CHAPTER

Template-Driven Agent-Based Modeling and Simulation with CUDA

21

Paul Richmond, Daniela Romano

This chapter describes a number of key techniques that are used to implement a flexible agent-based modeling (ABM) framework entirely on the GPU in CUDA. Performance rates equaling or bettering that of high-performance computing (HPC) clusters can easily be achieved, with obvious cost-to-performance benefits. Massive population sizes can be simulated, far exceeding those that can be computed (in reasonable time constraints) within traditional ABM toolkits. The use of data parallel methods ensures that the techniques used within this chapter are applicable to emerging multicore and data parallel architectures that will continue to increase their level of parallelism to improve performance. The concept of a flexible architecture is built around the use of a neutral modeling language (XML) for agents. The technique of template-driven dynamic code generation specifically using XML template processing is also general enough to be effective in other domains seeking to solve the issue of portability and abstraction of modeling logic from simulation code.

21.1 INTRODUCTION, PROBLEM STATEMENT, AND CONTEXT

Agent-based modeling is a technique for computational simulation of complex interacting systems through the specification of the behavior of a number of autonomous individuals acting simultaneously. The focus on individuals is considerably more computationally demanding than top-down system-level simulation, but provides a natural and flexible environment for studying systems demonstrating emergent behavior. Despite the potential for parallelism, traditionally frameworks for agent-based modeling are often based on highly serialized agent simulation algorithms operating in a discrete space environment. Such an approach has serious implications, placing stringent limitations on both the scale of models and the speed at which they may be simulated and analyzed. Serial simulation frameworks are also unable to exploit architectures such as the GPU that are shown within this article to demonstrate enormous performance potential for the agent modeling field.

Previous state-of-the-art work that demonstrates agent-based simulation on the GPU [4, 5] is also either very task specific or limited to only discrete spaced environments and is therefore unsuitable for a wider range of agent-based simulation. Little focus has previously been given to general techniques for agent modeling on the GPU, particularly those that address the issue of allowing heterogeneous agents without introducing large amounts of divergent code execution. This chapter presents a summary of techniques used to implement the Flexible Large-Scale Modeling Environment (FLAME) framework for GPU simulation that is both extendible and suitable for a wide range of agent simulation examples.

GPU Computing Gems

21.1.1 Core Method

This chapter describes a technique for using template-generated CUDA code for simulation of large-scale agent-based models on the GPU using CUDA. More specifically, it describes the FLAME process of translating formally specified XML descriptions of agent systems into CUDA code through the use of templates. The use of templates within work is essential because it allows the specification of agent systems to be abstracted from the simulation code and described using a common and portable specification format. Likewise, the use of multiple template sets also offers the potential to target multiple architectures (such as more traditional high-performance processing clusters or grids) that are used for comparison with our GPU implementation. With respect to agent modeling in CUDA, common parallel routines are described that achieve essential agent-based functionality, including agent birth and death processing as well as various interagent communication patterns. Most importantly, the issue of agent heterogeneity is addressed through the use of state machine-based agent representation. This representation allows agents to be separated into associated state lists that are processed in batches to allow very diverse agents while avoiding large divergence in parallel code kernels.

21.1.2 Algorithms and Implementations

A key aspect of implementing thread-level agents on the GPU in CUDA is to ensure heterogeneous agents can be simulated without introducing large amounts of thread-level divergence. Such divergence occurs during simulation when not all threads (or agents) within a processing group of 32 threads (a warp) fail to follow the same instruction path.

21.1.3 State-Based Agent Simulation in CUDA

Part of the solution to this is the expression of agents using a technique that allows a state-based grouping of agents that are likely to perform similar behaviors. To achieve this, FLAME GPU uses an abstract model of an agent that is based on finite state machines. More specifically a formal definition known as the X-Machine can be used to represent an agent that adds to the finite state machine definition a persistent internal memory. The addition of memory allows a state transition to include the mapping of a memory set (M) to a new set (M′) that is implemented in CUDA using a kernel. The separation of agents into distinct states reduces divergence and ensures agents can be processed efficiently. In addition to this, storage and processing of agents within state lists ensures that memory accesses from consecutive threads are linear and are therefore coalesced. Using this formal-based definition a simulation can be derived from a list of ordered transition functions that occur during each simulation step.

Because the movement of agents from one state to another occurs as a direct result of performing a state transition (or agent function), it is important to allow a filtering mechanism that may restrict the execution of a transition depending on some condition of the simulation or the agent's memory. This ensures that agents are free to follow heterogeneous paths between states independent of their initial state and neighboring agents. Within the FLAME GPU this is implemented using a *function condition kernel* (Figure 21.1) for each state transition. This kernel outputs a flag for each agent thread indicative of the result of performing the filtering process for the current list of agents. A parallel prefix sum algorithm [6] is then used to produce two new compact lists that represent a working list (agents that met the transition function condition) and a list with remaining agents that replaces the original state list. The working list is then able to perform the agent transition function (mapping of memory from M to M′) before the agents are finally appended to the state transitions next state list. The prefix sum technique is not limited to filtering of agents during transition functions and is also employed through

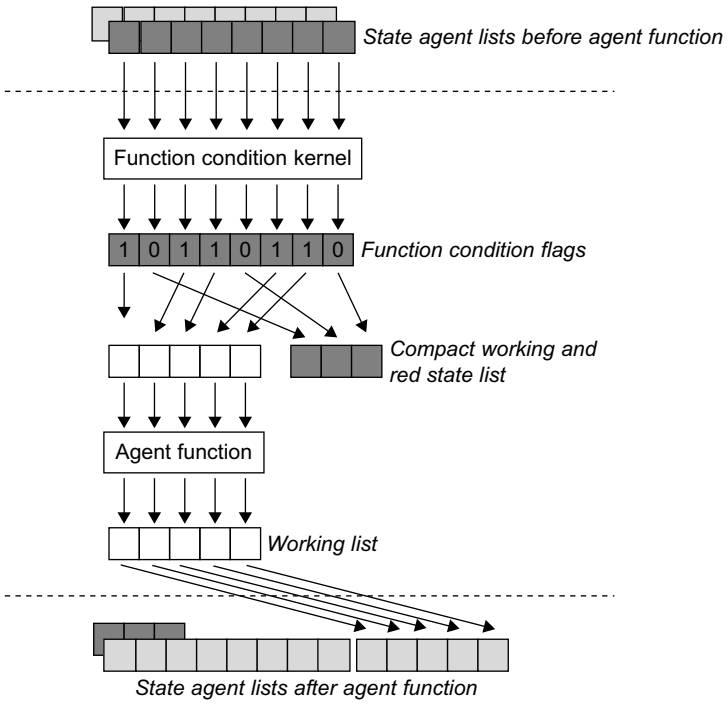

FIGURE 21.1

Use of function condition filters to compact an agent state list into a working list for transition function processing. Agents in the start state that meet the function condition are finally appended to the end state.

the simulation process to compact lists containing dead agents and on sparse lists containing new agents (agent births).

21.1.4 XML-Driven CUDA Code Generation

Representation of agents in a flexible and portal format is essential in ensuring agent models are suitable for simulation on a wide range of hardware architectures (including the GPU, HPC grids, and more traditional CPU architectures). XML-based technologies form a large part of providing this functionality and are used extensively with the FLAME GPU to both specify an agent model and generate agent code. Model specification is achieved through XML representation of agent structure, including agent memory variables, states, and transition functions. The syntax of this specification is guarded through a number of extendible XML schemas that based on polymorphic extension, allow a base specification to be extended with a number of GPU-specific elements. The base specification in this case forms the structure of a basic X-Machine agent model compatible with HPC and CPU versions of the FLAME framework. The GPU schema extensions add some specific information, such as the maximum population size (used to preallocate GPU memory), as well as some optional elements that indicate additional model details used to select between a number of lower-level algorithms at the code-generation stage. These details include a distinction between discrete and continuous agents and the type of communication that may exist between agents.

Given an XML-based agent model definition, template-driven code generation has been achived through Extensible Stylesheet Transformations (XSLT). XSLT is a flexible functional language based on XML and is most commonly used in the translation of XML documents into other HTML or other XML document formats on the Web. Despite this there is no limitation on the type of file that may be generated from an XSLT template, and it is hence suitable for the generation of source code (Figure 21.2). Translation of XSLT documents is possible through any compliant processor (such as Saxon, Xalan, Visual Studio, and even many common Web browsers) that will also validate the template itself using a W3C specified schema. The XSLT code sample in Figure 21.3 demonstrates how the iterative `for-each` control is used to generate a C structure for each agent and an agent list used for state-based global storage within the XML model document. The `select` attribute uses an XPath expression to match nodes in the document. Likewise, XPath expressions are used with XSLT to match nodes within the value-of attributes and any other XSLT elements that require XML document querying. Within FLAME GPU any global memory is stored using the Structure of Array (SoA) format rather than the Array of Structure format to ensure all memory access is coalesced. Data can then be translated into more accessible and logical structure format within registers or shared memory (so long as appropriate padding is used to avoid bank conflicts) without any performance penalties.

21.1.5 Agent Communication and Transition Function Behavior Scripting

While agents perform independent actions, interagent communication is essential in the emergence of group behavior. Communication between agents using the X-Machine notation within FLAME GPU

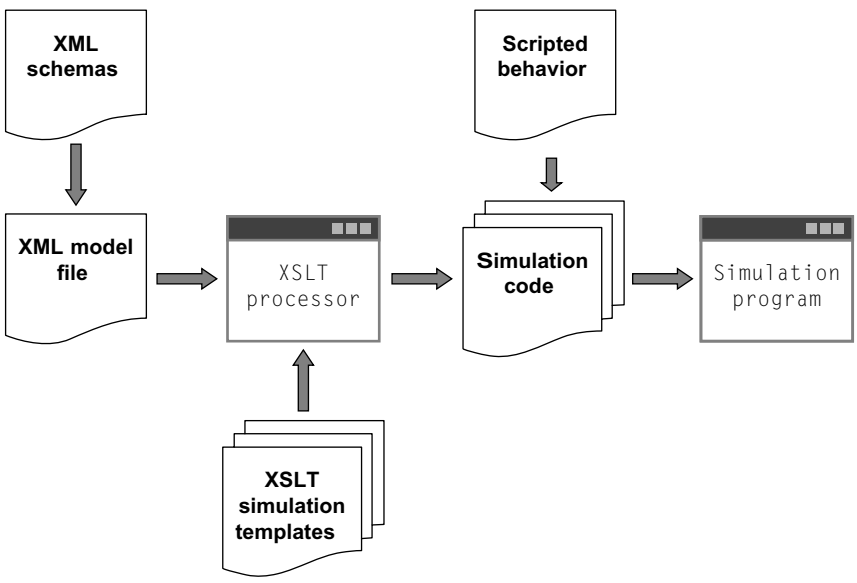

FIGURE 21.2

The FLAME GPU modeling and code-generation process.

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

//agent list structure (AoS)
<xsl:for-each select="gpu:xmodel/xmml:xagents/gpu:xagent"> struct
xagent_memory_<xsl:value-of select="xmml:name"/>_list {
    //Holds agents position in the 1D agent list
    int _position [<xsl:value-of select="gpu:maxPopulationSize"/>];
    <xsl:for-each select="xmml:memory/gpu:variable">
     <xsl:value-of select="xmml:type"/><xsl:text></xsl:text>
     <xsl:value-of select="xmml:name"/>[<xsl:value-of
                              select=" gpu:maxPopulationSize"/>];
     </xsl:for-each>
};
</xsl:for-each>
```

FIGURE 21.3

An example of XSLT template code used to generate an structure representing an agent's memory and a structure of arrays representing a list of agent memory data.

is introduced through the use of messages stored in global message lists. Agent transition functions are able to both output and input messages that in the case of the latter, requires a message iteration loop for the agent to process message information. The use of only indirect message-based communication between agents ensures that the scheduling of agents can in no way introduce bias or any other simulation artifacts based on the order of agent updates. Figures 21.4 and 21.5 show examples of two agent transition functions that demonstrate message output and a message input loop, respectively. The ordering of transition functions is used to ensure global synchronization of messages between consecutive transition functions, and as a result, a single-transition function can never perform both input and output of the same message type. Each agent transition function performs the memory mapping of M to M′ by updating the agent memory structure argument directly. Agent deaths can be signaled by the return value flag by returning any value other than 0 (the flag can then be used to compact the working list of agents).

Integration of the transition functions within automatically generated simulation code is made possible by wrapping the transition functions with global kernels (generated through the XSLT templates) that are responsible for loading and storing agent data from the SoA format into registers. The custom message functions (Figures 21.4 and 21.5) are also template generated, depending on the definition of a message within the XML model file. The custom message functions hide the same data-loading techniques as used for agent storage with each message having a structure and SoA definition consisting of a number of memory variables.

```
_FLAME_GPU_FUNC_
int outputdata(xagent_memory_Circle* memory,
              message_location_list* location_messages)
{
      /* Output a location message */
      add_location_message(location_messages,
                           xmemory->x,
                           xmemory->y,
                           xmemory->z);
      return 0;
}
```

FIGURE 21.4

An example of a scripted agent transition function demonstrating message output.

```
_FLAME_GPU_FUNC_
int inputdata(xagent_memory_Circle* memory,
              message_location_list* location_messages)
{
     /* Get the first message */
     message_location* message;
     message = get_first_location_message(location_messages);

     while(message)
     {
           /* Process the message */
           ...

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

    /* Update agent memory */
    memory->x += 1;
    ...

    /* Agent is not flagged to die */
    return 0;
}
```

FIGURE 21.5

An example of a scripted agent transition function showing message iteration through the template-generated custom message functions.

The use of message functions to hide the iteration of messages is particularly advantageous because it abstracts the underlying algorithms from the behavioral agent scripting. This allows the same functional syntax to be used for a number of different communication techniques between agents. The most general of these techniques is that of brute-force communication where an agent will read every single message of a particular type. Technically, this is implemented through the use of tiled batching

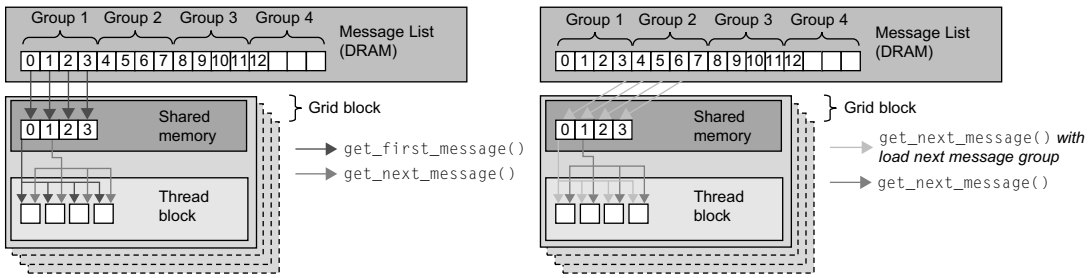

FIGURE 21.6

Brute-force message group loading when requesting the first and next message (left). Brute-force message group loading when requesting the next message from a new message group (right).

of messages into shared memory [8]. The message iteration functions are responsible for performing this tiled loading into shared memory that occurs at the beginning of the iteration loop and after each message from within a group has been serially accessed. Figure 21.6 demonstrates how this is performed and shows the access pattern from shared memory for the iteration functions, particularly when iteration through shared memory has been exhausted.

In addition to brute-force message communication, the FLAME GPU templates provide both a spatially partitioned message iteration technique and a discrete message partitioning technique. In the case of spatially partitioned messages, a 2-D or 3-D regular grid is used to partition the agent/message environment, depending on a prespecified message interaction range (the range in which agents read the message if it is used as a input during a transition function). When we use a parallel radix sort and texture cached lookups [7], the message iteration loop can ensure far higher performance for limited range interactions. Within discrete message communication, messages can be output only by discrete spaced agents (cellular automaton) with the message iteration functions operating by cycling through a fixed range in a discrete message grid. For discrete agents, the message iteration through the discrete grid is accelerated by loading a single large message block into shared memory. Interaction between continuous and discrete agents is possible by continuous agents using a texture cache implementation of discrete message iteration.

21.1.6 Real-Time Instanced Agent Visualization

In addition to improving the performance of simulation, simulation on the GPU provides the obvious benefit of maintaining agent information directly where it is required for visualization. The first step to performing visualization is making the agent data available to the rendering pipeline from CUDA. This can be achieved through the use of OpenGL Buffer Objects, which are able to share the same memory space as CUDA global memory. Simple agent visualization can be accomplished with a single draw call, rendering the positions as either OpenGL points or axis-aligned point sprites (which give the appearance of more complex geometry). By default the FLAME GPU visualization templates provide rendering of agents using more complex geometry. This requires an instancing-based technique that displaces sets of vertices for each agent rather than a single-vertex position. This is implemented by using a CUDA kernel to pass positional agent data to a texture buffer object (TBO) and then rendering instances of geometry with a vertex attribute (or index), which corresponds to the agent's position in the TBO texture data. The vertex shader uses this attribute to look up the agent position and applies the

```
uniform samplerBuffer displacementMap;
attribute in float index;
varying vec3 normal, lightDir;
void main()
{
      vec4 position = gl_Vertex;
      vec4 displacement = texelFetchBuffer(displacementMap, (int)index);

      //offset instanced gemotery by agent position
      displacement.w = 1.0;
      position += displacement;
      gl_Position = gl_ModelViewProjectionMatrix * position;

      //lighting
      vec3 mvPosition = vec3(gl_ModelViewMatrix * position);
      lightDir = vec3(gl_LightSource[0].position.xyz - mvPosition);

      //normal
      normal = gl_NormalMatrix * gl_Normal;
}
```

FIGURE 21.7

GLSL vertex program for instanced agent rendering used to displace agents depending on their location.

vertex transformations, with a further fragment shader providing per-pixel lighting. To minimize data transfer to the GPU model, data can be stored within a VBO; instances of the same model can then be created with a single draw call minimizing GPU data transfer. Figure 21.7 shows a GLSL (Open GL Shading Language) shader program that is used to displace instanced agents and perform the model view projection.

More advanced visualization is available within the FLAME GPU by extending the basic visualization source code produced by the templates (Figure 21.8). Levels of detail, where the fidelity of agent representation is varied with respect to the viewer position, can be integrated by considering the distance to the camera within an agent transition function. This distance can then be used to store an appropriate detail level within a single-agent memory variable with the total for the population of each level being calculated through a number of prefix sum operations. The FLAME GPU simulation templates generate a per-agent sort function that can be then used to sort the population by detail level allowing host code to draw the correct number of instances for each level sequentially. Simple key-frame animation of agents such as fish (Figure 21.8) can be further added by considering a number of model representations for each detail level and updating the vertex shaders to perform interpolation between these key frames depending on some agent memory variable such as velocity.

21.2 FINAL EVALUATION AND VALIDATION OF RESULTS

Performance and quality of the FLAME GPU framework can be measured through reduction in modeling/analysis time and directly through quantitative simulation performance and cost performance (when compared with supercomputing alternatives). Measuring the reduction in modeling time as a result of

FIGURE 21.8

Agent-based simulation of flocking within the FLAME GPU showing 65,000 low-resolution agents (left) and 5000 dynamically detailed (level of detail) agents (right).

using the FLAME GPU framework is difficult to quantify; however, the volume of modeling code provides a good indication of what is required to produce a complete simulation. For the case of a simple Boids [9] flocking model, the model can be described within 91 elements of XML code (roughly 100 lines, depending on formatting) and fewer than 100 lines of agent function script. The script consists of two agent state transition functions, of which the first outputs location information and a second cycles the location information to influence the Boids velocity and movement. After the full simulation code is generated from the simulation templates, the resulting source code (including real-time visualization) consists of roughly 2500 lines of C and CUDA code (2000 lines without visualization), of which about 660 lines consists of CUDA kernels. The ability to generate complete GPU-enabled agent simulations with such a small amount of time investment is massively advantageous and allows models to be developed without prior knowledge of the GPU architecture or CUDA. Likewise, the use of templates to produce CUDA-enabled code removes vast amounts of code repetition for common agent-based functionality (such as communication and birth and death allocation) that occurs when writing agent simulations from scratch.

Simulation performance of the FLAME GPU can be compared directly with similar FLAME implementations that exist for both a single CPU processor and a grid-based supercomputing architecture. For the purposes of benchmarking, a simple force resolution model (referred to as the Circles model) has been benchmarked (without visualization) on a single GPU core of an NVIDIA 9800GX2 and also on an NVIDIA GTX480. For the latter, full double precision was used for agent memory variables and arithmetic operations, whereas the former utilizes single precision floats throughout. The model consists of only a single agent and message type with three agent functions, which output and input a message containing location information with a final function that moves the agent according to inter-agent repulsive forces. Figure 21.9 shows a comparison of the FLAME GPU and FLAME for the CPU (AMD Athlon 2.51 GHz dual-core processor with 3 GB of RAM). Relative speedup is the percentage of performance improvement over the CPU timings calculated by considering the iteration time of a single time step of the Circles model at various population sizes.

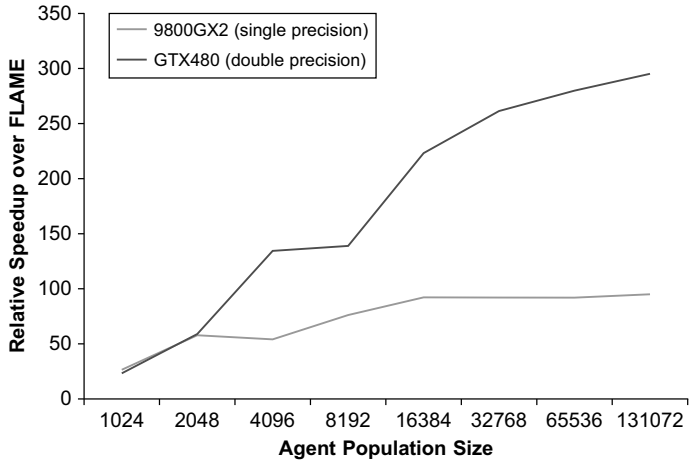

FIGURE 21.9

Relative speedup of the Circles benchmarking model for brute-force message iteration in both single and full precision on a NVIDIA 9800GX2 and a NVIDIA GTX480, respectively.

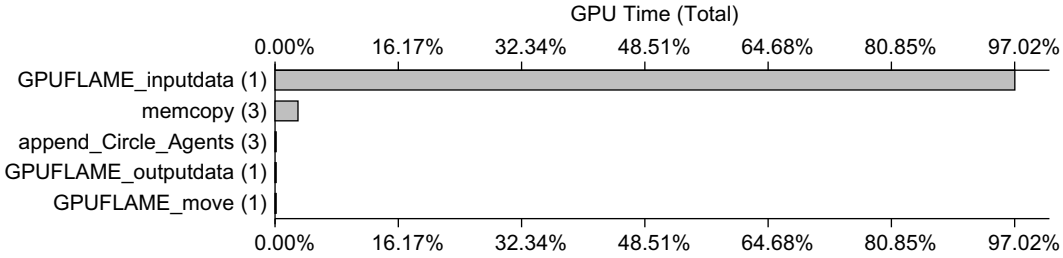

FIGURE 21.10

Breakdown of simulation time during a single simulation step of the Circles benchmarking model.

The initial fluctuation of the simulation performance in Figure 21.9 can be attributed to the fact that, at (relatively) low agent counts, the multiprocessors are underutilized during the most computationally expensive stage of the simulation (the inputdata function, shown in Figure 21.10), resulting in unpredictable amounts of idle time and global memory access latency coverage. In fact, for the 9800GX2, population sizes up to and including 4096 (and a thread block size of 128), the maximum number of thread blocks per multiprocessor is limited by the register count (of 8192 for compute capability 1.1 cards), to 2. This suggests that 4096 agents (or 32 blocks of 128 agent threads) are the minimum number required to fill all 16 of the multiprocessors. In the case of the GTX480 the number of thread blocks per multiprocessor is limited by shared memory to 6, suggesting that 11,520 agents (90 blocks of 128 agent threads) is the minimum number required to fill the 15 multiprocessors.

Performance of the FLAME GPU in contrast with the performance of the Circles model on distributed HPC architectures is considered by comparing FLAME GPU's spatially partitioned message functionality with a technique that partitions agents across multiple nodes on an HPC grid. Within the

Table 21.1 Simulation performance of spatially partitioned message iteration for the Circles benchmarking model.

Population Size	9800GX2 – Single Precision (ms)	GTX480 – Double Precision (ms)
1024	0.94	1.59
4096	1.24	2.75
16,384	2.45	2.12
65,536	9.09	5.19
262,144	33.74	10.53
1,048,576	136.28	34.41

HPC implementation each grid node processes a subset of agents and corresponding communication messages. Messages spanning multiple nodes are communicated through the use of a message board library (which is optimized to reduce transfer costs by communicating messages in batches rather than independently). Although the Circles model has been benchmarked on multiple architectures, both the SCARF and HAPU architectures show the best performance and are able to perform a single simulation step consisting of 1 million agents in double precision in just over 6 seconds, using a total of 100 processing cores.

Table 21.1 shows the performance timing of a single iteration of the Circles model using a thread block size of 32. Although some larger thread block size results in higher occupancy, experimentation shows that a smaller block size results in a higher-texture cache hit rate (and faster overall performance) during message reading owing to an increased locality between threads. Future work will improve upon this further through the use of space-filling curves, such as that demonstrated by Anderson *et al.* [2] which have been shown to dramatically increase the cache hit rate in molecular dynamics modeling. In summary, Table 21.1 demonstrates that 1 million agents can be simulated in single precision using a single GPU in less than 0.14 seconds. If we use full double-precision support within the Fermi architecture of the GTX480, it is possible to simulate a million agents in little over 0.03 seconds, a speedup of 200 times the SCARF and HAPU architectures. From this it is clear that the GPU is easily able to compete with and outperform supercomputing solutions for large population agent modeling using the FLAME GPU framework.

In the case of more complex continuous spaced agent models, the FLAME GPU has also shown massive potential, especially in the case of cell-based tissue modeling [10] where simulation performance in contrast with previous MATLAB models has improved from hours to seconds in the most extreme cases. Such performance improvements allow real-time visualization (which using instancing has a minimal performance effect) where it was previously not possible. This in turn allows a mechanism for fast parameter exploration and immediate face validation with the possibility of real-time steering (manipulation) of models by extending the dynamically produced simulation code.

21.3 CONCLUSIONS, BENEFITS AND LIMITATIONS, AND FUTURE WORK

This chapter has presented the FLAME GPU for large-scale agent-based simulation on GPU hardware using CUDA. Benchmarking has demonstrated a considerable performance advantage to using the

GPU for agent-based simulation. Future work will focus on addressing some of the current limitations of the framework. Specifically, the framework will be extended to support multi-GPU configurations. This will be possible by using concepts from the HPC FLAME code to perform distribution of agents and MPI-based communication between hardware devices. Recent work in performance optimization across multiple nodes connected by a network [1, 3] highlights performance potential for discrete/grid-based agent systems. The suitability of this toward more general state-based agent execution will have to be considered.

The use of multiple program multiple data kernel execution within the NVIDIA Fermi architecture is particularly exciting and lends itself well to the state-based storage and processing of agents within the FLAME GPU. The effect of this on performance will almost certainly be explored in future work. Currently, the framework's reliance on CUDA allows it to be used only on NVIDIA hardware. The introduction of OpenCL may offer a solution toward increased portability across heterogeneous core platforms and will be explored in the future. This will not require any major change in the framework and will be possible by simply specifying a new set of compatible templates.

References

[1] B.G. Aaby, K.S. Perumalla, S.K. Seal, Efficient simulation of agent-based models on multi-GPU and multi-core clusters, in: SIMUTools '10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, Belgium, Torremolinos, Malaga, Spain, 2010, pp. 1–10. Available from: http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8822.

[2] J. Anderson, D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (2008) 5342–5359.

[3] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, et al., Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, TX, IEEE Press, Piscataway, NJ, 2008, pp. 1–12.

[4] R.M. D'Souza, M. Lysenko, K. Rahmani, SugarScape on steroids: simulating over a million agents at interactive rates, in: Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence, Chicago, IL, 2007. Available from: http://www.me.mtu.edu/~rmdsouza/Papers/2007/SugarScape_GPU.pdf

[5] U. Erra, B. Frola, V. Scarano, I. Couzin, An efficient GPU implementation for large scale individual-based simulation of collective behavior, in: HIBI '09: Proceedings of the 2009 International Workshop on High Performance Computational Systems Biology, IEEE Computer Society, Washington, DC, Trento, Italy, 2009, pp. 51–58. Available from: http://dx.doi.org/10.1109/HiBi.2009.11

[6] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007, Chapter 39, pp. 851–876.

[7] S. Le Grand, Broad-phase collision detection with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007, Chapter 39, pp. 677–695.

[8] L. Nyland, M. Harris, J. Prins, Fast N-Body simulation with CUDA, in: H. Nguyen (Ed.), GPU Gems 3, Addison Wesley, 2007, Chapter 39, pp. 677–695.

[9] C. Reynolds, Flocks, herds and schools: a distributed behavioral model, in: M.C. Stone (Ed.), Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87, ACM, New York, 1987, pp. 25–34.

[10] P. Richmond, D. Walker, S. Coakley, D. Romano, High performance cellular level agent-based simulation with FLAME for the GPU, Brief. Bioinform. 11 (3) (2010) 334–347.