2017 International Conference on High Performance Computing & Simulation

FLAME GPU: Complex System Simulation Framework

Paul Richmond
Department of Computing Science
211 Portobello, University of Sheffield
Sheffield, S1 4DP, UK
Email: p.richmond@sheffield.ac.uk

Mozhgan K. Chimeh
Department of Computing Science
211 Portobello, University of Sheffield
Sheffield, S1 4DP, UK
Email: m.kabiri-chimeh@sheffield.ac.uk

***Abstract*—FLAME GPU is an agent based simulation framework that utilises the parallel architecture of Graphic Processing Unit (GPU) to enable real time model interaction and visualisation. In this paper, we provide an overview of the features of FLAME GPU and demonstrate its efficiency as a parallel agent based simulation platform. FLAME GPU abstracts the complexity of the GPU architecture from the users by offering a high level modelling syntax based on a formal state machine representation. A flocking model is presented showing how a simple multi-agent system is modelled.**

***Index Terms*—GPU; ABM; Complex System Simulation; Agent Based Modelling**

I. Introduction

Agent Based Modelling(ABM) is a technique for computational simulation of complex interacting systems, through the specification of the behaviour of a number of autonomous individuals acting simultaneously. ABM is a bottom-up approach, in contrast with a top-down approach of modelling the behaviour of the whole system through sets of dynamic mathematical equations. The modelling and simulation of individuals is considerably more computationally demanding but provides a natural and flexible approach for studying systems demonstrating emergent behaviour. Traditionally, frameworks for ABM fail to exploit parallelism in their design and often utilise highly serialised algorithms for manipulating mobile discrete agents. Serialised algorithms have serious performance implications, placing stringent limitations on both the scale of models and the speed at which they may be simulated.

FLAME GPU (a template driven framework for ABM on parallel architecture) is an extended version of the FLAME (Flexible Large-scale Agent-based Modelling Environment) framework [1]. It is a mature and stable agent-based modelling simulation platform that enables modellers from various disciplines like economics, biology and social sciences to easily write agent-based models, specifically for Graphics Processing Units (GPUs). The purpose of the FLAME GPU framework is to address the limitations of previous agent modelling software by targeting the high performance GPU architecture. The framework is designed with parallelism in mind and as such allows agent models to scale to massive sizes, ensuring simulations execute within reasonable time constraints. In addition to this, real-time visualisation is easily achievable as simulation data is held entirely within GPU memory where it can be rendered directly and efficiently.

FLAME GPU has been used in various domains within science and engineering where models have been validated against other modelling software. For example, Haywood et el. [2] developed a microscopic road network simulator using FLAME GPU to evaluate the impact of many core architectures on road network micro-simulation. It was cross validated against the reference Aimsun traffic modelling software [1] for a range of validation models. The GPU accelerated simulation achieved speedup of 27.1x for a model containing 394k detectors and 250k vehicles.

Chisholm, et al. [3] reports the performance results collected from implementations of a benchmark model validated against three different simulation frameworks including FLAME GPU. For large populations, FLAME GPU outperformed both Repast and MASON frameworks.

The are various other applications using the FLAME GPU framework. It has been used in simulating a large scale pedestrian simulation [4], [5], a Keratinocyte (cell) model of the in-vitro behaviour of skin epithelial cells [6], [7], [8], [9] and the immune system [10], [11].

II. High Level Overview of FLAME GPU

Technically the FLAME GPU[2] framework is not a simulator it is instead a template based simulation environment which maps formal descriptions of agents into simulation code. The formal representation of an agent is based on the concept of a communicating X-Machine (which is an extension to the Finite State Machine which includes memory) [12]. Whilst the X-Machine has a very formal definition, X-Machine agents can be summarised as state machines which are able to communicate via messages, stored in a globally accessible message lists.

Agent behaviour is exposed as a set of state transition functions which move agents from one state to another. Upon changing state, agents update their internal memory through the influence of messages which may be either used as input, by iterating message lists, or as output, where information may

[1]https://www.aimsun.com/
[2]https://github.com/FLAMEGPU/FLAMEGPU

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.12

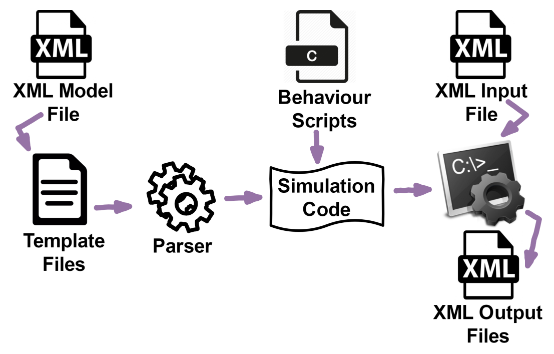

Fig. 1. The FLAME GPU Modelling Process. An XSLT template processor translates a user defined XMML model into simulation code to be linked with the behavioural function scripts to produce a custom simulation executable.

be passed to the message lists for other agents to read. FLAME GPU uses agent function scripting for this purpose. Agent functions scripts are defined in a number of agent function files.

Complete models are specified using the X-Machine Mark-up Language (XMML) which is XML syntax with corresponding Schemas governing the syntax. A typically XMML model file consists of a definition of a number of X-Machine agents (including state and memory information as well as a set of agent transition functions), a number of message types (each of which is stored in the globally accessible message lists during simulation) and a set of simulation layers which define the execution order of agent functions within a single simulation iteration.

Throughout a simulation agent data is persistent, however, message information is persistent only over the life-cycle of a single iteration. I.e. message lists are cleared at the end of each simulation loop. The overall simulation processes allows a mechanism for agents to iteratively interact in a way which emergent global group behaviour [13] can be demonstrated.

The process of generating a FLAME GPU simulation is described in Fig. 1. The use of XML schemas create polymorphic like extensions allows a base schema specification to be extended with a number of GPU specific elements. The base elements are common with the original FLAME framework providing a common syntax for model exchange. Given an XMML model definition, template driven code generation is achieved through Extensible Stylesheet Language Transformations (XSLT). XSLT is a flexible functional templating language based on XML (validated itself using a W3C specified Schema) and is suitable for the translation of XML documents into other document formats using a number of compliant processors. Within Windows environments the FLAME GPU SDK provides a lightweight .NET processor as part of Visual Studio custom build tools. Within Linux, FLAME GPU $make$ files are dependant on the installation of XSLTProc (packaged with all major Linux distributions). FLAME GPUs XSLT templates allow a dynamic simulation specific API to be generated which can be linked with the user provided agent function files to generate a simulation program. This dynamically generated API uses the CUDA programming interface to target GPU devices. Use of code generation in this way allows highly optimised code to be generated which is specific to the simulation avoiding code bloat and providing an obvious extension point for more advanced programmers to further develop the model or generated GPU code. Such extensions to the generated code are demonstrated through a number of *custom* visualisations for models such as pedestrian navigation provided within the SDK.

III. FLAME GPU Features and advantages

This section explores some of the FLAME GPU features and points out some of the existing techniques within FLAME GPU which directly address common difficulties in mapping multi-agent systems to GPU hardware.

A. Life and Death

The removal or creation of an agent will lead to non contiguous data layouts in memory. In other words, when agents change state, die or are born, data structures holding populations of agents become sparse with respect to the data stored within them. This results in non-coalesced memory access which is problematic for GPU performance.

In order to retain coalesced access, data structures need to avoid sparsity. This improves the overall use of GPU memory bandwidth as all GPU memory is moved in byte aligned cache lines. Contiguous data accesses ensure good utilisation of cache lines and therefore reduces the total number of memory cache line transactions which are required. FLAME GPU uses stream compaction, a parallel primitive (algorithm) provided by the Thrust [14] library to avoid sparse data. Stream compaction produces a smaller array of data containing indices of only the wanted/selected elements from a source array [15]. This stream compacted data can be used to write agents to new compressed unique locations in parallel without conflicted memory access writes.

B. Communication

When simulating a complex system on the GPU, absence of a global mechanism for GPU thread communication puts a constraint on the suitability of techniques for communication between all agents within a population. As such it is required to build data structures which can be used to ensure that data accesses can be made conflict free. I.e. avoiding race-conditions. FLAME GPU provides this functionality through the message abstraction which ensures that agents communication only indirectly. Message specialisations describe specific access patterns for reading messages from the global message lists which permits data structures to be built to improve the performance for the particular pattern. In designing data structures to permit fast message access, FLAME GPU relies on the use of a number of the GPU devices fast caches, namely shared memory and texture memory to exploit re-use of data and minimise data movements. The impact of which has considerable impact on overall execution time.

FLAME GPU currently supports the three message communication specialisations which are brute force, spatially distributed, and discrete (with respect to space) messages. With brute force messaging, each agent reads every available message. In spatially distributed messaging, agents within 2D or 3D continuous space, read messages within a fixed radius. Discrete messaging restricts messages to occupying a single unique space within a 2D discrete grid. This approach is suitable only for discrete cellular automaton agents to write messages (to avoid conflicts in writing) however non agents located in continuous space can read discrete messages conflict free. Combining continuous and discrete spaced agent is common in models such as pedestrian navigation where discrete agents communicate vector fields used by pedestrians to perform obstacle avoidance and global navigation [4], [16].

With respect to the optimisation of differing message access patterns, brute force messaging uses on-chip shared memory to load messages that are read/accessed by agents within a group of threads. Shared memory is much faster that global memory with about 100x lower latency. As it is allocated per thread block, groups of threads in a block can have access to the same shared memory loaded from global memory [17]. In spatially partitioned agent communication, FLAME GPU uses a fast counting sort to both re-order the agents and build a partition boundary matrix containing the indices of agents within fixed (message) radius sized discrete partitions representing the agents environment. During iteration of the spatially partitioned messages, the partition matrix is used to return only the agents within neighbouring discrete partitions. The messages are loaded via texture memory as the memory access pattern has excellent data locality and is hence ideally suited to this caching mechanism. Discrete messaging uses either shared memory or the texture cache to load a 2D grid of messages in discrete locations without the necessity of building a spatial data structure.

IV. An example: Implementing a model using FLAME GPU

The following worked example presents the modelling syntax required to implement a simple Boids model in FLAME GPU. We have chosen Boids for its simplicity. However, the general approach can be applied to more complex complex system models (examples of which are provided in the FLAME GPU SDK).

A. The Boids model

The Boids model was created by Craig Reynolds [18]. A visual representation of the Boids model appears in Fig. 2 which highlights agents demonstrating an emergent group flocking behaviour. Agents have a number of simple behaviour rules which include steering towards the centre of their perceived local group, avoiding collisions with other agents and matching the speed of neighbouring agents. Fig. 3 shows the state diagram for the FLAME GPU implementation of the Boids model which requires the agent functions (INPUT_DATA, MOVE, OUTPUT_DATA) describing the Boid

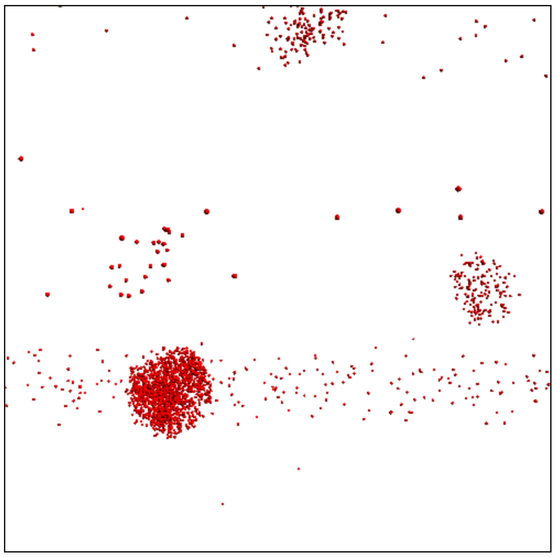
Fig. 2. Visual Representation of Flocking Boids Model.

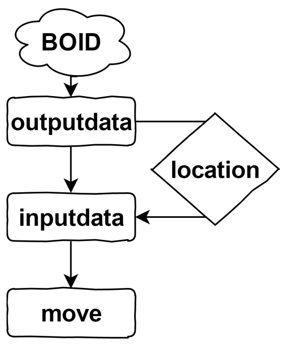

Fig. 3. Boids Model State Diagram showing the dependency relationship of functions required to implement the required flocking behaviour.

agent behaviours, the order of their execution, and the *location* message which permits indirect communication of information between the agents.

B. Model specification

In order to create the GPU implementation of the Boids model only two user specified files are required; the model description (e.g. `XMLModelFile.xml`) and the scripted (e.g. `function.c` behaviour functions file.

The model description file contains three major components which are *Agents*, *Messages*, and *Layers*. *Agents* are the description of what an agent is, *Messages* represent the information an agent will communicate to other agents and *Layers* are dependencies between functions which determine the order in which agents should perform behaviours. The description of an agent consists of *memory*, *states*, and *functions*. The following highlights the use of this syntax. For a full overview of the syntax readers are directed to the FLAME GPU user guide [19].

a) Memory: Memory describes the persistent variables which an agent contains over its life-cycle of the simulation. Variables are typically used to hold properties unique to the agent such as location, velocity, etc and are strictly typed using C basic data types. The Boids model has a variable for each component of the agents location (x, y, z) and velocity (fx, fy, fz) as well as a unique identifier (id) shown in Listing 1.

```
<memory>
  <gpu:variable>
    <type>int</type><name>id</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>x</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>y</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>z</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>fx</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>fy</name>
  </gpu:variable>
  <gpu:variable>
    <type>float</type><name>fz</name>
  </gpu:variable>
</memory>
```

Listing 1. XMML definition of an agents internal memory.

b) State: Agent states are used to differentiate between agents of the same type which may be in differing functional capacity of their life-cycle. For example, a biological cell agent may be in a state of normal behaviour or it may be in the process of dividing or dying. Representing agents in differing states ensures that complex systems exhibiting heterogeneous behaviour can be represented by groups of homogeneous behaviours without introducing large amounts of divergence in the behavioural scripts, an issue known to negatively effect performance on GPUs. Within the Boids model, all agents perform the same homogeneous behaviour throughout the life-cycle of the simulation. As such the model requires only a single state. *default*.

c) Functions: Agents functions represent a mapping (or update) of the agents internal memory which occurs when an agent transitions from one state to another. Functions are applied to agents in a specified *initial_state* and will result in the agent moving into a *next_state* (or the same state). The behaviour script which dictates how an agent updates its internal memory when transitioning between states will be described in the *functions* section. A description of the scripted function must however be provided in the model file to provide information on the functions operation. For example, functions can have either a single input or output (or neither). Inputs and outputs are in the form of a named message type described in the XMML as a collection of variables. Functions can have conditions as a mechanism for transiting subsets of agents into differing states. For example, a function which simulates agent death might check a variable incremented during each iteration called `life-cycles` to see if it has reached a maximum number. Only agents meeting this condition can be made to perform the behaviour and move into a dead state. Function conditions must always be deterministic.

The following XML code (Listing 2) presents the syntax required to specify the Boids `move` function description within the model file.

```
<gpu:function>
  <name>move</name>
  <currentState>default</currentState>
  <nextState>default</nextState>
  <gpu:reallocate>false</gpu:reallocate>
  <gpu:RNG>false</gpu:RNG>
</gpu:function>
```

Listing 2. Move Function Description Added to the XML Model File

The `reallocate` and `RNG` tags are used to specify if the agent function may result in an agent's death (reallocate) or if a random number generation is required (RNG). The presence of either flag will modify the prototype of the corresponding function definition within the user provided agent function file.

All defined agent *functions* must be added to the models execution *layers*. Layers represent global synchronisation points in the execution of the simulation. It is assumed that functions on the same layer execute simultaneously so functions which have a dependency via messages (or computation) must occur on different layers.

Listing 3 shows the `move` function which is added the third layer in the model description. The `move` function is dependant on the completion of the `inputdata` function, so it is added to a new layer which immediately follows the `inputdata` function layer. In this case the `move` function is not dependant on message data (as is the case between the `outputdata` and `inputdata` functions) but instead is dependant on computation of the velocity vectors carried out by the `inputdata` function and stored within the agents internal memory.

```
</layer>
  <layers>
    <layer>
      <gpu:layerFunction>
        <name>outputdata</name>
      </gpu:layerFunction>
    </layer>
    <layer>
      <gpu:layerFunction>
        <name>inputdata</name>
      </gpu:layerFunction>
    </layer>
    <layer>
            <gpu:layerFunction>
               <name>move</name>
          </gpu:layerFunction>
        </layer>
  </layers>
```

Listing 3. An Example of Separating Agent Functions into Layers

C. Model Behaviour

After adding a function to the model description a definition of the function containing agent agents behaviour must be defined within a suitable function script. E.g. `functions.c`. A blank `functions.c` file can be generated by FLAME GPU with the correctly formatted function prototypes to expedite model creation. All FLAME GPU functions are distinguished by the proceeding `__FLAME_GPU_FUNC__` attribute. Each FLAME GPU function represents the behavioural script that a single agent will perform. The individual agent data is passed to the function as the first argument in the form of a C structure (*xmemory*). The *xmemory* structure has a member variable for each of the agent variables defined in the model file. By modifying the members of this structure, an agents internal memory variables are updated. For example the structure represented int Listing 4 is generated from the Boids memory defined in Listing 1.

```
struct xmachine_memory_Boid
{
    int id;
    float x;
    float y;
    float z;
    float fx;
    float fy;
    float fz;
};
```

Listing 4. Automatically generated Boid data structure.

In Listing 5, `outputdata` function makes use of a dynamically generated function called `add_location_message`. The agent function is able to call `add_location_message` as the definition in the model file states that it will output a message of type *location*. The agent returns a value of 0 to indicate that it is still alive. Any non zero value will indicate that the agent is dead and that it should be removed from the simulation (assuming the reallocate value of the functions definition is not false).

```
__FLAME_GPU_FUNC__ int outputdata(
    xmachine_memory_Boid* xmemory,
    xmachine_message_location_list*
    location_messages)
{
  add_location_message(location_messages,
                       xmemory->id,
                       xmemory->x,
                       xmemory->y,
                       xmemory->z,
                       xmemory->fx,
                       xmemory->fy,
                       xmemory->fz);

  return 0;
}
```

Listing 5. Code snippet of OutputData Function in Boids Model

The `inputdata` function (Listing 6) demonstrates how to cycle a list of messages using the `get_first_location_message` and `get_next_location_message` functions. It has been simplified for brevity but shows the addition of the *xmachine_message_location_list* argument in the function definition as a a result of the model file indicating that this function inputs the *location* message. Note that the XSLT templates will automatically generate these functions and the associated agent and message structures during the build process.

```
__FLAME_GPU_FUNC__ int inputdata(
    xmachine_memory_Boid* xmemory,
    xmachine_message_location_list*
    location_messages)
{
 xmachine_message_location* location_message;
 location_message = get_first_location_message(
    location_messages);

 while(location_message){
  //check for messages from self
  if (location_message->id != xmemory->id){
   //check distance between message and self
   float separation = distance(xmemory -
    location_message);
   if (separation < INTERACTION_RADIUS){
    //update perceived global centre
    //update average velocity
    //update avoidance vector
    ...
   }
  }

  location_message = get_next_location_message(
    location_message, location_messages);
 }
//update the velocity vx, vy, and vz
...
}
```

Listing 6. Code snippet of InputData Function in Boids Model

Within the Boids model, the final `move` function updates the agents position by using the agents velocity components (fx, fy, fz) previously calculated by the *inputdata* function. The position and the velocity are extracted from the agents memory variables into vectorised types as shown in Listing 7. This will create a 3 component vector using the `glm` library for the position and velocity (Line 3-8). The agent's position is then updated by adjusting it by some small fraction of the velocity (Line 10-14) using the $TIME\_SCALAR$ model variable (see later experimentation sub section). Finally, the agents position is written back to the *xmemory* structure (Line 16-19).

```
1  __FLAME_GPU_FUNC__ int move(
     xmachine_memory_Boid* xmemory)
2  {
3    glm::vec3 pos = glm::vec3(xmemory->x,
4                              xmemory->y,
5                              xmemory->z);
6    glm::vec3 vel = glm::vec3(xmemory->fx,
7                              xmemory->fy,
8                              xmemory->fz);
9
10   //Apply the velocity
11   pos += vel * TIME_SCALE;
12
13   //Bound position
14   pos = boundPosition(pos);
15
16   //Update agent structure
17   xmemory->x = pos.x;
18   xmemory->y = pos.y;
19   xmemory->z = pos.z;
20
21   return 0;
22 }
```

Listing 7. Positing and Velocity Calculation of Boid Agents

D. Model Execution and Visualisation

FLAME GPU simulations require a number of arguments depending on the requirement for either console or visualisation mode. In both cases the first argument is always a file location for the initial agent XML file containing the initial agent data. Extra optional CUDA arguments (i.e. `device=1`) may also be passed to specify the CUDA enabled GPU device. The default value for the CUDA device argument is 0. The syntax of the initial XML model file is dependant on the agents internal memory. Agents should be contained within $xagent$ tags naming the agent within the $name$ element. Initial variables values for the agent should be contained within tags using the variables name. An example of an a Boid agent taken from an initial agent file is shown in Listing 8.

```
<xagent>
  <name>Boid</name>
  <id>0</id>
  <x>-0.12465215</x>
  <y>-0.2607972</y>
  <z>-0.39554715</z>
  <fx>0.1592794</fx>
  <fy>-0.29218417</fy>
  <fz>-0.29697996</fz>
</xagent>
```

Listing 8. An initial state of a Boid agent taken from an initial agent XML file provided as an argument when executing the simulation.

Within console mode an additional argument is required which represents the number of iterations which should be performed. After execution of the simulation in console mode a number of XML output files containing the state of the agents at each iteration will be generated (in the same location of the initial input file). Output files can be used as input for subsequent runs and therefore provide a a method of check-pointing.

Statistical information about the simulation can be obtained by writing scripts which parse the agent output files. For example, a Python script can be specified to iterate the output files and plot the the average distance between agents over time. Within FLAME GPU, there are various macros within the generated simulation header file ($header.h$) which can be modified to output to console extra information about the population. E.g: agent count per iteration or more detailed timing results.

Simulation executable for visualisation only requires the initial agent XML file. The number of simulation iterations is not required as the simulation will run indefinitely until the visualisation window is closed. Building for visualisation requires only the specification of a *_VISUALISATION* macro to the compiler. Different build configurations are provided both for Visual Studio and within the provided $make$ files. Visualisation of agents is through a basic representation using a 3D spherical object of agents in 3D space. The visualisation technique adopted within FLAME GPU uses instance based rendering to avoid any reading back of the agents data back through host (CPU) memory and is hence extremely efficient. Fig. 4 shows the visualisation of Boids model.

E. Experimenting with the Model

Model variables constitute variable's which are uniform across all agents within a simulation. For example the $TIME\_SCALAR$ introduced in Listing 7 which can be used to balance the speed or accuracy of agents movements. Model variables can be expressed either a macro definitions written directly within the `function.c`) file or preferably as a FLAME GPU global variable. Global variables are defined as an environment variable (Listing 9) in the XMML model file rather than as macro definition in `function.c`. In either case they are referenced in exactly the same way. The significant advantage of using FLAME GPU global variables is that they can be specified in the initial agent XML file containing initial agent data (Listing 10).

```
...
<gpu:environment>
  <gpu:constants>
    <gpu:variable>
      <type>float</type>
      <name>TIME_SCALE</name>
    </gpu:variable>
    </gpu:constants>
</gpu:environment>
...
```

Listing 9. Specification of an environment variable for the Boids model within the XMML model file

```
<states>
<itno>0</itno>
<environment>
<TIME_SCALE>0.0005f</TIME_SCALE>
</environment>
...
```

Listing 10. Code Snippet for initial agent data file (0.xml) showing the experimental modification of the $TIME\_SCALAR$ global variable.

V. Conclusion

In this paper, we have shown the features and capabilities of the FLAME GPU simulation platform through an example.

FLAME GPU is an open source software which is managed and contributed to via Github [3]. This allows anyone to contribute to the main project, or propose extensions on top of it. Future releases of the FLAME GPU will integrate improvements which include:

- code portability across various GPU architectures
- load balancing of models across multiple GPU devices
- behind the scene memory management and GPU kernel optimisation
- scaling performance for communication between multiple GPU devices

Furthermore, the code generated by this state of art framework will be optimised for the latest GPU hardware including P100s and for large shared memory multi-GPU equipment such as the DGX-1.

[3]https://github.com/FLAMEGPU/FLAMEGPU

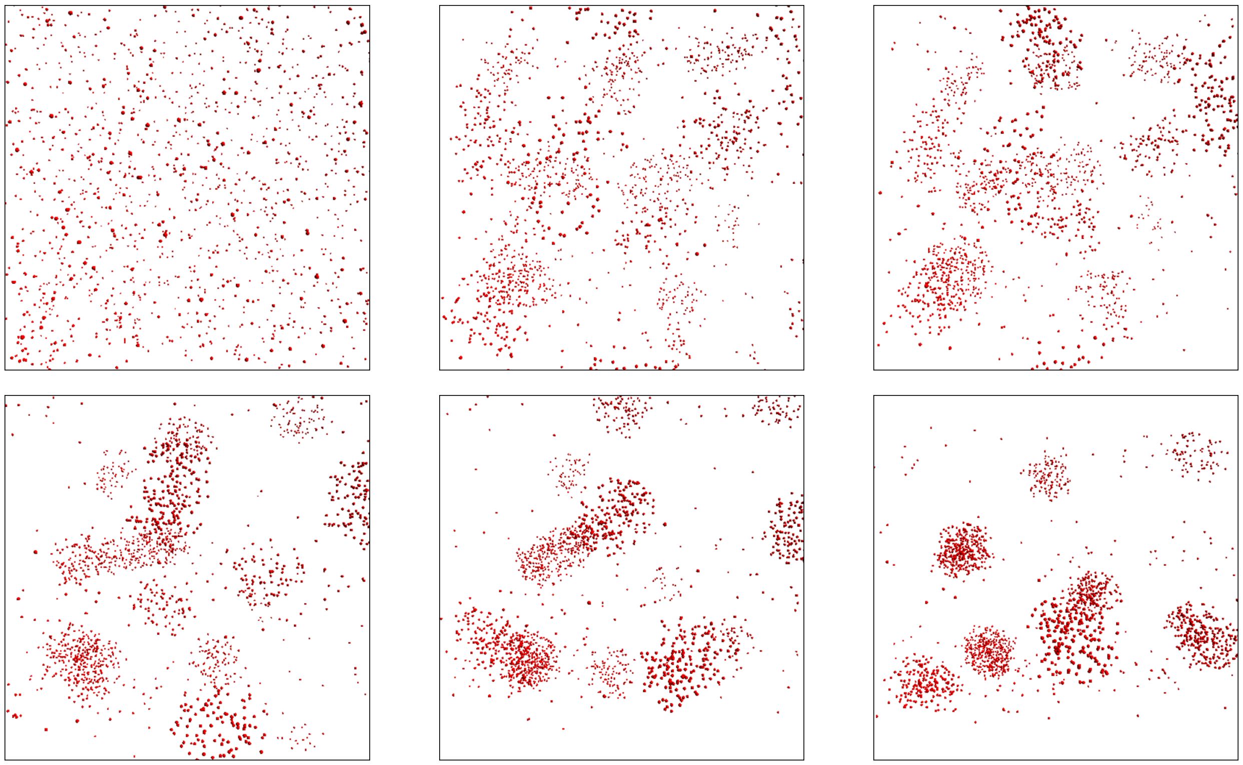

Fig. 4. Visualisation of Boids model. The figures show progression from an initial random state to clustered grouping as a result of the Boids local interaction model. Agents are rendered as spheres.

References

[1] S. Coakley, P. Richmond, M. Gheorghe, S. Chin, D. Worth, M. Holcombe, and C. Greenough, *Large-Scale Simulations with FLAME*. Cham: Springer International Publishing, 2016, pp. 123–142.

[2] S. M. M. B. J. C. D. G. Peter Heywood, Paul Richmond, "White paper: Accelerating transport microsimulation: Demonstrating the impact of future many-core simulations," in press.

[3] S. M. Robert Chisholm, Paul Richmond, "A standardised benchmark for assessing the performance of fixed radius near neighbours," in press.

[4] T. Karmakharm, P. Richmond, and D. M. Romano, "Agent-based large scale simulation of pedestrians with adaptive realistic navigation vector fields," in *TPCG*, J. P. Collomosse and I. J. Grimstead, Eds. Eurographics Association, 2010, pp. 67–74.

[5] M. Kiran, L. S. Chin, P. Richmond, M. Holcombe, D. Worth, and C. Greenough, "Flame: Simulating large populations of agents on parallel hardware architectures," 2010.

[6] P. Richmond, S. Coakley, and D. Romano, "Cellular level agent based modelling on the graphics processing unit," in *2009 International Workshop on High Performance Computational Systems Biology*, Oct 2009, pp. 43–50.

[7] P. Richmond, S. Coakley, and D. M. Romano, "A high performance agent based modelling framework on graphics card hardware with cuda," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, ser. AAMAS '09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1125–1126. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558109.1558172

[8] P. Richmond and D. Romano, "Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu," in *In Proceedings International Workshop on Super Visualisation (IWSV08) 2008. In.* Press, 2008.

[9] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334–347, May 2010. [Online]. Available: http://dx.doi.org/10.1093/bib/bbp073

[10] S. Tamrakar, P. Richmond, and R. M. DSouza, "Pi-flame: A parallel immune system simulator using the flame graphic processing unit environment," *SIMULATION*, vol. 93, no. 1, pp. 69–84, 2017.

[11] A. Oliveira and P. Richmond, "Feasibility study of multi-agent simulation at the cellular level with flame gpu," 2016.

[12] T. Balanescu, A. J. Cowling, H. Georgescu, M. Gheorghe, M. Holcombe, and C. Vertan, "Communicating stream x-machines systems are no more than x-machines," *Journal of Universal Computer Science*, vol. 5, no. 9, pp. 494–507, sep 1999.

[13] P. Richmond and D. Romano, "Template-Driven Agent-Based Modeling and Simulation with CUDA," in *GPU Computing Gems Emerald Edition*, 1st ed., ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed. Morgan Kaufmann, Feb. 2011, ch. 21, pp. 313–324.

[14] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," 2010. [Online]. Available: http://www.meganewtons.com/

[15] M. Harris, S. Sengupta, and J. Owens, "Parallel prefix sum (scan) with CUDA," *GPU Gems*, vol. 3, no. 39, p. 851876, 2007.

[16] T. Karmakharm and P. Richmond, "Large Scale Pedestrian Multi-Simulation for a Decision Support Tool," in *Theory and Practice of Computer Graphics*, H. Carr and S. Czanner, Eds. The Eurographics Association, 2012.

[17] L. Nyland and M. Harris, "Chapter 31 Fast N-Body Simulation with CUDA," 2007.

[18] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 25–34, Aug. 1987. [Online]. Available: http://doi.acm.org/10.1145/37402.37406

[19] P. Richmond, "FLAME GPU Technical Report and User Guide (CS-11-03)," Department of Computer Science, University of Sheffield, Tech. Rep., 2011.