Received: 17 November 2019 | Revised: 4 April 2020 | Accepted: 7 April 2020 | Published on: 2 June 2020

DOI: 10.1002/cpe.5807

SPECIAL ISSUE PAPER

WILEY

OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms

Jiajian Xiao[1,2] | Philipp Andelfinger[3] | Wentong Cai[4] | Paul Richmond[5] | Alois Knoll[2,4] | David Eckhoff[1,2]

[1]TUMCREATE, Singapore, Singapore

[2]Department of Informatics, Technische Universität München, Munich, Germany

[3]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee,

[4]School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

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

Correspondence
Jiajian Xiao, TUMCREATE, 1 Create Way, 10-02, 138602 Singapore.
Email: jiajian.xiao@tum-create.edu.sg

Summary

The execution of agent-based simulations (ABSs) on hardware accelerator devices such as graphics processing units (GPUs) has been shown to offer great performance potentials. However, in heterogeneous hardware environments, it can become increasingly difficult to find viable partitions of the simulation and provide implementations for different hardware devices. To automate this process, we present OpenABLext, an extension to OpenABL, a model specification language for ABSs. By providing a device-aware OpenCL backend, OpenABLext enables the co-execution of ABS on heterogeneous hardware platforms consisting of central processing units, GPUs, and field programmable gate arrays (FPGAs). We present a novel online dispatching method that efficiently profiles partitions of the simulation during run-time to optimize the hardware assignment while using the profiling results to advance the simulation itself. In addition, OpenABLext features automated conflict resolution based on user-specified rules, supports graph-based simulation spaces, and utilizes an efficient neighbor search algorithm. We show the improved performance of OpenABLext and demonstrate the potential of FPGAs in the context of ABS. We illustrate how co-execution can be used to further lower execution times. OpenABLext can be seen as an enabler to tap the computing power of heterogeneous hardware platforms for ABS.

KEYWORDS

agent-based simulation, FPGA, heterogeneous hardware, OpenABL, OpenCL

1 | INTRODUCTION

In agent-based simulation (ABS), each agent (eg, a pedestrian or a vehicle) is an autonomous entity that is defined by models that specify how the agent behaves and how it interacts with other agents and the environment. This type of simulation has found application in various domains such as transport, computer networks, biology, and social sciences.[1] With increasingly complex models, larger number of agents, and bigger simulation spaces, ABS often suffers from long execution times.

The prevalent way to approach this problem and speed up ABS to meet these performance needs is to make use of parallel and distributed simulation techniques. The increasing availability of heterogeneous hardware composed of central processing units (CPUs) and *accelerators* such as graphics processing units (GPUs) or field programmable gate arrays (FPGAs) has opened up new possibilities to enhance the performance of ABS.[2] For instance, it can be beneficial to offload certain segments to an accelerator where they can be executed faster or to enable co-execution of the simulation using more than one device in parallel.

The challenge for the simulationist is then to optimize the simulation code for the target hardware platform, degrading maintainability as well as portability to other hardware platforms.[3] The introduction of a new abstraction layer in the form of a domain-specific language can help alleviate this problem. In the context of ABSs, OpenABL[4] has been proposed to enable code generation from high-level model and scenario specifications using a C-like syntax. It features a number of *backends* to generate parallelized code for CPUs, GPUs, clusters, or cloud environments.

The original OpenABL targeted one specific type of hardware platform, that is, co-execution on combinations of CPUs, GPUs, and FPGAs was not possible. This leaves a large range of computational resources untapped, even though previous work has demonstrated high hardware utilization using co-execution.[5] However, once enabled, co-execution poses the additional challenge of determining a suitable combination of hardware devices for execution to maximize performance. Furthermore, the original OpenABL limited the simulation environment to continuous 2D or 3D spaces, which excludes graph-based simulation spaces as commonly used in domains such as road traffic and social sciences. Finally, OpenABL did not provide a mechanism for conflict resolution, requiring modelers to manually provide code to detect and resolve situations where multiple agents request the same resources.

In this article, we address these limitations by introducing OpenABLext, an extension to the OpenABL language. Our contributions can be summarized as follows:

- We improve the declaration of simulation environments to feature user-defined types, the possibility to support graph-based simulation spaces, and an efficient neighbor search for GPUs and FPGAs to achieve efficient memory access.
- We present an OpenCL backend to support automatic code generation for CPUs, GPUs, and FPGAs.
- We propose an online dispatching method to optimize the hardware assignment during co-execution.
- We define an interface that enables the generation of parallel conflict resolution code based on user-specified rules.

OpenABLext is open-source and available online (https://github.com/xjjex1990/OpenABLext). This manuscript constitutes an extended version of our previous work,[6] including simulation code generation and execution for FPGAs, online dispatching to dynamically assign step functions to hardware devices, parallel conflict resolution for both 2D/3D and graph-based simulations, and a more comprehensive evaluation of the performance gains achieved with different hardware configurations.

The remainder of the article is organized as follows: In Section 2, we introduce OpenABL, OpenCL, and FPGA fundamentals and discuss related work in the respective fields. We present OpenABLext in detail in Section 3 and evaluate its performance in Section 4. Section 5 summarizes our work and concludes the article.

2 | RELATED WORK AND BACKGROUND

2.1 | Related work

The acceleration of ABS through parallelization has received wide attention from the research community. A number of CPU-based frameworks such as MASON,[7] Repast,[8] Swarm,[9] or FLAME[10] simplify the process of developing parallel ABS. Variants that exploit CPU-based parallelization or distributed execution include D-MASON[11] and Repast-HPC.[12] However, these frameworks target CPU-based hardware and require modelers to be knowledgeable in parallel or distributed computing.

A comprehensive survey of existing techniques to accelerate ABSs with the use of hardware accelerators is presented in Reference 2. A number of papers propose frameworks that abstract away from hardware specifics in order to simplify the porting to hardware accelerators. One of these frameworks is FLAME GPU,[13] which is an extension of FLAME. It provides a template-driven framework for agent-based modeling targeting GPUs based on a state machine model called X-machine. The many-core multi-agent system (MCMAS)[14] provides a Java-based toolkit that supports a set of pre-defined data structures and functions called plugins to abstract from native OpenCL code. Agent models can be implemented using these data structures or plugins. In contrast to our work, MCMAS and FLAME GPU target GPUs only.

To achieve domain-independent code generation for heterogeneous platforms, methods such as pattern-matching to detect parallelizable C snippets[15] and the use of code templates[16] have been proposed. Most of these approaches focus on detecting localized sections of the source code that can be parallelized, for example, nested loops with predictable control flows, whereas our work addresses the more irregular control flow of ABSs.

Generating OpenCL code for FPGAs from sequential loop-based code for grid-based applications was studied in References 17 and 18. To the best of our knowledge, we are the first to generate OpenCL code for FPGAs in the context of ABS, which on the one hand usually exhibits more irregular workloads, but on the other hand also exhibits common computational patterns that can be utilized for better performance. Open accelerators (OpenACC)[19] and open multi-processing (OpenMP, version 4.0 and above) provide directive-based application programming interfaces to parallelize general sequential code to run on heterogeneous systems. There are also efforts to translate OpenACC code to OpenCL targeting FPGAs.[20] Unlike

OpenCL, both OpenACC and OpenMP provide only limited control over low-level performance-critical aspects such as memory access patterns and control flow. Furthermore, as general programming standards, they do not include domain-specific optimizations as our framework does for ABS. Domain-specific languages (DSLs) are another group of methods to simplify development for high-performance code generation on heterogeneous hardware platforms.[21,22] However, to the best of our knowledge, none of the existing works consider ABS.

One of the major challenges for the efficient use of heterogeneous hardware environments is the partitioning of the workload and the assignment of these partitions to the most suitable hardware device. To this end, machine-learning based approaches[23-25] have been proposed. However, these methods usually require substantial offline training. The runtime behavior of ABS not only potentially varies based on input parameters but can also substantially change over the course of a simulation run, requiring regular retraining of machine-learning models to achieve good performance. To partition generic programs dynamically at runtime, several authors have shown that partitioning on a data level is a viable option for both regular and irregular problems.[26-29] Some works[30-32] tackle the problem of accelerating numerical algorithms such as matrix multiplication and the fast Fourier transform on heterogeneous systems using data partitioning approaches. In ABSs, however, we observe a strong locality of dependencies, as agents primarily interact with nearby agents, that is, their neighbors. This can cause additional application-specific overhead when generic data-level partitioning methods are applied to ABS, for example, by regularly repartitioning data geographically.[33] Therefore, we propose a generalizable approach to partition the workload on the function level. Other works that partition on the function level either require periodic re-evaluation of the current assignment to adjust to the irregularly evolving workload,[34] or they need to profile the hardware and program offline.[35]

To reduce overhead, we propose a heuristic to dynamically trigger re-evaluation of the hardware assignment. Furthermore, the computations carried out as part of the re-evaluation at runtime are used to advance the simulation. Unlike the above mentioned methods, our approach is tailored to ABSs to reduce the assignment overhead.

Another challenge in parallel ABS is that conflicting actions may occur, for example, when two agents move to the same position at the same point in time. Approaches proposed to detect and resolve such conflicts typically rely either on the use of atomic operations during the parallel agents updates[36] or on enumerating the agents involved in conflicts once an update cycle has completed.[37] In both cases, the winner of each conflict is determined according to a tie-breaking policy, which may be stochastic or rely on model-specific tie-breaking rules. A taxonomy and performance evaluation of the conflict resolution methods from the literature is given by Yang et al.[38] In the present work, we provide a generic interface to define a conflict search radius and a tie-breaking policy from which low-level code is generated automatically.

2.2 | OpenABL

OpenABL is a domain-specific language to describe the behavior of ABSs and a framework to generate code targeting various execution platforms.[4] It acts as an intermediate layer to generate parallel or distributed time-stepped ABS, given sequential simulation code written in a C-like language. An overview of the OpenABL framework is shown in Figure 1. The framework consists of a *frontend* and a *backend*. Listing 1 shows an example of frontend OpenABL code, where users can define agents with a mandatory position attribute (keyword `agent`, L.1), constants (keyword `param`, L.3-5), simulation environments (keyword `environment`, L.7), step functions (keyword `step`, L.9-13), and a main function (keyword `main()`, L.15-18).

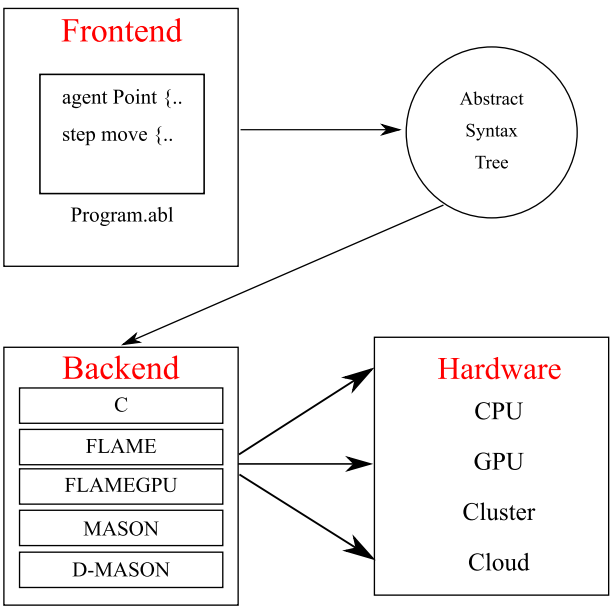


FIGURE 1 An overview of the OpenABL framework

The OpenABL compiler parses this code and compiles it to an Intermediate representation (IR) called an abstract syntax tree (AST). The AST IR is then further relayed to one of the available backends. The backend reconstructs simulation code from the AST IR and generates parallel code for the step functions that can then be executed on CPUs, GPUs, clusters, or cloud environments. OpenABL supports the following backends: C, FLAME,[10] FLAME GPU,[13] MASON,[7] and D-MASON.[11]

```
agent Agent { position float2 coordinate; }

param int num_of_agents = 1000;
param int sim_steps = 100;
param float env_size = 100;

environment { max : float2 (env_size) }

step stepFunc1(Agent in -> out) {
  for (Agent neighbours: near (in, 5) {
        /* do something to the agent */
      }
}
step stepFunc2(Agent in -> out) { ... }

void main() {
  ... /* initialise the agents */
  simulate(sim_steps) { stepFunc1, stepFunc2 }
}

```

LISTING 1 Example OpenABL simulation definition

2.3 | Open computing language (OpenCL)

The OpenCL is a framework that allows users to write parallel programs in a C-like language without considering low-level hardware specifics. OpenCL is supported by a wide range of hardware types including CPUs, GPUs, accelerated processing units (APUs), and FPGAs, making it the logical choice for heterogeneous hardware environments. An OpenCL execution environment is comprised of a host (usually a CPU) and one or multiple devices (eg, other CPUs, GPUs, or FPGAs). A host program initializes the environment, control, memory, and computational resources for the devices. A device program consists mainly of the so-called kernels that implement the computational tasks. Threads that process the tasks are referred to as work-items. Performance improvements over sequential execution are achieved by processing many work-items in parallel (for GPUs) or by pipelining (for FPGAs). However, as different devices usually have their own memory, performance penalties are to be expected when moving data between devices, usually realized via a latency-prone PCI-E link.

2.4 | Field-programmable gate array

An FPGA is a programmable integrated circuit comprised of programmable logic gates and reconfigurable connections between those gates. A typical way to program an FPGA is to write code in a hardware description language (HDL) such as Verilog[39] or VHDL.[40] FPGA vendors such as Intel or Xilinx also provide SDKs to translate OpenCL programs to HDL code and then to bitstream. The framework proposed in this article makes use of this capability in order to target FPGAs along with CPUs and GPUs based on the same high-level model specification. In this work, we will utilize the terminology and programming syntax from Xilinx. Similar terminology and syntax is used by the Intel OpenCL SDK for FPGAs.

Unlike OpenCL programs targeting GPU, which launch large numbers of work-items to process a large volume of data in parallel yielding the so-called single instruction multiple data (SIMD) parallelism, high-level frameworks targeting FPGAs typically achieve parallelism through pipelining:[41] Given a loop repeating a computational task, the involved memory or arithmetic operations are compiled to circuits called functional units. The pipeline is formed by sequencing the functional units according to the data flow required by the task. Each iteration of the task exercises the pipeline stages sequentially, allowing multiple tasks to be overlapped. In an optimally pipelined case, as soon as a functional unit becomes unoccupied, a subsequent task makes use of the unit. The number of clock cycles between two consecutive tasks is called the initiation interval (II). The performance of a pipelined FPGA design is strongly affected by the II and the operating frequency. Dependencies across tasks may cause larger II due to one needing to wait for the input from the previous task. The operating frequency is dictated by the cycles spent on the most time-consuming pipeline stage.

The pipelining described above is applied by the compiler if the OpenCL program is specified in terms of the so-called *single work-item* kernels containing loops, which permit code development in a sequential style. SIMD parallelism is exploited by *NDRange* kernels, which can be executed

by multiple work-items in parallel and thus require the developer to carefully consider dependencies. In contrast to pipelining, NDRange kernels require functional units to be replicated to enable the parallel execution of work-items, which can increase the required FPGA hardware resources for a given functionality. As pipelining is the dominant method to achieve parallel execution through HLS for FPGAs, our framework relies on single work-item kernels.

3 | FROM OPENABL TO OPENABLEXT

In this section, we propose OpenABLext, an extension to the OpenABL language and framework, to support a wider range of simulation models as well as additional types of hardware. Our extensions, each described in detail in the following subsections, include: (i) The support of a more extensive environment declaration, featuring user-defined types, the possibility to support graph-based simulation space, and a more efficient neighbor search to achieve efficient memory access. (ii) The introduction of an OpenCL backend with specific compilation rules for GPUs and FPGAs as well as support for multi-device co-execution. (iii) An online dispatcher that profiles step functions and assigns them to the most suitable hardware device. (iv) A conflict resolution mechanism which automatically detects conflicts and resolves them using a tie-breaking function.

3.1 | User-specified environments

The original OpenABL limits the simulation environment to continuous 2D or 3D spaces, parameterized by the `max, min`, and `granularity` attributes in the `environment` declaration. Furthermore, user-defined types can only be used to define agent types, and not in function bodies or the environment, further complicating model specification. We extend the OpenABL syntax and frontend to lift these limitations.

User-defined types for arbitrary variables in function bodies as well as in the definition of the simulation environment can be specified as shown in the following example:

```
Lane {
int laneId;
float length;
int nextLaneIds[MAX_LANE_CONNECTIVITY]; }
environment { env : Lane lanes[env_size] }
```

The keyword `env` inside the environment declaration defines the simulation environment. It accepts an environment array of all native types supported by the original OpenABL as well as user-defined types. In this example, the environment is defined as an array of the user-defined type `Lane`. The `Lane` type encapsulates a lane's identifier, its length, and its connections to other lanes.

Accelerators typically employ a memory hierarchy composed of *global memory* accessible to all work-items and one or more types of *local memory* accessible to groups or individual work-items. *Global memory* is used for massive data storage, for example, an array of all agents, while *local memory* only holds the agent and a set of relevant agents to be processed by each or a group of work-items. Due to the high latency of global memory accesses, data locality is an important consideration in ABS development:[4] accesses of adjacent work-items to adjacent memory addresses can frequently be coalesced, that is, translated to a single memory transaction, allowing for peak memory performance on OpenCL devices such as GPUs. In common ABS models, agents tend to only interact with agents within a certain radius or with agents on the same edge in a graph environment. To achieve data locality during execution, we implemented the efficient neighbour search method presented in.[42] Spatial locality is exploited by partitioning the simulation space into a grid of cells. Each cell's side length equals the largest search radius that appears in the model. When searching for neighbors, only the agents in the same or adjacent cells are considered. In the original OpenABL, data locality is achieved in 2D or 3D space by specifying a radius using the following neighbor search query:

```
for (AgentType neighbours : near (currentAgent, radius))
```

We extend the language to allow for a similar neighbor search query for graph-based models:

```
for (AgentType neighbours : on (env))
```

Given traffic simulation as an example where edges in a graph represent lanes, the following query retrieves all agents on a lane:

```
for (Vehicles neighbors : on (lanes[currentVehicle.currentLane]))
```

Coalesced memory access is achieved by always keeping the array of agents in the global memory sorted according to the individual dimensions of the `position` attributes. Each element in the environment array keeps track of its start and end address in global memory. As illustrated in Figure 2A, two attributes `mem_start` and `mem_end` record the start and end address of each single lane in the global array of agents. The two attributes are updated after all step functions have terminated. When the neighbor search query is called, instead of iterating through global memory, only a chunk of memory needs to be visited. In a graph-based setting, the chunk of memory is indicated by `env.mem_start` and `env.mem_end`. For 2D or 3D simulation spaces, we load the chunks of memory holding the agents in the current cell and all the neighboring cells (Figure 2B).

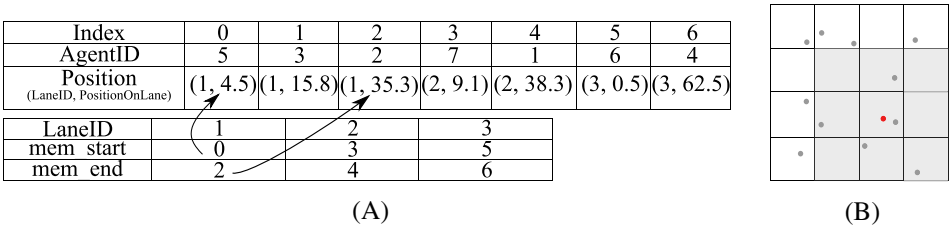


FIGURE 2 Coalesced memory access in the generated OpenCL code. A, Agents are sorted by their position (eg, EdgeID and PositionOnEdge). Each element in the environment array keeps a `mem_start` and a `mem_end` pointer to its agents in global memory. B, In a grid with cell width at least the search radius, the neighbor search of the red agent loads itself and adjacent cells

3.2 | OpenCL code generation for heterogeneous hardware

The architecture of OpenABL allows the addition of new backends without modifying the frontend, which allows us to support heterogeneous hardware environments by adding an OpenCL backend. OpenCL enables *co-execution* across multiple devices of different types, while the existing backends only support single-platform execution. With this new backend, we aim to allow modelers to fully utilize the available hardware without specifying simulation code for each device manually.

To allow co-execution, we extend the syntax of the `simulate(sim_steps)` statement so that each step function can be annotated with the identifier of the OpenCL device on which the step function should execute, for example, `simulate(sim_steps) {stepFunc1(0), stepFunc2(1)}`.

If all the step functions are assigned to the same hardware, OpenABLext will generate code targeting this single device. Otherwise, OpenABLext will assume a co-execution setting with the requirement for the users to define a `merge` function. The `merge` function describes how the output data from different devices should be merged per simulation step. The co-execution scenario will be discussed in Section 3.2.3.

In the following, we will discuss how the structure of the generated code changes depending on the target device (GPU, FPGA, and co-execution). To this end, we assume Listing 1 as the input of OpenABLext.

3.2.1 | OpenCL for GPUs and CPUs

The OpenCL backend takes as input the AST IR generated by the OpenABL frontend. The output of the OpenCL backend consists of a host program and a device program, compiled for the respective devices. Listing 2a and Listing 2b show pseudo code for the host and device programs (extraneous details omitted; assumed input OpenABLext syntax as shown in Listing 1). Agents, the environment, constant declarations, and all auxiliary functions are duplicated in both the host and device programs, as they may be referenced on either side.

The generated host program initializes the device, allocates the required memory, and initializes the agent state variables as well as the environment.

```
... /* function and variable declarations */
int main()
{
  initialise(agentArray[LENGTH]);
  initialise(clEnvironment, clDeviceBuffers);
  clWriteBuffer(agentArray[LENGTH],
   clDeviceBuffers);

  for (int i=0;i<NUM_STEPS;i++) {
    clExecuteNDRangeKernel(compute_kernel_1);
    clExecuteNDRangeKernel(compute_kernel_2);
  }

  clReadBuffer(clDeviceBuffers,
  agentArray[LENGTH]);
}
```

(A) Host code

```
... /* function and variable declarations */
void stepFunc1(Agent *agent) {
  for( agents inside the nine surrounding cells ) {...}
}
void stepFunc2(Agent *agent) {...}

__kernel void compute_kernel_1(__global Agent *agentArray)
{
  if ( get_global_id(0) >= LENGTH ) return;
  stepFunc1(&agentArray[id]);
}
__kernel void compute_kernel_2(__global Agent *agentArray)
{
  if ( get_global_id(0) >= LENGTH ) return;
  stepFunc2(&agentArray[id]);
}
```

(B) Device caode.

LISTING 2 Generated (pseudo) code for host and devices programs for GPUs and CPUs

One `compute_kernel` function is created for each step function in the device program. They are called in the sequence defined in the `simulate(sim_steps)` body to ensure dependencies between step functions are respected. In the step functions, the appearances of `on` or `near` statement is replaced by the neighbour search method detailed in Section 3.1. Parallel bitonic sort is used to efficiently sort the agents in the global memory. The work-items execute in parallel with each one processing one agent's step functions on the device.

The main loop of the simulation located in the host program calls the `compute_kernel` iteratively until the step count defined in the parameter of `simulation()` has been reached.

3.2.2 | OpenCL code generation for FPGAs

Just as the OpenCL code for GPUs and CPUs, the code generated for the FPGA also takes the AST IR as input. Again, a host program (Listing 3a) for initiating the OpenCL and simulation environment and a device program (Listing 3b) with one `compute_kernel` function is generated. The difference compared to GPU or CPU code is that the main loop of the simulation resides in the body of the `compute_kernel` function, following the single-work-item kernel guidelines described in Section 2.4. The main body of the `compute_kernel` function is a nested loop with the outer loop counting simulation steps. Each step function in the `simulate` body creates an inner loop which iterates through the agents, calling the designated step function. OpenABL employs a double buffering mechanism where two different buffers are used to store agent's states before and after executing each step function.[4] The read and write buffers are, therefore, swapped after each step function as shown in Listing 3b. The `on` or `near` statement is replaced by the efficient neighbor search method. Radix sort, rather than bitonic sort is used to sort the agents, which has been shown to perform well on FPGAs.[43]

When pipelining a loop on FPGAs, dependencies across tasks increase the initial interval (II) between tasks. As in each simulation time step, the agents react to the states updated in the last time step, pipelining the outer loop causes the pipeline to stall until the last time step finishes, resulting in a sequential execution of the outer loop. We thus only consider pipelining the inner loop. All step functions as well as the updating of `mem_start` and `mem_end` pointers are pipelined with the minimum possible IIs. Notably, dependencies inside step functions, for example, looping through agents to accumulate a value, can still cause a large II. Furthermore, radix sort iterates through the data digit by digit. Since each iteration relies on the results produced in the last iteration, sorting of agents is also not fully pipelined.

```
... /* function and variable
   declarations */
int main()
{
  initialise(agentArray[LENGTH]);
  initialise(clEnvironment,
   clDeviceBuffers);
  clWriteBuffer(agentArray[LENGTH],
    clDeviceBuffers);
  clExecuteKernel(compute_kernel);
  clReadBuffer(clDeviceBuffers,
   agentArray[LENGTH]);
}
```

(A) Host code

```
... /* function and variable declarations */
void stepFunc1(Agent *agentRead, Agent *agentWrite) {
#pragma pipeline_loop
  for( agents inside the nine surrounding cells )
  {...}
}
void stepFunc2(Agent *agent) {...}

__kernel void compute_kernel(__global Agent *agentArrayReadBuffer,
   __global Agent *agentArrayWriteBuffer)
{
  for (int i = 0 ; i < NUM_STEPS ; i++)
  {
    for (int j = 0 ; j < NUM_AGENTS ; j++)
      stepFunc1(&agentArrayReadBuffer[j], &agentArrayWriteBuffer[j]);
    agentArrayReadBuffer = agentArrayWriteBuffer;
    for (int j = 0 ; j < NUM_AGENTS ; j++)
      stepFunc2(&agentArrayReadBuffer[j], &agentArrayWriteBuffer[j]);
  }
  agentArrayReadBuffer = agentArrayWriteBuffer;
}
```

(B) Device code

LISTING 3 Generated (pseudo) code for host and devices programs for FPGA

3.2.3 | Code generation for co-execution

In a co-execution setting, the output of the OpenCL backend consists of a host program (Listing 4a) and multiple device programs, one for each available device (Listing 4b and Listing 4c). One `compute_kernel` function is created for each step function assigned to this device. On each device, the work-items execute in parallel with each one processing one agent's step function. This means that parallelization is achieved not only on the device-level with many devices working in parallel but also inside the devices where multiple work-items are executed simultaneously. It is also possible that two or more devices process different step functions on the same agent at the same time (Figure 3).

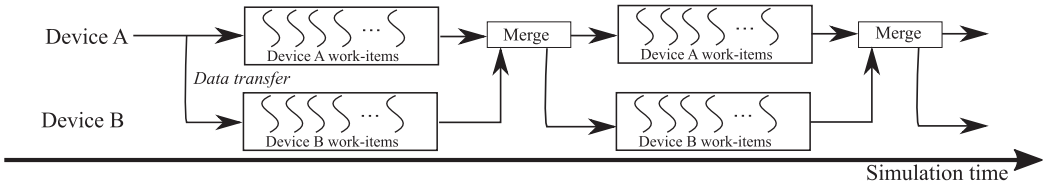


FIGURE 3 Co-execution on device A (acts as host) and B. Each work-item of device A and device B processes the step functions assigned to them in parallel. After that, the data is merged at the host and assigned to device A and B respectively for the next step

The host program orchestrates the data exchange between devices. After each simulation iteration, data processed by different devices are transferred back to the host. Users are required to specify a `merge_function`, which takes as arguments the states of a single agent returned by different devices at the end of each simulation iteration and describes how those states should be combined. The merging is carried out on the host by iterating through all agents and calling the `merge_function` in parallel using OpenMP.

The parallel co-execution of step functions may introduce inconsistent agent states across kernels. Some inconsistencies can be resolved also by the `merge_function`. As an example, in a traffic simulation, two models guide the movement of an agent: a car-following model and a lane-changing model. When executing these models in parallel, an inconsistency occurs when the car-following model advances an agent to a lane on the next road whereas the lane-changing model moves the agent to a parallel lane on the current road. In this situation, the `merge_function` could simply use the lane given by the car-following model in the merged agent state. However, if the step functions defined by the model are fully sequentially dependent, co-execution cannot be applied.

3.2.4 | Online dispatcher

While co-execution can lead to better hardware utilization, it may not lead to better performance. In the OpenABLext syntax, we allow users to manually assign each step function to a piece of hardware. When a step function (ie, one OpenCL kernel function) that would perform well on a GPU is assigned to a CPU, simulation performance potentially suffers. The burden is again put on simulationists to find the best suitable hardware for each step function. To alleviate this problem, we propose a light-weight online dispatcher that assigns step functions to the most suitable device.

In ABS, most of the workload occurs in the main simulation body, which iteratively calls the step functions. In these functions, an agent's behavior and its interaction with near-by agents is defined. The computing intensity of step functions therefore depends on two factors: the complexity of the behavior models and the amount of agent interactions. While the complexity of a behavioral model is fixed, the distribution of agents in the simulation space varies over the course of a simulation, resulting in a nonconstant number of agent interactions per time-step. Therefore, the workload of ABS shows irregularities. We use a dynamic assignment approach that re-evaluates the hardware assignment to better fit into the irregular workloads.

Our proposed online dispatcher has two phases: a profiling phase and an execution phase. Unlike existing generic approaches that evaluate the hardware assignment periodically, we can reduce the evaluation frequency by evaluating the assignment only if there is a sharp change in the amount of agent interactions. The dispatcher learns about the performance of each device through its profiling phase, which profiles all step functions on all available hardware devices in parallel over a few simulation steps. All profiling is done with the most up-to-date agent data so that after profiling, the devices can continue their execution by reusing the results generated during profiling. This way the profiling contributes to advancing the simulation at the cost of the data transfer overheads to distribute the most up-to-date agent data to all available hardware before the profiling process starts. The workflow of the proposed online dispatcher is depicted in Figure 4.

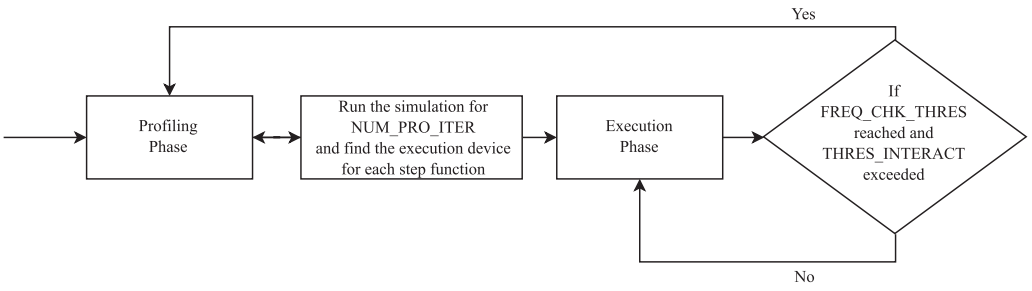


FIGURE 4 Workflow of the online dispatcher

```
... /* function and variable declarations */
int main()
{
  initialise(agentArray[LENGTH]);
  initialise(clEnvironment, clDeviceBuffersDev0,
  clDeviceBuffersDev1);
  clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev0);
  clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev1);
  for (int i=0;i<NUM_STEPS;i++) {
    clExecuteNDRangeKernel(compute_kernel_dev0);
    clExecuteNDRangeKernel(compute_kernel_dev1);
    clReadBuffer(clDeviceBuffersDev0, agentArray0[LENGTH]);
    clReadBuffer(clDeviceBuffersDev1, agentArray1[LENGTH]);
    agentArray[LENGTH] = merge(agentArray0[LENGTH],
  agentArray1[LENGTH]);
    clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev0);
    clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev1);
  }
}
```

(A) Host code

```
... /* function and variable declarations */
void stepFunc1(Agent *agent) {
  for(agents inside the 9 surrounding cells)
    {...}
}

__kernel void compute_kernel_dev0(__global
  Agent *agentArray)
{
  if ( get_global_id(0) >= LENGTH ) return;
  stepFunc1(&agentArray[id]);
}
```

(B) Code for device ID 0

```
... /* function and variable declarations */
void stepFunc2(Agent *agent) {...}

__kernel void compute_kernel_dev1(__global
  Agent *agentArray)
{
  if ( get_global_id(0) >= LENGTH ) return;
  stepFunc2(&agentArray[id]);
}
```

(C) Code for device ID 1

LISTING 4 Generated (pseudo) code for co-execution with one host device and two other devices

Users are required to specify three parameters for the online dispatcher: the number of profiling iterations `NUM_PRO_ITER`, a threshold for the change of agent interactions in percentage `THRES_INTERACT`, and a frequency to check this threshold `FREQ_CHK_THRES`. At the beginning of the simulation, initial agent data are distributed among all available devices. The simulation enters the profiling phase in which each hardware redundantly executes all step functions for `NUM_PRO_ITER` iterations. After `NUM_PRO_ITER`iterations, the host assigns each step function to the piece of hardware on which it executed the fastest. Once all step functions are assigned a `current_device`, the simulation can enter the execution phase.

In the execution phase where there is a one-to-one assignment of step function to hardware device, an agent interaction counter is maintained for each step function, monitoring the amount of agent interactions per simulation step. Every `FREQ_CHK_THRES` simulation steps, the host will check these counters and if one counter has changed more than `THRES_INTERACT` percent compared to the last profiling, the profiling phase is reactivated for the corresponding step function of this counter. The up-to-date agent data on `current_device` are distributed among all available devices and the profiling starts for another `NUM_PRO_ITER` iterations. Again, the hardware device that computes the step function the fastest is selected as the new `current_device`. The execution of this step function continues on the new `current_device`. In the scope of this article, we do not consider resource limitations of each device but assume that each connected device will have enough capacities to execute the assigned step functions. We leave the accounting for resources and other constraints as future work.

3.3 | Conflict resolution

In parallel ABS, simultaneous updates of multiple agents can result in multiple agents being assigned the same resource at the same time, for example, a position on a road or consumables.[44] Unlike desired spatial collisions, for example, in particle collision models, conflicts introduced purely by the parallel execution must be resolved to achieve results consistent with a sequential execution.

Conceptually, conflict resolution involves two steps: First, *conflict detection* determines pairs of conflicting agents, and second, *tie-breaking* determines the agent that acquires the resource. The loser of a conflict can be rolled back to its previous state. Since roll-backs may introduce additional conflicts, this process repeats until no further conflicts occur. A number of approaches for conflict resolution on parallel hardware have been studied in Reference 38. Here, we propose a generic interface for the users to specify a spatial range for conflict detection and a policy for tie-breaking, from which low-level implementations are generated.

Users can define a conflict resolution as follows:

```
conflict_resolution(env, search_radius, tie_breaking)
```

All pairs of agents residing on the same element in the `env` array are checked for conflicts based on the agents' state variables. In 2D and 3D simulation environments, the environment array is comprised of the internally generated partitions of the simulation space as described in Section 3.1, with `search_radius`specifying the search radius. `tie_breaking` has different meanings for graph-based simulations and for 2D/3D

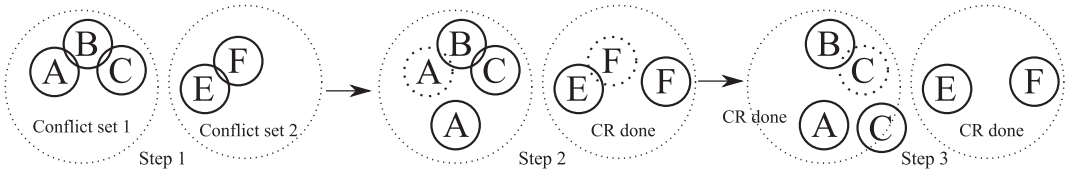


FIGURE 5 Illustration of conflict resolution (CR) for 2D/3D simulations

simulations. For graph-based simulations, it is a binary predicate with two agents *A* and *B* as arguments. The predicate returns `true` if *A* should be rolled back and `false` if the agents are not in conflict or agent *B* needs to be rolled back. For 2D/3D simulations, it returns `true` if agents *A* and *B* are in conflict while returning `false` stands for no conflict. The generated conflict resolution runs in parallel for both types of simulation spaces. By default, conflict resolution runs on GPUs. It can also run on CPUs, if no GPU is present.

As an example, in a graph-based traffic simulation, the simulation environment consists of an array of `roads[]`. Assuming the desired position of an agent is indicated by the state variables `(LaneID, PositionOnLane)`, the `tie_breaking` function can be defined so that the agent with a larger `PositionOnLane` wins the conflict. The position and velocity of the other agent involved in the conflict are reverted to their previous values. The generated conflict resolution code is executed once all step functions have been executed. The conflict detection relies on the neighbour search methods introduced in Section 3.1. As the step functions may change the agents' positions, the environment array is sorted and the `mem_start` and `mem_end` pointers are updated after each iteration (Section 3.1). As the parallel execution of conflict resolution may also result in conflicts, the process of conflict resolution is executed iteratively until there is no conflict detected.

For 2D/3D simulations, conflict resolution can be achieved in the following way: all agents are checked in parallel against other agents within `search_radius` using the `tie_breaking` function. If agent *A* and agent *B* are in conflict, they are added to a *conflict set*. Hence, if another agent *C* conflicts with either agent *A* or agent *B*, agent *C* is also added to the set. The result of the conflict detection step is a list of non-overlapping conflict sets (Figure 5, Step 1) on each of which the conflict resolution can be carried out simultaneously. In each set, a randomly selected agent is rolled back to its previous state and the remaining agents inside the set are again checked for conflicts (Figure 5, Step 2). If no more conflicts exist, the processing of this conflict set ends, otherwise, another randomly selected agent is rolled back (Figure 5, Step 3). This repeats until there is only one agent left in this conflict set.

4 | EXPERIMENTS

In this section, we use three typical hardware platforms (Table 1) to conduct a comprehensive performance analysis and evaluate the performance of OpenABLext, including co-execution and online dispatching. We compare the performance of the new OpenCL backend with the existing backends, that is, C with OpenMP, FLAME GPU, and MASON. For this, we evaluate a range of different simulation models: *Circle*, a benchmark for accessing neighbors within a certain radius provided in Reference 45; Conway's *Game of Life*,[46] a cellular-automata based simulation of the evolution of agents based on the neighbouring agents' states; *Sugarscape*,[47] a social model where each agent searches for nearby resources (a piece of sugar) to metabolize; and *Ants*,[48] which simulates the foraging behavior of ants by leaving pheromones between their home and the food. We based our implementation on the OpenABL code provided in the OpenABL repository (https://github.com/OpenABL/OpenABL). To gain insights into the performance of our co-execution method as well as the online dispatcher, we utilize two additional simulation models: *Traffic*, a traffic simulation based on the implementation in Reference 49 and *Crowd*, which simulates the flocking behavior of people following leaders (fire wardens) during a building

TABLE 1 Configuration of the tested platforms

	Platform 1	Platform 2	Platform 3 (Amazon AWS F1)
CPU	Intel Core i7-4770	Ryzen Threadripper 2950X	Intel Xeon E5-2686 v4
RAM	16 GB DDR3	64 GB DRR4	122 GB DDR4
Accelerator	NVIDIA GTX 1060 GPU	NVIDIA GTX 2080Ti GPU	Xilinx Virtex UltraScale + VU9P FPGA
GCC Version	5.4	7.4	4.8
OpenCL Version	1.2	1.2	1.2
Accelerator Compiler	CUDA 10.0	CUDA 10.1	XOCC 2018.2

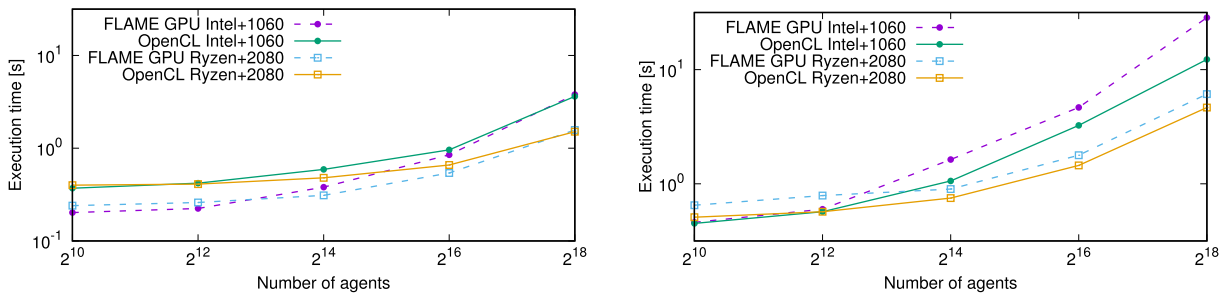
evacuation.[50] While the *Traffic* simulation is graph-based, all others exhibit 2D/3D simulation spaces. OpenABLext detects the simulation type without any user intervention by checking if a user-specified environment is used. All neighbor search queries, indicated by the keywords **on** or **near** in the step functions, are replaced with the respective efficient neighbor search algorithm as described in Section 3.1. Additionally, if users specify whether to generate simulation code with conflict resolution, OpenABLext detects the conflict resolution interface and generates the respective code based on the simulation type.

OpenCL backend During preliminary experiments, we observed that the performance of FLAME GPU is severely affected by I/O to store intermediate simulation statistics. Since these have no effect on the simulation results, we disabled them in our measurements. The generated FPGA code for the traffic simulation consumed 11% of the available look up tables (LUTs) and less than 4% of other resources such as random access memory, flip flops, etc., and ran at a frequency of around 288 MHz. For the other 2D/3D simulations, 6%-17% of LUTs and at most 6% of other resources were utilized and the FPGA ran at a frequency of 310-318MHz. We ran all simulations at least five times for 100 time steps to enable comparison with the existing results in Reference 4. All 95% confidence intervals of the execution time measured are smaller than 16% of the mean values for C backend and 8% for other backends.

In a first experiment, we evaluate the performance of the OpenCL backend compared with the existing FLAME GPU backend. In preliminary experiments, executing the generated OpenCL code on CPUs was slower than on GPUs in all cases. Therefore, we only show the GPU performance. Our results are shown in Figure 6. We observe that the OpenCL backend is on par with the FLAME GPU backend as both implement an efficient neighbour search and make use of the massive parallelism of the GPU. When agents are evenly distributed in the simulation space (Figure 6A), FLAME GPU performed slightly better compared to OpenCL on the older hardware platform (Intel + 1060) and similarly on the more recent one (Ryzen + 2080) in large scale settings. In a second setting, we changed agents' spawn point to be closer together (Figure 6B). We observe that the performance of FLAME GPU is more sensitive to these higher agent densities as it uses a message passing mechanism that generates one message per agent in the current cell. When the amount of messages is too high to fit in local memory, more accesses to global memory are required, causing a decrease in performance. In contrast, the OpenCL backend sorts all agents in global memory after each simulation step to ensure their correct cell assignment. The performance of sorting is thus barely affected by the density of agents. The OpenCL backend performed better on both hardware platforms for this application. For the sake of readability and since all other simulation models showed similar trends, we primarily show the performance of the OpenCL backend on hardware platform 1 and omit curves for FLAME GPU in the following experiments.

In a second experiment, we put the performance of the OpenCL backend into perspective by comparing it to the C with OpenMP as well as the MASON backend. Furthermore, we show how feasibly ABSs can be executed on an FPGA using OpenABLext. Our findings are summarized in Figure 7. First, we demonstrate the performance of three simulation models with a 2D/3D environment (Figure 7A-C). Unsurprisingly, the GPU performed best for higher agent numbers, as the relative initialization overhead decreased and the workload was sufficient to benefit from the massive parallelization. We confirm the findings from Reference 4 that the MASON backend performed considerably better than the C with OpenMP backend when the agent number is high enough. Interestingly, the performance of the FPGA, which is restricted by the relatively low operating frequency and slow off-chip global memory access, is able to catch-up with OpenMP at 2^{14} agents and even outperform it for the 2^{16} agent scenario in both Game-of-Life and Sugarscape (Figure 7A-C). This is caused by the efficient neighbor search and the pipelining parallelism the FPGA implements. As mentioned in Section 3.2.2, we employ a double-buffering design where we swap read/write buffers after each step function. The three step functions in the Ants application (Figure 7C) are computationally lightweight so that the FPGA is not fully utilized while they cause more memory-intensive buffer swaps, slowing down the performance of the FPGA. As to our knowledge, these measurements represent the first results on FPGA-accelerated ABSs from high-level model specifications, and the performance results demonstrate the promise of FPGAs in this context.

As described above, OpenABLext enables modelers to utilize graph-based simulation environments. As a proof of concept, we developed a traffic simulation based on our previous implementation.[49] Our results are shown in Figure 7D. Note that we do not include MASON or C with OpenMP as these backends are unable to support graph-based simulation spaces. We observe that the FPGA performs comparably to the OpenCL CPU variant



(A) Circle (evenly distributed agents). (B) Circle (high agent density).

FIGURE 6 Comparison of the OpenCL backend with the FLAME GPU using the Circle application

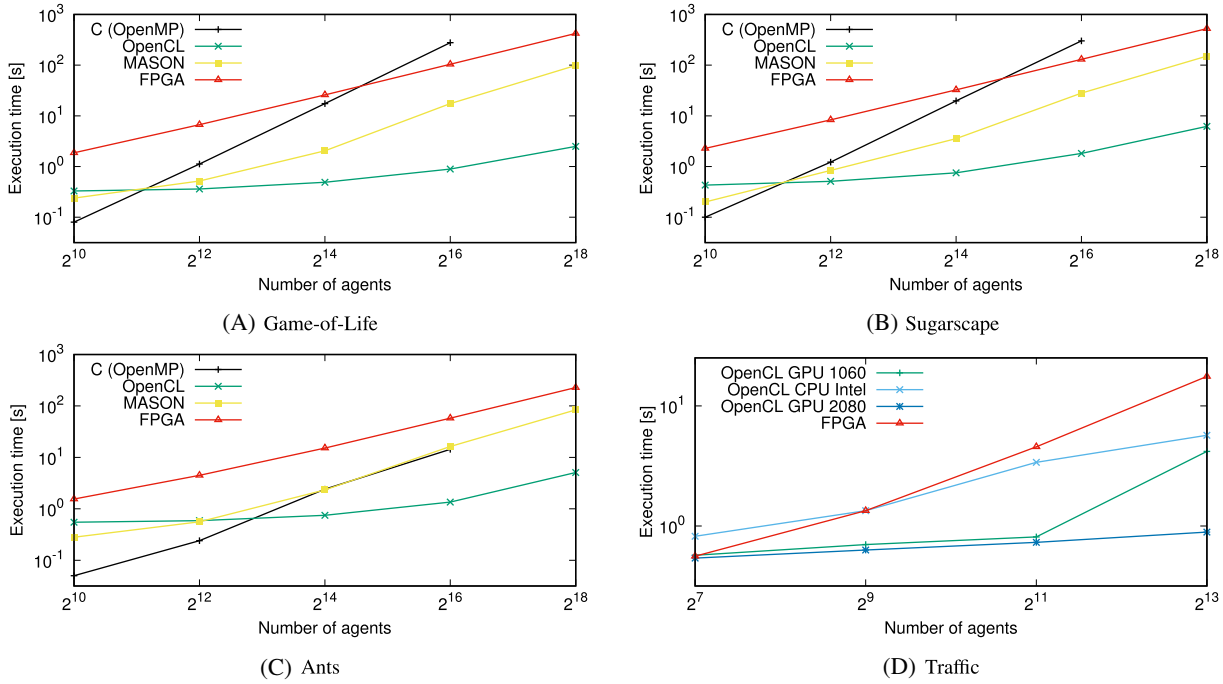


FIGURE 7 Performance of the OpenCL backend, compared with the C, MASON, FPGA (where applicable)

in small scale scenarios ($\leq 2^{10}$ agents). The performance of both GPUs is similar. However, with a larger number of agents, platform 2 could benefit from its larger memory, avoiding the performance drop we experienced on platform 1.

Conflict resolution Due to limited space, we do not show figures for the OpenABLext automated conflict resolution from user-specified rules, but instead briefly report our findings: On the traffic application, we specified a tie-breaking function so that the winner of each conflict is the agent further ahead on the same lane. The performance of the generated conflict resolution is evaluated by running 2^{13} agents in both a low agent density and a high agent density scenario. The conflict resolution takes 6% on a CPU and 9% on a GPU of the overall runtime with an average of 0.06 rollbacks per agent in 100 simulation steps for the low agent density setting. In the high density setting, the percentages are 13.67% for a CPU and 39% for a GPU with an average of 2.23 rollbacks per agent.

Online dispatcher Finally, we present results on the performance of the co-execution schemes introduced with OpenABLext where an online dispatcher automatically assigns each step function to a suitable piece of hardware. We demonstrate the benefit of the online dispatcher as well as the co-execution capability using three simulation models: *Circle*, *Crowd*, and *Traffic*. The former two were simulated using 2^{16} agents, while the more complex traffic simulation was populated by 2^{13} agents. The `merge_functions` for *Crowd* and *Traffic* are defined as: combining the results from the path finding model and the social force model for *Crowd* and taking the result of the car-following model for *Traffic*. To further understand the impact of agent interactions on the simulation performance, we use three different *Traffic* scenarios: short, medium, and long average agent trip length. As shown in Figure 8A, in the short trip-length setting, most of the agents finish their trips in the first one-third of the simulation, resulting in a sharp reduction of agent interactions with simulation time passing. In the medium setting, around half of all agents stayed active throughout the entire simulation time while in the long trip setting, almost all agents remained. In these experiments, we set the online dispatcher parameters as follows: `NUM_PRO_ITER` = 20, `THRES_INTERACT` = 0.4, and `FREQ_CHK_THRES` = 100. We run the above three applications for 1000 steps.

Naturally, co-execution requires at least two step functions to be distributed among different devices. However, the online dispatcher can still be employed in cases where there is only one step function in order to find the most suitable device. Figure 8B shows the speed up over running the generated OpenCL code on CPU. For the circle application that has only one step function, the online dispatcher manages to identify the GPU as the most suitable device with only 7% overhead introduced by profiling. In the *Crowd* simulation with its two step functions, we notice a considerable speed-up of the co-execution scheme over the pure GPU (2.28x) or CPU (1.47x) variants, though 10% of the total execution time is spent on merging. The online dispatcher variant was able to identify the most suitable device for each step function with only 1% overhead compared with a manual assignment.

In the **traffic-med** and **traffic-long** scenarios, we also observe an advantage of co-execution over pure GPU and CPU execution with a merging overhead of 1%. The online dispatcher succeeds in finding the co-execution setting with less than 3% overhead in both cases. Interestingly, this does not hold for the **traffic-short** scenario, as during the simulation the number of agents drops below a point where parallelization can no longer make up for the data merging overhead. In these cases, co-execution will effectively slow down the simulation. As the online dispatcher recognizes the

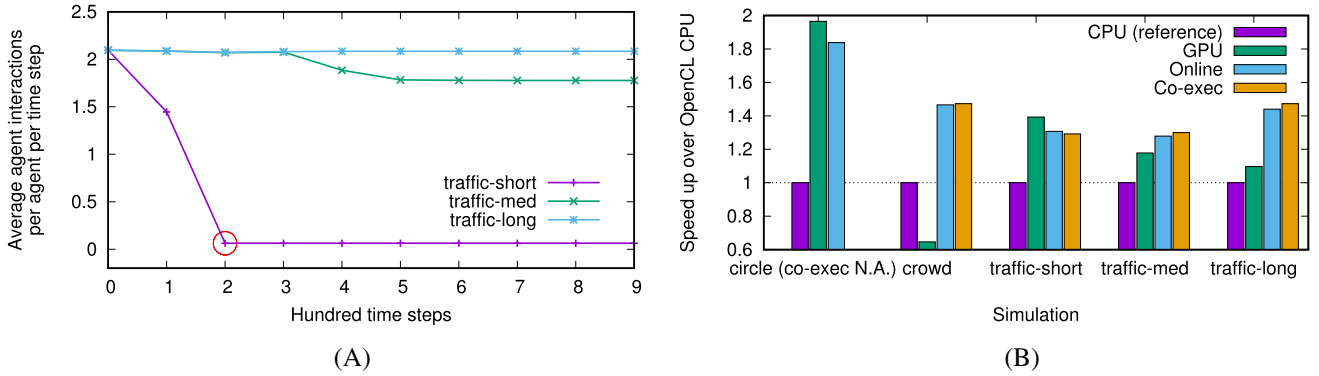


FIGURE 8 Evaluation of the proposed online dispatcher on Platform 1

decrease of agent interactions, it switches from co-execution to a pure GPU execution (the switch point is marked in red in Figure 8A), resulting in a total overhead of about 7%.

5 | CONCLUSION AND FUTURE WORK

In this article, we presented OpenABLext, an open-source automatic code-generation framework for ABS on heterogeneous hardware environments. OpenABLext extends the OpenABL framework to overcome limitations in terms of the supported hardware platforms and opens up new possibilities such as multi-device co-execution. The addition of an OpenCL backend to the OpenABL framework enables the execution on CPUs and accelerators such as GPUs and FPGAs. Furthermore, OpenABLext features automated conflict resolution based on user-specified rules, supports graph-based simulation spaces, and utilizes an efficient neighbor search algorithm. To ease deployment in a co-execution setting, a light-weight online dispatcher is proposed, which automatically chooses the most suitable hardware assignment.

We evaluated the performance of OpenABLext using a range of different simulation models. On GPUs, the new OpenCL backend outperforms FLAME GPU in high agent density settings and also delivers a better and more stable total deployment time. We showed that using FPGAs for ABS is a promising approach as our experiments exhibited shorter execution times for scenarios with a larger agent count compared with the OpenMP backend. The main reason for this is the more efficient neighbor search and the pipeline parallelism of the FPGA. Finally, we demonstrated that for some applications, co-execution can outperform single-device simulation, and that our online dispatcher was able to find the best hardware assignment in all tested cases with less than 7% overhead.

Future work includes leveraging the vectorization ability of CPUs and optimizing FPGA performance by generating more compute units per simulation instance. This may require a reconsideration of some parts of the simulation design, for example, dealing with limited on-chip memory bandwidth of FPGAs when multiple compute units are present. Exploring more co-execution options such as data-level co-execution is another direction for future work. Currently, our online dispatcher selects hardware devices solely based on performance; however, other criteria such as energy efficiency constitute interesting research avenues.

ACKNOWLEDGEMENTS

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence and Technological Enterprise (CREATE) programme. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

ORCID

Jiajian Xiao https://orcid.org/0000-0003-4893-9757

REFERENCES

1. Macal C M, North M J. Tutorial on agent-based modelling and simulation. *Journal of Simulation*. 2010;4 (3):151–162. http://dx.doi.org/10.1057/jos.2010.3.
2. Xiao J, Andelfinger P, Eckhoff D, Cai W, Knoll A. A survey on agent-based simulation using hardware accelerators. *ACM Comput Surv*. 2019;51(6):131:1-131:35.

3. Fujimoto R, Bock C, Chen W, Page E, Panchal JH. *Research Challenges in Modeling and Simulation for Engineering Complex Systems*. Berlin, Germany: Springer; 2017.
4. Cosenza B, Popov N, Juurlink B, et al. OpenABL: a domain-specific language for parallel and distributed agent-based simulations. *LNCS*. Vol 11014. Springer, Cham; 2018:505-518.
5. Belviranli ME, Bhuyan LN, Gupta R. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans Archit Code Optim*. 2013;9(4):57.
6. Xiao J, Andelfinger P, Cai W, Richmond P, Knoll A, Eckhoff D. Advancing automatic code generation for agent-based simulations on heterogeneous hardware. Paper presented at: Proceedings of the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar); 2019; Springer, Goettingen, Germany.
7. Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G. MASON: a multiagent simulation environment. *Simulation*. 2005;81(7):517-527.
8. North MJ, Collier NT, Vos JR. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans Model Comput Simul*. 2006;16(1):1-25.
9. Minar N, Burkhart R, Langton C, Askenazi M. *The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations Tech Report*. Santa Fe, NM: Santa Fe Institute1399 Hyde Park Rd; 1996.
10. Kiran M, Richmond P, Holcombe M, Chin LS, Worth D, Greenough C. FLAME: simulating large populations of agents on parallel hardware architectures. Paper presented at: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems; IFAAMAS:1633–1636 IFAAMAS; 2010.
11. Cordasco G, De Chiara R, Mancuso A, Mazzeo D, Scarano V, Spagnuolo C. A framework for distributing agent-based simulations. *LNCS*. 7155, Heidelberg, Germany: Springer; 2011:460-470.
12. Collier NT, North MJ. *Large-Scale Computing Techniques for Complex System Simulations*. Hoboken, NJ: Wiley; 2011.
13. Richmond P, Walker D, Coakley S, Romano D. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings Bioinf*. 2010;11(3):334-347.
14. Laville G, Mazouzi K, Lang C, Marilleau N, Herrmann B, Philippe L. MCMAS: a toolkit to benefit from many-core architecure in agent-based simulation. *LNCS*. Vol 8374. Heidelberg, Germany: Springer; 2013:544-554.
15. Grewe D, Wang Z, O'Boyle MFP. Portable mapping of data parallel programs to opencl for heterogeneous systems. Paper presented at: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems; IFAAMAS; 2013:1-10; IEEE.
16. Li P, Brunet E, Trahay F, Parrot C, Thomas G, Namyst R. Automatic OpenCL code generation for multi-device heterogeneous architectures. Paper prsented at: Proceedings of the International Conference on Parallel Processing; 2015:959-968; IEEE.
17. Krommydas K, Sasanka R, Feng WC. Bridging the FPGA programmability-portability Gap via automatic OpenCL code generation and tuning. Paper presented at: Proceedings of the 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP); vol I, 2016:213-218; IEEE.
18. Rohde J, Martinez-Peiro M, Gadea-Girones R. SOCAO: source-to-source OpenCL compiler for intel-altera FPGAs. Paper presented at: Proceedings of the FSP 2017 4th International Workshop on FPGAs for Software Programmers; VDE; 2017:1-7; VDE.
19. Farber R. *Parallel Programming with OpenACC*. Newnes; 2016.
20. Lee S, Kim J, Vetter JS. Openacc to fpga: a framework for directive-based high-performance reconfigurable computing. Paper prsented at: Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2016:544-554; IEEE.
21. Sujeeth AK, Brown KJ, Lee H, et al. Delite: a compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans Embed Comput Syst*. 2014;13(4s):134.
22. DeVito Z, Joubert N, Palacios F, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. Paper presented at: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2011:9; IEEE.
23. Johnston B, Falzon G, Milthorpe J. OpenCL performance prediction using architecture-independent features. Paper presented at: Proceedings of the 2018 International Conference on High Performance Computing & Simulation (HPCS); 2018:561-569; IEEE.
24. Moren K, Göhringer D. Automatic mapping for OpenCL-programs on CPU/GPU heterogeneous platforms. Paper presented at: Proceedings of the International Conference on Computational Science; 2018:301-314; Springer.
25. Ahmad M, Dogan H, Michael CJ, Khan O. HeteroMap: a runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators. Paper presented at: Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); 2019:268-281; IEEE.
26. Pandit P, Govindarajan R. Fluidic kernels: cooperative execution of opencl programs on multiple heterogeneous devices. Paper presented at: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization; 2014:273; ACM.
27. Pérez B, Stafford E, Bosque JL, et al. Auto-tuned OpenCL kernel co-execution in OmpSs for heterogeneous systems. *J Parall Distrib Comput*. 2019;125:45-57.
28. Guzman MAD, Nozal R, Tejero RG, Villarroya G María, Gracia Darío S, Bosque JL. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *J Supercomputing*. 2019;75(3):1732-1746.
29. Navarro A, Corbera F, Rodriguez A, Vilches A, Asenjo R. Heterogeneous parallel_for template for CPU–GPU chips. *Int J Parall Prog*. 2019;47(2):213-233.
30. Liu W, Vinter B. A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. *J Parall Distrib Comput*. 2015;85:47-61.
31. Khaleghzadeh H, Manumachu RR, Lastovetsky A. A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms. *IEEE Trans Parall Distrib Syst*. 2018;29(10):2176-2190.
32. Beaumont O, Boudet V, Rastello F, Robert Y. Matrix multiplication on heterogeneous platforms. *IEEE Trans Parall Distrib Syst*. 2001;12(10):1033-1051.
33. Xu Y, Cai W, Aydt H, Lees M. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. Paper presented at: Proceedings of the Winter Simulation Conference; 2014:3483-3494; IEEE.
34. Huchant P, Barthou D, Counilh MC. Adaptive partitioning for iterated sequences of irregular OpenCL kernels. Paper presented at: Proceedings of the 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD); 2018:262-265; IEEE.
35. Aji AM, Pena AJ, Balaji P, Feng WC. Automatic command queue scheduling for task-parallel workloads in opencl. Paper presented at: Proceedings of the 2015 IEEE International Conference on Cluster Computing; 2015:42-51; IEEE.

36. Lysenko M, D'Souza RM. A framework for megascale agent based model simulations on graphics processing units. *J Artif Soc Soc Simul*. 2008;11(4):10.
37. Richmond P. Resolving conflicts between multiple competing agents in parallel simulations. *LNCS*. Vol 8805. Cham: Springer; 2014:383-394.
38. Yang M, Andelfinger P, Cai W, Knoll A. Evaluation of conflict resolution methods for agent-based simulations on the GPU. Paper presented at: Proceedings of the Conference on Principles of Advanced Discrete Simulation; 2018:129-132; ACM.
39. Thomas D, Moorby P. *The Verilogss® Hardware Description Language*. Springer Science & Business Media; 2008.
40. NZ. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill Inc; 1997.
41. Kastner R, Matai J. Neuendorffer S. Parallel programming for FPGAs; 2018. arXiv 1805.03648.
42. Li Xi, Cai W, Turner SJ. Efficient neighbor searching for agent-based simulation on GPU. Paper presented at: Proceedings of the International Symposium on Distributed Simulation and Real Time Applications; 2014:87-96; IEEE.
43. Mahony AO, Popovici E. Power analysis of sorting algorithms on FPGA using OpenCL. Paper presented at: Proceedings of the 2018 29th Irish Signals and Systems Conference (ISSC); 2018:1-6; IEEE.
44. Epstein JM, Axtell R. *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press; 1996.
45. Chisholm R, Richmond P, Maddock S. A standardised benchmark for assessing the performance of fixed radius near neighbours. *LNCS*. Vol 10104. Cham: Springer; 2016:311-321.
46. Gardner M. Mathematical games: the combinations of John Conway's new solitaire game "Life". *Sci Am*. 1970;223(4):120-123.
47. Epstein JM, Robert A. Artificial societies and generative social science. *Artif Life Robot*. 1997;1(1):33-34.
48. Panait L, Luke S. A pheromone-based utility model for collaborative foraging. Paper presented at: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004; 2004:36-43; IEEE.
49. Xiao J, Andelfinger P, Eckhoff D, Cai W, Knoll A. Exploring execution schemes for agent-based traffic simulation on heterogeneous hardware. Paper presented at: Proceedings of the International Symposium on Distributed Simulation and Real Time Applications; 2018:1-10; IEEE.
50. Pelechano N, Badler NI. Modeling crowd and trained leader behavior during building evacuation. *IEEE Comput Graph Appl*. 2006;26(6):80-86.

How to cite this article: Xiao J, Andelfinger P, Cai W, Richmond P, Knoll A, Eckhoff D. OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms. *Concurrency Computat Pract Exper*. 2020;32:e5807. https://doi.org/10.1002/cpe.5807