Received: 31 March 2022 | Revised: 7 February 2023 | Accepted: 23 March 2023

DOI: 10.1002/spe.3207

RESEARCH ARTICLE

WILEY

# FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs

**Paul Richmond**[1] | **Robert Chisholm**[1] | **Peter Heywood**[1] | **Mozhgan Kabiri Chimeh**[2] | **Matthew Leach**[1]

[1]The Department of Computer Science, University of Sheffield, Sheffield, UK

[2]NVIDIA, Bristol, UK

**Correspondence**
Paul Richmond, The Department of Computer Science, University of Sheffield, Sheffield, UK.
Email: p.richmond@sheffield.ac.uk

**Funding information**
Engineering and Physical Sciences Research Council, Grant/Award Number: EP/N018869/1; European Commission, Grant/Award Numbers: 777123, 826494

**Abstract**

Agent based modelling (ABM) offers a powerful abstraction for scientific study in a broad range of domains. The use of agent based simulators encourages good software engineering design such as separation of concerns, that is, the uncoupling of the model description from its implementation detail. A major limitation in current approaches to ABM simulation is that of the trade off between simulator flexibility and performance. It is common that highly optimised simulations, such as those which target graphics processing units (GPU) hardware, are implemented as standalone software. This work presents a software framework (FLAME GPU 2) which balances flexibility with performance for general purpose ABM. Methods for ensuring high computational efficacy are demonstrated by, minimising data movement, and ensuring high device utilisation by exploiting opportunities for concurrent code execution within a model and through the use of ensembles of simulations. A novel hierarchical sub-modelling approach is also presented which can be used to model certain types of recursive behaviours. This feature is shown to be essential in providing a mechanism to resolve competition for resources between agents within a parallel environment which would otherwise introduce race conditions. To understand the performance characteristics of the software, a benchmark model with millions of agents is used to explore the use of simulation ensembles and to parametrically investigate concurrent code execution within a model. Performance speedups are demonstrated of 3.5× and 10× respectively over a baseline GPU implementation. Our hierarchical sub-modelling approach is used to demonstrate the implementation of a recursive algorithm to resolve competition of agent movement which occurs as a result of agent desire to simultaneously occupy discrete areas high in a 'resource'. The algorithm is used to implement a classical socio-economics model, Sugarscape, with populations of up to 16M agents.

**KEYWORDS**

agent based modelling, GPUs, large scale simulation

## 1 | MAIN

Agent based modelling (ABM) presents a powerful abstraction for modelling complex systems which has a emphasis on the specification of behaviour from the bottom up. Agents are described using behaviours at the individual level, rather than attempting to model system level phenomena from the top down. The process of simulation allows individuals to interact within a virtual environment in which the system dynamics often demonstrate important emergent phenomena. The ABM approach is analogous with micro-simulation and has been the subject of extensive reviews with respect to; domains and methods,[1] and simulation tooling.[2] Reference 1 describes the benefits of an agent based system as being; (i) ABM captures emergent phenomena; (ii) ABM provides a natural description of a system; and (iii) ABM is flexible. As a result agent based systems have seen widespread use in a broad range of domains. Computational biology in particular has seen a growth in the use of bottom up approaches.[3] Many of these do not use the explicit terminology of ABM and simulation, for example, spiking neural simulation,[4] but share the same modelling principles of microscopic modelling and observation of emergent properties through simulation.

With the emphasis of behaviour specification at an individual level, agent based simulation is computationally expensive. As a result there are an increasing number of research papers which apply parallelisation approaches to agent based simulations to achieve greater performance.[5] Parallelisation can permit larger ABM and simulation studies to be undertaken, both in terms of model size and in the exploration of model parameters. With respect to the latter, independent simulations of the same model under different parameters can be parallelised via traditional distributed parallelism. Improving the performance of larger and more complex of models presents a much greater challenge. More performant single model simulation requires efficient use of dense computing nodes or designing approaches in which distributed models can avoid the latency of data movement between logical processors.[5-7] A popular and generalisable approach towards higher computational efficiency has been the application of hardware accelerators such as graphics processing units (GPUs) and field programmable gate arrays (FPGAs) to large scale computational challenges.[8] GPUs in particular have had an explosive growth in prevalence, and usage, within top supercomputers (see the Top500 project[*]) and in desktop computing. This has been prompted by a number of factors including; early adaptation of the architectures for general purpose usage,[9] investment in tools development, delivery of dense compute within a smaller energy window than CPU counterparts, and importantly the ability for GPUs to be deployed with extraordinary efficiency towards the popular area of artificial intelligence (AI). An accelerator's ability to demonstrate higher performance does however come at a cost which is usually borne by the developer. GPUs are throughput, rather than latency oriented and as such require skilled approaches to programming them to hide the costs of (mostly memory) latency in order to maximise the GPUs performance benefits. The recent surge in AI on GPUs has no doubt benefited from the development of excellent tooling[10,11] which abstracts the complexities of GPU architecture, and indeed any concept of parallelism, entirely away from the end user. It is the aim of this work to provide a similar quality of tooling to allow non specialists to accelerate ABM simulations using GPUs. There have been a number of attempts to use accelerators for agent based simulation of specific models, with Reference 12 providing a review of approaches. It is concluded that although a significant number of specific models have implemented aspects of GPU parallelism only a very small number have attempted to integrate these approaches into a flexible framework[13,14] for use by general modellers without a background in GPU development. Given the complexity of development, a generalised approach to ABM which allows for simple execution on parallel hardware is preferable.

In previous work we described the first version of the FLAME GPU framework,[15] a generalised template driven ABM and simulation framework which attempted to abstract the GPUs complexity. Central to this approach was the use of state based representation which permits modellers to describe an agent based system as a communicating stream X-machine,[16] a form of state machine with internal memory, updated by functions communicating via a global mechanism. The state based description of a model requires a set of agents, each with a set of state variables (their memory), states and functions. Functions transition an agent from one state to another by updating agent memory by performing a scripted set of mathematical operations. Functions can include inter-agent communication by reading and writing to a set of global message lists. Each message list groups a set of values into a message object providing an abstraction of indirect agent communication via messages. State transition functions and their use of messages, along with any functional dependencies (e.g., function A must occur before function B) allows a directed acyclic graph (DAG) to be inferred which dictates the required execution order of functions within the model. State based representation is less common for agent modelling than object-orientation model design.[17] Although object-orientation is orthogonal to state based representation, functions within an object are not typically coupled with the object state and as such inferring dependencies between

[*]http://www.top500.org

objects/agents and hence the process of parallelisation is not straight-forward. Object-oriented agent based simulation typically loops through agent objects sequentially, randomly permuting agent order, and performing execution of specific object member functions. Conversely, the approach of state based representation is inherently parallel. The generated DAG allows functions to become analogous to GPU kernels and the use of indirect communication through messages prevents race conditions. The previous version of FLAME GPU used XML templates to generate a CUDA GPU implementation from a state based model description and was successful in demonstrating high performance without a modeller having to understand the intricacies of data parallelism and GPU optimisation. A limiting factor of previous work was that the state based representation enforced users to describe the model in a declarative XML format rather than using a more intuitive API. In addition, the transition of a model to a state based representation was conceptually difficult for models designed upon serial concepts. Well known examples of serial models can be observed in discrete space examples such as Sugarscape[18] or Schellings model of segregation.[19] In both of these models, agents exist within a discrete 2D environment where movement of agents to areas which are favourable in terms of resource, is determined through a serial process in which agent order is randomised for fairness. Within a parallel implementation of this model conflict naturally occurs as numerous agents compete to try to move to desirable locations. More generally, conflicts can occur in the case of other agent interactions, for example, in pair-wise bonding of agents which may occur in biological models[20] and which requires a process to handle situations where multiple agents desiring pairing with another. In each case movement or conflict must be resolved explicitly within the model's design in order to reproduce the equivalent behaviour of the serial version.

In this article, we present FLAME GPU 2. A complete rewrite of the FLAME GPU software which permits more flexible API driven modelling without sacrificing performance. The contribution of the article is unique in that the software provides general purpose agent based simulations capable of transparently leveraging GPUs. The article provides an overview of the software architecture and then presents four novel methods that enable performant and flexible large-scale simulation of general ABMs on GPU architectures. The proposed methods address the two challenges identified above, namely; high performance simulation for a general modelling case, and, abstraction of parallelism within state based representation to provide robust conflict resolution. Our first method of achieving high simulation performance is realised by considering efficient data movement during simulation. The approach proposes movement of data at a fine grained, variable level (as opposed to the agent level), as well as an extensible design for message communication to support agents within different forms of continuous and discrete environments. The second and third methods focus on the ability to better utilise the high level of parallelism available on modern GPUs. In 10 years the number of CUDA processing cores has increased by $\sim 14\times$[†]. This increase in raw performance offers the potential to simulate larger and more complex agent based systems however it also poses a challenge of ensuring good device utilisation of smaller agent based models when utilising an agent per thread mapping the GPU device. Our second method improves device utilisation for small populations by enabling ensembles of simulations to target a single GPU device. Ensembles can consist of stochastic runs of the same simulation, separate simulations constituting parameter exploration, or combinations of both. The third method presented improves device utilisation for large heterogeneous models. The approach taken uses concurrency exposed within the state based model description to provide opportunity for concurrent overlapping of kernel execution. Our last method introduces a mechanism for abstracting and resolving movement conflicts and more general forms of inter agent conflict which result in the transition of serial models to the state based representation. The approach uses a hierarchical modelling description to provide re-usable sub-models which ensure that state based representation of the main model does not require a conceptually different description to that of any serial counterpart. Our results are demonstrated through a range of example models which are representative of a broader class of ABMs.

## 2 | FLAME GPU 2

### 2.1 | Framework overview

The FLAME GPU 2 software is engineered to promote a clear separation of model description from implementation. In the first generation of the FLAME GPU software models were described using declarative (XML) syntax. FLAME GPU 2 favours a procedural process to generate and define model descriptions through a modelling API (which is shown in Figure 1. The API has a clear distinction between the model description and simulation (or experimental process).

[†]When comparing an NVIDIA Tesla M2090 GPU in 2011 with 512 CUDA core with an NVIDIA A100 GPU in 2021 with 6912 CUDA Cores.

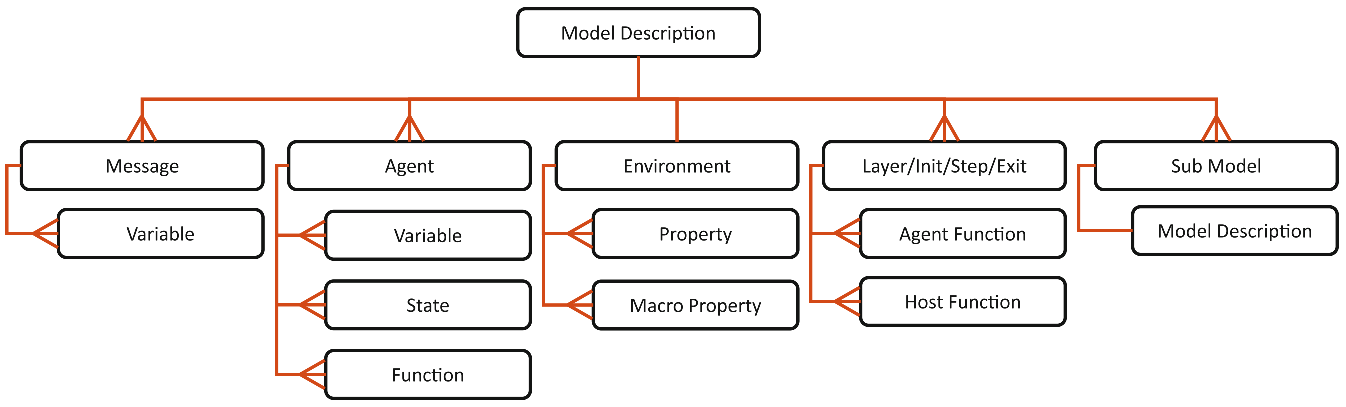


FIGURE 1 A simplified hierarchy of the FLAME GPU 2 model description API. Crowfoot notation defines a one to many relationship otherwise there is a one to one relationship between modelling objects. Full API documentation is provided within the user guide at https://docs.flamegpu.com/.

A FLAME GPU 2 model description requires the specification of agents, messages, an environment, functions including their dependencies to other functions and any hierarchical sub models (described in detail in Section 3.4). Following the state based method of agent representation, the API allows agents to have a number of states, functions which transition agents between states, and typed variables (both scalar and vector). The API allows messages to be described as a collection of typed variables. FLAME GPU 2 does not impose restrictions on the spatial form of a model and agents can exist within any dimension of space in either continuous or discrete environments. In order to ensure agents can communicate efficiently within their environment the framework provides message specialisation which dictates how information is accessed from the global message lists. The choice of message specialisation can have a significant impact on the efficiency of data movement and is one of the key considerations of Section 3.1.

The API provides a mechanism to specify and represent environment variables (or properties) which are not owned by any particular instance of an agent. The environment is also able to represent macro environment properties which are intended for models which require large amounts of environment data to be stored. Macro properties provide an alternative to messages for lightweight indirect communication between agents via a globally accessible variable. Atomic operations permit agents to manipulate values directly. This form of communication is well suited to agents consuming some form of resource within the environment ensuring that race conditions are not introduced. The order of the atomic operations is non deterministic and determined by the GPU's execution model, as such, if agent priority plays an important role in manipulating the environment then this 'conflict' between agents (in obtaining a resource) is best resolved using the sub-model approach described later in this article.

Within the API, the order of execution of agent functions can either be specified explicitly using layers or can be inferred from the DAG which describes dependencies between functions (where a function may depend either on another function or on input or output of a message). Functions which have a dependence on other functions or which operate on the same agent cannot exist within the same function layer, however independent functions may. In addition to agent functions it is possible to run arbitrary 'host' code by inserting 'host layer functions' into the dependency graph. In the context of the API, device refers to code executing on the GPU and host refers to code executing on the CPU. Additionally, host functions can be added to the model execution at initialisation (init), after each step of the simulation (step) or after simulation has completed (exit). Host functions can manipulate agents and the environment or gather statistics or data from the simulation through the run-time (host) API.

### 2.1.1 | Simulation API

In order to perform the simulation of a model in FLAME GPU 2, the simulation API (show in Figure 2) allows the configuration of a computational experiment for execution. Simulations can be executed as single simulations or as multiple instantiations of the same simulation (i.e., ensembles). A single simulation execution is initialised with one or more vectors of agents (a population of agents in a given state), agent vectors can also be obtained after the simulation has

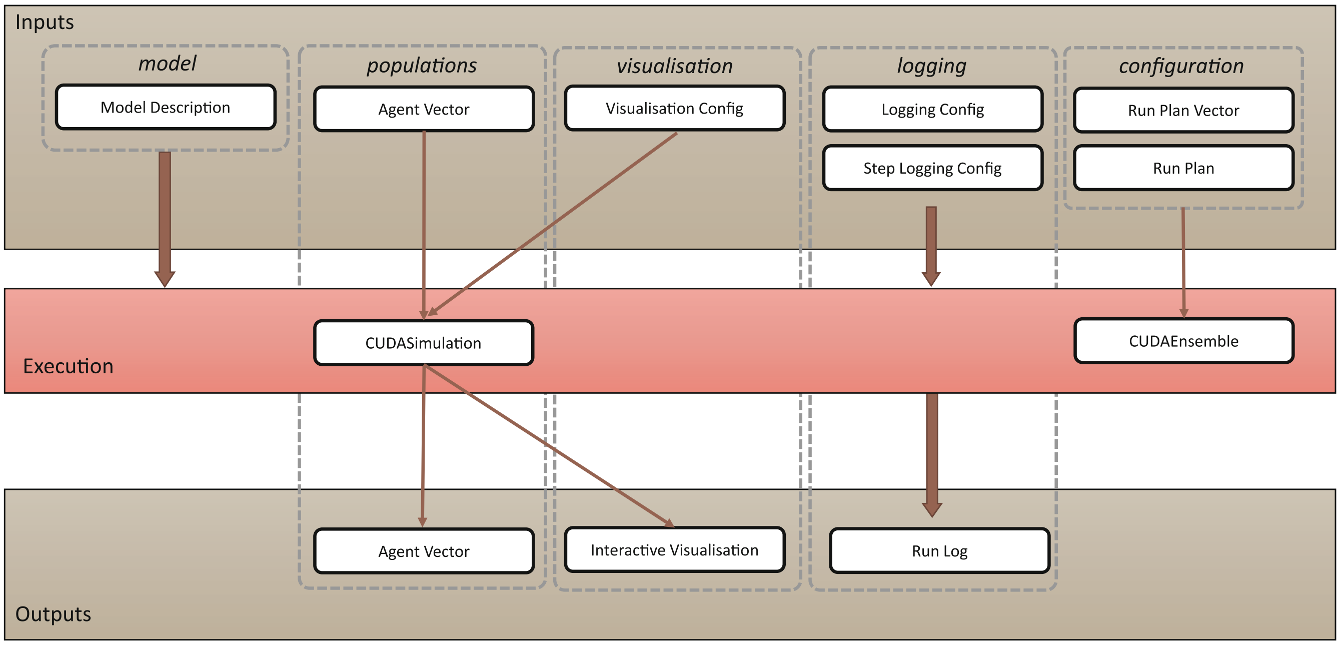


**FIGURE 2** The FLAME GPU simulation (execution) API components. The components are broadly categorised as inputs, execution, and outputs. The dotted lines show functional clustering of components which span over the categories. The model and logging options are used by by `CUDASimulation` and `CUDAEnsemble`. Population and visualisation components are unique to single simulations with `CUDASimulation` and run plans are unique to ensembles of simulation with `CUDAEnsemble`.

completed for analysis. Ensembles of simulations require a run plan configuration which describes how properties of the simulation should be varied over the ensemble. Agent populations must then be dynamically created using initialisation functions. Both single simulations and ensembles require a model definition and both can be configured to perform logging either at each simulation step or at simulation completion. Real-time interactive visualisation is supported for single simulations and can be configured using the API without writing any OpenGL code. The visualisation can be configured to render the desired agent types and states by linking agent variables to a set of specific visualisation properties. For example, location, scale, direction, and colour.

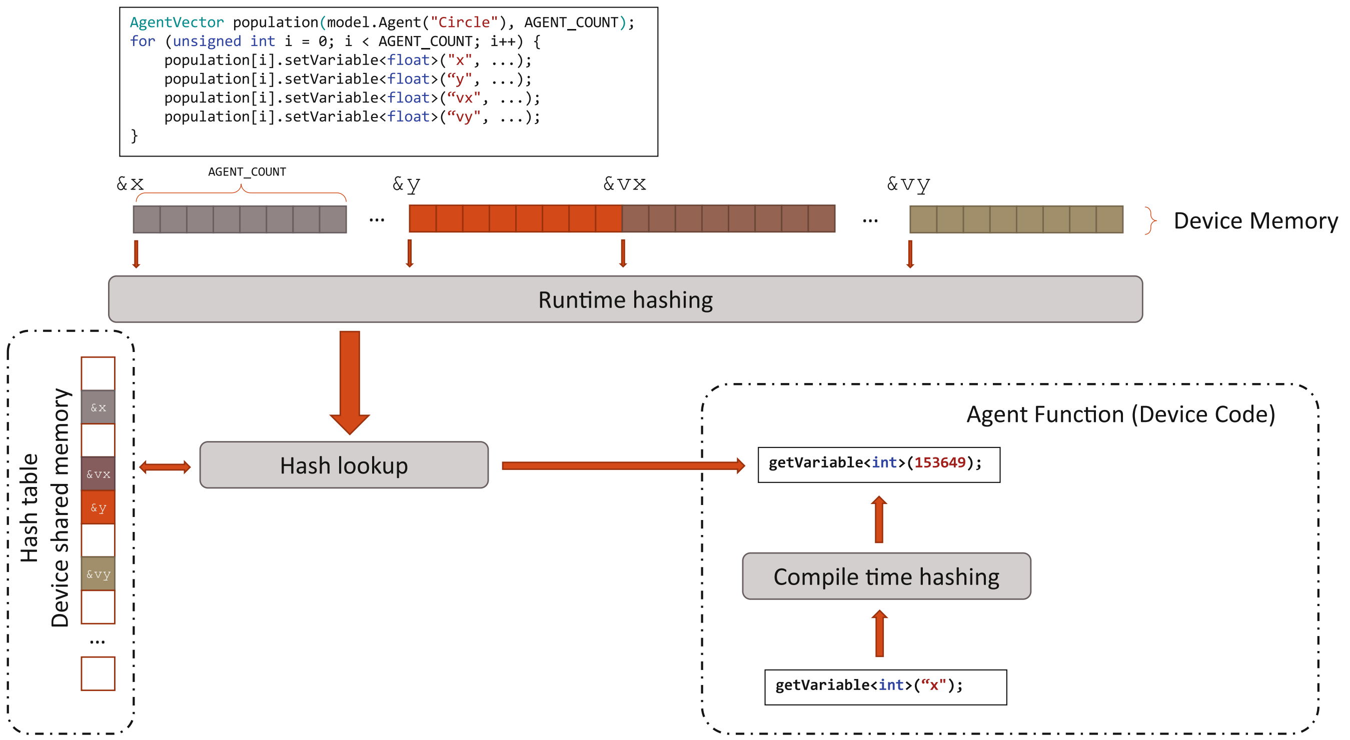
### 2.1.2 | Run-time API

A key design goal for FLAME GPU 2 was to ensure that modellers have a simple method of scripting agent functions. The run-time API provides a documented set of library functions which can be used to describe agent function behaviour. This includes the ability to query agent variables, iterate message lists and interact with the environment. The following code listing demonstrates an agent function called `move`, from the circles benchmark model used in the results section. The function performs iteration over a message list in order to calculate a repulsive force which is used to effect the agent's position. Only agents within a fixed spatial radius are considered (line 13). Agent functions are passed into the simulation API agent description objects as a structure containing a common device function macro (shown on line 1). The approach is inspired by that used by Thrust and CUB custom operators and ensures the compiler can potentially perform whole kernel optimisation. From the modelling perspective this appears as though the function is a function pointer and is hidden by using the `FLAMEGPU_AGENT_FUCNTION` macro. Each agent function requires three arguments, a function name, a message input type and a message output type. If Message input or output is not required for the function the `flamegpu::MessageNone` type should be used. All agent functions have access to a `FLAMEGPU` singleton object. The singleton is configured within the simulator prior to execution of the function so that it can be used to read agent variables (e.g., lines 2 and 3), read environment variables (e.g., lines 3 and 4), and to write agent variables (e.g., lines 23 and 24). In the provided example all agent and message variables are three component floating point vector types available by including the `glm` library which also defines the function `length`. The listing below also shows how the message input argument can be used to iterate messages (line 8) as well as how to query message variables (line 10). Although not shown, the run-time API supports creation of new agents (using `FLAMEGPU->agent_out`) and the use of random number generation (using `FLAMEGPU->random`). Random number generation includes uniform and normal distributions.

```
1  FLAMEGPU_AGENT_FUNCTION(move, flamegpu::MessageBruteForce, flamegpu::MessageNone) {
2      const flamegpu::id_t ID = FLAMEGPU->getID();
3      const vec3 p1 = FLAMEGPU->getVariable<vec3>("position");
4      const float REPULSE_FACTOR = FLAMEGPU->environment.getProperty<float>("repulse");
5      const float RADIUS = FLAMEGPU->environment.getProperty<float>("radius");
6      vec3 f = {0.0f, 0.0f, 0.0f};
7      int count = 0;
8      for (const auto &message : FLAMEGPU->message_in) {
9          if (message.getVariable<flamegpu::id_t>("id") != ID) {
10             const vec3 p2 = message.getVariable<vec3>("position");
11             vec3 displacement = p2 - p1;
12             const float distance = length(displacement);
13             if (distance < RADIUS && distance > 0.0f) {
14                 float k = sinf((distance)*M_PI*-2)*REPULSE_FACTOR;
15                 // Normalise without recalculating distance
16                 displacement /= distance;
17                 f += k * displacement;
18                 count++;
19             }
20         }
21     }
22     f /= count > 0 ? count : 1;
23     FLAMEGPU->setVariable<vec3>("position", p1 + f);
24     FLAMEGPU->setVariable<float>("drift", sqrtf(f.x*f.x + f.y*f.y + f.z*f.z));
25     return flamegpu::ALIVE;
26 }
```

One of the main challenges of the run-time API implementation, is providing a mechanism to map an agent function's named variables to GPU memory. That is, the variable names which are specified as a string literal within the simulation APIs agent description and referred to within an agent function. In order to provide this mapping with compiled agent functions in C++, compile-time string hashing is used to resolve a string literal into a numeric value (see Figure 3). Compile time string hashing is implemented through recursive C++ template meta programming functions operating on each character of the string literal. The resulting hashed variable is queried within a hash table constructed in shared memory in order to reduce latency. The hash table query returns a region of memory which corresponds to a compact array of individual agent values for the given agent variable (and state). Individual agent values are obtained by indexing into the region of memory using the thread index.

In order to support run-time compilation of agent functions, a necessity when providing language bindings in non-compiled languages such as Python, an alternative approach to variable and address mapping is required. Where agent functions are defined at run-time, CUDA code is compiled using CUDA run-time compilation (NVRTC). From the perspective of a modellers there is no difference in the specification of the agent function other than the agent function being provided at run-time as a string or external file. In order to support run-time compilation (or RTC), each agent function is compiled to a separate PTX (the low-level parallel thread execution virtual machine and instruction set architecture used for CUDA) module using a dynamically generated version of the run-time API. The RTC version of the runtime API includes an alternative variable name to memory address mapper which is generated at runtime, once the model is fully specified. The mapper directly embeds a lookup table of variable name strings to device (constant) memory locations. Use of an intermediary constant memory location is preferable to embedding a memory address directly within the PTX as the addresses may change throughout a simulation's life-cycle due to reallocation as population sizes grow. RTC generated code makes heavy use of C++ template meta programming to ensure that entire variable getter and setter functions can be resolved to address look-ups through compiler processing (e.g., compile time string comparison) to avoid introducing any function call overhead in the generated PTX code. This reduces any latency in obtaining values from memory. RTC functions are also cached to disk between invocations of a simulation to avoid the RTC overhead cost of compilation between simulation runs.



```
AgentVector population(model.Agent("Circle"), AGENT_COUNT);
for (unsigned int i = 0; i < AGENT_COUNT; i++) {
    population[i].setVariable<float>("x", ...);
    population[i].setVariable<float>("y", ...);
    population[i].setVariable<float>("vx", ...);
    population[i].setVariable<float>("vy", ...);
}
```

```
getVariable<int>(153649);
```

```
getVariable<int>("x");
```

FIGURE 3 Figure shows the process of runtime variable hashing to obtain an agent's unique variable value from memory. The agent's four variables (x, y, vx, and vy) are organised in memory using a structure of arrays to promote efficient memory coalescing. The figure shows how variable x is hashed at compile time (from within an agent function) to the value 153,649 (via use of compile time hashing). The hashed value is then queried in shared memory to return the area of memory representing the storage of the named agent variable. An individual agent value is obtained by indexing into the common area of memory. The shared memory hash table is constructed at runtime and loaded into global memory. Prior to the agent function execution the runtime API transparently loads the hash table into shared memory.

## 3 | DESIGN CONSIDERATIONS

### 3.1 | Efficient data movement

Core to the design of FLAME GPU 2 is the principle of optimising data movement. GPUs are high throughput architectures and as such a common optimisation approach for simulation is to minimise latency by reducing data movement to increase arithmetic intensity. Previous work[14,15] has utilised state based representation of agents with a data model at the granularity of individual agents and messages. This has had significant advantages in the design of a user facing API as agents and messages can be encapsulated as objects with agent and message variables accessed directly as member variables. Whilst conceptually simple, this level of data granularity requires that entire agents and messages are moved between memory subsystems and registers. In many cases, particularly with complex agents and messages, agent behaviour may operate on only a subset of variables with the agent or message object. As such, it is desirable that data movement should be managed at the individual variable level. FLAME GPU 2 therefore proposes this level of data granularity. Agent and message variables are moved around the memory hierarchy only when directly accessed. The previous Section 2.1.2 and Figure 3 demonstrated how agent variables can be queried using this approach.

When considering the performance of an ABM on the GPU it is typical that message list iteration (message input) is, in the majority of cases, the most expensive operation that is performed. From a simplistic perspective this can be attributed to volume of data movement. That is, message input functions typically require the reading of variables from multiple message, requiring high volumes of data movement adding latency to the agent function. As such, a general optimisation approach for agent communication is to reduce message latency by ensuring that:

1. Only a subset of (potentially relevant) messages are returned to agents.
2. That the GPUs cache hierarchy is utilised to reduce movement of messages read by adjacent (in memory) agents within a state list.

An extendable design for message access patterns is incorporated into FLAME GPU 2. This allows a large number of message iteration patterns to be implemented to ensure that the first point (above) is satisfied. Users are able to select the most appropriate access pattern depending on the agent type. For example, an agent in 3D space interacting within a limited spatial neighbourhoods can opt for a messaging type suited to their communication pattern. Spatial messaging for example, transparently builds the necessary data-structures and performs periodic sorting of agent data to ensure good data locality for message variable reads. Iteration of messages within a user defined agent function uses C++ iterators in order to provide a consistent mechanism for parsing message data regardless of the underlying message access pattern. Internally within the framework, message iterators on the device are responsible for ensuring that message variable requests traverse the necessary data structures to accelerate access. On the host, the message implementation is responsible for building and maintaining any data structures required to provide efficient device access.

The base implementation for messaging is the brute force case which requires no acceleration data structure. This all-to-all form of messaging ensures that agent functions reading this type of message will iterate the whole message list which will have messages from any agent performing a function where the message has been output. Spatial messaging is used to allow agents within a 2D or 3D environment a filter ensuring only messages within a fixed radius are returned. The approach for optimising spatial messaging has been the subject of previous work[21] and is relevant to other fields of research where particle like models are simulated, for example, fluid simulation and molecular dynamics. Array messaging supports communication of agents within discrete 1D/2D/3D space and bucket messaging supports agents communicating via buckets which are assigned a unique integer key. The latter acts as a foundation to build more complex communication strategies such as networks.

## 3.2 | Improving device utilisation for small populations

In order to ensure that highly parallel modern GPU devices can be used effectively, there is a necessity to ensure that a large number of threads are active at any one time. It has been highlighted in our introduction that the traditional mapping of a single agent to a single GPU thread is unable to provide sufficient parallelism for modern GPUs when considering small populations sizes. Small populations occur as either the result of small simulations (e.g., with low number of agents), or potentially as a result of a complex model in which there exist smaller groups of agents in a large number of states (or represented by different agent types). The latter case is discussed in the next section. To ensure high device utilisation for small simulations, FLAME GPU 2 allows the specification of model ensembles. Ensembles permit a single device to execute multiple parameterisations or randomly seeded runs of a model a concurrently. Arithmetically this exposes a high level of parallelism to the device leading to higher utilisation. From a simulation perspective this is highly advantageous as ABMs typically require multiple stochastic runs to statistically demonstrate global or emergent behaviours for a given parameter configuration of a model. Additionally parameters of ABMs are usually explored through simulation in order to discover or infer behaviours. The ensemble abstraction supports both use cases. Additionally, external processes such genetic algorithms (or more generally, heuristic search) can be used to drive ensembles to provide system level optimisation via parameter space search. Within the implementation of ensembles, independent simulations share the same GPU context from within their own CPU thread. This places the emphasis for scheduling of GPU resources on the run-time driver and has the advantage that simulations running for different duration's are not required to remain in lockstep. Figure 4 highlights how simulations within an ensemble may be allocated to resources. Execution of ensembles can utilise multiple GPU devices which are assigned using round robin scheduling. Despite sharing a context, independent simulations within an ensemble have their own dedicated blocks of memory and as such there is no need to introduce isolation mechanisms, synchronisation or locking mechanisms.

## 3.3 | Improving device utilisation for large heterogeneous models

As agent based models become increasingly complex, their population level behaviour tends to become less homogeneous. Heterogeneity is problematic for GPU simulation as it has the potential to introduce code divergence within the vector lanes (warps). State based representation of agent based systems[15] provides a potential solution to the issue of divergence. Agents are naturally grouped by their state providing the opportunity for a larger number of less divergent functions to apply to each state list. A significant and limiting factor of the use of states to avoid divergence is that as the number of states increases, the likelihood of being able to fully utilise the device with any one agent function is significantly

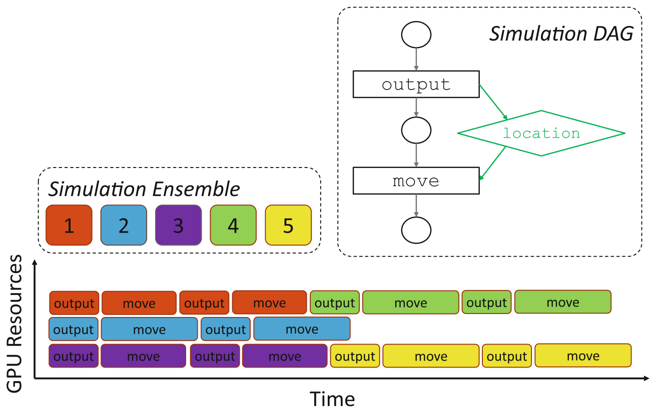


FIGURE 4 Figure shows the scheduling of concurrent simulations within an ensemble in order to improve device utilisation. Each numbered simulation (1–5) utilise a common model description but can have different inputs or environmental properties (i.e., modelling parameters). The model description has a directed acyclic graph (DAG) of agents functions which occur for each timestep. The two agents functions (output and move) have dependency on a location message requiring their execution to be sequential within a single simulation. The resource graph (bottom) is a simplified version of a profiling timeline which demonstrates how each simulation instance from the ensemble is able to be overlapped according to availability of GPU resources. Each simulation is shown to execute for two iterations. Within the timeline each simulation runs to completion. Simulations from the ensemble are scheduled as GPU resources become available.

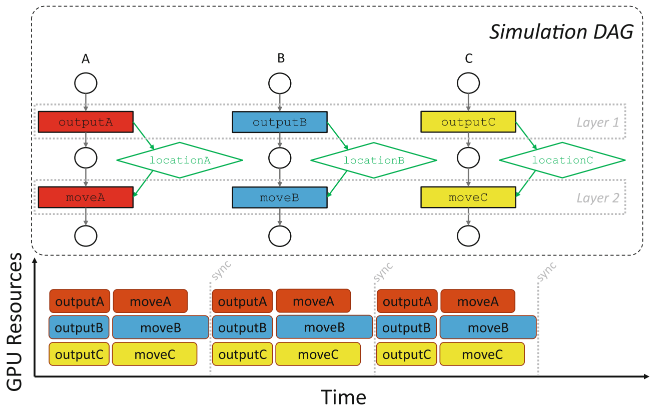


FIGURE 5 Figure shows the scheduling of agent functions within a heterogeneous model by exploiting concurrent execution of independent agent functions. Within the directed acyclic graph (DAG) of the simulations agent functions three non-interacting agents perform the same agent functions of 'output' and 'move'. There are two layers containing agent functions. Each agent function is called asynchronously within its own stream within each layer, allowing overlap of execution. The diagram shows three iterations of the model. Synchronisation after the execution of each layer is required to ensure race conditions are not introduced. The non concurrent execution equivalent would require each function to execute sequentially making poor use of the available GPU resources.

diminished. For very large simulations (e.g., within computational biology) this is not a consequential problem. For small population sizes, device under utilisation suffers the same problem as described in the previous section.

FLAME GPU 2 addresses the issue of under utilisation by ensuring that where possible, independent agent functions can be executed concurrently. Figure 5 shows an example of how non-interacting agent functions can be executed concurrently to provide better device utilisation. This is implemented using CUDA streams. Each agent function within a simulation layer (which is known to have no function or message dependencies) is launched asynchronously within a unique stream. This asynchronous launching allows functions to overlap within the timeline subject to the availability of device resources. A synchronisation mechanism is utilised after the execution of each layer to ensure that agents can then asynchronously launch functions in the next layer without introducing race conditions. The concurrency mechanism for heterogeneous models is different to that of the previous section. It exploits parallelism within the model rather than over multiple models. The two approaches can however be combined to support concurrency within a model at the same time as concurrency between independent simulations. With respect to mapping concurrent streams, or indeed simulations, to hardware each SM is able to have multiple resident blocks (up-to the maximum SM limit of threads in flight). From a modelling perspective, concurrency is typically limited by either functional or message dependencies described by the execution DAG. When the control flow of a FLAME GPU 2 model is specified as a dependency graph, the execution order/schedule of function execution is generated. This feature maximises the potential for concurrency within a model and ensures that users do not have to manually infer dependencies between agents, messages, and functions.

### 3.4 | Hierarchical modelling for recursive problems

The need for recursion exists within state based representation for a range of reasons. Within the Sugarscape model recursive behaviour is required to resolve competition for resources and to avoid collisions within movement. More generally the recursive resolution of competition is referred to as conflict resolution. Biological mechanical models of cell behaviour typically require a conflict resolution algorithm to provide iterative movement of densely packed cells to within a margin of acceptable overlap. Although introducing recursive elements within state based modelling does not require understanding of the underlying parallel architecture, it adds a level of modelling complexity which in most cases would not be required for a non parallel simulator. Cellular biology models an example of this References 14 and 22 where concepts such as 'non-linear' loops or multi-scale modelling[23] have been utilised. The latter relies on the separation of the mechanical models of movement from the agent based cell behaviour. In the case of 'non-linear' loops, a simulation time-step may filter agents into a state where they will undertake a set of behaviours to resolve conflict or competition (for example movement on a discrete grid), but which does not advance the global simulation time. Although functional, this approach is limited in flexibility and fundamentally requires a different model specification to that of a simple sequential implementation.

FLAME GPU 2 aims to ensure a clear separation between model specification and implementation which includes abstracting recursive behaviours needed to reproduce serial counterparts. As such, a hierarchical modelling abstraction is proposed which allows a model to contain a re-usable sub-model for conducting conflict resolution or other forms of recursive behaviours. A sub-model is defined in much the same way as a standard FLAME GPU 2 model and acts as a fully specified model nested inside a parent model. The only additional requirement is that the model contains at least one exit condition. Agents and a subset of their variables can be mapped along with required environment properties. Each model is also able to have private components not visible to the parent/sub model. Sub-models are executed as part of the main model by being placed in an independent simulation layer where they are executed until the exit condition is met. An example sub-model is shown in Figure 6 which represents the Sugarscape model used in the results section.

## 4 | RESULTS AND SHOWCASES

The following section considers measuring performance of the FLAME GPU 2 simulator v2.0.0-rc[24] through the implementation of a number of benchmark and classic agent based simulation models. The models are chosen and adopted to demonstrate the novel features and design considerations proposed within this article.

All results have been obtained from execution on the Bessemer GPU system which is part of the University of Sheffield's HPC system. The nodes are Dell PowerEdge C4140 nodes with two Intel Xeon Gold 6138 CPUs and four NVIDIA V100 (32 GB) GPUs hardware. The hardware is typical of that of modern GPU nodes within HPC facilities. Our job submission scripts request a single dedicated V100 and a single CPU core (with the exception of Figure 9 which uses a single GPU and 1/4 of the CPU cores. E.g., 10 cores). For all simulation runs the FLAME GPU 2 was compiled with the configuration option `FLAMEGPU_SEATBELTS=OFF`. This disables a number of run-time checks (e.g., bounds checking, run-time warnings) to improve run-time performance. Prior to benchmarking models were executed with `FLAMEGPU_SEATBELTS=ON` to verify the implementation would not encounter these types of errors.

### 4.1 | Benchmark modelling

Benchmarking of FLAME GPU 2 is undertaken by considering a scalable benchmark model, 'circles'. The circles model is based upon a simple force based particle model where each agent, $i$, has a position ($p$) clamped within a defined environment. The position is updated each time step by considering a force ($F_i$) which acts on the agent to cause movement.

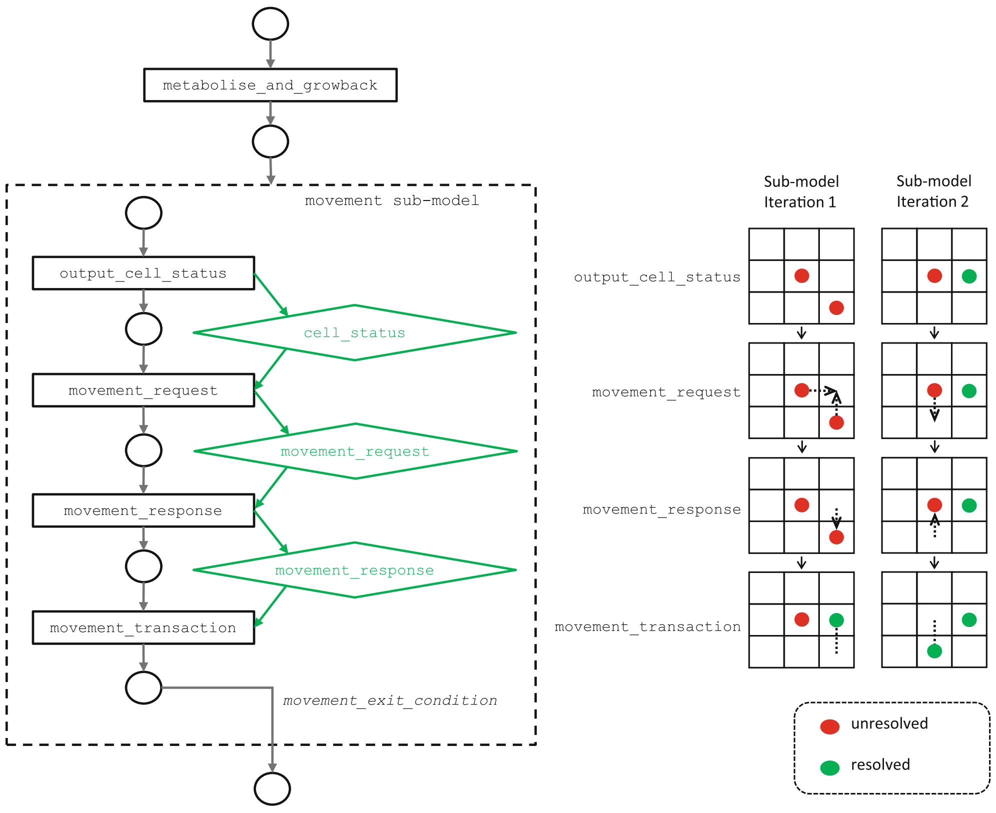


FIGURE 6 Figure shows the directed acyclic graph (DAG) of agent functions within a FLAME GPU 2 implementation of the classic Sugarscape model. A sub-model is used to recursively allow movement on a 2D grid to prevent collision. The main model consists of a single function describing the behaviour of a discrete cell. The cell is either unoccupied, in which case it grows a natural resource, or it is occupied by an agent which will move if resource is higher elsewhere. The sub-model process is shown on the right hand side using a simplistic two agent example. In this case, each agent initially requests to move to the same cell location. The cell responds by selecting the agent with highest priority therefore allowing a single agent to move in the first iteration of the model. An exit condition only allows the main simulation to resume once all agents are 'resolved' and as such a second iteration of the sub-model is required for the second agent to find a collision free location (and the exit condition to pass).

$$\vec{p_i}(t+1) = \vec{p_i}(t) + \vec{F_i} \tag{1}$$

The force $F_i$ is calculated the sum of forces acting on the agent as a result of pairwise interactions and can be determined through the following equations.

$$\vec{F_i} = k\sum_{i \neq j} \vec{F_{ij}} \tag{2}$$

$$\vec{F_{ij}} = \sin\left(\frac{-2\pi\left|\left|\vec{p_j} - \vec{p_i}\right|\right|}{r}\right)\frac{\vec{p_j} - \vec{p_i}}{\left|\left|\vec{p_j} - \vec{p_i}\right|\right|}. \tag{3}$$

The parameter $k$ is a damping term used to parameterise the magnitude of agent's velocity. Pairwise forces are calculated by the sine of the separation of vector positions $p_i$ and $p_j$ normalised by the search radius $r$, multiplied by the unit vector from $p_i$ to $p_j$. The purpose of the sine term is to slow agents as they approach a reciprocal distance of $r$, without this agents can overshoot the desired position resulting in oscillations. The model is initialised by a density term $\rho$ and environment boundary diameter $W$ which is used to initialise agents using a random uniform distribution. In all experiments, unless otherwise stated, the damping term $k$ used a value of 0.05, $r$ fixed at 2.0 and a density $\rho$ of 1.0 agent per unit volume. The default value of $W$ is 40.0.

### 4.1.1 | Results

To investigate the performance of efficient data movement, we consider a 3D benchmark model; circles, in which agents apply a force with an attractive and repulsive component to reach an equilibrium state. Figure 7 demonstrates the behaviour of the model which converges towards a stable state by forming spherical clusters with a radius influenced by the parameter $r$ (the interaction radius). The measure of drift (shown in Figure 7A) highlights how convergence varies over the lifetime of a simulation. The interaction radius $r$ impacts convergence as it controls the size of the cluster formations and their ability to merge once formed (shown in Figure 7B–E). The model is representative of a much wider class of force based models typical within multi agent systems. It is implemented within FLAME GPU 2 as a single agent type with two agent functions The first function simply outputs the agents position as a *location* message. The second function inputs the *location* messages, performs the summation of the force term and updates the agent's position.

Figure 8A,B demonstrate the overall performance of the circles benchmark model. The results highlight the considerable performance difference between the choice of appropriate message communication, in this case between brute force (A) and spatial (B) messaging. In addition to highlighting the importance of message communication type as a key performance indicator of a model, the graphs also highlight the performance improvement offered by run time compilation (RTC) of agent functions. In the case of brute force communication, RTC models are $\sim 2\times$ faster than the non RTC equivalent. This can be attributed to a more efficient mapping of agent and message variable names to memory, as well as improvements in register usage. Figure 8C explores the choice of message communication technique further by varying the interaction radius $r$ and comparing the relative performance of brute force and spatial messaging. The plot demonstrates how the cost of dynamically building a data structure is advantageous over brute force messaging for the majority of smaller interaction radii. At the point at which brute force communication becomes close to the performance of spatial, a large percentage of the total message data ($\sim$50%) is considered by every agent (Figure 8D). The initial ramp up of the brute force message performance is attributed to increased cache coherence at smaller interaction radii (presumably as there is a higher probability that all agents in a warp fail the distance check and remain coherent). Stepping of the performance within the spatial model is attributed to the changing size of the messaging data structure which results in different levels of device utilisation and occupancy. Periodic sorting of agent data, shown in Figure 8E, has a significant impact on the performance of spatial messaging. Although periodic sorting adds extra work, having agent data sorted using the same spatial binning as the message output increases the cache efficiency. The optimal frequency is likely to vary per model as a function of agent movement speed between spatial partitions. In the case of the benchmark model movement is highly chaotic within the 200 steps of execution and as such per simulation step sorting is optimal. Users are

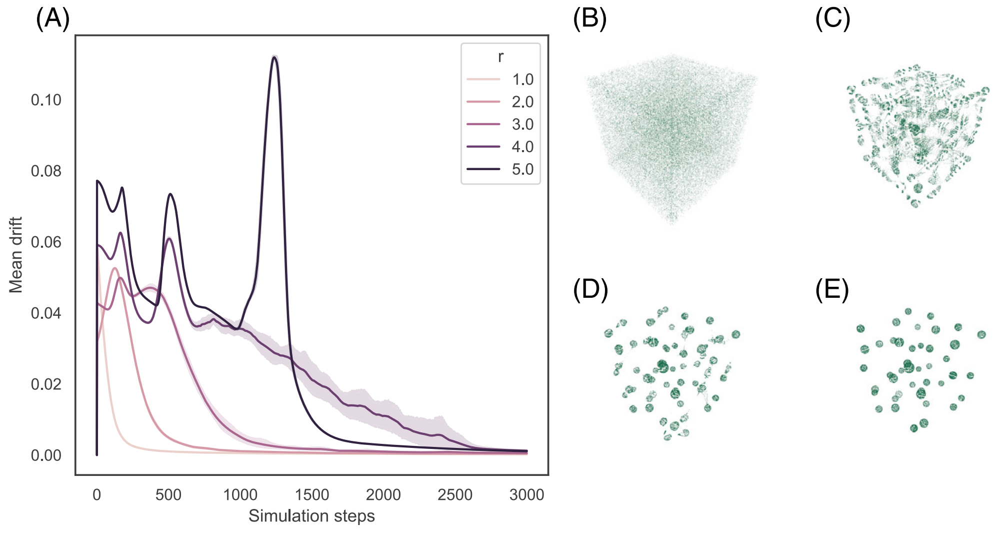


**FIGURE 7** Visualisation and emergent behaviour of the circles benchmark model. (A) Shows the mean drift which is a measure of the mean movement of the agent population per time step and acts as an indicator of movement stability and hence convergence. The mean drift is calculated over 100 simulation runs with different seeds using a varying interaction radius ($r$). Population size is fixed at 64,000 and the environment width at $W = 40.0$. (B–E) Show visualisation of the model corresponding with the plot of $r = 3.0$ at steps 0, 350, 650, and 2500 respectively.

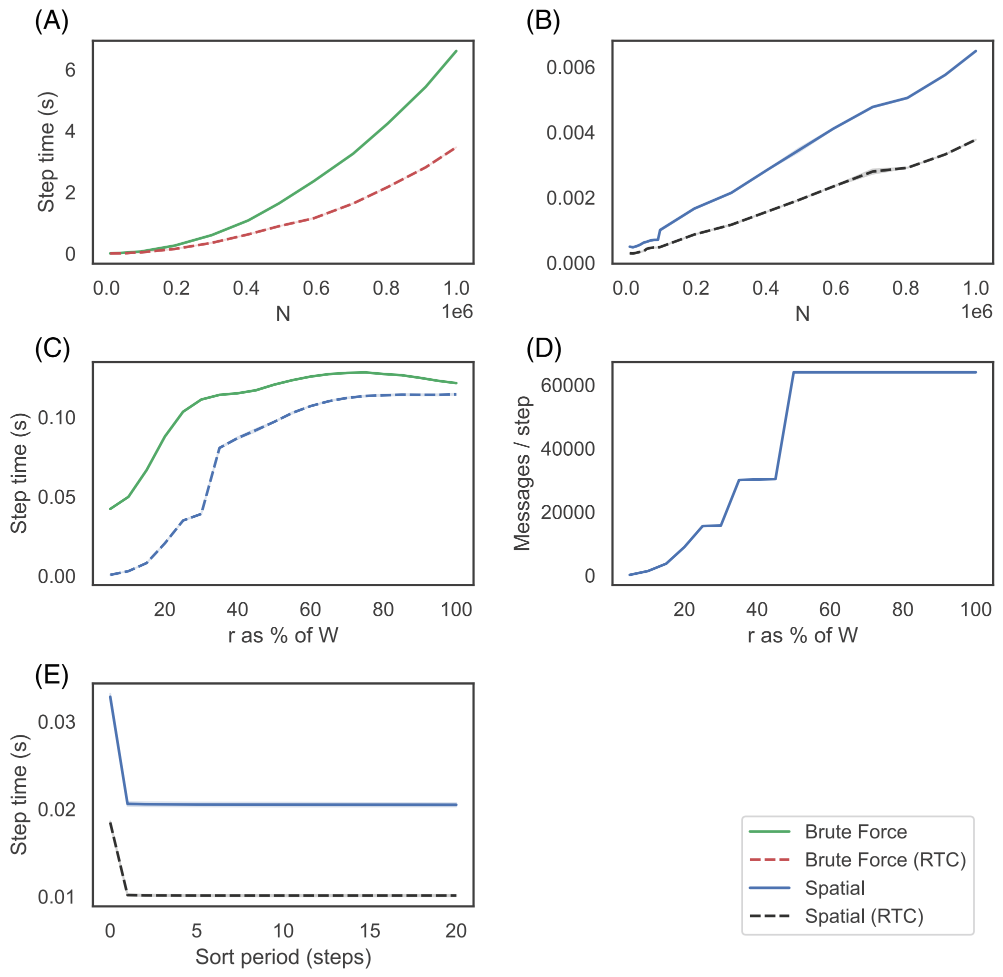


**FIGURE 8** (A, B) Performance scaling of the circles model by varying the total population size ($N$) but maintaining a fixed density ($\rho = 1.0$) by scaling the environment scale ($W$). (C) Varying the interaction radius of the model ($r$) which has the impact of increasing the workload per agent shown in (D) by considering the average number of messages passing the distance calculation check per step. For (C, D), Population size is fixed at 64,000 and the environment width at $W = 40.0$. (E) Shows the impact of varying the frequency in which agent data is sorted, to maintain cache coherence when reading spatial messages, for a fixed radius of $r = 2.0$.

encouraged to experiment with the sort frequency parameter. Although not shown, the observed memory usage differs between the spatial and brute force implementations as spatial requires additional memory for the spatial data structure which is constructed. Up to 200M agents can be simulated on a single 32GB V100 GPU. Following the trend of memory usage, it is anticipated that ~500M circle agents could be simulated on an 80GB A100 device.

## 4.2 | Enhancing parallelism example

In order to understand how FLAME GPU 2 is able to enhance parallelism by exploring both simultaneous simulation (i.e., ensembles) and function execution, a classic Boids flocking model is derived from Reynolds original distributed behaviour model for flocks, herds, and schools.[25] The model has been modified to use an inverse-square separation force. Each agent maintains a vector position $p$ and velocity $v$. The velocity term is updated each time step by considering the rules of; separation: to avoid near neighbours, cohesion: by steering to the perceived centre of the flock, and alignment: matching the velocity of the perceived flock. The terms $s_i$, $c_i$, and $a_i$ define these rules as velocity contributions within Equation (4). The parameters $S$, $C$, and $A$ are defined weighted modifiers for each of the respective rules.

$$\vec{v}_{i(t+1)} = \vec{v}_{i(t)} + S.\vec{s_i} + C.\vec{c_i} + A.\vec{a_i} \tag{4}$$

Each of the three velocities representing the behavioural rules can be defined as follows;

$$\vec{s_i} = \sum_{i,j \in O}^{n} \left(1.0 - \frac{\|\vec{p_i} - \vec{p_j}\|}{r_{sep}}\right)^2 \vec{p_i} - \vec{p_j} \tag{5}$$

$$\vec{c_i} = \left(\frac{1}{n}\sum_{i,j \in N}^{n} \vec{p_j}\right) - \vec{p_i}, \tag{6}$$

$$\vec{a_i} = \frac{1}{n}\sum_{i,j \in N}^{n} \vec{v_j}, \tag{7}$$

where the sets $O$ and $N$ are defined as

$$O = \{i,j : \|\vec{p_j} - \vec{p_i}\| < r_{sep}\} \tag{8}$$

$$N = \{i,j : \|\vec{p_j} - \vec{p_i}\| < r\} \tag{9}$$

The set $O$ represents agents within a small separation distance $r_{sep}$ defined as 0.5× the value of the interaction radius $r$. The set $N$ represents all agents within the interaction radius $r$. Each of the Equations (5)–(7) operate by calculating an average position or location from neighbours. Equations (5) and (6) represent a displacement vector from these perceived average locations and the agents own position.

### 4.2.1 | Simulation scaling performance

To demonstrate the impact of using ensembles to improve device utilisation and overall simulation performance Figure 9 shows the results of varying the number of ensembles of a simple 3D flocking model.[25] The performance timing shows

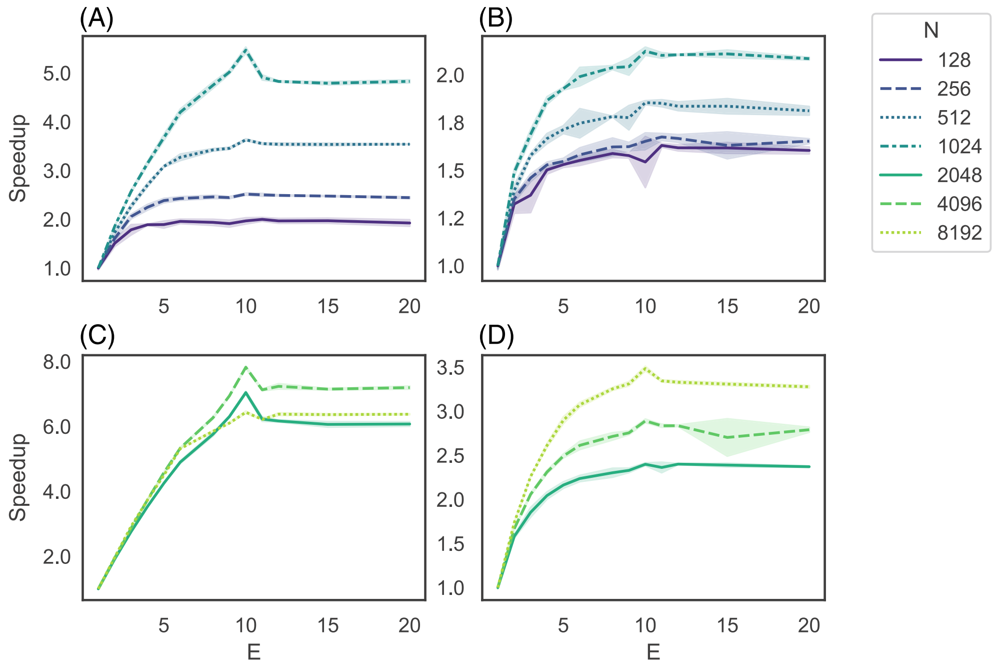


**FIGURE 9** Mean speedup of a 60 Boids simulation models of flocking agents where ensemble size (E) represents the number of ensembles in which the 60 simulations are distributed. Speedup is measured by calculating a baseline for each population size from the mean of simulation runs where $E = 1$ (e.g., where each simulation is run sequentially). The results are based on 10 repeats of each configuration with standard deviation shown. (A) Shows small population sizes using brute force messaging. (B) Shows small population sizes using spatial messaging. (C) Shows large population sizes using brute force messaging. (D) Shows large population sizes using spatial messaging.

that as the ensemble size increases the overall simulation time is universally reduced until a point where it stabilises. The performance improvement can be attributed to the increased number of simulations being able to better utilise the device. This primarily comes from the fact that small simulations result in kernel launches that utilise only a small number of the available streaming multi-processors (SMs). Increasing the ensemble size allows more of the SMs to be utilised increasing the overall throughput and reducing the average simulation time. The V100 device used for this benchmark is able to support up to a maximum of 128 resident grids per device. This number is greater than the 80 SMs suggesting that it is possible that a single SM is also able to switch between blocks of threads from different grids to hide latency. Performance is observed to level off as a result of the device becoming saturated at which stage the overhead of launching and serialisation of workload becomes a significant factor. Beyond the saturation point, total simulation time (for all ensembles) increases linearly as shown by the stable average run-time. The impact of improving performance is most pronounced with ensembles for large population sizes (e.g., Figure 9C,D), especially in the case of brute force messaging (e.g., Figure 9C). This can be attributed to the longer running kernels at larger populations sizes and that brute force messaging has a higher ratio of kernel execution to overhead and therefore more opportunity for kernel overlap. It can be concluded from the results that ensembles therefore are most beneficial to larger population sizes of densely communicating agents.

#### 4.2.2 | Concurrency performance results

The impact of concurrent function execution can be measured by considering a variant of the flocking model in which the level of concurrent behaviour can be controlled experimentally. This is achieved by adding a parameterised number of non-interacting species. Each species follows the standard Boids rules. By ensuring no interaction between species there is opportunity for concurrent scheduling of agent functions within the DAG. Figure 10 demonstrates that a maximum speedup of ∼14× is achieved for a population size of 2048 with 25 species. More broadly, Figure 10A,B show a consistent level of speedup for small populations but with diminishing returns as the device approaches full utilisation. This is a result of threads competing for hardware resources (i.e., cache and registers). For large population sizes (Figure 10E,F) the same initial profile of speedup is observed however there are significant peaks and troughs as the number of species is increased. There correspond with plateaus in the step times show in Figure 10G,H. The plateaus in step time are a result of device utilisation and represent points in which a new full wave of thread blocks is required to simulate the entire grid of threads. In all cases, brute force messaging sees an improved speedup and favourable run-time over spatial messaging for the same populations. In previous scaling experiments (i.e., Figure 8) the opposite was true of run-times when comparing spatial with brute force messaging. An important distinction within the concurrency experiment is the relatively low population sizes and resulting small message lists. Smaller message lists result in shorter kernel execution reducing the opportunity for overlapping kernel execution.

### 4.3 | Hierarchical modelling example

In order to demonstrate the effectiveness and performance of hierarchical modelling, a classic variant of the Sugarscape model is chosen (the Epstein and Axel Immediate Growback model[18]). It follows the same implementation as that provided by the popular agent based modelling toolkit NetLogo.[26] Within this model, $N$ agents occupy a 2D grid environment of 'cells'. Each cell contains a level of sugar $s$ which has a maximum value of $S_{\max}$. The grid wraps so as to form a torus shaped environment without boundaries. At each time-step sugar grows back within a cell at a rate of $\alpha$. Agents have a fixed vision distance of $V$ cells in the north, south, east, and west directions as well as a sugar wealth, $w$, which represents a quantity of obtained sugar. The serial implementation of this model requires that at each time-step the order of agents is randomised so that sequentially, each agent can choose to move to a new empty cell with a preference for cells where there is a higher quantity of sugar available than their current location. If there are multiple potential cells which an agent could move to, then one is chosen from the candidate cells at random. Upon moving to a new cell, or if an agent remains stationary, the agent consumes all of the sugar at the agent's current location and decreases its own sugar wealth by a metabolism $m$. If at this stage the agents sugar wealth is not greater than zero it dies and is removed from the simulation. The Sugarscape model and movement sub-model function dependency graph within FLAME GPU 2 are shown graphically in Figure 6.

Initialisation of the originally described Sugarscape model in an environment size of $49 \times 49$ requires the creation of a sugar distribution that defines a topography of two peaks separated by a valley. In order to define a scalable equivalent

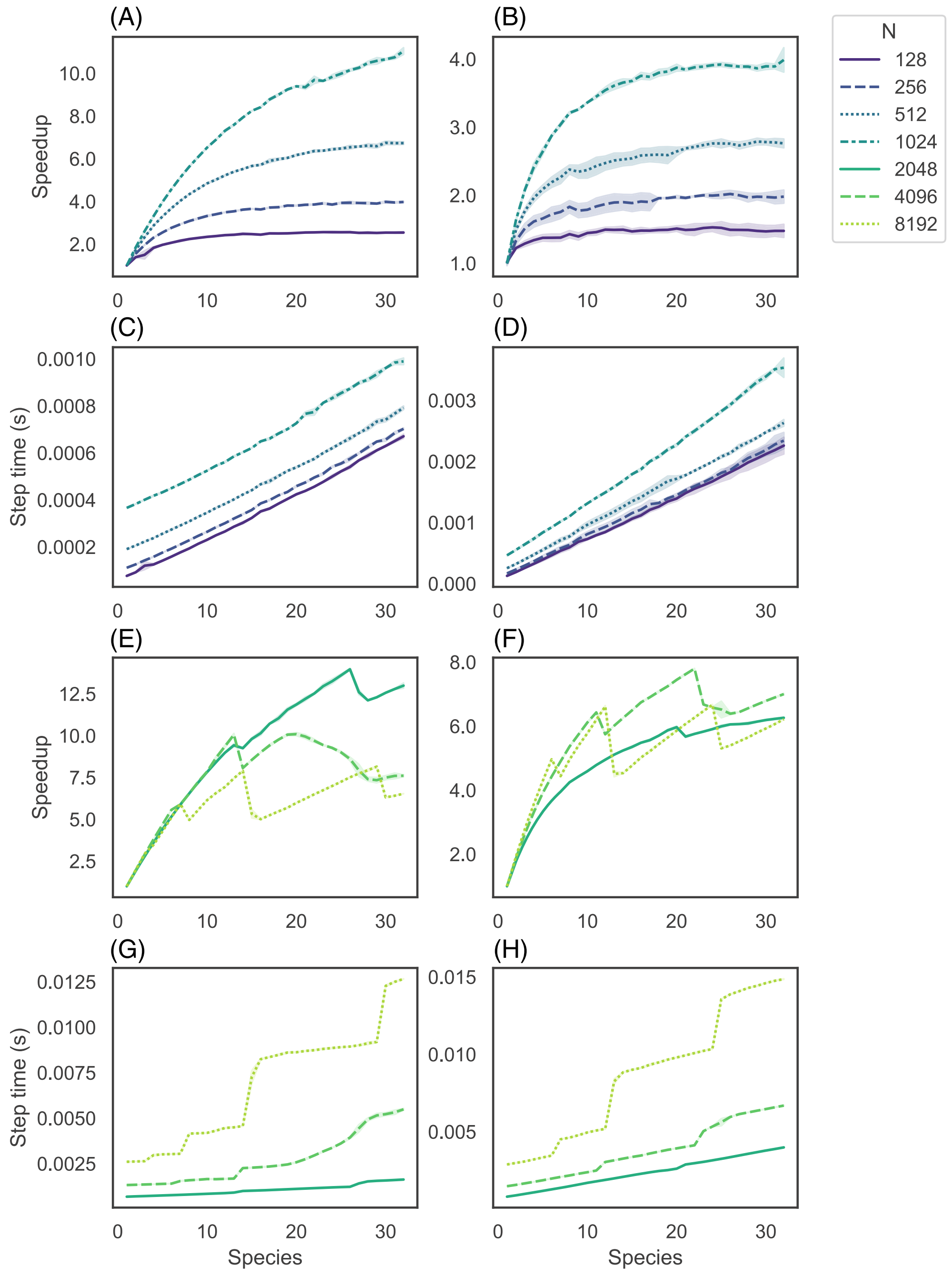


**FIGURE 10** Speedup and step timing of a Boids model where the number of non-interacting species is varied to consider concurrent execution of populations. Speedup represents the impact of concurrency compared to a baseline where each species is simulated in serial within the simulation. Step time plots show the mean simulation step time over the 1000 steps of simulation and 3 repetitions (each with different random seed controlling initial positions) and represent the timings used for the speedup plot which are directly below. Left hand side plots (ACEG) are brute force messaging, right hand side plots (BDFH) are spatial messaging, top half of figure (ABCD) is small population sizes, bottom half of figure (EFGH) is large population sizes.

model in which the environment size (of $D \times D$) can be dynamically specified, the number of sugar peaks with a uniform random position, $P$, can be determined as $\frac{2D^2}{49*49}$. Peaks are not allowed to exist within an Euclidean distance of 20 of each other within the torus environment to ensure that peaks do not exceed $S_{\max}$. Each position $x$ within the environment has an initial sugar volume, $s_x$ influenced by position $p$ of the $P$ peaks according to the following equation;

$$s_x = \sum_{i}^{P} 4 - \left(4 \wedge \left\lfloor \frac{\|p_i - x\|}{5} \right\rfloor\right). \tag{10}$$

That is, that initial sugar density is a sum of the sugar distributions from each peak given some fixed boundaries where $\wedge$ indicates the a minimum operator, ensuring that sugar value contributions from each peak are between 4 and 0. The value of $S_{\max}$ is set at 4. Each cell has a probability\$\rho$ (with a default value of 0.17) of being occupied by a Sugarscape agent during initialisation. $N_{init}$ represents the initial population size of Sugarscape agents and has an expected value of $D^2\rho$. Each Sugarscape agent has an initial $w$ drawn from a uniform random distribution of in the range of 5 and 25 with a metabolism $m$ drawn from a uniform random distribution with a range of 1 and $M_{\max}$. The model parameter value of $M_{\max}$ is 5, $S_{\max} = 7$, $\alpha$ is 1 and all agents have a fixed vision $V = 1$.

### 4.3.1 | Results

The flexibility of our hierarchical modelling approach can be demonstrated through the implementation of the Sugarscape model[18] which uses a sub-model to resolve conflict in movement within the discrete grid. Conflict occurs naturally as the movement of agents in parallel (rather than in randomised order) creates competition for areas of the grid with high levels of resource. In parallel, conflict can be resolved through a process of bidding, in which agents simultaneously 'bid' for their desired location with only the highest priority agent allowed to perform the movement. Priorities can be randomly assigned to agents and stored as an agent variable to achieve fairness, as in the serial case. The sub-model allows a dynamic number of transactional bidding steps to take place to ensure that every agent has an opportunity to move. Without the use of a dynamic number of steps provided by the sub-model, the parallel implementation would require a fixed number of bidding steps (i.e., the worst case of nine steps) to guarantee that agents have the opportunity to resolve conflicts. The transactional bidding process consists of agent functions to allow agents to (1) propose a movement to a new cell location, (2) for a cell location to rank competing agents by the agent's assigned priority and for the cell to communicate a confirmation of its preferred choice of agent, (3) perform movement of the agent whose preference is confirmed. Within the sub-model implementation this transactional process can be fully defined and an exit condition specified which uses the host API's reduction operator (count) to determine the number of unresolved agents (i.e., proposed to move but were unable to due to competition from a higher priority agent).

The results of the Sugarscape model in Figure 11B, demonstrate that resolution of movement as a result of competition, can be achieved in far fewer resolution steps than the theoretical maximum of 9. This remains true even for very high initial densities which result in large early population decreases as a result of agent deaths (show in Figure 11A). Performance scaling of the model is linear as demonstrated by Figure 11C. The impact of multiple waves of blocks is not obvious due to the short kernel execution times. This is a result of the array messaging which is extremely efficient at reducing the number of messages iterated by each agent. World grid sizes well in excess of the total ~16M can be simulated with the main constraint on being on memory. A conservative estimate for a V100 device is that population sizes exceeding 100M are feasible.

## 5 | DISCUSSION

FLAME GPU 2 is inspired by its predecessors FLAME GPU 1 and FLAME (for distributed CPUs) and shares the common goal of most generalised ABM simulators, of providing a layer of abstraction which permits modellers to focus on the development of agent based models rather than on their implementation. Related work has focused on high performance implementations of agent based simulation tools within distributed environments,[2,5] or alternatively has sought to accelerate simulations within particular domains or for specific models.[27-29] There have been other recent works which focus on software engineering challenges related to agent based simulation tools, in particular making software accessible in

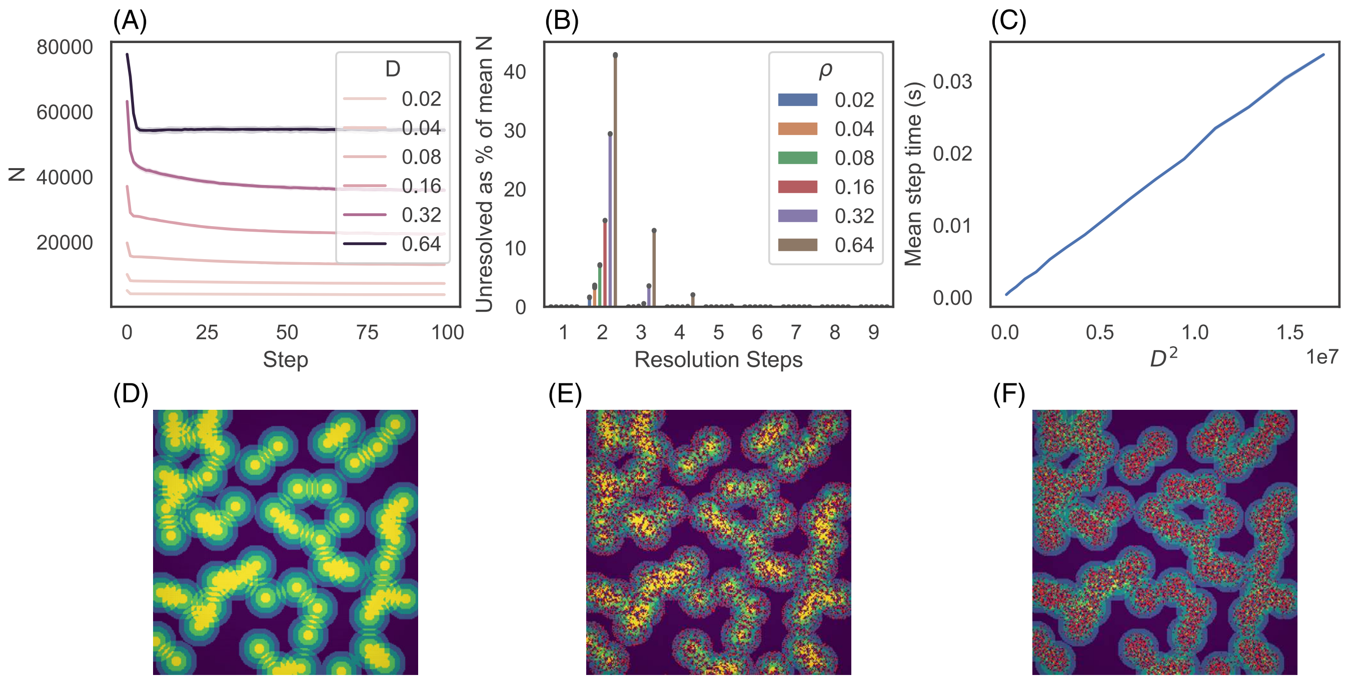


**FIGURE 11** (A) the agent population size $N$ over 100 simulation steps for varying $N_{init}$ via the initial probability of occupation $\rho$. The environment width ($D$) is fixed at 512. (B) The mean number of resolution steps (over three iterations of the model) required within the sub-model to resolve conflict in agent movement when competing for locations with high resources. The number of unresolved agents is calculated as a percentage of the mean population size over three iterations of the model. The initial probability of occupation $\rho$ is varied. (C) Performance scaling of the Sugarscape model showing the mean step time for 3 runs of the model over 100 benchmark steps. (D) Visualisation of the initial sugar distribution $s_x$ for each location of the Sugarscape environment with an environment width $D = 256$. Image shows the positions of $P = 54$ sugar peaks with a colour distribution of $s_x = S_{\max} = yellow$ and $s_x = 0 = darkblue$. (E, F) Visualisation of the Sugarscape agents (red) and the Sugarscape environment after iteration 1 and 50 respectively.

high level languages such as Python.[30,31] This article is unique in attempting to address the software engineering challenge of providing a general purpose simulator capable of utilising GPU acceleration. A significant challenge of GPU accelerated programming is ensuring flexibility without sacrificing performance. In this work we have presented four novel methods to address this challenge. Our method of balancing performance with flexibility through minimisation of data movement demonstrates an ability to simulate millions of agents with a single GPU device paving the way for large scale simulations. Future work will consider whole tissue simulations of multi-scale biological cellular systems.[32] It has been shown that our benchmark model, which is representative of a wide range of continuous space, particle type models, is able to simulate 1$M$ agents in ~0.003 s per time step using spatial partitioning and our run time compilation approach. Our Sugarscape model demonstrating the use of sub modelling is able to simulate an environment size of 16M agents in ~1 second per time step. Run-time compilation of agent functions has been show to provide the greatest performance, as the mapping of an agent variable to memory avoids the need for a hash table approach. The use of run-time compilation is essential for supporting languages bindings in languages such as Python. A disadvantage of the run-time compilation approach is that there is a roughly one second run-time overhead on first execution of a simulation. This overhead is due to the expensive compilation process occurring during the simulation but this is quickly amortised in the cost of long running simulations or through re-use of the run-time compiled kernel cache on subsequent experiments.

Simulation sizes beyond those demonstrated in this work are feasible even on single GPU devices, subject to memory availability. To increase the scale of simulation beyond a single device, it is feasible that multi-device approaches could be applied.[33] The use cases for simulations within research at this scale are however far fewer than those of smaller population sizes which do not fully occupy a single GPU device. This is partially the result of modellers migrating from existing ABM simulators with limited support for large scale simulation and partly due to ever increasing parallelism within GPU architectural design. Pedestrian simulation models, for example, have previously been a popular area of application for GPU agent based simulation.[34] As device parallelism has improved, GPUs have quickly exceeded the demands of even very large crowd simulations. A Volta series GV100 GPU can require up to 160k resident threads to achieve full device occupancy. [‡] To ensure good utilisation of GPU devices for even small models, we have demonstrated two methods which focus on exploiting better use of GPU resources. The first permits efficient utilisation of GPUs for small simulations. In this case 'ensembles' are able to increase device utilisation by a factor of up to 8× which was shown to be most effective for small populations where communication is less dense. The second method exploits additional concurrency within the model itself to achieve higher utilisation demonstrating speedups of up to 14× compared to a non-concurrent case.

Our use of hierarchical modelling for conflict resolution and recursive behaviour abstracts these processes into reusable sub-models, permitting models to be expressed in a format which closely resembles their serial counterparts. FLAME GPU 2 also provides modellers with an extensive public simulation API, completely abstracting the GPU from modellers. The use of an API differs considerably from the approach taken in previous GPU accelerated modelling libraries. That is, in computational neuroscience, where GPU simulations are code generated from declarative model descriptions using templates. Instead modern C++ is used to provide a library which permits procedural and dynamic model specification with a run-time library which is not dependent on any external code generation process. The key to this approach is the ability to provide a mapping process of user-defined (string) variables to memory addresses at run-time. A significant advantage of an API based approach from a software engineering perspective is that code repetition is massively reduced as the library contains only single implementations of behaviours required for a particular model feature. Within FLAME GPU 1, the XML/XSLT template generated code could very easily become extremely large. For example, a recent FLAME GPU 1 implementation of an immune system model extended to over 60k lines of template generated code. The API based approach also permits more modular software design permitting extensions to the software. The implementation of new message communication patterns is an example of this which can be achieved with relative ease and without modifying complex templates. A modular architecture facilitates good software engineering design, promoting a separation of concerns and ensuring that verification of the simulator can take place independently of model validation. Future work will focus on graphical interfaces to lower the barrier to entry of high performance ABM further as well as providing domain specific models and interfaces for modelling using the FLAME GPU 2 API as simulation middle-ware. Further performance improvements will be realised by considering finer grained dependency analysis at the variable level. This would permit concurrent execution of agent functions from the same agent which operate on different variables but would require static analysis or manually labelling variable usage within agent functions.

## 6 | CONCLUSIONS AND FUTURE WORK

In this article we introduced the FLAME GPU 2 software. The article introduced novel methods which are incorporated into the software in order to demonstrate high performance without sacrificing modelling flexibility. This is vitally important for a general purpose simulator and ensures that the software can be applied to a wide range of research domains and modelling challenges. High computational performance has been demonstrated by ensuring efficient data movement as well as providing a mechanisms for targeting the increasing levels of parallelism which are available in each new generation of GPU architecture. A method for hierarchical and reusable sub-modelling has been demonstrated which acts as a generalised way to abstract conflict resolution and recursion. This ensures that models can be cleanly defined without conflating the model with the process of resolving conflict. The result of the work ensures that there can be a separation of concerns between modelling and simulation for performant agent based modelling on GPUs.

In future work we will consider alternative approaches to ensemble simulation which may permit higher device utilisation. One such approach may be to launch all simulations within the same kernel launch. This would minimise launch overhead but may be less preferential where simulations have different run-time duration. It is possible that run-time analysis of the model may be required to execute ensembles using the most efficient technique. An obvious area for further exploration is the use of multiple GPUs for simulation of a single large model. Although the single GPU implementation has been shown to support many millions of agents (and is constrained only by the availability of memory) only a small percentage of agents, in the order of 100's of thousands, can be in flight at once. Multi-GPU simulation offers the potential to split models to have increasing levels of simultaneous computation but requires additional research into how model splitting can be achieved to minimise data transfer overheads and ensure that computation is balanced.

[‡]Based on 80 Streaming multi-processors each with a maximum of 2048 threads in flight. Full device utilisation can be achieved with fewer threads if occupancy is limited by resources such as SM or registers.

## AUTHOR CONTRIBUTIONS

**Paul Richmond:** Conceptualization; investigation; software; experimentation; writing-original draft; writing-review and editing. **Robert Chisholm:** software; validation; experimentation, writing-review and editing. **Peter Heywood:** software; validation; experimentation, writing-review and editing. **Matthew Leach:** software; validation; experimentation, writing-review. **Mozhgan Kabiri Chimeh:** software; validation; writing-review.

## ACKNOWLEDGMENTS

The work in this article has been undertaken as part of an EPSRC RSE Fellowship (EP/N018869/1) held by Paul Richmond and which has supported Mozhgan Kabiri-Chimeh and Matthew Leach. Peter Heywood has contributed to the work via the EU H2020 STriTuVaD project and Robert Chisholm has contributed via the EU H2020 PRIMAGE project. We acknowledge IT Services at The University of Sheffield for the provision of the High Performance Computing Service.

## DATA AVAILABILITY STATEMENT

The data used for generating the plots, along with plotting scripts are available in separate code repositories (see next section). The raw data from our simulation runs is committed to the repository used to generate the figures with details of how to reproduce the figures included within the included `README.md` document.

FLAME GPU 2 v2.0.0-rc[24] is open source under a permissive MIT licence but is available for download from GitHub with links on the FLAME GPU website http://www.flamegpu.com. The main simulator code and all benchmarks used within this article are hosted as separate GitHub repositories. The simulator code repository actively uses issues to track bugs and feature requests with contribution guidelines published in the repository. The code has been developed by a team of Research Software Engineers and embraces the FAIR for Software principles.[35] The software used within this article makes the following attempts to apply the FAIR4RS principles.

- Findable: The simulator has a release tag and DOI, each figure repository has a pre-release tag (and will be assigned a DOI on publication). The software and figure repositories have a defined licence, and authorship is described in the `Readme.md` along with version control commit history. F3: The simulator `Readme.md` defines a DOI to the latest release.
- Accessible: The software simulator and models used for the figures are available as open source code on a public repository with a clearly defined and permissive MIT licence.
- Interoperable: The simulator API is documented both with a user guide and with detailed API documentation: http://docs.flamegpu.com.
- Reusable: The software is built using CMake with clearly defined versioning of dependencies where necessary. The code used to generate figures has a provided `requirements.txt` file and instructions within the *Readme.md* on creating a Python environment using conda.

Each of the figures produced for this article have their own GitHub repository containing, model code, results and figures. Each of the figure repositories uses a tagged release (or commit) of the main simulator and have a release tag for submission to ensure results are reproducible. DOIs will be created for each release tag at publication stage. The current tags are available in the links for each figure respectively; Figure 7 (https://github.com/FLAMEGPU/FLAMEGPU2-circles-benchmark-vis/releases/tag/v1), Figure 8 (https://github.com/FLAMEGPU/FLAMEGPU2-circles-benchmark/releases/tag/v1), Figure 9 (https://github.com/FLAMEGPU/FLAMEGPU2-ensemble-benchmark/releases/tag/v1), Figure 10 (https://github.com/FLAMEGPU/FLAMEGPU2-concurrency-benchmark/releases/tag/v1), and Figure 11 (https://github.com/FLAMEGPU/FLAMEGPU2-submodel-benchmark/releases/tag/v1).

## ORCID

*Paul Richmond* https://orcid.org/0000-0002-4657-5518
*Robert Chisholm* https://orcid.org/0000-0003-3379-9042
*Peter Heywood* https://orcid.org/0000-0001-9277-8394
*Mozhgan Kabiri Chimeh* https://orcid.org/0000-0002-2561-7926
*Matthew Leach* https://orcid.org/0000-0002-8901-5609

## REFERENCES

1. Bonabeau E. Agent-based modeling: Methods and techniques for simulating human systems. *Proc Natl Acad Sci*. 2002;99(supp 3):7280-7287. doi:10.1073/pnas.082080899
2. Abar S, Theodoropoulos GK, Lemarinier P, O'Hare GMP. Agent based modelling and simulation tools: a review of the state-of-art software. *Comput Sci Rev*. 2017;24:13-33. doi:10.1016/j.cosrev.2017.03.001.
3. Bausch AR, Kroy K. A bottom-up approach to cell mechanics. *Nat Phys*. 2006;2(4):231-238. doi:10.1038/nphys260
4. Ghosh-Dastidar S, Adeli H. Spiking neural networks. *Int J Neural Syst*. 2009;19(04):295-308. doi:10.1142/S0129065709002002
5. Collier N, North M. Parallel agent-based simulation with repast for high performance computing. *Simulation*. 2013;89(10):1215-1235. doi:10.1177/0037549712462620
6. Cosenza B, Cordasco G, De Chiara R, Scarano V. Distributed load balancing for parallel agent-based simulations. Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing; 2011:62–69. doi:10.1109/PDP.2011.22
7. Coakley S, Gheorghe M, Holcombe M, Chin S, Worth D, Greenough C. Exploitation of high performance computing in the FLAME agent-based simulation framework. Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems; 2012:538–545. doi:10.1109/HPCC.2012.79
8. Mittal S, Vetter JS. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput Surv*. 2015;47(4):69. doi:10.1145/2788396
9. Owens JD, Luebke D, Govindaraju N, et al. A survey of general-purpose computation on graphics hardware. *Comput Graph Forum*. 2007;26(1):80-113. doi:10.1111/j.1467-8659.2007.01012.x
10. Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library. In: Wallach H et al., eds. *Advances in Neural Information Processing Systems*. Vol 32. Curran Associates, Inc.; 2019:8024-8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
11. Abadi M, Agarwal A, Barham P, et al. TensorFlow: large-scale machine learning on heterogeneous systems; 2015. https://www.tensorflow.org/
12. Xiao J, Andelfinger P, Eckhoff D, Cai W, Knoll A. A survey on agent-based simulation using hardware accelerators. arXiv preprint arXiv:1807.01014 [cs.MA], 2018.
13. Li X, Cai W, Turner SJ. Supporting efficient execution of continuous space agent-based simulation on GPU. *Concurr Comput*. 2016;28(12):3313-3332. doi:10.1002/cpe.3808
14. Richmond P, Walker D, Coakley S, Romano D. High performance cellular level agent-based simulation with FLAME for the GPU. *Brief Bioinform*. 2010;11(3):334-347. doi:10.1093/bib/bbp073
15. Richmond P, Romano D. Template-driven agent-based modeling and simulation with CUDA. In: Hwu W-m W, ed. *GPU Computing Gems Emerald Edition*. Elsevier; 2011:313-324.
16. Balanescu T, Cowling AJ, Georgescu H, Gheorghe M, Holcombe M, Vertan C. Communicating stream X-machines systems are no more than X-machines. *J Univers Comput Sci*. 1999;5:494-507.
17. Wooldridgey M, Ciancarini P. Agent-oriented software engineering: the state of the art. In: Ciancarini P, Wooldridge MJ, eds. *Agent-Oriented Software Engineering*. Springer; 2000:1-28.
18. Epstein JM, Axtell RL. *Growing Artificial Societies: Social Science from the Bottom Up*. Vol 1. The MIT Press; 1996.
19. Schelling TC. Dynamic models of segregation. *J Math Sociol*. 1971;1(2):143-186. doi:10.1080/0022250X.1971.9989794
20. Chimeh MK, Heywood P, Pennisi M, Pappalardo F, Richmond P. Parallel pair-wise interaction for multi-agent immune systems modelling. Proceedings of the 2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM); 2018:1367–1373. doi:10.1109/BIBM.2018.8621404
21. Chisholm R, Richmond P, Maddock S. A standardised benchmark for assessing the performance of fixed radius near neighbours. In: Desprez F, Dutot P-F, Kaklamanis C, et al., eds. *Euro-Par 2016: Parallel Processing Workshops*. Springer International Publishing; 2017:311-321.
22. Christley S, Lee B, Dai X, Nie Q. Integrative multicellular biological modeling: a case study of 3D epidermal development using GPU algorithms. *BMC Syst Biol*. 2010;4(1):107. doi:10.1186/1752-0509-4-107
23. Ghaffarizadeh A, Heiland R, Friedman SH, Mumenthaler SM, Macklin P. PhysiCell: an open source physics-based cell simulator for 3-D multicellular systems. *PLoS Comput Biol*. 2018;14(2):1-31. doi:10.1371/journal.pcbi.1005991
24. Richmond P, Chisholm R, Heywood P, Leach M, Chimeh MK. FLAME GPU. Version 2.0.0-rc. If you use this software, please cite it using these metadata; December 2022. doi:10.5281/zenodo.7434228
25. Reynolds CW. Flocks, herds and schools: a distributed behavioral model. *ACM SIGGRAPH Comput Graph*. 1987;21(4):25-34.
26. Tisue S, Wilensky U. Netlogo: a simple environment for modeling complexity. Proceedings of the International Conference on Complex Systems, Vol. 21, Boston, MA; 2004:16–21.
27. Stack M, Macklin P, Searles R, Chandrasekaran S. OpenACC acceleration of an agent-based biological simulation framework. arXiv preprint arXiv:2110.13368 [cs.DC], 2021.
28. Husselmann A, Hawick K. Simulating species interactions and complex emergence in multiple flocks of boids with GPUS. Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems; September 2011. doi:10.2316/P.2011.757&hyphen;012
29. De Chiara R, Erra U, Scarano V, Tatafiore M. Massive simulation using GPU of a distributed behavioral model of a flock with obstacle avoidance. Proceedings of Vision, Modeling and Visualization 2004 (VMV); January 2004:233–240.

30. Nourisa J, Zeller-Plumhoff B, Willumeit-Römer R. CppyABM: An open-source agent-based modeling library to integrate C++ and Python. *Softw Pract Exp*. 2022;52(6):1337-1351. doi:10.1002/spe.3067
31. Jaxa-Rozen M, Kwakkel JH. PyNetLogo: linking NetLogo with Python. *J Artif Soc Soc Simul*. 2018;21(2):4. doi:10.18564/jasss.3668
32. Martí-Bonmatí L, Alberich-Bayarri Á, Ladenstein R, et al. PRIMAGE project: predictive in silico multiscale analytics to support childhood cancer personalised evaluation empowered by imaging biomarkers. *Eur Radiol Exp*. 2020;4(1):1-11.
33. Aaby BG, Perumalla KS, Seal SK. Efficient simulation of agent-based models on multi-GPU and multi-core clusters. Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools'10). Torremolinos, Malaga: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering); 2010. doi:10.4108/ICST.SIMUTOOLS2010.8822
34. Karmakharm T, Richmond P, Romano DM. Agent-based large scale simulation of pedestrians with adaptive realistic navigation vector fields. In: Collomosse J, Grimstead I, eds. *Theory and Practice of Computer Graphics*. Vol 10. The Eurographics Association; 2010:67-74.
35. Neil P, Hong C, Katz DS, et al. FAIR principles for research software (FAIR4RS Principles); 2021. doi:10.15497/RDA00065

**How to cite this article:** Richmond P, Chisholm R, Heywood P, Chimeh MK, Leach M. FLAME GPU 2: A framework for flexible and performant agent based simulation on GPUs. *Softw Pract Exper*. 2023;1-22. doi: 10.1002/spe.3207