FLAMEGPU Documentation

Release

FLAME GPU

Oct 17, 2018

Contents

This is the FLAME GPU technical report and user guide documentation. This documentation has been migrated from the old LaTex documents. Changes can be made via pull request on the GitHub repo. If you are looking for the main FLAME GPU website then go to www.flamegpu.com.

CHAPTER 1

Introduction

Agent Based Modelling is a technique for the 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 constrains. 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, it is instead a template based simulation environment that maps formal descriptions of agents into simulation code. The 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). Whilst the X-Machine has very formal definition X-Machine agents can be thought of a state machines which are able to communicate via messages which are stored in a globally accessible message lists. Agent functionality is exposed as a set of state transition functions which move agents from one internal 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 be passed to the message lists for other agents to read). FLAME GPU uses agent function scripting for this purpose where script is defined in a number of *Agent Function Files*. Simulation models are specified using a format called X-Machine Mark-up Language (*XMML*) which is XML syntax with Schemas governing the content. 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 has a globally accessible message list) and a set of simulation layers which define the execution order of agent functions (which constitutes a single simulation iteration). Throughout a simulation, agent data is persistent however message information (and in particular message lists) is persistent only over the lifecycle of a single iteration. This allows a mechanism for agents to iteratively interact in a way which allows emergent global group behaviour.

The process of generating a FLAME GPU simulation is described by the 1. The use of XML schemas forms a large part of the process where polymorphic like extension allows a base schema specification to be extended with a number of GPU specific elements. Given an XMML model definition, template driven code generation is achieved through Extensible Stylesheet Transformations (XSLT). XSLT is a flexible functional 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 (although the FLAME GPU SDK provides its own). Through the specification of a number of *XSLT Simulation Templates* a *Dynamic Simulation API* is generated which links with the *Agent Function Files* to generate a simulation program.

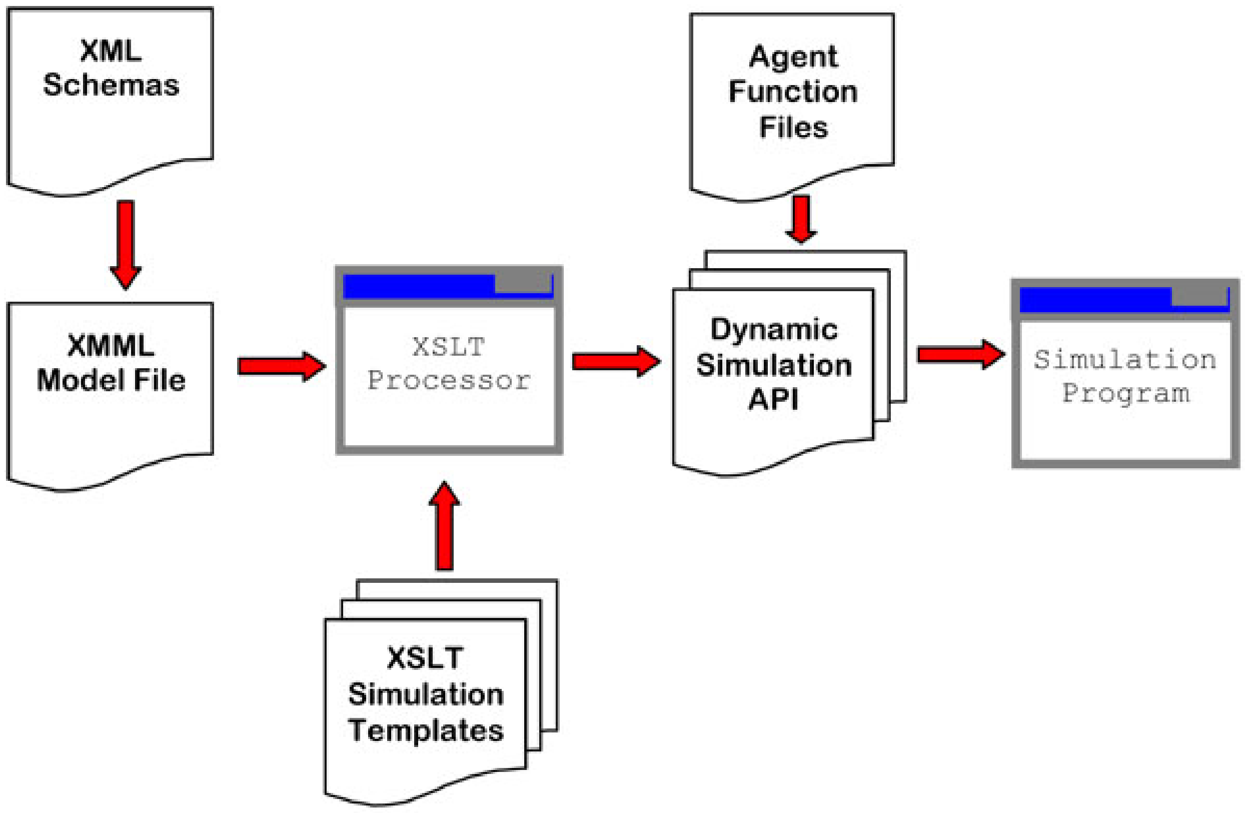


Fig. 1.1: FLAME GPU Modelling and Simulation Processes

Purpose of This Document

The purpose of this document is to describe the functional parts which make up a FLAME GPU simulation as well as providing guidance on how to use the FLAME GPU SDK. describes in detail the syntax and format of the XMML Model file. describes the syntax of use of agent function scripts and how to use the dynamic simulation API and describes how to generate simulation code and run simulations from within the Visual Studio IDE. This document does not act as a review of background material relating to GPU agent modelling, nor does it provide details on FLAME GPUs implementation or descriptions of the FLAME GPU examples. For more in depth background material on agent based simulation on the GPU, the reader is directed towards the following document;

Richmond Paul, Walker Dawn, Coakley Simon, Romano Daniela (2010), "High Performance Cellular

Level Agent-based Simulation with FLAME for the GPU", Briefings in Bioinformatics, 11(3), pages 334-47.

For details on the implementation including algorithms and techniques the reader is directed towards the following publication;

Richmond Paul (2011), "Template Driven Agent Based Modelling and Simulation with CUDA", GPU Computing Gems Emerald Edition (Wen-mei Hwu Editor), Morgan Kaufmann, March 2011, ISBN: 978-0-12-384988-5

Richmond Paul, Coakley Simon, Romano Daniela (2009), "A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA", Proc. of 8th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009), May, 10–15, 2009, Budapest, Hungary

Some examples of FLAME GPU models are described in the following publications;

Richmond Paul, Coakley Simon, Romano Daniela (2009), "Cellular Level Agent Based Modelling on the Graphics Processing Unit", Proc. of HiBi09 - High Performance Computational Systems Biology, 14-16 October 2009,Trento, Italy (additional detail in the BiB paper)

Karmakharm Twin, Richmond Paul, Romano Daniela (2010), " Agent-based Large Scale Simulation of Pedestrians With Adaptive Realistic Navigation Vector Fields", To appear in Proc. of Theory and Practice of Computer Graphics (TPCG) 2010, 6-8th September 2010, Sheffield, UK

CHAPTER 2

FLAMEGPU Model Specification

Introduction

FLAME GPU models are specified using XML format within an XMML document. The syntax of the model file is governed by two XML Schemas, an abstract base Schema describes the syntax of a basic XMML agent model (compatible with HPC and CPU versions of the FLAME framework) and a concrete GPU Schema extension this to add various bits of additional model information. Within this chapter the XML namespace (xmlns) gpu is used to qualify the XML elements which extend the basic Schema representation. A high level overview of a an XMML model file is described below with various sections within this chapter describing each part in more detail.

```
<gpu:xmodel
    xmlns:gpu="http://www.dcs.shef.ac.uk/~paul/XMMLGPU"
    xmlns="http://www.dcs.shef.ac.uk/~paul/XMML">
    <name>Model Name</name>    <!-- optional -->
    <gpu:environment>...</gpu:environment>
    <xagents>...</xagents>
    <messages>...</messages>
    <layers>...</layers>
</gpu:xmodel>
```

The Environment

The environment element is used to hold global information which relates to the simulation. This information includes, zero or more constant, or global, variables (which are constant for all agents over the period of either the simulation or single simulation iteration), a single non optional function file containing agent function script (see *Agent Function Scripts and the Simulation API*), an optional number of initialisation, step and exit functions and an optional number of graph data structures.

```
<gpu:environment>
    <gpu:constants>...</gpu:constants>             <!-- optional -->
    <gpu:functionFiles>...</gpu:functionFiles>     <!-- required -->
    <gpu:initFunctions>...</gpu:initFunctions>     <!-- optional -->
    <gpu:stepFunctions>...</gpu:stepFunctions>     <!-- optional -->
    <gpu:exitFunctions>...</gpu:exitFunctions>     <!-- optional -->
    <gpu:graphs>...</gpu:graphs>                   <!-- optional -->
</gpu:environment>
```

Simulation Constants (Global Variables)

Simulation constants are defined as (global) variables and may be of type int, float or double (on GPU hardware with double support i.e. CUDA Compute capability 2.0 or beyond). Constant variables must each have a unique name which is used to reference them within simulation code and can have an optional static array length (of size greater than 0). Table [var_types] summarizes the supported data types for the environment variables. The description element arrayLength element and defaultValue element are all optional. The below code shows the specification of two constant variables: the first represents a single `int` constant (with a default value of `1`), the second indicates an `int` array of length `5`. Simulation constants can be set either as default values as show below, within the initial agent XML file (see *Initial Simulation Constants*) or at run-time are described in *Host Simulation Hooks*. Values set in initial XML values will overwrite a default value and values which are set at runtime will overwrite values set in initial agent XML files.

```
<gpu:constants>
    <gpu:variable>
        <type>int</type>
        <name>const_variable</name>
        <description>none</description>
        <defaultValue>1</defaultValue>
    </gpu:variable>
    <gpu:variable>
        <type>int</type>
        <name>const_array_variable</name>
        <description>none</description>
        <arrayLength>5</arrayLength>
    </gpu:variable>
</gpu:constants>
```

Function Files

The `functionFiles` element is not optional and must contain a single file element which defines the name of a source code file which holds the scripted agent functions. More details on the format of the function file are given in *Agent Function Scripts and the Simulation API*. The example below shows the correct XML format for a function file named `functions.c`.

```
<gpu:functionFiles>
    <file>functions.c</file>
</gpu:functionFiles>
```

Initialisation Functions

Initialisation functions are user defined functions which can be used to set constant global variables. Any initialisation functions defined within the `initFunctions` element are called a single time by the automatically generated simulation code in the order that they appear during the initialisation of the simulation. If an `initFunctions` element is specified there must be at least a single `initFunction` child element with a unique name. *Initialisation Functions (API)* demonstrates how to specify initialisation functions within a function file.

```
<gpu:initFunctions>
    <gpu:initFunction>
        <gpu:name>initConstants</gpu:name>
    </gpu:initFunction>
</gpu:initFunctions>
```

Step Functions

Step functions are similarly defined to initialisation functions, requiring at least a single `stepFunction` child element if the `stepFunctions` element is defined. These functions are called at the end of each iteration step, i.e. after all the layers, as defined in section *Step Functions (API)*, are executed each step. Example uses of this function are to calculate agent averages during the iteration step or sort functions.

```
<gpu:stepFunctions>
    <gpu:stepFunction>
        <gpu:name>some_step_func</gpu:name>
    </gpu:stepFunction>
</gpu:stepFunctions>
```

Exit Functions

Exit functions are again like the other function types defined above, requiring at least a single `exitFunction` child element if the `exitFunctions` element is defined. These functions are called at the end of the whole simulation. An example use of this function would be to calculate final averages of agent variables or print out final values. *Exit Functions (API)* demonstrates how to specify initialisation functions within a function file.

```
<gpu:exitFunctions>
    <gpu:exitFunction>
        <gpu:name>some_exit_func</gpu:name>
    </gpu:exitFunction>
</gpu:exitFunctions>
```

Graph Data Structures

Some agent based models may contain environmental data structures as a graph. To ensure high performance access of this data, and enable communication restricted to a graph based data structure FLAME GPU 1.5.0 introduces a list of graphs to the environment.

Graphs are implemented using the Compressed Spares Row (CSR) data, enable high performance read access. Currently it is not possible to pragmatically modify (or update) the graph data structure at runtime.

The following example shows the definition of a static graph with the name `graph`, with a text description. The `<gpu:loadFromFile>` tag defines that the graph is to be loaded from a `json` file stored on disk, called `network.json`. This path is relative to the initial states file. Alternatively the graph can be loaded from an XML format via `<gpu:xml>relative/path/to/file.xml</gpu:xml>`.

The `<gpu:vertex>` and `<gpu:edge>` tags define the list of `<variables>` which the data structure contains, and the maximum number of each type of element via the `<gpu:bufferSize>` tag. Vertices require a variable called *id*, with an integer based type, such as `int`, `unsigned int`, `unsigned long long int` etc. Edges require an `id` variable of an integer type, a `source` variable of an integer type referring to a vertex id, and a `destination` variable of an integer type referring to a vertex id.

```xml
<gpu:graphs>
  <gpu:staticGraph>
    <gpu:name>graph</gpu:name>
    <gpu:description>A graph containing some static data</gpu:description> <!-- Optional -->
    <gpu:loadFromFile>
        <gpu:json>graph.json</gpu:json> <!-- or <gpu:xml>graph.xml</gpu:xml> -->
    </gpu:loadFromFile>
    <gpu:vertex>
      <variables>
        <gpu:variable>                    <!-- vertices require an id variable of an integer type -->
          <type>unsigned int</type>
          <name>id</name>
          <defaultValue>0</defaultValue>
        </gpu:variable>
        <gpu:variable>
          <type>float</type>
          <name>x</name>
          <defaultValue>1.0f</defaultValue>
        </gpu:variable>
        <gpu:variable>
          <type>float</type>
          <name>y</name>
          <defaultValue>1.0f</defaultValue>
        </gpu:variable>
      </variables>
      <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:vertex>
    <gpu:edge>
      <variables>
        <gpu:variable>                    <!-- edges require an id variable of an integer type -->
          <type>unsigned int</type>
          <name>id</name>
          <defaultValue>0</defaultValue>
        </gpu:variable>
        <gpu:variable>                    <!-- edges require a source variable of an integer type -->
          <type>unsigned int</type>
          <name>source</name>
          <defaultValue>0</defaultValue>
        </gpu:variable>
        <gpu:variable>                    <!-- edges require a destination variable of an integer type -->
          <type>unsigned int</type>
          <name>destination</name>
          <defaultValue>0</defaultValue>
        </gpu:variable>
      </variables>
      <gpu:bufferSize>256</gpu:bufferSize>
    </gpu:edge>
  </gpu:staticGraph>
</gpu:graphs>
```

Defining an X-Machine Agent

A XMML model file must contain a single `xagents` element which in turn must define at least a single `xagent`. An `xagent` is an agent representation of an X-Machine and consists of a name, optional description, an internal memory set (*M* in the formal definition), a set of agent functions (or next state partial functions, *F*, in the formal definition) and a set of states (*Q* in the formal definition). In addition to this FLAMEGPU requires two additional pieces of information (which are not required in the original XMML specification), a `type` and a `bufferSize`. The `type` element refers to the type of agent with respect to its relation with its spatial environment. An agent type can be either `discrete` or `continuous`, discrete agents occupy non mobile 2D discrete spatial partitions (cellular automaton) where as continuous agents are assumed to occupy a continuous space environment (although in reality they may in fact be non spatial more abstract agents). As all memory is pre-allocated on the GPU a `bufferSize` is required to represent the largest possible size of the agent population. That is the maximum number of x-machine agent instances of the format described by the XMML model. There is no performance disadvantage to using a large `bufferSize` however it is the user's responsibility to ensure that the GPU contains enough memory to support large populations of agents. It is recommended that the bufferSize always be a power of two number (i.e. `1024`, `2048`, `4096`, `16384`, etc) as it will most likely be rounded to one during simulation. For discrete agents the bufferSize is strictly limited to only power of 2 numbers which have squarely divisible dimensions (i.e. the square of the bufferSize must be a whole number). If at any point in the simulation exceeds the stated `bufferSize` then the user will be warned at the simulation will exit. Care must be taken when defining the value of bufferSize. Any datatype which would exceed the stack limit of 2GB (calculated as bufferSize*sizeof(agent variable data type) will fail to build under windows. E.g. This limits the bufferSize for 4byte variables (int, float, etc) to 62.5 million.

Each expandable aspect of an XMML agent representation in the below example is discussed within this section with the exception of agent functions, which due to their dependence of the definition of messages, are discussed later in *Defining an Agent function*.

```xml
<xagents>
    <gpu:xagent>
    <name>AgentName</name>
        <description>optional description of the agent</description>
        <memory>...</memory>
        <functions>...</functions>
        <states>...</states>
        <gpu:type>continuous</gpu:type>
        <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:xagent>
    <gpu:xagent>
        <!-- ... -->
    </gpu:xagent>
</xagents>
```

Agent Memory

Agent memory consists of a number of variables (at least one) which are use to hold information. An agent `variable` must have a unique `name` and may be of `type` `int`, `float` or `double` (CUDA compute capability 1.3 or beyond). Table [var_types] summarizes the supported data types for the agent variables. Default values are always `0` unless a `defaultValue` element is specified or if a value is specified within the XML input states file (which supersedes the default value). There are no specified limits on the maximum number of agent variables however the performance tips noted in *Performance Tips* should be taken into account. Agent memory can also be defined as static sized array. Below shows an example of agent memory containing four agent variables representing an agent identifier, two positional values (one with a default value) and a list of numbers.

```xml
<memory>
    <gpu:variable>
```

```
        <type>int</type>
        <name>id</name>
        <description>variable description</description>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>x</name>
        <defaultValue>1.0f</defaultValue>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>y</name>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>nums</name>
        <arrayLength>64</arrayLength>
    </gpu:variable>
</memory>
```

Agent States

Agent states are defined as a list of `state` elements (Q in the X-Machine formal definition) with a unique and non optional name. As simulations within FLAMEGPU can continue indefinitely (or for a fixed number of iterations), terminal states (T in the formal definition) are not defined. The initial state q_0 must however be defined within the initialState element and must correspond with an existing and unique state name from the list of states above it.

```
<states>
    <gpu:state>
        <name>state1</name>
    </gpu:state>
    <gpu:state>
        <name>state2</name>
    </gpu:state>
    <initialState>state1</initialState>
</states>
```

Defining Messages

Messages represent the information which is communicated between agents. An element `messages` contains a list of at least one `message` which defines a non optional `name` and an optional `description` of the message, a list of `variables`, a `partitioningType` and a `bufferSize`. The `bufferSize` element is used in the same way that a `bufferSize` is used to define an X-Machine agent, i.e. the maximum number of this message type which may exist within the simulation at one time. The `partitioningType` may be one of four currently defined message partition schemes, i.e. non partitioned (`partitioningNone`), discrete 2D space partitioning (`partitioningDiscrete`), 2D/3D spatially partitioned space (`partitioningSpatial`) or graph edge partitioned (`partitioningGraphEdge`). Message partition schemes are used to ensure that the most optimal cycling of messages occurs within agent functions. The use of the partitioning techniques is described within this section, as are message variables.

```
<messages>
    <gpu:message>
        <name>message_name</name>
        <description>optional message description</description>
        <variables>...</variables>
        ...<partitioningType/>... <!-- replace with a partitioning type -->
        <gpu:bufferSize>1024</gpu:bufferSize>
    </gpu:message>
    <gpu:message>...</gpu:message>
</messages>
```

Message Variables

The message `variables` element consists of a number of `variable` elements (at least one) which are use to hold communication information. A `variable` must have a unique `name` and may be of `type {int`, `float` or `double` (CUDA Compute capability 2.0 or beyond). Unlike with agent variables, message variables support only scalar single memory values (i.e. no static or dynamic arrays). Table [var_types] summarizes the supported data types for the message variables. There are no specified limits on the maximum number of message variables however increased message size will have a negative effect on performance in all partitioning cases (and in particular when non partitioned messages are used). The format of message variable specification shown below is identical to that of agent memory. The only exception is the requirement of certain variable names which are required by certain partitioning types. Non partitioned messages have no requirement for specific variables. Discrete partitioning requires two `int` type variables of name `x` and `y`. Spatial partitioning requires three `float` (or `double`) type variables named `x`, `y` and `z`. The example below shows an example of message memory containing two message variables named `id` and `message_variable`.

```
<variables>
    <gpu:variable>
        <type>int</type>
        <name>id</name>
        <description>variable description</description>
    </gpu:variable>
    <gpu:variable>
        <type>float</type>
        <name>message_variable</name>
    </gpu:variable>
</variables>
```

Non partitioned Messages

None partitioned messages do not use any filtering mechanism to reduce the number of messages which will be iterated by agent functions which use the message as input. None partitioned messages therefore require a brute force or $O(n^2)$ message iteration loop wherever the message list is iterated. As non partitioned messages do not require any message variables with location information the partition type is particularly suitable for communication between non spatial or more abstract agents. Brute force iteration is obviously computationally expensive, however non partitioned message iteration requires very little overhead (or setup) cost and as a result for small numbers of messages it can be more efficient than either limited range technique. There is no strict rule governing performance and different GPU hardware will produce different results depending on it capability. It is therefore left to the user to experiment with different message partitioning types within a simulation. The example below shows the format of the partitioningNone element tag.

```
<gpu:partitioningNone/>
```

Discrete Partitioned Messages

Discrete partitioned messages are messages which may only originate from non mobile discrete agents (cellular automaton). A discrete partitioning message scheme requires the specification of a radius which indicates the range (in in 2D discrete space) which a message iteration will extend to. A radius value of `0` indicates that only a single message will be returned from message iteration. A value of greater than `0` indicates that message iteration will loop through radius directions in both the `x` and a `y` dimension, but ignore the centre cell (e.g. a range of `1` will iterate `(3x3)-1=8` messages, a range of `2` will iterate `(5x5)-1=24`). When iterating messages, the environment is wrapped in the `x` and `y` axis to form a torus. This means that the radius value used should be less than or equal to `floor((sqrt(bufferSize) - 1) / 2)` to avoid the same message being read multiple times. In addition to this the agent memory is expected to contain an `x` and `y` variable of `type int`. As with discrete agents it is important to ensure that messages using discrete partitioning use only supported buffer sizes (power of 2 and squarely divisible). The width and height of the discrete message space is then defined as the square of the `bufferSize` value.

```
<gpu:partitioningDiscrete>
    <gpu:radius>1</gpu:radius>
</gpu:partitioningDiscrete>
```

Warning: Outputting messages with `x` or `y` values outside of the environment bounds (greater than the square of the `bufferSize`) is undefined and may result in unexpected behaviour.

Spatially Partitioned Messages

Spatially partitioned messages are messages which originate from continuous spaced agents in a 3D environment (i.e. agents with continuous value `x`, `y` and `z` variables). A spatially partitioned message scheme requires the specification of both a radius and a set of environment bounds. The `radius` represents the range in which message iteration will extend to (from its originating point). The environment bounds represent the size of the space which massages may exist within. If a message falls outside of the environment bounds then it will be bound to the nearest possible location within it. The space within the defined bounds is partitioned according to the radius with a total of `P` partitions in each dimension, where for each dimension;

$$P = ceiling((max\_bound - min\_bound)/radius)$$

The partitions dimensions are then used to construct a partition boundary matrix (an example of use within message iteration is provided in *Spatially Partitioned Message Iteration*) which holds the indices of messages within each area of partitioned space. The value of `P` must not exceed 62.5 million due to limitations on the size of stack memory. The value of `P` must be at least 3 in both the `x` and `y` axis, and at least `1` in the `z` axis, else a compilation error will occur. If the desired configuration does not meet these critera consider using Non Partitioned Messages. Spatially partitioned message iteration can then iterate a varying number of messages from a fixed number of adjacent partitions in partition space to ensure each message within the specified radius has been considered. When iterating messages, the environment is wrapped in the `x` and `y` axis to form a torus. No wrapping occurs in the `z` axis.

The following example defines a spatial partition in three dimensions. For continuously spaced agents in 2D space `P` in the x z dimension should be equal to `1` and therefore a `zmin` of `0` would require a `zmax` value equal to `radius` (even in this case a message variable with name `z` is still required).

```
<gpu:partitioningSpatial>
    <gpu:radius>1</gpu:radius>
    <gpu:xmin>0</gpu:xmin>
    <gpu:xmax>10</gpu:xmax>
    <gpu:ymin>0</gpu:ymin>
    <gpu:ymax>10</gpu:ymax>
    <gpu:zmin>0</gpu:zmin>
    <gpu:zmax>10</gpu:zmax>
</gpu:partitioningSpatial>
```

> **Warning:** Outputting messages with `x`, `y` or `z` values outside of the environment bounds is undefined and may result in unexpected behaviour. Currently messages are clamped to the final partition in the relevant dimension, however this should not be relied upon.

Graph Edge Partitioned Messaging

Graph Edge Partitioned messages are messages which originate from continuous spaces agents in an environment where communication is restricted to the structure of a graph. I.e agents which traverse along a network such as a road network. A graph edge partitioned message scheme requires the specification of a graph and the corresponding message variable which refers to the graph edge id.

Messages are sorted by the `messageEdgeId` variable, which enables high performance access to messages on the edge. Using the graph data structure it is then possible to traverse the graph accessing messages from multiple edges.

The following example defines a graph edge partitioning scheme corresponding to a `staticGraph` named `graph` where the message variable `edge_id` contains the edge from which the message corresponds.

```
<gpu:partitioningGraphEdge>
  <gpu:environmentGraph>graph</gpu:environmentGraph>
  <gpu:messageEdgeID>edge_id</gpu:messageEdgeID>
</gpu:partitioningGraphEdge>
```

> **Warning:** Outputting messages with `messageEdgeID` values greater than the `bufferSize` of the corresponding `<gpu:environmentGraph>` is undefined and may result in unexpected behaviour.

Message Partitioning and Agent Type Compatibility

Different types of agent (`CONTINUOUS` & `DISCRETE_2D`) agents can output different types of message, and may need to use templated functions to read messages of certain types. The following table shows which message types may be output, and when templated accessor functions are required.

Message Type	Output		Input Template Argument	
	CONTINUOUS	DISCRETE_2D	CONTINOUS	DISCRETE_2D
partitioningNone	**Yes**	No		
partitioningDiscrete	No	**Yes**	<CONTINOUS>	<DISCRETE_2D>
partitioningSpatial	**Yes**	No		
partitioningGraphEdge	**Yes**	No		

Defining an Agent function

An optional list of agent `functions` is described within an X-Machine agent representation and must contain a list of at least a single agent `function` element. In turn, a function must contain a non optional `name`, an optional `description`, a `currentState`, `nextState`, an optional single message input, and optional single message output, an optional single agent output, an optional global function condition, an optional function condition, a reallocation flag and a random number generator flag. The current state is defined within the `currentState` element and is used to filter the agent function by only applying it to agents in the specified state. After completing the agent function agents then move into the state specified within the `nextState` element. Both the current and `nextState` values are required to have values which exist as a state/name within the state list (states) definition. The `reallocate` element is used as an optional flag to indicate the possibility that an agent performing the agent function may die as a result (and hence require removing from the agent population). By default this value is assumed `true` however if a value of false is specified then the processes for removing dead agents will not be executed even if an agent indicates it has died (see agent function definitions in *Defining an Agent function*). The `RNG` element represents a flag to indicate the requirement of random number generation within the agent function. If this value is `true` then an additional argument (demonstrated in *Using Random Number Generation*) is passed to the agent function which holds a number of seeds used for parallel random number generation.

```
<functions>
    <gpu:function>
        <name>func_name</name>
        <description>function description</description>
        <currentState>state1</currentState>
        <nextState>state2</nextState>
        <inputs>...</inputs>                          <!-- optional -->
        <outputs>...</outputs>                        <!-- optional -->
        <xagentOutputs></xagentOutputs>               <!-- optional -->
        <gpu:globalCondition>...</gpu:globalCondition> <!-- optional -->
        <condition>...</condition>                    <!-- optional -->
        <gpu:reallocate>true</gpu:reallocate>         <!-- optional -->
        <gpu:RNG>true</gpu:RNG>                       <!-- optional -->
    </gpu:function>
</functions>
```

Agent Function Message Inputs

An agent function message input indicates that the agent function will iterate the list of messages with a name equal to that specified by the non optional messageName element. It is therefore required that the `messageName` element refers to an existing (XPath) `messages/message/name` defined within the XMML document. In addition to this an agent function cannot iterate a list of messages without specifying that it is an `input` within the XMML model file (message iteration functions are parameterised to prevent this).

```
<inputs>
    <gpu:input>
        <messageName>message_name</messageName>
    </gpu:input>
</inputs>
```

Agent Function Message Outputs

An agent function message output indicates that the agent function will output a message with a name equal to that specified by the non optional `messageName` element. The `messageName` element must therefore refer to an existing message/name defined within the XMML document. It is not possible for an agent function script to output a message without specifying that it is an output within the XMML model file (message output functions are parameterised to prevent this). In addition to the `messageName` element a message output also requires a `type`. The type may be either‘‘single_message‘‘ or `optional_message`, where `single_message` indicates that every agent performing the function outputs exactly one message and `optional_message` indicates that agent’s performing the function may either output a single message *or no message*. The type of messages which can be output by discrete agents are not restricted however continuous type agents can only output messages which do not use discrete message partitioning (e.g. no partitioning or spatial partitioning). The example below shows a message output using `single_message` type. This will assume every agent outputs a message, if the functions script fails to output a message for every agent a message with default values (of `0`) will be created instead.

```
<outputs>
    <gpu:output>
        <messageName>message_name</messageName>
        <gpu:type>single_message</gpu:type>
    </gpu:output>
</outputs>
```

Agent Function X-Agent Outputs

An agent function `xagentOutput` indicates that the agent function will output an agent with a name equal to that specified by the non optional `xagentName` element. This differs slightly from the formal definition of an x-machine which does not explicitly define a technique for the creation of new agents but adds functionality required for dynamically changing population sizes during simulation runtime. The `xagentName` element belonging to an `xagentOutput` element must refer to an existing (XPath) `xagents/agent/name` defined within the XMML document. It is not possible for an agent function script to output a agent without specifying that it is an `xagentOutput` within the XMML model file (agent output functions are parameterised to prevent this). In addition to the `xagentName` element a message output also requires a `state`. The `state` represents the state from the list of state elements belonging to the specified agent which the new agent should be created in. Only `continuous` type agents are allowed to output new agents (which must also be of type `continuous`). The creation of new discrete agents is not permitted. An `xagentOutput` does not require a type (as is the case with a message output) and any agent function outputting an agent is assumed to be optional. I.e. each agent performing the function may output either one or zero agents.

```
<xagentOutputs>
    <gpu:xagentOutput>
        <xagentName>agent_name</xagentName>
        <state>state1</state>
    </gpu:xagentOutput>
</xagentOutputs>
```

Function Conditions

An agent function `condition` indicates that the agent function should only be applied to agents which meet the defined condition (and in the correct state specified by `currentState`). Each function condition consists of three parts a left hand side statement (`lhs`), an `operator` and a right hand side statement (`rhs`). Both the `lhs` and `rhs` elements may contain either a `agentVariable` a value or a recursive condition element. An `agentVariable` element must refer to a agent variable defined within the agents list of variable names (i.e. the XPath equivalent of `xagent/memory/variable/name`). A `value` element may refer to any numeric value or constant definition (defined within the agent function scripts). The use of recursive conditions is demonstrated below by embedding a condition within the `rhs` element of the top level condition.

```
<condition>
    <lhs>
```

```
        <agentVariable>variable_name</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
        <condition>
            <lhs>
                <agentVariable>variable_name2</agentVariable>
            </lhs>
            <operator>+</operator>
            <rhs>
                <value>1</value>
            </rhs>
        </condition>
    </rhs>
</condition>
```

In the above example the function condition generates the following pseudo code function guard;

```
(variable_name) < ((variable_name2)+(1))
```

The `condition` element may refer to any logical operator. Care must be taken when using angled brackets which in standard form will cause the XML syntax to become invalid. Rather than the left hand bracket (less than) the correct xml syntax of `&lt;` should be used. Likewise the right hand bracket (greater than) should be replaced with `&gt;`.

Note: *Discrete* agents **cannot** have agent functions with conditions.

Global Function Conditions

An agent global function condition is similar to an agent function in its syntax however it acts as a global switch to determine if the function should be applied to either **all** or **none** of the agents (within the correct state specified by `currentState`). In the case of *every* agent evaluating the global function condition to `true` (or to the value specified by the `mustEvaluateTo` element) the agent function is applied to **all** of the agents. In the case that *any* of the agents evaluate the global function condition to `false` (or to the logical opposite of the value specified by the `mustEvaluateTo` element) then the agent function will be applied to **none** of the agents. As with an agent function condition a `globalCondition` consists of a left hand side statement (`lhs`), an `operator` and a right hand side statement (`rhs`). The syntax of the left hand side statement (`lhs`), the `operator` and the right hand side statement (`rhs`) is the same as with an agent function condition and may use recursion to generate a complex conditional statement. The `maxItterations` element is used to limit the number of times a function guarded by the global condition can be avoided (or evaluated as the logical opposite of the value specified by the `mustEvaluateTo` element). For example, the definition at the end of this section, resulting in the following pseudo code condition;

```
(((movement) < (0.25)) == true)
```

May be evaluated as false up to `200` times (i.e. in `200` separate simulation iterations) before the global condition will be ignored and the function is applied to every agent. Following maximum number of iterations being reached the iteration count is reset once the agent function has been applied.

```
<gpu:globalCondition>
    <lhs>
        <agentVariable>movement</agentVariable>
    </lhs>
    <operator>&lt;</operator>
    <rhs>
```

```
        <value>0.25</value>
    </rhs>
    <gpu:maxItterations>200</gpu:maxItterations>
    <gpu:mustEvaluateTo>true</gpu:mustEvaluateTo>
</gpu:globalCondition>
```

Function Layers

Function layers represent the control flow of the simulation processes and hence describes any functional dependencies. The sequence of layers defines the sequential order in which agent functions are executed. Complete execution of every layer of agent functions represents a single simulation iteration which may be repeated any number of times. Synthetically within the model definition a single layers element must contain at least one (or more) layer element. Each layer element may contain at least one (or more) `gpu:layerFunction` elements which defines only a `name` which must correspond to a function name (e.g. the XPath equivalent of `xagents/xagent/functions/function/name`. Within a given layer, the order of execution of layer functions should not be assumed to be sequential (although in the current version of the software it is, future versions will execute functions within the same layer in parallel). For the same reason functions within the same layer should not have any communication or internal dependencies (for example via message communications or execution order dependency) in which case they should instead be represented within separate layers which guarantee execution order and global synchronisation between the functions. Functions which apply to the same agent must therefore also not exist within the same layer. The below example demonstrates the syntax of specifying a simulation consisting of three agent functions. There are no dependencies between `function1` and `function2` which in this case can be thought of as being functions from two different agents definitions with no shared message inputs or outputs.

```
<layers>
    <layer>
        <gpu:layerFunction>
            <name>function1</name>
        </gpu:layerFunction>
        <gpu:layerFunction>
            <name>function2</name>
        </gpu:layerFunction>
    </layer>
    <layer>
        <gpu:layerFunction>
            <name>function3</name>
        </gpu:layerFunction>
    </layer>
</layers>
```

Initial XML Agent Data

The initial agent data information is stored in an XML file which is passed to the simulator as a parameter before running the simulation. Within this initial agent data XML file, a single `states` element contains a single iteration number `itno` and any number of (including none) `xagent` elements. The syntax of the `xagent` element depends on the agent definitions contained within the XMML model definition file. A `name` element is always required and must represent an agent name contained within the XPath equivalent of `xgents/agent/name` in the XMML model definition. Following this an element may exist for each of the named agents memory variables (XPath) `xagents/agent/memory/variable/name`). Each named element is then expected to contain a value of the same `type` as the agent memory variable defined. If the initial agent data XML file neglects to specify the value of a variable defined within an agents memory then the value is assumed to be the `defaultValue` otherise `0`. If an element defines a variable name which does not exist within the XMML model definition then a warning is generated and the value is ignored. The example below represents a single agent corresponding to the agent definition in *Defining an X-Machine Agent*.

```xml
<states>
    <itno>0</itno>
    <xagent>
        <name>AgentName</name>
        <id>1</id>
        <x>21.088</x>
        <y>12.834</y>
        <z>5.367</z>
    </xagent>
    <xagent>...</xagent>
</states>
```

Care must be taken in ensuring that the set of initial data for the simulation does not exceed any of the defined `bufferSize` (i.e. the maximum number of a given type of agents) for any of the agents. If buffer size is exceeded during initial loading of the initial agent data then the simulation will produce an error.

Another special case to consider is the use of 2D discrete agents where the number of agents within the set of initial agent data must match exactly the `bufferSize` (which must also be a power of 2) defined within the XMML models agent definition. Furthermore the simulation will expect to find initial agents stored within the XML file in row wise ascending order.

Initial Simulation Constants

Simulation constants (or global variables) specified within the model file (as described in *Simulation Constants (Global Variables)*) can be set within the initial XML agent data within an environment label between the `itno` and `xagent` elements. Environment variables should be set within an XML element with a name corresponding to the environment variable name. E.g. An environment variable defined within the model file as;

```xml
<gpu:constants>
    <gpu:variable>
        <type>int</type>
        <name>my_variable</name>
        <description>none</description>
        <defaultValue>1</defaultValue>
    </gpu:variable>
</gpu:constants>
```

Could have a value specified within an initial XML agents file as follows;

```xml
<states>
<itno>0</itno>
    <environment>
        <my_variable>2</my_variable>
    </environment>
...
```

Note: that the value obtained from the initial XML agents file will supersede any default value.

Host-based Agent Creation

As of FLAME GPU 1.5.0 it is possible to create agents on the host using Init or Step functions, rather than loading from XML. This is described by *Agent Creation from the Host*.

Loading StaticGraph Data from Disk

Static Graph data is loaded from disk during the initialisation phase of a FLAME GPU simulation. The data can be loaded from either XML or JSON formats. If a static graph is defined, the file **must** be present and valid for the simulation to continue.

The following examples show the data for a graph containing 2 vertices with variables `id`, `x` & `y` and 1 edge with variables `id`, `source`, `destination` & `length`.

Listing 2.1: graph.xml

```xml
<graph>
    <vertices>
        <vertex>
            <id>0</id>
            <x>0.0</x>
            <y>0.0</y>
        </vertex>
        <vertex>
            <id>1</id>
            <x>0.0</x>
            <y>10.0</y>
        </vertex>
    </vertices>
    <edges>
        <edge>
            <id>0</id>
            <source>0</source>
            <destination>1</destination>
            <length>10.0</length>
        </edge>
    </edges>
<graph>
```

Listing 2.2: graph.json

```json
{
    "vertices": [
        {
            "id": 0,
            "x": 0.0,
            "y": 0.0
        },
        {
            "id": 1,
            "x": 0.0,
            "y": 10.0
        }
    ],
    "edges": [
        {
            "id": 0,
            "source": 0,
            "destination": 1,
            "length": 10.0
        }
    ]
}
```

FLAME GPU variable types

FLAME GPU supports the commonly used scalar types and a set of vector types (currently provided by GLM), as defined in the following tables.

Scalar Type	Description
bool	Conditional type with values of true or false
(unsgined) char	Integer type using (typically) 8 bits. Can be signed or unsigned
(unsgined) short	Integer type using (typically) 16 bits. Can be signed or unsigned
(unsgined) int	Integer type using (typically) 32 bits. Can be signed or unsigned
(unsigned) long long int	Integer type using (typically) 64 bits. Can be signed or unsigned
float	Signed single precision floating point type using (typically) 32 bits
double	Signed double precision floating point type using (typically) 64 bits

Vector Type	Scalar Type	Elements
ivec2	int	2
ivec3	int	3
ivec4	int	4
uvec2	unsigned int	2
uvec3	unsigned int	3
uvec4	unsigned int	4
fvec2	float	2
fvec3	float	3
fvec4	float	4
dvec2	double	2
dvec3	double	3
dvec4	double	4

In addition FLAME GPU supports array variables, for agent member variables, environment constants and as member variables of staticGraphs. Array variables are **not** supported for message variables.

	Scalar & Vector Types	Array Variables
Agent Variables	Yes	Yes
Environment Constants	Yes	Yes
Graph Variables	Yes	Yes
Message Variables	Yes	**No**

CHAPTER 3

Agent Function Scripts and the Simulation API

Introduction

Agent function scripts define the behaviour of agents by describing changes to memory and through the iteration and creation of messages and new agents. The behaviour of the agent function is described from the perspective of a single agent however the simulator will apply in parallel the same agent function code to each agent which is in the correct start state (and meets any of the defined function conditions). Agent function scripts are defined using a simple C based syntax with the agent function declarations, and more specifically the function arguments dependant on the XMML function definition. The use of message input and output as well as random number generation will all change the function arguments in a way which is described within this section. Likewise the simulation API functions for message communication are dependent on the definition of the simulation model contained with the XMML model definition. A single C source file is required to hold all agent function declarations and must contain an include directive for the file `header.h` which contains model specific agent and message structures. Agent functions are free to use many features of common C syntax with the following important exceptions:

- Globally Defined Variables: i.e. Variables declared outside of any function scope are not permitted and should instead be defined as global variables within the XMML model file and used as described in *Simulation Constants (Global Variables)*. *Note: The use of pre-processor macro directives for constants is supported and can be freely used without restriction.*
- Include Directives: Are permitted however as agent functions are functions which are ran on the GPU during simulation they may not call non GPU code. This implies that including and linking with non CUDA libraries is not supported.
- External Function Calls: As above external function calls may only be made to CUDA `__device__` functions. Many common math functions calls such as `sin`, `cos`, etc. are supported via native GPU implementations and can be used in exactly the same way as standard C code. Likewise additional *helper* functions can be defined and called from agent functions by prefixing the helper function using the `__FLAME_GPU_FUNC__` macro (which signifies it can be run on the GPU device).

The following chapter describes the syntax and use of agent function scripts including any arguments which must be passed to the agent or simulation API functions. As agent functions and simulation API functions are dynamic (and based on the XMML model definition) it is often easier to first define a model and use the technique described within *Generating a Functions File Template* to automatically generate a functions file containing prototype agent function files and API system calls. Alternatively *Summary of Agent Function Arguments* describes fully the expected argument order for agent function arguments.

Agent and Message Data Structures

Access to agent and message data within the agent function scripts is provided through the use of agent and message data structures which contain variables matching those defined within the XMML definitions. For each agent in the simulation a structure is defined within the dynamically generated `header.h` with the name `xmachine_memory_agent_*name*`, likewise each message defines a structure with the name `xmachine_message_message_*name*` where `*name**` represents the agent or message `name` from the model description respectively. In both cases the structures contain a number of private variables prefixed with an underscore (e.g. _) which are used internally by the API functions and should not be modified by the user. In addition to this the simulation API defines structures of arrays to hold agent and message list information. Agent lists are named `xmachine_memory_agent_*name*_list` and message lists are named ‘‘xmachine_message_message_*name*_list‘‘. These lists are passed as arguments to agent functions and should only be used in conjunction with the simulation API functions for message iteration and the adding of messages and agents. List structures should never be accessed directly as doing so will produce undefined behaviour.

A Basic Agent Function

The following example shows a simplistic agent function `function1` which has no message input or output and only updates the agents internal memory. All FLAME GPU agent functions are first prefixed with the macro definition `__FLAME_GPU_FUNC__`. In this basic example the agent function has only a single argument, a pointer to an agent structure of type `xmachine_memory_myAgent` called `xmemory`. In the below example the agent `name` is `myAgent` and the agent memory contains two variables `x` and `no_movements` of type `float` and `int` respectively. The return type of FLAME GPU functions is always `int`. A return value of anything other than `0` indicates that the agent has died and should be removed from the simulation (unless the agent function definition had specifically set the reallocate element value to false in which case any agent deaths will be ignored).

```
__FLAME_GPU_FUNC__ int function1(xmachine_memory_myAgent* xmemory)
{
    xmemory->x = xmemory->x += 0.01f;
    xmemory->no_movements += 1;
    return 0;
}
```

Use of the Message Output Simulation API

Within an agent function script, message output is possible by using a message output function. For each message type defined within the XMML model definition the dynamically generated simulation API will create a message output function of the following form;

```
add_message_*name*_message(message_*name*_messages, args...);
```

Where `*name*` refers to the value of the messages `name` element within the message specification and `args` is a list of named arguments which correspond to the message variables (see *Message Variables*). Agent functions may only call a message output function for the message name defined within the function definitions output (see *Agent Function Message Outputs*). This restriction is enforced as message output functions require a pointer to a message list which is passed as an argument to the agent function. Agents are only permitted to output at most a single message per agent function and repeated calls to an add message function will result in previous message information simply being overwritten. The example below demonstrates an agent function `output_message` belonging to an agent named `myAgent` which outputs a message with the name `location` defined as having four variables. For clarity the message output function prototype (normally found in `header.h`) is also shown.

```c
//header.h
add_location_message(xmachine_message_location_list* location_messages, int id, float x, float y, float z);
```

```c
//functions.c
__FLAME_GPU_FUNC__ int output_message(xmachine_memory_myAgent* xmemory, xmachine_message_location_list* location_messages)
{
    int id;
    float x, y, z;
    id = xmemory->id;
    x = xmemory->x;
    y = xmemory->y;
    z = xmemory->z;

    add_location_message(location_messages, id, x, y, z);

    return 0;
}
```

Use of the Message Input Simulation API

As with message outputs, iterating message lists (message input) within agent functions is made possible by the use of dynamically generated message API functions. In general two functions are provided for each named message, a `get_first_*name*_message(args...)` and `get_next_*name*_message(args...)` the second of which can be used within a while loop until it returns a `NULL` (`0`) value indicating the end of the message list. The arguments of these functions differ slightly depending on the partitioning scheme used by the message. The following subsections describe these in more detail. Regardless of the partitioning type a number of important rules must be observed when using the message functions. Firstly it is essential that message loop complete naturally. I.e. the `get_next_*name*_message` function must be called without breaking from the while loop until the end of the message list is reached. Secondly agent functions must not directly modify messages returned from the get message functions. Changing message data directly will result in undefined behaviour and will most likely crash the simulation

Non Partitioned Message Iteration

For non partitioned messages the dynamically generated message API functions are relatively simple and the arguments which are passed to the API functions are also required by all other message partitioning schemes. The get first message API function (i.e. `get_first_*name*_message`) takes only a single argument which is a pointer to a message list structure (of the form `xmachine_message_*name*_list`) which is passed as an argument to the agent function. The get next message API function (i.e. `get_next_*name*_message`) takes two arguments, the previously returned message and the message list. The below example shows a complete agent function `input_messages` demonstrating the iteration of a message list (where the message `*name*` is `location`). The while loop continues until the get next message API function returns a `NULL` (or false) value. In the below example the location message is used to calculate an average position of all the locations specified in the message list. The agent then updates three of its positional values to move toward the average location (cohesion).

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_myAgent* xmemory, xmachine_
↪message_location_list* location_messages)
{
    int count;
    float avg_x, avg_y, agv_z,

    /* Get the first location messages */
    xmachine_message_location* message;
    message = get_first_location_message(location_messages);

    /* Loop through the messages */
    while(message)
    {
        if((message->id != xmemory->id))
        {
            avg_x += message->x;
            avg_y += message->y;
            avg_z += message->z;
            count++;
        }

        /* Move onto next location message */
        message = get_next_location_message(message, location_messages);

    }

    if (count)
    {
        avg_x /= count;
        avg_y /= count;
        avg_z /= count;
    }

    xmemory->x += avg_x * SMALL_NUMBER;
    xmemory->y += avg_y * SMALL_NUMBER;
    xmemory->z += avg_z * SMALL_NUMBER;

    return 0;
}
```

Spatially Partitioned Message Iteration

For spatially partitioned messages the dynamically generated message API functions rely on the use of a Partition Boundary Matrix (PBM). The PBM holds important information which determines which agents are located within the spatially partitioned areas making up the simulation environment. Wherever a spatially partitioned message is defined as a function input (within the XMML model definition) a PMB argument should directly follow the input message list in the list of agent function arguments. As with non partitioned messages the first argument of the get first message API function is the input message list. The second argument is the PBM and the subsequent three arguments represent the position which the agent would like to read messages from (which in almost all cases is the agent position). The get next message API function differs only from the non partitioned example in that the PBM is passed as an additional parameter. The example below shows the same example as in the previous section but using a spatially partitioned message type (rather than the non partitioned type). The differences between the function arguments in the previous section are highlighted in red as is the use of a helper function `in_range`. The purpose of the `in_range` function is to check the distance between the agent position and the message. This is important as the messages returned by the get next message function represent any messages within the same or adjacent partitioning cells (to the position specified by the get first message API function). On average roughly $1/3$ of these values will be within the actually range specified by the message definitions range value.

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_location* xmemory, xmachine_
↪message_location_list* location_messages, xmachine_message_location_PBM* partition_
↪matrix)
{
    int count;
    float avg_x, avg_y, agv_z,

    /* Get the first location messages */
    xmachine_message_location* location_message;
    message = get_first_location_message(location_messages,
        partition_matrix,
        xmemory->x,
        xmemory->y,
        xmemory->z);
    /* Loop through the messages */
    while(message)
    {
        if (in_range(message, xmemory))
        {
            if((message->id != xmemory->id))
            {
                avg_x += message->x;
                avg_y += message->y;
                avg_z += message->z;
                count++;
            }
        }

        /* Move onto next location message */
        message = get_next_location_message(message,
        location_messages,
        partition_matrix);
    }
    if (count)
    {
        avg_x /= count;
        avg_y /= count;
        avg_z /= count;
    }
    xmemory->x += avg_x * SMALL_NUMBER;
    xmemory->y += avg_y * SMALL_NUMBER;
    xmemory->z += avg_z * SMALL_NUMBER;
    return 0;
}
```

Discrete Partitioned Message Iteration

For discretely partitioned messages the dynamically generated message API functions differ from those of non partitioned only in that two additional parameters must be passed to the get first message API function. The two integer arguments represent the position which the agent would like to read messages from within the cellular environment (as with spatially partitioning this is usually the agent position). These values of these arguments must therefore be within the width and height of the message space itself (the square of the messages `bufferSize`). In addition to the additional arguments, the discrete message API functions also make use of template parameterisation to distinguish between the type of agent requesting message information. The template parameters which may be used are either `DISCRETE_2D` (as in the example below) or `CONTINUOUS`. This parameterisation is required as underlying implementation of the message API functions differs between the two agent types. The example below shows an agent function (`input_messages`) of a discrete agent (named `cell`) which iterates a message list (of state messages) to count the number neighbours with a state value of `1`.

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_cell* xmemory, xmachine_
↪message_state_list* state_messages)
{
    int neighbours = 0;
    xmachine_message_state* state_message;
    message = get_first_state_message<DISCRETE_2D>(state_messages, xmemory->x,␣
↪xmemory->y);

    while(message){
        if (message->state == 1){
            neighbours++;
        }
        message = get_next_state_message<DISCRETE_2D>(message, state_messages);
    }
    xmemory->neighbours = neighbours;
    return 0;
}
```

Graph Edge Partitioned Message Iteration

For graph edge partitioned messages the dynamically generated message API functions rely on the use of a message boundary s

As with other message types the first argument is the input message list. The second argument is the message boundary structure, and the third argument is the id of the edge for which messages should be loaded (usually the edge where the agent is located). The `get_next` message API function differs only from the non partitioned example in that the message boundary structure is passed as an additional parameter.

The example below shows the use of graph edge partitioned messaging to access messages from the edge as the agent currently resides, counting the number of agents.

```
__FLAME_GPU_FUNC__ int input_messages(xmachine_memory_Agent* agent, xmachine_message_
↪location_list* location_messages, xmachine_message_location_bounds* message_bounds){
    // Initialise a variable
    unsigned int count = 0;

    // Get the first message from the message list for the target edge
    xmachine_message_location* current_message = get_first_location_message(location_
↪messages, message_bounds, agent->edge_id);

    // Loop through the messages
    while(current_message){
        // No need to check that the current_message->edge_id matches, as this is␣
↪guaranteed by the partitioning scheme
        // Increment the counter
        count++;

        // Get the next message from the message list.
        current_message = get_next_location_message(current_message, location_
↪messages, message_bounds);
    }

    // Store the count
    agent->count = count;

    return 0;
}
```

Message Type Macro Definition

To increase the portability of agent function scripts, a preprocessor macro is defined in `src/dynamic/header.h` detailing which message partitioning scheme is used for each message type.

I.e.

```
#define xmachine_message_MESSAGE_partitioningNone
#define xmachine_message_MESSAGE_partitioningDiscrete
#define xmachine_message_MESSAGE_partitioningSpatial
#define xmachine_message_MESSAGE_partitioningGraphEdge
```

These macros can then be used to write a single `functions.c` file which can be used with different partitioning shchemes in the `XMLModelFile.XML`.

```
#if defined(xmachine_message_MESSAGE_partitioningNone)
    __FLAME_GPU_FUNC__ int readMessages(xmachine_memory_agent* agent, xmachine_
↪message_MESSAGE_list* MESSAGE_messages){
#elif defined(xmachine_message_MESSAGE_partitioningSpatial)
    __FLAME_GPU_FUNC__ int readMessages(xmachine_memory_agent* agent, xmachine_
↪message_MESSAGE_list* MESSAGE_messages, xmachine_message_MESSAGE_PBM* partition_
↪matrix){
#endif
    // ...
    #if defined(xmachine_message_MESSAGE_partitioningNone)
        xmachine_message_MESSAGE* current_message = get_first_MESSAGE_
↪message(MESSAGE_messages);
    #elif defined(xmachine_message_MESSAGE_partitioningSpatial)
        xmachine_message_MESSAGE* current_message = get_first_MESSAGE_
↪message(MESSAGE_messages, partition_matrix, agent->x, agent->y, agent->z);
    #endif
    while (current_message) {
        // ...
        #if defined(xmachine_message_MESSAGE_partitioningNone)
            current_message = get_next_MESSAGE_message(current_message, MESSAGE_
↪messages);
        #elif defined(xmachine_message_MESSAGE_partitioningSpatial)
            current_message = get_next_MESSAGE_message(current_message, MESSAGE_
↪messages, partition_matrix);
        #endif
    }
    // ...
}
```

Use of the Agent Output Simulation API

Within an agent function script, agent output is possible on the host from Init and Step functions, and on the device by using a agent output API function.

Agent Creation from the Host

Within `__FLAME_GPU_INIT_FUNC` and `__FLAME_GPU_STEP_FUNC` (or within custom visualisation code) it is possible to generate one or more agents of a specific type and state, and transfer them to the device for the next simulation iteration.

Several steps must be followed to make use of this feature.

1. Allocate enough host (CPU) memory for all as many agents as you would like to create within the host function.
2. Populate the agent data on the host.
3. Copy agent data from the host to the device.
4. Deallocate host memory when it is no longer required.

If agents are only create in an `INIT` function, then the above procedure can be local to that `INIT` function.

If agents are going to be created in `STEP` functions, it is more efficient to split this procedure over an `INIT` function, a `STEP` function and an `EXIT` function. In this case, in `functions.c` you should declare a host memory in the global scope. An `INIT` function is then used to allocate sufficient memory, agents are created in the `STEP` function and lastly the `EXIT` function is used to deallocate and free resources.

If you are only creating a single agent of type `Agent` using the `default` state, the relevant data types and functions are:

```
// Declare a pointer to a single agent structure, and allocate the memory.
xmachine_memory_Agent * h_agent = h_allocate_agent_Agent();
// Populate the agent values as desired.
// Copy the single agent to the default in a synchronous operation.
h_add_agent_Agent_default(h_agent);
// Free the host memory when no longer required.
h_free_agent_Agent(&h_agent);
```

If you would like to create multiple (`N`) agents of type `Agent` to the `default` state in a single init/step function, the relevant data types and functions are:

```
// Declare a pointer to an array of agent structures.
xmachine_memory_Agent ** h_agent_AoS;
// Allocate enough memory on the host for N agents
h_agent_AoS = h_allocate_agent_Agent_array(N);
// Populate the agents as required.
// Copy the agents to the device. Here count is an integer less than or equal to N.
h_add_agents_Agent_default(h_agent_AoS, count);
// Deallocate memory.
// The total number of agents is required to avoid memory leaks.
h_free_agent_Agent_array(&h_agent_AoS, N);
```

For an example of this being used please see the `HostAgentCreation` example.

Note: Creating agents from the host is a relatively expensive process, as host to device memory copies are required. Higher performance is achieved when the number of copies as minimised, by batching creating multiple agents at once rather than many copies of individual agents.

Agent Creation from the Device

Agent functions can be defined with the `xagentOutputs` tag containing one or more `gpu:xagentOutput` tags, allowing the agent function to create new agents of the specified `<xagentName>` and `<state>`.

For each agent type defined within the XMML model definition the dynamically generated simulation code will create an agent output function of the following form;

```
add_*name*_agent(*name*_agents, args...);
```

Where `*name*` refers to the value of the agents `name` element within the agent specification and `args` is a list of named arguments which correspond to the agents memory variables (see *Agent Function X-Agent Outputs*). Agent functions may only output a single type of agent and are only permitted to output a single agent per agent function. As with message outputs, repeated calls to an add agent function will result in previous agent information simply being overwritten. The example below demonstrates an agent function (`create_agent`) for an agent named `myAgent` which outputs a new agent by creating a clone of itself. For clarity the agent output API function prototype (normally found in `header.h`) is also shown.

```
//header.h
add_myAgent_agent(xmachine_memory_myAgent_list* myAgent_agents, int id, float x, float y, float z);
```

```
//functions.c
__FLAME_GPU_FUNC__ int output_message(xmachine_memory_myAgent* xmemory, xmachine_memory_myAgent_list* myAgent_agents)
{
    int id;
    float x, y, z;
    id = xmemory->id;
    x = xmemory->x;
    y = xmemory->y;
    z = xmemory->z;
    add_myAgent_agent(myAgent_agents, id, x, y, z);
    return 0;
}
```

Generating new agent IDs

For many models it can be useful to provide each agent a unique identifier, to track progress in output files, or to avoid reading self-messages during message list iteration. FLAME GPU provides a mechanism to generate a new unique identifier, for host or device agent creation, if certain conditions are met.

Agent types must have an agent variable named `id`, of a signed or unsigned integer type (i.e, *int*, *unsigned int*, *unsigned long long int*, etc.) for the function to be generated.

For an agent type with name `agentName` with an `unsigned int` variable `id` , a new function `unsigned int generate_agentName_id()` will be generated. When called from a host function (init, step, exit) or device agent function this will return the next available id from the sequence.

The initial value for each agent type will either be *0* if no agents of that type are provided, or if the initial states file does contain agents, the maximum value plus one will be used. I.e. if `0.xml` contains agents with a maximum id value of `128`, the first call to `generate_agentName_id()` will return *129*.

It is possible to specify a custom starting point for the first id, using the generated `set_initial_agentName_id(unsigned int first)` function.

Using Random Number Generation

Random number generation is provided via the `rnd` API function which uses template parameterisation to distinguish between either discrete (where a template parameter value of `DISCRETE_2D` should be used) or continuous (where a template parameter value of CONTINUOUS should be used) spaced agents. If a template parameter value is not specified then the simulation will assume a DISCRETE_2D value which will work in either case but is more computationally expensive. The API function has a single argument, a pointer to a RNG_rand48 structure which contains random seeds and is passed to agent functions which specify a true value for the RNG element in the XMML function definition. The example below shows a simple agent function (with no input or outputs) demonstrating the random number generation to determine if the agent should die.

```
#define DEATH_RATE 0.1f

__FLAME_GPU_FUNC__ int kill_agent(xmachine_memory_myAgent* agent, RNG_rand48* rand48)
{
    float random;
    int die;

    die = 0; /* agent does not die */
    random = rnd<CONTINUOUS>(rand48);
    if (random < DEATH_RATE)
        die = 1; /* agent dies */

    return die;
}
```

Summary of Agent Function Arguments

Agent functions may use any combination of message input, output, agent output and random number generation resulting in a large number of agent function arguments which are expected to be in a specific and predefined order. The following pseudo code demonstrates the order of a function containing all possible arguments. When specifying an agent function declaration this order must be observed.

```
__FLAME_GPU_FUNC__ int function(xmachine_memory_*agent_name* *agent,
                                xmachine_memory_*agent_name* _list* output_agents,
                                xmachine_message_*message_name*_list* input_messages,
                                xmachine_message_*message_name*_PBM* input_message_
↪PBM,
                                xmachine_message_*message_name*_list* output_
↪messages,
                                RNG_rand48* rand48);
```

Host Simulation Hooks

Host simulation hooks functions which are executed outside of the main simulation iteration. More specifically they are called by CPU code during certain stages of the simulation execution. Host simulation Hooks should be defined in your *functions.c* file and should also be registered in the model description. There are numerous hook points (*init*, *step* and *exit*) which can are be explained in the proceeding sections.

Initialisation Functions (API)

Any initialisation functions defined within the XMML model file (see *Initialisation Functions*) is expected to be declared within an agent function code file and will automatically be called before the first simulation iteration. The initialisation function declaration should be preceded with a *__FLAME_GPU_INIT_FUNC__* macro definition, should have no arguments and should return void. The below example demonstrated an initialisation function named *initConstants* which uses the simulation APIs dynamically created constants functions to set a constant named *A_CONSTANT*.

```
__FLAME_GPU_INIT_FUNC__ void initConstants()
{
    float const_value = 8.25f;
    set_A_CONSTANT(&const_value);
}
```

Step Functions (API)

If a step function was defined in the XMMl model file (section *Step Functions*}) then it should be defined in a similar way to the initialisation functions as described above in section *Initialisation Functions (API)*. These functions will be called after each iteration step. An example is shown below. A common use of a step functions is to output logs from analytics functions when full agent XML output is not required. In this case an init or step function can be used for creating and closing a file handle respectively.

```
__FLAME_GPU_STEP_FUNC__ void some_step_func()
{
    do_step_operation();
}
```

Exit Functions (API)

If an exit function was defined in the XMMl model file (section *Exit Functions*) then it should be defined in a similar way to the initialisation and step functions as described above. It will be called upon finishing the program. An example is shown below.

```
__FLAME_GPU_EXIT_FUNC__ void some_exit_func()
{
    calculate_agent_position_average();
    print_to_file();
}
```

Note that Exit functions are not executed by the default visualisation.

Runtime Host Functions

Runtime host functions can be used to interact with the model outside of the main simulation loop. For example runtime host functions can be used to set simulation constants, gather analytics for plotting or sorting agents for rendering. Typically these functions are used within step, init or exit functions however they can also be used within custom visualisations. In addition to the functionality in this section it is also possible to create agents on the host which are injected into the simulation (see *Agent Creation from the Host*).

Getting and Setting Simulation Constants (Global Variables)

Simulation constants defined within the environment section of the XMML model definition (or the initial agents state file) may be directly referenced within an agent function using the name specified within the variables definition (see *Simulation Constants (Global Variables)*). It is not possible to set constant variables within an agent function, however, the simulation API creates methods for setting simulation constants which may be called from *Host Simulation Hooks*.

E.g. At start of the simulation (either manually or within an initialisation function) or between simulation iterations (for example as part of an interactive visualisation). The code below demonstrates the function prototype for setting a simulation constant with the name *A_CONSTANT*.

```
extern "C" void set_A_CONSTANT (float* h_A_CONSTANT);
```

The function requires a pointer to a host variable (or array in the case of an environment variable array). An equivalent function is created for the getting of simulation constants on the host. E.g.

```
extern "C" float* get_A_CONSTANT();
```

This function returns a host pointer to the variable (or array in the case of an environment variable array).

The functions for getting and setting constants are all declared using the *extern* keyword which allows them to be linked by externally compiled code such as a custom visualisation or custom simulation loop.

Getting Static Graph Data

To access data from a staticGraph defined in environment tag of the XMLModelFile, several functions are defined, which can be called from host or device functions. The following examples are for a graph named `GRAPH`.

The following 4 methods are always defined, for any graph.

```
// Get the number of vertices in the graph data structure, less than or equal to the␣
↪bufferSize
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_vertex_
↪count();

// Get the number of edges in the graph data structure, less than or equal to the␣
↪bufferSize
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_edge_
↪count();

// Get the index of the first edge which leaves a given vertex (index)
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_vertex_
↪first_edge_index(unsigned int index);

// Get the number of edges which leaves a given vertex (index)
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_vertex_
↪num_edges(unsigned int index);
```

In addition, for each member variable defined for each vertex, and each edge a function is defined, which returns the variable of the appropriate type. If the variable is an array an additional parameter is provided for the element of the array.

```
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_vertex_
↪VARIABLE(unsigned int vertexIndex);
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_vertex_
↪ARRAY(unsigned int vertexIndex, unsigned int element);

__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_edge_
↪VARIABLE(unsigned int edgeIndex);
__FLAME_GPU_HOST_FUNC__ __FLAME_GPU_FUNC__ unsigned int get_staticGraph_GRAPH_edge_
↪ARRAY(unsigned int edgeIndex, unsigned int element);
```

Sorting Agents

Each *CONTINUOUS* type agent can be sorted based on key value pairs which come from agent variables. This can be particularly useful for rendering. A function for sorting each agent (named **agent**) state list (in the below example the state is named *default*) is created with the folowing format.

```
void sort_*agent*_default(void (*generate_key_value_pairs)(unsigned int* keys,␣
↪unsigned int* values, xmachine_memory_*agent*_list* agents))
```

The function takes as an argument a function pointer to a GPU *__global__* function. This function it points to takes two unsigned int arrays in which it will store the resulting key and value data, and *xmachine_memory_*agent*_list* which contains a structure of arrays of the agent. This type is generated dynamically depending on the agent variables defined in the XML model file (*Agent Memory*). For an agent with two float variables *x* and *y*, it has the following structure:

```
struct xmachine_memory_*agent*_list
{
    float x [xmachine_memory_*agent*_MAX];
    float y [xmachine_memory_*agent*_MAX];
}
```

The value *xmachine_memory_agent_MAX* is the buffer size of number of agents (section *Defining an X-Machine Agent*). This struct can be accessed to assign agent data to the key and value arrays. The following example is given within a FLAME step function which sorts agents by 1D position

```
__global__ void gen_keyval_pairs(unsigned int* keys, unsigned int* values, xmachine_
↪memory_agent_list* agents) {
    int index = (blockIdx.x*blockDim.x) + threadIdx.x;

    //Number of agents
    const int n = xmachine_memory_agent_MAX;

    if (index < n) {
        //set value
        values[index] = index;
        //set key
        keys[index] = agents->x[index];
    }
}

__FLAME_GPU_STEP_FUNC__ void sort_func() {

    //Pointer function taking arguments specified within sort_agent_default
    void (*func_ptr)(unsigned int*, unsigned int*, xmachine_memory_agent_list*) = &
↪gen_keyval_pairs;

    //sort the key value pairs initialized within argument function
    sort_agent_default(func_ptr);

    //Since we run GPU code, make sure all threads are synchronized.
    cudaDeviceSynchronize();
}
```

Analytics Functions

A dynamically generated *reduce* function is made for all agent variables for each state. A dynamically generated *count*, *min* and *max* functions will only be created for single-value (not array) variables. Count functions are limited to *int* type variables (including short, long and vector type variants), min and max functions are limited to non vector type variables (e.g. no dvec2 type of variables). Reduce functions sum over a particular variable variable for all agents in the state list and returns the total. Count functions check how many values are equal to the given input and returns the quantity that match. These *analytics* functions are typically used with init, step and exit functions to calculate averages or distributions of a given variable. E.g. for agent agent with a *name* of *agentName*, *state* of *default* and an *int* variable name *varName* the following analytics functions will be created.

```
reduce_agentName_default_varName_variable();
count_agentName_default_varName_variable(int count_value);
min_agentName_default_varName_variable();
max_agentName_default_varName_variable();
```

Accessing Agent Data

As of FLAME GPU 1.5.0 it is possible to directly access agent data from the device in Host functions (Init, Step and Exit). It is not possible to modify agent data from host functions.

For each agent type (`AGENT`), state (`STATE`) and each agent variable (`VARIABLE`) a function is generated to access the variable, i.e. `get_AGENT_STATE_variable_VARIABLE(index)` which returns the value, by transparently copying the agent data from the host to the device, for the given variable from the relevant state list, which has potentially significant performance implications.

For non-array agent variables, a single argument `index` refers to the position within the state list for the requested agent. Out of bounds accesses will return the default value for the agent variable.

For array agent variables, two arguments are required, `index` and `element`, where `index` is the agent position within the state list, and `element` is the 0 indexed element of the agent array. I.e. `get_AGENT_STATE_variable_ARRAY(0, 2)` would return the 2nd element of the agent variable `ARRAY` for the 0th `AGENT` agent in the state `STATE`.

This enables the creation of custom agent output functions as step functions, if you do not require all agent data in output XML files. For instance, it can be used to create a CSV file. See the `customOutputStepFunc` step function for the `HostAgentCreation` example.

Exiting the Simulation Early

It is possible to exit the simulation earlier than specified (specified as command-line argument for console mode, or on exit for visualisation mode). This is done by calling `set_exit_early()` from CPU code in one of the runtime host functions. When called, the remainder of the simulation iteration is called, then exit functions are called and the simulation ends. To check the status of this, `get_exit_early()` can be called, which returns a boolean value of true if it set to exit after this simulation iteration.

An example of when this function is useful is if the simulation has a fixed end state. Start by running the simulation for many more iterations than necessary. Upon reaching the desired system state, which can be checked by *Accessing Agent Data*, call `set_exit_early()` to avoid simulating more iterations than necessary.

Instrumentation for timing and population sizes

It is possible to obtain information of population and timings of different functions by taking advantage of CUDA timing events. Per-iteration and per-function (init/agent/step/exit functions) timing using CUDA events, and also the population size for each agent state per iteration printed to *stdout*.

This instrumentation is enabled with a set of defines. The value must be a positive non-zero integer (i.e. 1) to be enabled.

When enabled, the relevant measures are printed to *stdout*, which can then later be parsed (or redirected) to produce graphs, etc.

```
#define INSTRUMENT_ITERATIONS 1
#define INSTRUMENT_AGENT_FUNCTIONS 1
#define INSTRUMENT_INIT_FUNCTIONS 1
#define INSTRUMENT_STEP_FUNCTIONS 1
#define INSTRUMENT_EXIT_FUNCTIONS 1
#define OUTPUT_POPULATION_PER_ITERATION 1
```

will print out, for example (using the circles benchmark model)

```
processing Simulation Step 1
Instrumentation: Circle_outputdata = 0.304128 (ms)
Instrumentation: Circle_inputdata = 16.849920 (ms)
Instrumentation: Circle_move = 0.261120 (ms)
FLAME GPU Step function. Average circle position is (4115.978027, 4139.279785, 512.
↪000000)
Instrumentation: stepFunction = 27.652096 (ms)
agent_Circle_default_count: 1024
Instrumentation: Iteration Time = 46.309376 (ms)
Iteration 1 Saved to XML
```

CHAPTER 4

FLAME GPU Simulation and Visualisation

Introduction

The processes of building and running a simulation is made easier described within this chapter as are a number of tools and procedures which simplify the simulation code generation and compilation of simulation executables. In order to use the FLAME GPU SDK it should be placed in a directory which does not contain any spaces (preferably directly within the C: drive or root or root operating system drive). The host machine must also be running windows with a copy of the .NET runtime (used within the XSLT template processor) and must contain NVIDIA GPU hardware with Compute level 1.0.

Generating a Functions File Template

Chapter *Summary of Agent Function Arguments* previously described the exact argument order for agent function declarations however in most cases it is sensible to use the provided XSLT template `functions.xslt` located in the `FLAMEGPU/templates` directory within the FLAME GPU SDK) to generate a agent function source file with empty agent function declarations automatically using your XMML model file. Once this has been generated the agent function scripts can be implemented within the function declarations rather easily. Care must however be taken in ensuring that if the XMML model file is later modified that the agent function arguments are updated manually where necessary. Likewise be careful not to overwrite any existing function source file when generating a new one using the XSLT template. Generation of blank function source files is not incorporated into the visual studio template project and must be manually accomplished. A .NET based XSLT processor is provided within the FLAME GPU SDK for this purpose (`XSLTProcessor.exe` located in the `tools` directory) and can be used via the command line as follows (or via the `GenerateFunctionsFileTemplate` batch file located in the `tools` directory of the FLAME GPU SDK);

```
XSLTProcessor.exe XMLModelFile.xml functions.xslt functions.c
```

Alternatively any compliant XSLT processor such as Xalan, Unicorn or even Firefox web browser can be used.

FLAME GPU Template Files

The FLAME GPU SDK contains a number of XSLT templates which are used to generate the dynamic simulation code. A brief summary of the functionality and contents of each template file is as follows:

- `header.xslt` This template file generates a header file which contains any agent and message data structures which are common in many of the other dynamically generated simulation source files. The template also generates function prototypes for simulation functions and functions which are visible externally within custom C or C++ code.
- `main.xslt` This template file generates a source file which defines the main execution entry point function which is responsible for handling command line options and initialising the GPU device.
- `io.xslt` This template file generates a source file which contains functions for loading initial agent XML data files (see *Initial XML Agent Data*) into the simulation and saving the simulation state back into XML format.
- `simulation.xslt` This template file generates a source file containing the host side simulation code which includes loading data to and from the GPU device and making a number of CUDA kernel calls which perform the simulation process.
- `FLAMEGPU\_kernels.xslt` This template file generates a CUDA header file which contains the CUDA kernels and device functions which make up the simulation.
- `visualisation.xslt` This template file generates a source file which will allow basic visualisation of the simulation using sphere based representation of agents in 3D space. The source file is responsible for CUDA OpenGL interoperability and rending using OpenGL. The source file includes a *visualisation.h* file containing a number of definitions and variables which is not generated by any templates and should be specified manually.

Compilation Using Visual Studio

The FLAME GPU SDK and examples are targeted at a specific CUDA and Visual Studio version. The Visual Studios XML editor includes validation support and XML tag auto completion which makes defining an XMML model incredibly easy. The following subsections describe the various aspects of a FLAME GPU project file and describe the build processes.

Visual Studio Project Build Configurations

The FLAME GPU examples and template project file contain build configurations 64 bit Windows (`x64`) environments. 32 bit windows has been removed due to limitations on GPU memory addressing since version 1.3.0. For each platform the project also contains four configurations for debugging (`Debug`) and release versions (`Release`) of both `console` based simulation and `visualisation` simulation. The two debug options disable all compiler optimisations and generate debug information for debugging host (non GPU) code and enables CUDA device emulation for GPU (device) debugging. The visualisation configurations enable building of visualisation code and specify a pre processor macro (`VISUALISATION`) which is used by a number of pre-processor conditionals to change the simulations expected arguments (see *Simulation Execution Modes and Options*).

Visual Studio Project Virtual File Structure

Within the FLAME GPU examples and template projects code is organised into the following virtual folders;

- `FLAME GPU` Consisting of a folder containing the FLAME GPU XML schemas and Code generating templates. These files are shared amongst all examples so editing them will change simulation code generated for other projects.

- `FLAMEModel` Contains the XMML model file and the agent functions file (usually called `functions.c`). Note that the `functions.c` file is actually excluded from the build processes as it is built by the dynamically generated `simulation.cu` source file which includes it.
- `Dynamic Code` Contains the dynamically generated FLAME GPU simulation code. This code will be overwritten each time the project is built so any changes to this files will be lost unless template transformation is turned off using the FLAME GPU build rule (see *FLAME GPU Build Rule Options*).
- *Additional Source Code* This folder should contain any hard coded simulation specific source or header files. By default the FLAME GPU project template defines a single `visualisation.h` file in this folder which may be modified to set a number of variables such as viewing distance and clipping. Within the FLAME GPU examples this folder is typically used to sore any model specific visualisation code which replaces the dynamically generated visualisation source file.

The physical folders of the SDK structure a self explanatory however it is worth noting that executable files generated by the Visual Studio build processes are output in the SDKs `bin` folder which also contains the CUDA run time `dlls`.

Build Process

The Visual Studio build process consists of a number of stages which call various tools, compilers and linkers. The first of these is the FLAME GPU build tool (described in more detail in the following section) which generates the dynamic simulation code from the FLAME GPU templates and mode file. Following this the simulation code (within the Dynamic Code folder) is built using the CUDA build rule which compiles the source files using the NVIDIA CUDA compiler `nvcc`. Finally any C or C++ source files are compiled using MSVC compiler and are then linked with the CUDA object files to produce the executable. To start the build processes select the `Build` menu followed by `Build Solution` or use the `F7` hotkey. If the first build step in the Visual Studio skips the FLAME GPU build tool a complete rebuilt can be forced by selecting the `Build` menu followed by `Rebuild Solution` (or `Ctrl + Alt + F7`).

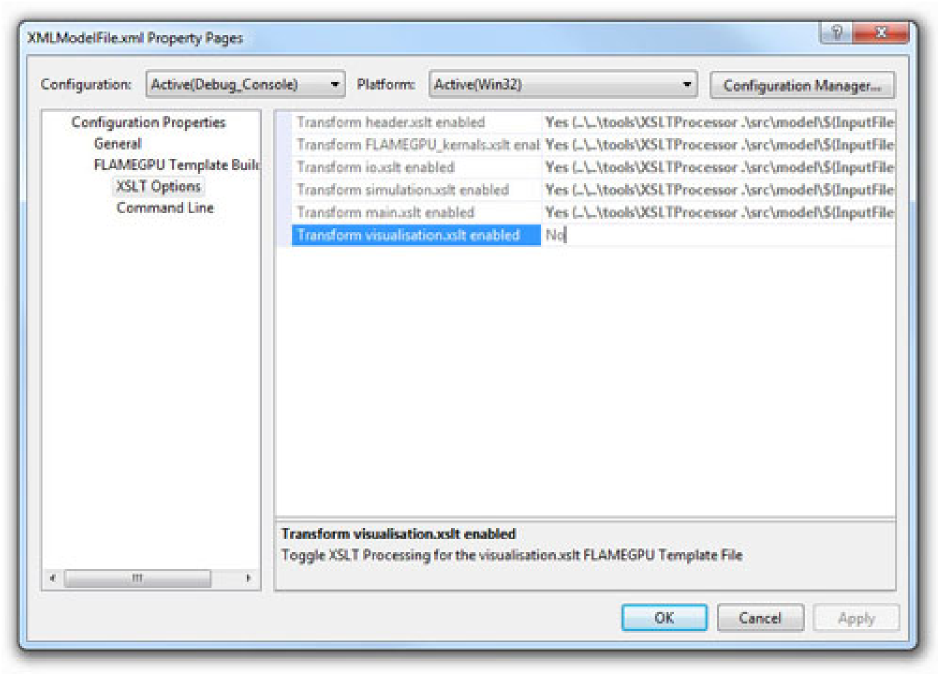
FLAME GPU Build Rule Options

The FLAME GPU build rule is configured by selecting the XMML model file properties. Within the Build rule the XSLT options tab (see Figure) allows individual template file transformations to be toggled on or off. These options are configuration specific and therefore console configurations by default do not processes the visualisation template.

Visual Studio Launch Configuration Command Arguments

In order to set the execution arguments (described in the next section) for simulation executable in any one of one of the four launch configurations, the `Command Arguments` property can be set form the Project Properties Page (Select `Project` Menu followed by `FLAMEGPU\_Project Properties`). The `Command Arguments` property is located under `Configuration Properties -> Debug` (see *Agent Function Scripts and the Simulation API*). Each configuration has its own set of `Command Arguments` so when moving between configurations these will need to be set. Likewise the `Configuration Properties` are computer and user specific so these cannot be preset and must be specified the first time each example is compiled and run. The Visual Studio macro `$InputDir` can be used to specify the working directory of the project file which makes locating initial agent data XML files for many of the examples much easier (these are normally located in the iterations folders of each example).

The Command Arguments have been set the simulation executable can be launched by selecting `Start Debugging` from the `Debug` menu or using the `F5` hotkey (this is the same in both release and debug launch configurations).



XMLModelFile.xml Property Pages

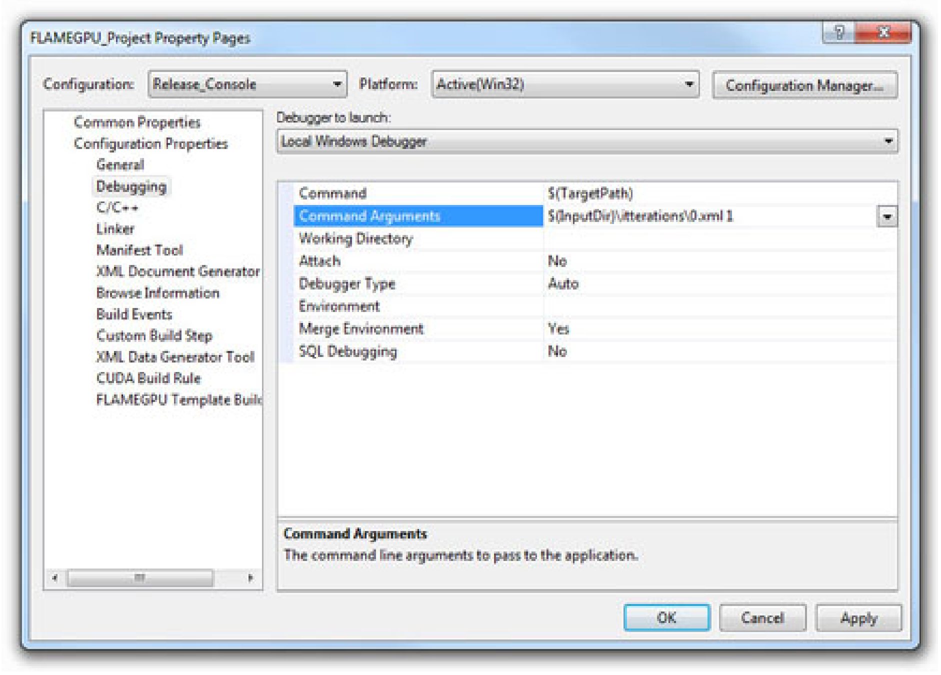
Configuration: Active(Debug_Console) Platform: Active(Win32) Configuration Manager...

- Configuration Properties
  - General
  - FLAMEGPU Template Buil
    - XSLT Options
    - Command Line

Transform header.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform FLAMEGPU_kernals.xslt enal	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform io.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform simulation.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform main.xslt enabled	Yes (..\..\tools\XSLTProcessor .\src\model\$(InputFile
Transform visualisation.xslt enabled	No

Transform visualisation.xslt enabled
Toggle XSLT Processing for the visualisation.xslt FLAMEGPU Template File

Fig. 4.1: FLAME GPU Build Rule XSLT Options Tab



FLAMEGPU_Project Property Pages

Configuration: Release_Console Platform: Active(Win32) Configuration Manager...

- Common Properties
- Configuration Properties
  - General
  - Debugging
  - C/C++
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - XML Data Generator Tool
  - CUDA Build Rule
  - FLAMEGPU Template Build

Debugger to launch:
Local Windows Debugger

Command	$(TargetPath)
Command Arguments	$(InputDir)\iterations\0.xml 1
Working Directory	
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No

Command Arguments
The command line arguments to pass to the application.

Compilation using Make (for Linux and Windows)

`make` can be used to build FLAME GPU under linux and windows (via a windows implementation of `make`).

Makefiles are provided for each example project `examples/project/Makefile`), and for batch building all examples (`examples/Makefile`).

To build a console example in release mode:

1. Download the FLAME GPU SDK release or alternatively clone the project using Git (it will be cloned into the folder `FLAMEGPU`):

   cd examples/EmptyExample/ make console

Or for a visualisation example in release mode:

   cd examples/EmptyExample/ make Visualisation

Debug mode executables can be built by specifying `debug=1` to make, `make all debug=1`. The generated executable can then be debugged using `cuda-gdb`.

In the project specific portion of the Makefile (i.e `examples/EmptyExample/Makefile`) several variables exist which allow the project to be customised.

- `EXAMPLE`: Controls the name of the project / executables generated.
- `HAS_VISUALISATION`: Determins if a visualisation mode should be supported or not.
- `CUSTOM_VISUALISATION`: Determins if a custom or the default visualisation should be used.
- `FLAMEGPU_ROOT`: The relative path from the Makefile to the main `FLAMEGPU` directory. I.e. `../../`
- `EXAMPLE_BIN_DIR`: Path to the location to place executables.
- `EXAMPLE_BUILD_DIR`: Path to the build directory for this project.
- `SMS`: Set the CUDA Compute Capabilities to build executables for
- `TRANSFORM_*_XLS`: Prevent the relevant `XSLT` file from being transformed
  - `TRANSFORM_HEADER_XSLT_DISABLED`: `header.xslt`
  - `TRANSFORM_FLAMEGPU_KERNALS_XSLT_DISABLED`: `flamegpu_kernals.xslt`
  - `TRANSFORM_IO_XSLT_DISABLED`: `io.xslt`
  - `TRANSFORM_SIMULATION_XSLT_DISABLED`: `simulation.xslt`
  - `TRANSFORM_MAIN_XSLT_DISABLED`: `main.xslt`
  - `TRANSFORM_VISUALISTION_XSLT_DISABLED`: `visualistion.xslt`

For more information on building FLAME GPU via make, run `make help` in an example directory.

Creating a New FLAME GPU Example Project

The simplest way to create a new FLAME GPU example project is to copy and modify an existing project, renaming visual studio solution / project files, and modifying the Makefile.

A python script is provided to simplify this process for you, makeing the required changes. I.e. to create a new example projected called `NewExample`, based on the `EmptyExample` run the following command.

```
python tools/new_example.py --base EmptyExample NewExample
```

Simulation Execution Modes and Options

FLAME GPU simulations require a number of arguments depending on either console or visualisation mode. Both are described in the following subsections.

Console Mode

Simulation executables built for console execution require two arguments, with several optional arguments.

```
usage: EmptyExample [-h] [--help] input_path num_iterations [cuda_device_id] [XML_
↪output_override]

required arguments:
  input_path           Path to initial states XML file OR path to output XML directory
  num_iterations       Number of simulation iterations

options arguments:
  -h, --help           Output this help message.
  cuda_device_id       CUDA device ID to be used. Default is 0.
  XML_output_override  Flag indicating if iteration data should be output as XML
                       0 = false, 1 = true. Default 1
```

The result of running the simulation will be a number of output XML files which will be numbered from `1` to `n`, where `n` is the number of simulations specified by the `Iterations` argument. It is possible to turn XML output on or off by changing the definition of the `OUTPUT_TO_XML` macro located within the main.xslt template to true (`1`) false (`0`).

Visualisation Mode

Simulation executables built for visualisation require only a single argument (usage shown below) which is the same as the first argument for with console execution (an initial agent XML file). The number of simulations iterations is not required as the simulation will run indefinitely until the visualisation is closed. As with console execution there are additional optional arguments available.

```
usage: EmptyExample [-h] [--help] input_path [cuda_device_id]

required arguments:
  input_path           Path to initial states XML file OR path to output XML directory

options arguments:
  -h, --help           Output this help message.
  cuda_device_id       CUDA device ID to be used. Default is 0.
```

Many of the options for the default visualisation are contained within the `visualisation.h` header file and include the following;

- `SIMULATION_DELAY` Many simulations are executed extremely quickly making visualisation a blur. This definition allows an artificial delay by executing this number of visualisation render loops before each simulation iteration is processed.
- `WINDOW_WIDTH` and `WINDOW_HEIGHT` Specifies the size of the visualisation window
- `NEAR_CLIP` and `FAR_CLIP` Specifies the near an far clipping plane used for OpenGL rendering.
- `SPHERE_SLICES` The number of slices used to create the sphere geometry representing a single agent in the visualisation.

- `SPHERE_STACKS` The number of stacks used to create the sphere geometry representing a single agent in the visualisation.
- `SPHERE_RADIUS` The physical size of the sphere geometry representing a single agent in the visualisation. This will need to be a sensible value which corresponds with the environment size and agent locations within your model/simulation.
- `VIEW_DISTANCE` The camera viewing distance. Again this will need to be a sensible value which corresponds with the environment size and agent locations within your model/simulation.
- `LIGHT_POSITION` The visualisation will contain a single light source which will be located at this position.
- `PAUSE_ON_START` If defined the simulation is paused on launch, allowing the simulation to be visualised one iteration at a time.

The colour of spheres in the default visualisation is determined using an agent variable `colour` (or alternatively `type` or `state`, however `colour` is the preferred option.) This can be an `int` or a `float`, with a set of distinct colours available, using the following defined values:

- `FLAME_GPU_VISUALISATION_COLOUR_BLACK`
- `FLAME_GPU_VISUALISATION_COLOUR_RED`
- `FLAME_GPU_VISUALISATION_COLOUR_GREEN`
- `FLAME_GPU_VISUALISATION_COLOUR_BLUE`
- `FLAME_GPU_VISUALISATION_COLOUR_YELLOW`
- `FLAME_GPU_VISUALISATION_COLOUR_CYAN`
- `FLAME_GPU_VISUALISATION_COLOUR_MAGENTA`
- `FLAME_GPU_VISUALISATION_COLOUR_WHITE`
- `FLAME_GPU_VISUALISATION_COLOUR_BROWN`

Creating a Custom Visualisation

Customised visualisation can easily be integrated to a FLAME GPU project by extending the automatically generated visualisation file (the output of processing `visualisation.xslt`). *Note: When doing this within Visual Studio it is important to turn off the template processing of the ‘‘visualisation.xslt‘‘ file in each of the launch configurations as processing them will overwrite any custom code!*. Many of the FLAME GPU SDK examples use customised visualisations in this way. As with the default visualisations any custom visualisation must define the following function prototypes defined in the automatically generated simulation header.

```
extern "C" void initVisualisation();

extern "C" void runVisualisation();
```

The first of these can be used to initialise any OpenGL memory and CUDA OpengGL bindings as well as displaying the user interface. The second of these functions must take control of the simulation by repeatedly calling the draw and singleIteration (which advances the simulation by a single iteration step) functions in a recursive loop. A more detailed description of the default rendering technique is provided within other FLAME GPU documentation (listed in *Purpose of This Document*).

Performance Tips

The GPU offers some enormous performance advantages for agent simulation over more traditional CPU based alternatives. With this in mind it is possible to write extremely sub optimal code which will reduce performance. The following is a list of performance tips for creating FLAME GPU model files;

General Usage of FLAME GPU

- FLAME GPU is optimal where there are very large numbers of relatively simple agents which can be parallelised.
- Populations of agents with very low numbers will perform poorly (in extreme cases slower than if they were simulated using the CPU). If you require an agent population with very few agents consider writing some custom CPU simulation code and transferring any important information into simulation constants to be read by larger agent populations during the FLAME GPU simulation step.
- Outputting information to disk (XML files) is painfully slow in comparison with simulation speeds so consider outputting information visually or only after larger numbers of simulation iterations.

Model Specification

- Minimise the number of variables with agents and message data where possible.
- Try to conceptualise and fully specify the model before completing the agent functions script to avoid making mistakes with agent function arguments. Try to think in terms of X-Machines agents!

Agent Function Scripting

- Small compute intensive agent functions are more efficient than functions which only iterate messages. Try to minimise the number of times message lists are iterated.
- Keep agent functions small and do not define more local variables than is strictly required. Reuse local variables where possible if they are no longer needed and before they go out of scope.

Message Iteration

- For small populations of agents (generally less than 2000 but dependant on hardware and the model) non partitioned messaging has less overhead and is similarly comparable to spatial partitioning.
- For large populations of distributed agents with limited communication spatially partitioned message communication will be much faster.

Detailed profiling using NVTX

Additional profiling information can be exported for the visual profiler using the Nvidia Tools Extension Library (NVTX). NVTX markers and ranges can be optionally enabled to provide enhanced profiling.

Enabling NVTX Markers via makefile

To achieve this using the `Makefile`, simply add `profile=1` as an argument to make, on any platform:

```
make console profile=1
```

Enabling NVTX Markers in Visual Studio

To enable NVTX markers in visual studio the solution must be modified to add the relevant definition, include path and linker flags as follows:

- **`C/C++ > Preprocessor > Preprocessor Definitions`**
  - Add `PROFILE`
- **`CUDA C/C++ > Common > Additional Include Directories`**
  - Add `$(NVTOOLSEXT_PATH)include`
- **`Linker > General > Additional Library Directories`**
  - Add `$(NVTOOLSEXT_PATH)lib/x64`
- **`Linker > Input > Additional Dependencies`**
  - Add `nvToolsExt64_1.lib`

Parameter Exploration

Agent Based Simulations typically have many parameters which control certain aspects of the simulation, which can be used for calibration. As of FLAME GPU 1.5.0 the simplest method to achieve this is to use multiple initial states files for separate simulations which contain different values for environmental constants, and run the simulation on each of the files.

For instance, for a model with 2 environmental constants representing model parameters called `SEED` and `INIT_POPULATION` which are defined in `XMLModelFile.XML` within the `<gpu:environment>` tag as follows:

```xml
<gpu:constants>
  <gpu:variable>
    <type>unsigned int</type>
    <name>SEED</name>
    <defaultValue>0</defaultValue>
  </gpu:variable>
  <gpu:variable>
    <type>unsigned int</type>
    <name>INIT_POPULATION</name>
    <defaultValue>1</defaultValue>
  </gpu:variable>
</gpu:constants>
```

If we wish to run this with `SEED` values `0`, `1` & `2` and `INIT_POPULATION` values `10`, `100` and `1000` this could be achieved with 9 initial states files (stored in separate folders to avoid overwriting output). A script could be used to create these files for large parameter sweeps.

This could have the following structure:

```
iterations
    - 0-10
    |   - 0.xml
    - 0-100
    |   - 0.xml
    - 0-1000
    |   - 0.xml
    - 1-10
    |   - 0.xml
    - 1-100
    |   - 0.xml
    - 1-1000
    |   - 0.xml
    - 2-10
    |   - 0.xml
    - 2-100
    |   - 0.xml
    - 2-1000
        - 0.xml
```

The contents of each file would then be different. Assuming agents are created via an `INIT` function, each `0.xml` file could look as follows.

`0-10/0.xml` would contain:

```xml
<states>
    <itno>0</itno>
    <environment>
        <SEED>0</SEED>
        <INIT_POPULATION>10</INIT_POPULATION>
    </environment>
</states>
```

`0-100/0.xml` would contain:

```xml
<states>
    <itno>0</itno>
    <environment>
        <SEED>0</SEED>
        <INIT_POPULATION>100</INIT_POPULATION>
    </environment>
</states>
```

`0-1000/0.xml` would contain:

```xml
<states>
    <itno>0</itno>
    <environment>
        <SEED>0</SEED>
        <INIT_POPULATION>1000</INIT_POPULATION>
    </environment>
</states>
```

And so on. Simulations could then be launched in batch via a script, either sequentially or concurrently depending upon the memory requirements of each model, and the availability of GPUs.