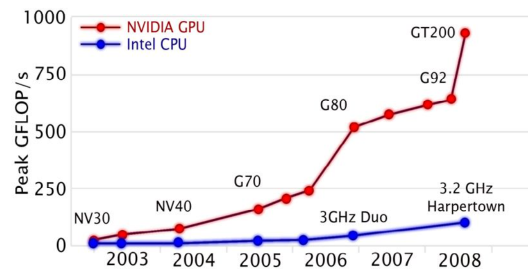

Figure 1. Peak Performance of NVIDIA GPU Hardware (Red) vs. Intel CPU Hardware (Blue). Data from [12]

The GPU programming model is described in detail in the CUDA Programming Guide [12] where it is presented as a parallel coprocessor. The GPU device architecture is described as Single Program Multiple Data (SPMD) where the program, or kernel, is some function native to the device and operating on multiple parallel threads. In order to generalise CUDA to multiple hardware implementations (with varying parallel capabilities) the idea of a grid of thread blocks is used to group threads. Whilst there is no global communication between threads, threads within the same block are able to cooperate by sharing information through the use of fast on chip shared memory. For each kernel execution the CUDA API provides a template based mechanism for grid and block size specification. Each block is then optimally assigned to the underlying multiprocessor hardware for parallel evaluation.

At a hardware level blocks are split into smaller units of 32 threads called warps. Blocks are then distributed amongst multiprocessors with the amount of blocks per multiprocessor 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. During execution on the multiprocessor, warps are processed a single instruction at a time which allows the multiprocessor to switch between warps which are ready for execution. This warp interleaving allows global memory access latency to be effectively hidden providing the multiprocessor can be kept busy with non latent arithmetic instructions. As each instruction across a warp is executed in parallel, any conditionals branches between threads must follow the same path to attain maximum performance. In the case of divergent branches (or warp serialisations) between

threads, instructions must either be serialised or multiple paths evaluated by every thread.

3. Agent Specification

Formal agent based specification is important within agent based modelling as it allows a simple and intuitive way of defining agents and their associated behaviour. Rather than creating a new framework and rewriting existing models, our work extends the already established FLexible Agent Modelling Environment (FLAME) [4]. The choice to extend FLAME not only aids better collaboration and understanding, but provides a basis for formal validation and verification of code [17]. FLAME itself is not a modelling platform instead it uses templates to produce simulation code. Models are described using a XML specification language (XMML) based around a formal modelling extending Finite State Machines (FSMs) concept called the X-Machine [6]. More specifically, FLAME builds upon a smaller subclass of X-Machines known as Communication Stream X-Machines (CSXMS) [2]. Due to their design targeted at streaming data, CSXMS are well suited for integration within parallel systems. Within FLAME agents are defined as a set of states with internal memory. A transition function determines an agent's next state and performs internal memory updates. The communication between agents is handled through the use of messages. Rather than use a large communication matrix [2] this is instead implemented in FLAME by more flexible variable length Message Lists.

One of the weaknesses of FLAME, resides within the main template system which maps XMML model specifications into compliable C code. The problem with this technique is that when extensions are made to the XMML language the template parser (XParser) needs to be modified to handle the additional XML tags. As the FLAME project is continually under development and being extended by multiple research projects we have defined an extendable system using XML Schema. This parallels Object Oriented techniques by allowing extensions to XMML to play the role of subclasses from a base XMML definition [20]. Through XML validation, model files can be checked to ensure they contain syntactically correct XML including element structure and value types. The base XML class itself comprises of a modular design of globally defined Complex Types. Figure 2 shows the Schema definition for the base `xagent` element which is referenced within the non extendible `xagents` list element.

```
<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>

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

<element name="xagents">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="1"
               ref="xagent" />
    </sequence>
  </complexType>
</element>
```

Figure 2 – XMML Schema definition of the xagent element

In this example an `xagent` can be extended though a new complex type which uses derivation by extension. Figure 3 shows an extract from our GPUXMML schema which imports the original XMML schema under the '`xmml`' namespace. This allows redefinition of the `xagent_type` and `xagent` element. The use of substitution groups allows the new `xagent` definition to replace the original producing valid XML code when run though an XML valuator such as Xerces or the .NET validator in Visual Studio.

```
<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" />
```

Figure 3 – GPU XMML Schema extension of the xagent element

Documents validating against our new XMML schema are intentionally very similar to original FLAME XML models. This enables a simpler migration of old models to our newer syntax. The

majority of minor changes are a result of parameterisation of agent functions which were previously not required in task parallel code. Both message and agent, inputs and outputs, must now be explicitly specified within XMML. In order to maintain high performance, GPU memory is pre allocated and as a result messages and agents require a buffer size representing an upper bound on their population size. The full XMML and GPUXMML definition files are available online at http://www.dcs.shef.ac.uk/~paul/FLAMEGPU.

To avoid the problems inherent with the XParser we have defined our code templates using Extensile Stylesheet Language Translations (XSLT). This removes the dependency on a specific template parser as any XSLT (version 1.0) compliant processor can convert an XMML model file to compliable code using our predefined templates. This change not only provides a more robust and standardised mechanism for template translation but also ensures validation of the templates though the same XMML schemas as the model files. Relationships defined with the XMML schema provide further assurance on the validity of model files by ensuring references to state changes and message/agent, inputs and outputs refer to valid XML elements defined in the same document.

In contrast with FLAMEs previous use of DTD validation and use of the XParser our technique 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 our technique no longer relies on the XParser our 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.

4. Agent Behaviour Scripting

Agent behaviour is determined though the definition of any number of agent functions within the XMML model specification. Each function defined requires a corresponding C code implementation. All other aspects of simulation such as memory allocation, importing of XML agent data and any pre/post processing required before agent functions are automatically generated by the extensive XSLT templates. Within this code both agent and message variables are stored within a Structure of Arrays (SoA) rather than the more intuitive Array of Structures (AoS). This allows a more efficient memory access pattern (coalescing) for both reading and writing data in global GPU memory [10]. Automatically generated code also handles post processing of agent births and deaths by using a parallel prefix sum algorithm [9]. This calculates a new index value for sparse data which is scattered to the appropriate SoA list.

Each agent function is parameterised containing a structure representing agent memory and a structure for message input/output and agent output lists. The agent structure represents a single agent instance and its member variables are determined directly by the content of the XMML model specification. An additional `RNG_rand48` structure can be used with a random number function implementing the GNU rand48 random number generator [21]. Figure 4 represents the skeleton structure of an agent function taken from our later example. The use of automatically generated message functions (red) ensures message iteration performance is optimal (described in Section 5). In addition to the message fetching functions in the example, each message and agent type has an additional function to add messages/agents to an appropriate SoA list.

```
__FLAME_GPU_FUNC__ int differentiate(
  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,
                               partition_matrix,
                               x1, y1, z1);
  /* Repeat untill there are no more messages */
  while(location_message)
  {
    /* Process the message */

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

  /* Update xmemory variables */

  return 0;
}
```

Figure 4 – An example of an agent function showing message iteration

5. Agent Communication

Whilst it would be possible for agents to process message lists within the agent function through simple iteration. This technique fails to exploit the maximum performance from GPU hardware. We have provided two efficient techniques for message iteration. The

first utilises per multiprocessor shared memory to improve the performance of brute force message processing. The second technique uses spatial partitioning to reduce the number of messages processed per agent depending on a pre specified message range and spatial boundary.

5.1. Brute Force Message Communication

In order to process messages using the brute force technique, a tiling method inspired by Nyland et al. [13] utilises shared memory by serialising message access across threads. Technically this requires that messages are split into groups with the first message group being loaded into shared memory by the `get_first_message()` function (Figure 5). Following this each thread within the same thread block sequentially reads messages from shared memory using the `get_next_message()` function. After each thread has exhausted the messages within the group (or tile) the `get_next_message()` function synchronises threads in the block and loads the next group of messages into shared memory (Figure 6).

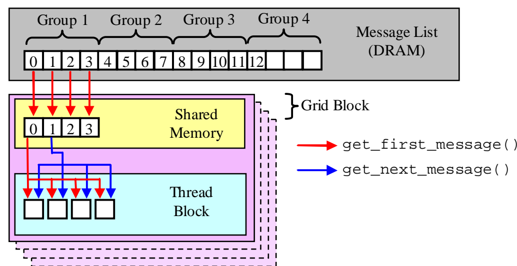

Figure 5 - Message group loading when requesting the first and next message.

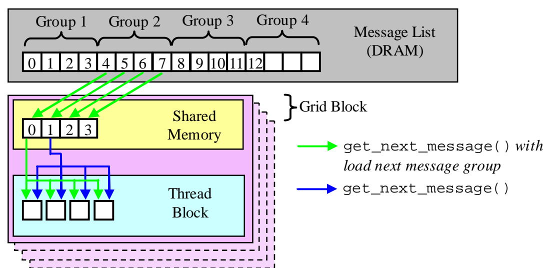

Figure 6 - Message group loading when requesting the next message from the beginning of a new message group.

Message group size and thread block size are equal so individual threads are responsible for loading shared memory values concurrently. A thread synchronisation is performed after loading any data into shared memory and ensures that all messages are available to all threads within the block. 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. Thread blocks beginning mid way through the message list load each message group sequentially from their starting group before circulating back to the first. The `get_next_message()` function then returns false after the same number of messages across the entire agent population have been processed.

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 follows 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 agents 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. Limited Range Message Communication

In cellular tissue modelling brute force message communication is highly inefficient as cells interact only within a limited interaction radius. In order to reduce the $O(n^2)$ processing a spatial partitioning technique [7, 1] has been adopted which we have previously demonstrated with swarm modelling [16]. The agent environment is split into cellular partitions (often referred to as buckets) and the message list is sorted depending on the cellular position of each message [7]. A cellular boundary matrix which contains the index of the first and last message per cell is then determined using scattered writes. As each cellular dimension is equal to the message range this guarantees that processing of each message within the 27 neighbouring cells of an agent (including its own)

will include all messages within the message radius. Figure 7 shows the pseudocode algorithm which given an existing message returns the next message from the neighbouring cellular partitions. More specifically it loops through the neighbouring cells looking for a cell which contains messages (line 21). When a cell containing messages is found or if there are more messages in the previous messages cell (line 10) then a message is returned. The variable `relative_cell` holds a vector of integers in the range $-1 \le x \ge 1$ 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).

```
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 7 - Pseudocode algorithm for spatial message loading

The function `cellPosition` calculates the cellular position of a continuous valued point within the message partition space. In the case that the point lies outside of the partition bounds (specified in the GPU XMML model file) the position is wrapped. The function `hashCellPosition` performs a hash function mapping the cell identifier to a unique integer [11]. It should be noted that 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 distance check within the agent function as they would be when using the brute force technique.

6. Non Linear Time Modelling

As with their formal definition, X-Machine agent functions have a start state and end state as well as a possible function condition. Function conditions allow agents to take different paths through a single iteration step by allowing multiple functions from a single state. As agents are stored and processed according to their state it is important that function conditions split agents into two working lists to avoid divergence during function execution. For each agent function this is implemented by using a filter kernel which maps agents meeting the function condition to a working list. Both the sparse working list and list of agents remaining in the start state list are then compacted using the same prefix sum technique used for birth and death allocation. After the working agent list is processed by the agent function the agents are finally appended to the end state list.

In the case of force resolution standard agent conditions do not provide enough flexibility to ensure agents are able to reach a stable state. Force resolution requires a minimum of two agent functions. The first of these is required to output a positional message and 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. Previously, multiple force resolution steps have been used. however careful scrutiny has suggested that to ensure a stable condition has been met a large amount of resolution steps are required. Using multiple resolution steps also has the problem of introducing a large amount of code repetition as each resolution step requires a separate agent function.

Our force resolution technique uses a recursive method to perform any number of force resolution steps. As recursive behaviour can not be achieved through agent function dependencies and conditions, it is instead achieved by removing the constraint that a single simulation step represents a single fixed length of time. Technically this implies that each simulation step may follow either a regular path through each agent function or if the population is unresolved perform only a force resolution step. 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 8). In the case of inter cellular force resolution this global condition can simply be a check to ensure agents have moved 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, global conditions have an optional fixed number of times that the condition can produce the same global result. This effectively allows a large upper bound to be placed on the number of force resolution steps which can be performed between normal simulation iterations.

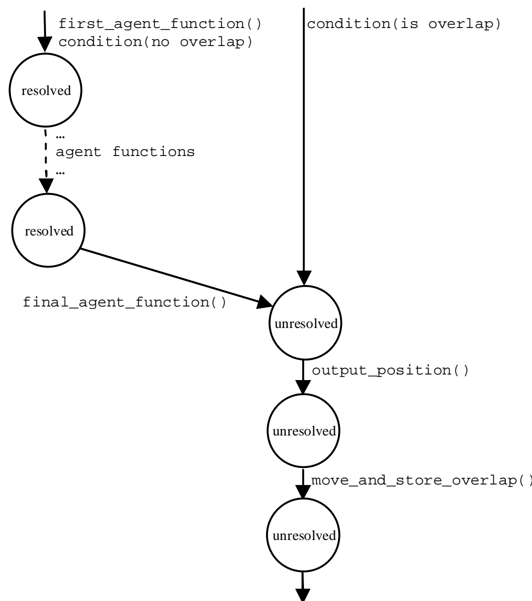

Figure 8 – Non linear simulation showing separate force resolution path

7. 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 effects the population sizes which 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).

As agent data is stored in CUDA global memory 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. The most simplistic method is to use a kernel (per agent type) to pass agents positional data into a Vertex Buffer Object (VBO) mapped into CUDA as a float array. This VBO can then be rendered with a single draw call rendering the positions as either OpenGL points or axis aligned point sprites which can give the appearance of more complex geometry. As an alternative to this we have used a more complex technique of mapping agent positions to texture data which in turn is used to displace sets of vertices which specify 3D geometry, see figure 9.

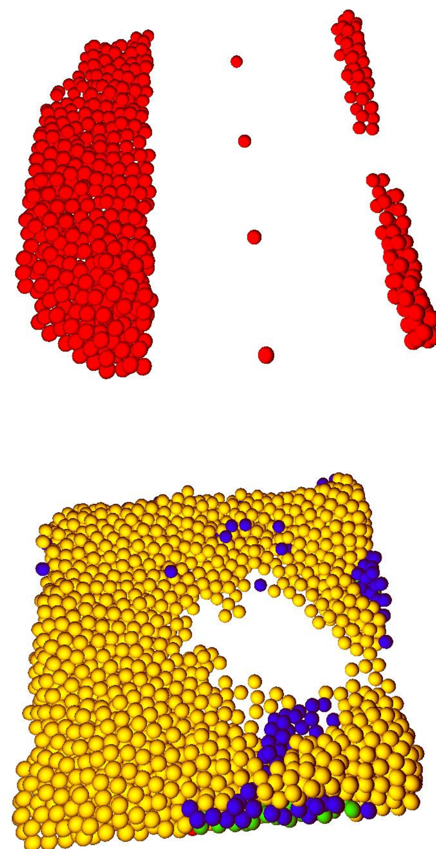

Figure 9 - Keratinocyte Model at iteration 0 and 1500 rendered as spheres. Red spheres represent stem cells, green represent TA cells, blue represent committed cells and yellow represents corneocyte cells.

This is achieved through using a CUDA kernel to pass agent data to a Texture Buffer Object (TBO) and rendering all vertices of a model with a vertex attribute which corresponds to the agent's position in the TBO texture data. The vertex shader uses this attribute to determine which agent position is used to offset the vertex, 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 are 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.

8. Performance Results

In order to evaluate our work we have implemented a modified version of the Keratinocyte colony model [19]. Whilst the functionality of this model remains the same as described by Sun, the model has been modified to limit agent functions to only a single message input and output (according to our XMML specification). All agent functions have also been rewritten to avoid any conditional dependency on message iteration or breaking from the message loops. Our modified version of the Keratinocyte model consists of a single agent, two message types, a single initialisation function and seven agent functions which include multiple birth and death allocations.

All results were obtained on a single 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 which halts GPU kernels exceeding five seconds in execution time. Original FLAME models have been updated to match our FLAME GPU function code as closely as possible and have been compiled using GCC with MingGW using full compiler optimisations.

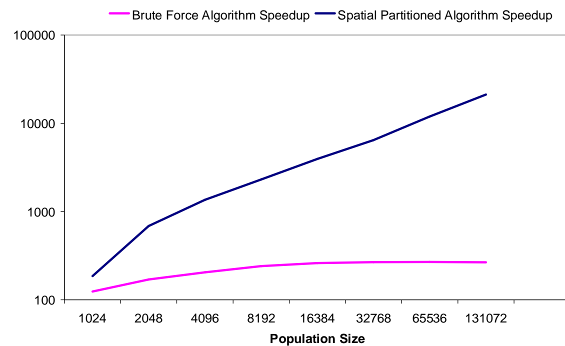

Figure 10 – Relative performance of the Keratinocyte model (logarithmic scale)

Figure 10 demonstrates the relative speedup of the Keratinocyte model achieved using the two message communication techniques. Each measurement consists of an initial configuration state consisting of stem cells 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 FLAMEs CPU iteration time. As the FLAME message processing for uses an $O(n^2)$ algorithm the result of our brute force algorithm gives the best 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 FLAME 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 131072 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 our spatially partitioned alternative took little over a second.

In order to evaluate a more realistic scenario than randomly distributed agents, we have measured the performance over an entire simulation using an initial configuration representing a scratch wound (300µm wide). Force resolution was done by testing agent's movement to ensure they had moved less than 0.25 µm with a maximum of 200 resolution steps. Figure 11 shows the performance of this simulation which took roughly 1500 iterations (not including force resolution iterations) to reach a stable state (shown in

Figure 9). 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 Frames per Second (FPS) inclusion of the force resolution steps ensures simulation remains interactive at over 60FPS 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.

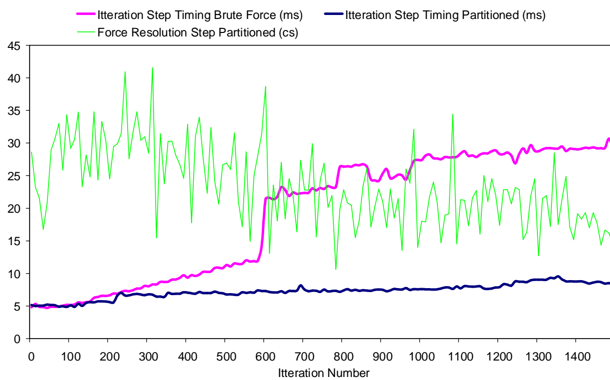

Figure 11 – Timing of simulation and force resolution steps during scratch would simulation

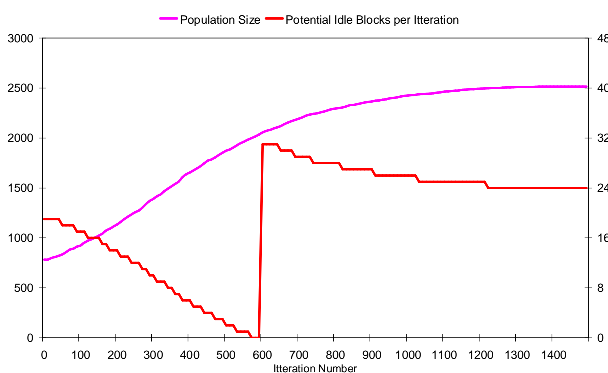

Figure 12 – Population size and potential idle blocks with respect to the iteration number

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 12). At iteration 600 the agent count 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 12, 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 our 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

9. Conclusion

We have presented FLAME GPU, an extension to the FLAME framework providing a flexible agent based architecture entirely on the GPU. Formal agent specification techniques have been used which are suitable for general continuous valued agent based systems. A technique for accurate force resolution has been presented which is suitable for resolving forces in cellular tissue models. This has been demonstrated through the implementation of a Keratinocyte colony model which has benefit from a direct 250x speedup in contrast with FLAME and an exponential speedup when more efficient spatially partitioned message communication is exploited.

In contrast with similar work [5] we have described the implementation of a flexible architecture that addresses both reusability as well as performance. As the trend towards faster GPU hardware continues we expect the performance of our framework to continue to improve. GT200 hardware already boasts almost twice the GFLOP/s performance over the hardware used within our evaluation. In contrast with PC grids this allows a single hardware upgrade to offer an enormous effect on simulation performance for a relatively small cost.

The reduction in simulation time will allow us to extend the variety and complexity of models in the future. With respect to cellular tissue modelling this

will allow larger simulations as well as increasing the complexity of physical cell representation. In addition to this it is expected that the public release of this framework will result in the implementation and evaluation of more advanced models which will provide feedback for further optimisations.

10. References

[1] Anderson, J., Lorenz, C. and Travesset, A. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008) 5342-5359

[2] Barnard, J., Whitworth, J., and Woodeard, M. 1996. Communicating X-Machines. Journal of Information and Software Technology, Vol 38. no. 6

[3] Chiara, R., Erra, U., Scarano, V. and Tatafiore, M.2004. Massive Simulation using GPU of a distributed behavioural model of a flock with obstacle avoidance. In Proceedings of 9th International Fall Workshop Vision, Modelling, And Visualization

[4] Coakley, S., Smallwood, R., and Holcombe, M. 2006. Using {X}-Machines as a Formal Basis for Describing Agents in Agent-Based Modelling, Proceedings of the 2006 Spring Simulation Multiconference, April 2006, pages 33-40.

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

[6] Eilenberg, S. 1974. Automata, Languages and Machines. volume A. Academic Press.

[7] Green, S. 2007. CUDA Particles, NVIDIA Whitepaper, November 2007.

[8] Harris, M., Coobe, G., Scheuermann, T., and Lastra, A. 2002. Physically-Based Visual Simulation on Graphics Hardware. In Procedings SIGGRAPH 2002 / Eurographics Workshop on Graphics Hardware 2002.

[9] Harris, M., Sengupta, S., and Owens, J. 2007. Parallel Prefix Sum (Scan) with CUDA. GPU Gems 3. Chapter 39.

[10] Howes, L. 2007. Loading Structured Data Efficiently With CUDA. NVIDIA Technical Report. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/vectorLoads/doc/vectorLoads.pdf

[11] Kipfer, P., Segal, M., and Westermann, R. 2004. UberFlow: a GPU-based particle engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Grenoble, France, August 29 – 30

[12] NVIDIA Corporation. 2007. CUDA Programming Guide Version 2.0, http://www.nvidia.com/object/cuda_develop.html

[13] Nyland, L., Harris, M., and Prins, Jan. 2007. Fast N-Body Simulation with CUDA, GPU Gems 3, Addison Wesley Professional, Chapter 31

[14] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., and Purcell, T. 2005. A Survey of General-Purpose Computation on Graphics Hardware. In Proceedings of Eurographics 2005, State of the Art Reports, pages 21-51

[15] Richmond, P., Coakley, S. And Romano, D. 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, in press.

[16] Richmond, P., and Romano, D. 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. Kos Island, Greece. June 2008.

[17] Rouff, C., Hinchey, M., Truszkowski, W., and Rash, J. 2005. Verifying large number of cooperating adaptive agents. 11th International Conference on Parallel and Distributed Systems. June 2005.

[18] Stone, J., Phillips, J., Freddolino, P., Hardy, D., Trabuco, L. and Schulten. K. 2007. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*. 28:2618-2640

[19] Sun, T., McMinn, P., Coakley, S., Holcombe, M., Smallwood, R., and MacNeil, S. 2006. Development and validation of an agent-based computational model of normal human keratinocytes organisation in vitro. European Cells and Materials, Volume 11, Number 3. Pages 28

[20] Van der Vlist, E. 2002. XML Schema. O'Riley Media ISBN:0-596-00252-1. pages 197-223

[21] Van Meel, J., Arnold, A., Frenkel, D., Zwart, S. and Belleman, R. 2008. Havesting graphics power for MD simulations. Molecular Simulation, Volume 34, Issue 3, p259-266

[22] Walker, D., Hill, G., Wood, S., Smallwood, R. and Southgate, J. 2004. Agent-based computational modelling of wounded epithelial cell monolayers. IEEE Trans Nanobioscience 3:153-163