FLAME GPU Introduction Tutorial

Authors: Paul Richmond, Mozhgan K Chimeh

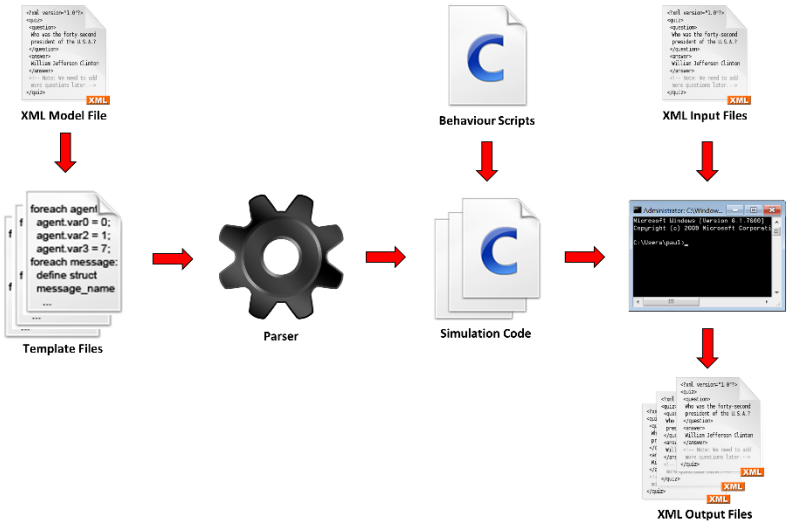
http://www.flamegpu.com/

Introduction and Background

Agent Based Modelling is a technique for computational simulation of complex interacting systems, through the specification of the behaviour of a number of autonomous individuals acting simultaneously. This is a bottom up approach, in contrast with the top down one of modelling the behaviour of the whole system through dynamic mathematical equations. The focus on individuals is considerably more computationally demanding but provides a natural and flexible environment for studying systems demonstrating emergent behaviour. Despite the obvious parallelism, traditionally frameworks for ABM fail to exploit this and are often based on highly serialised algorithms for manipulating mobile discrete agents. Such an approach has serious implications, placing stringent limitations on both the scale of models and the speed at which they may be simulated. 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 and ensures simulations run within reasonable time constraints. In addition to this, visualisation is easily achievable as simulation data is held entirely within GPU memory where it can be rendered directly.

High Level Overview of FLAME GPU

Technically the FLAME GPU framework is not a simulator. Instead, it is a template based simulation environment that maps formal descriptions of agents into simulation code. The following figure shows the components which are required to specify a FLAME GPU simulation. A user specifies a FLAME GPU model within an XML file and describes the behaviour of agents within a function script. Templates are used to generate highly efficient CUDA GPU code. Importantly, the writing of GPU code is abstracted from the user so that the modeller can concentrate on writing models without having to understand the complexities of the GPU architecture.



To execute a FLAME GPU simulation, a user builds the code generated from the templates and provides a set of initial states for the agents in the population. These initial states will include information such as the agent's location and any properties it has.

Getting started with FLAME GPU

To get started with FLAME GPU, open a suitable AWS image (with GPU capabilities and a CUDA install) and download the latest FLAME GPU working branch from GitHub using the following shell command:

```
git clone https://github.com/FLAMEGPU/FLAMEGPU.git
```

The folder structure of FLAME GPU is as follows:

FLAME GPU: contains the templates and XML schemas thatare used to generate CUDA GPU code. These should not be modified by the users.

Bin/x64: The location of the console and visualisation binaries for each of the examples. There is a Linux shell script for each example which will start the simulation with an initial states file (and the number of iterations to simulation in console mode)

Doc: The FLAME GPU technical report and user guide

Examples: The location of the model files for FLAME GPU examples and the location to create your own models.

Include: Some common include files required by FLAME GPU

Lib: Any library dependencies required by FLAME GPU

Media: 3D models used for some of the visualisations

Tools: A number of tools for generating function script files from XML model files and running template code generation in windows.

To start with, we are going to work with a model called `Boids`. This is a simple model in which agents demonstrate flocking behaviour. They have a number of simple rules which include steering towards the centre of their perceived local group, avoiding collisions with other agents and matching the speed of neighbouring agents. Navigate to the `examples/Boids_BruteForce` directory and call `make`. E.g.

```
cd examples
cd Boids_BruteForce
make
```

This will process the XML model and build a console and visualisation version of the model in release mode. Navigate back to the FLAME GPU bin directory and call the `Boids_BruteForce_console.sh` script which will have been generated by the make process.

```
cd ../../bin/x64/
./Boids_BruteForce_console.sh
```

The output will be an XML file (saved in the location of the initial input file) which will contain the state of the agents after applying a single simulation iteration to the agents. You can view this file to see how the agent positions and other properties have changed.

Examine the run script by looking at the parameters passed to the simulation. The parameters are the initial model file and the number simulation runs (iterations). In order to modify the number of iterations, simply pass an argument to the shell script as follows:

```
./Boids_BruteForce_console.sh 100
```

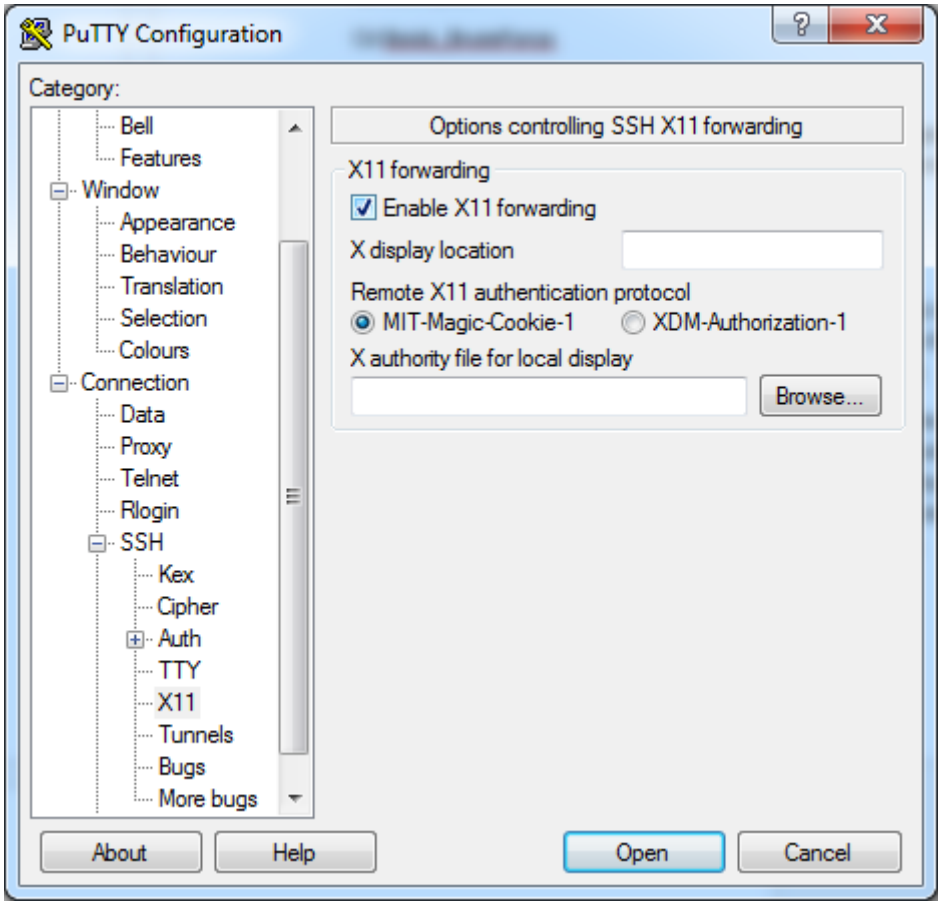
Multiple output files (100 in the above example) will be generated, one for each simulation step. Alternatively, you can modify the script use the output of the previous iteration as the input for a new one.

Visualising a Model

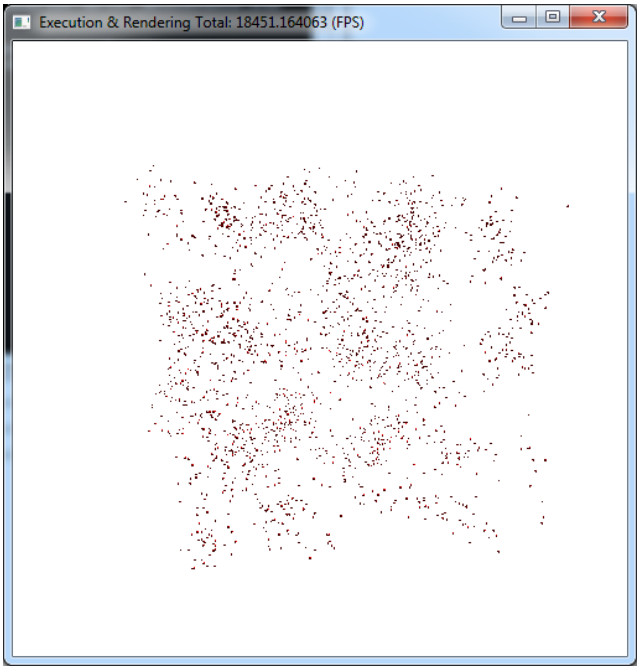
Note: This will not work on Amazon AWS images unless the Graphics driver is updated. Skip this section for the tutorial unless you are running on your own Ubuntu system.

In order to visualise the model, we will need to configure X11 window forwarding so that the OpenGL windows is opened on our local machine. If you are using Linux as your host operating system then you do not need to install anything. If you are using Windows you will need to install an X11 window server so that the remote machine where we are executing FLAME GPU can display on your local machine. The XMing software is free and is recommended.

You should disconnect your SSH session to the AWS image and connect again making sure to enable X11 forwarding. From the command line, this requires that you use the ssh `-X` option. From Putty in Windows, this can be enabled from the Connection->SSH->X11 dialogue shown below.



From your SSH session, navigate to the FLAME GPU bin folder (bin/x64/) and run the `Boids_BruteForce_viz.sh` scriptvisualisation shell script. This will launch the simulation in visualisation mode. The visualisation windows will open on your local machine.



Changing a FLAME GPU Model

To change the Boids model, we only need to modify two files in the model's `src/model` directory. We will start by modifying the model description. Open the XMLModelFile.xml in your favourite text editor (e.g. nano). The XML syntax is governed by a schema. If you use an intelligent XML editor, the schema will make suggestions as to what elements can be added to the file in specific locations.

The model file contains three major components which are `Agents`, `Messages`, and `Layers`. Agents are the description of what an agent is, Messages are what an agent will communicate to other agents and Layers are the order in which agents should perform behaviours.

The description of an agent consists of `Memory`, `States`, and `Functions`. Memory describes the persistent variables which an agent has over its life-cycle of the simulation. Variables are typically used to hold properties unique to the agent such as location, velocity, etc. The Boids model has a variable for each component of the agent's location (`x`, `y`, `z`) and velocity (`fx`, `fy`, `fz`) as well as a unique identifier (`id`). Agent's states are used to differentiate between agents of the same type which may be in differing functional states. 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. Agents in differing states represent the same type individuals within a population. However, they will likely have differing behaviours. The Boids model has only a single state of '`default`' this is because all agents perform the same homogenous behaviour throughout the life-cycle of the simulation.

The Functions aspect of the agent description describes the properties of the behaviours that the agent will have. *Note: This is not where the actual behaviour is described but the properties relating to the behaviour functions*. 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). Furthermore, functions can have conditions. For example, a function which simulates agent death might check a variable incremented each iteration called `life_cycles` to see if it has reached a maximum number. Only agents meeting this condition would perform the behaviour and move into a dead state. The Boids model is a simple model and does not have any function conditions.

Functions can have either a single input or output (or neither). Inputs and outputs are in the form of messages which are a collection of variables which are persistent from the point in which they are output to the end of the simulation iteration (at which point they are destroyed). A function description within a model requires that any inputs or outputs are fully specified.

To better clarify the structure of a Function description in a model, let have a look at the below XML code. In order to add a new agent function called '`move`', add the following XML code to the model after the existing function definitions.

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

The additional `reallocate` and `RNG` tags are used to specify if the agent function may result in agent death (`reallocate`) or if a random number generator is required (`RNG`).

We now need to add the function to the layers. Layers represent synchronisation points in the execution of the agent functions. It is assumed that functions on the same layer execute simultaneously so functions which have a dependency via messages should not be within the same layer. The `outputdata` and `inputdata` functions are in different layers for this reason. We will add our new `move` function so that it executes in a new layer. Add this new layer after the layer containing the `inputdata` function as follows. This will ensure our `move` function only begins executing after all agents have completed execution of the `inputdata` function.

```
<layer>
  <gpu:layerFunction>
        <name>move</name>
  </gpu:layerFunction>
</layer>
```

Changing FLAME GPU Model Behaviour

Having added our new function in the model description, we now need to implement the behaviour of the function by defining a suitable function script in `functions.c`. Open `functions.c` in your favourite text editor and take a look at the existing functions. FLAME GPU functions are proceeded with `__FLAME_GPU_FUNC__`. There are two functions (`inputdata` and `outputdata`) which match the function names in the model file. Each FLAME GPU function represents the behavioural script that an individual agent will perform. The individual agent data is passed to the function as the first argument in the form of a C structure (xmemory) which has a member variable for each of the agent variables defined in the model file. By modifying the members of this structure we can update an agent's internal memory variables.

Now let's have a look at the `outputdata` function. It 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 `inputdata` function demonstrates how to cycle a list of messages using the `get_first_`**`location`**`_message` and `get_next_`**`location`**`_message` functions. The templates will automatically generate these functions during the build process.

Add a new FLAME GPU function with the following definition which matches the name of the new `move` function we added in the model file.

```
__FLAME_GPU_FUNC__ int move(xmachine_memory_Boid* xmemory)
{
  //todo: add some behaviour

  return 0;
}
```

Remove lines 178-187 from `functions.c` as we are going to move the code that updates the agent's position into our new function. The `inputdata` function now no longer changes the agent's position. However, it updates the agent's velocity components (`fx`, `fy` and `fz`) by modifying the `xmemory` structure. We need to extract the position and the velocity from the agent's memory variables in our `move` function. Now, add the following code to the move function.

```
glm::vec3 agent_position = glm::vec3(xmemory->x, xmemory->y, xmemory->z);
```

```
    glm::vec3 agent_velocity = glm::vec3(xmemory->fx, xmemory->fy,
xmemory->fz);
```

This will create a 3 component vector using the `glm` library for the position and velocity. We can now update the position by adjusting it by some small fraction of the velocity.

```
    //Apply the velocity
    agent_position += agent_velocity * TIME_SCALE;

    //Bound position
    agent_position = boundPosition(agent_position);
```

Finally, we can update the agent's position by writing it to the `xmemory` structure as follows:

```
    //Update agent structure
    xmemory->x = agent_position.x;
    xmemory->y = agent_position.y;
    xmemory->z = agent_position.z;
```

Rebuild the model using make. The behaviour of the model will not have changed but have separated out the velocity and position calculation by adding the new function.

Experimenting with the Model

Try changing the global rule scalars on lines 36-38 of `functions.c`. This will change the behaviour of the Boids agents causing the behaviours to change. Although, this is easier to observe visually, one can simply write a script to process the XML output files to obtain statistical information about the population. For example, you could calculate the average distance between agents and the average speed and see how these changes dependant on the rule scalars.

More Information

For more information on FLAME GPU see the FLAME GPU website and documentation which gives detailed instructions on all aspects of FLAME GPU modelling.