Advancing Automatic Code Generation for Agent-Based Simulations on Heterogeneous Hardware

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

[1] TUM CREATE Singapore
{jiajian.xiao,philipp.andelfinger,david.eckhoff}@tum-create.edu.sg
[2] Technische Universität München, Germany
knoll@in.tum.de
[3] Nanyang Technological University, Singapore
aswtcai@ntu.edu.sg
[4] University of Sheffield, UK
p.richmond@sheffield.ac.uk

Abstract. The performance of agent-based simulations has been shown to benefit immensely from execution on hardware accelerator devices such as graphics processing units (GPUs). Given the increasingly heterogeneous hardware platforms available to researchers, it is important to enable modellers to target multiple devices using a single model specification, and to avoid the need for in-depth knowledge of the hardware. Further, key modelling steps such as the definition of the simulation space and the specification of rules to resolve conflicts among agents should be supported in a simple and generic manner, while generating efficient code. To achieve these goals, we extend the OpenABL modelling language and code generation framework by three aspects: firstly, a new OpenCL backend enables the co-execution of arbitrary agent-based models on heterogeneous hardware. Secondly, the OpenABL language is extended to support graph-based simulation spaces. Thirdly, we specify a generic interface for specifying conflict resolution rules. In a performance comparison to the existing OpenABL backends, we show that depending on the simulation model, the opportunity for CPU-GPU co-execution enables a speedup of up to 2.0 over purely GPU-based simulation.

Keywords: Agent-based simulation · Parallel and Distributed simulation · Heterogeneous hardware · OpenABL · OpenCL

1 Introduction

Agent-based simulation (ABS) is widely used for system analysis and the answering of what-if questions in domains such as transport, computer networks, biology, and social sciences [17]. Each agent, e.g., a vehicle or a pedestrian, is an autonomous entity that makes decisions based on its environment, other agents,

and a number of behavioural models. Due to increasingly complex models and large numbers of agents, large-scale ABS often suffer from long execution times.

There exists an ample body of methods to speed up agent-based simulation to meet increasing performance needs, commonly based on parallel and distributed simulation techniques. In the last decade, the increasing prevalence of heterogeneous hardware composed of CPUs and *accelerators* such as GPUs or FPGAs opens up new possibilities to accelerate ABSs [28]. For instance, computationally intensive segments can be offloaded to run on an accelerator where they can be executed faster or ran in parallel to other parts being executed on the CPU.

However, placing the burden of tailoring the simulation to the target hardware platform on the simulationist degrades the maintainability of the simulation code as well as the portability to other hardware platforms. To avoid these issues, the modelling language OpenABL [6] has been proposed to enable code generation from high-level model and scenario specifications using a C-like syntax. A number of *backends* are provided to generate parallelised code targeting CPUs, GPUs, clusters, or cloud environments.

Previous to the work presented in this paper, each backend supported by OpenABL targeted one specific type of hardware platform, i.e., co-execution on combinations of CPUs, GPUs, FPGAs was not possible. This leaves a large range of computational resources untapped, even though previous work has demonstrated high hardware utilisation using co-execution [2]. Further, the simulation environment was limited to continuous 2D or 3D spaces, which excludes graph-based simulation spaces as commonly used in domains such as road traffic and social sciences. Lastly, OpenABL did not provide a mechanism for conflict resolution, requiring modellers to manually provide code to detect and resolve conflicts in situations where multiple agents request the same resources.

We address these limitations and contribute to the state of the art as follows:

- We extend OpenABL by an OpenCL backend to support automatic code generation for heterogeneous hardware.
- We provide new syntactic elements to support graph-based simulation spaces.
- We define an interface that enables conflict resolution code to be generated from user-specified rules.

Our extensions are open-source and available online[1]. The remainder of the paper is organised as follows: In Section 2, we introduce OpenABL as well as OpenCL and give an overview of related work in the field. In Section 3, we describe our extensions to OpenABL. We evaluate the performance of the extended OpenABL in Section 4. Section 5 summarises our work and concludes the paper.

2 Related Work and Background

The acceleration of ABS through parallelisation has received wide attention from the research community. A number of frameworks simplify the process of developing ABSs, e.g. MASON [15], Repast [19], Swarm [18], or FLAME [11]. Simulator variants that exploit CPU-based parallelisation or distributed execution

[1] https://github.com/xjjex1990/OpenABL_Extension

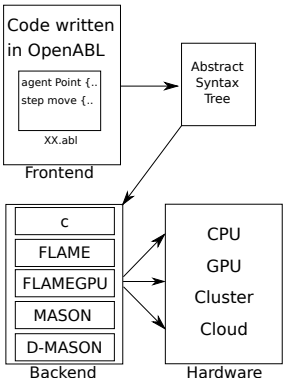


Fig. 1: An overview of the OpenABL.

```
1   agent Agent {
2   position float2 coordinate; }
3
4   param int num_of_agents = 1000;
5   param int sim_steps = 100;
6   param float env_size = 100;
7
8   environment { max : float2 (
    env_size) }
9
10  step move(Agent in -> out) {..}
11
12  void main() {
13  .../* initialise the agents */
14  simulate(sim_steps) {move} }
15
```

LISTING 1: Example OpenABL code.

include D-MASON [5] and Repast-HPC [4]. Making full use of those frameworks requires modellers to be knowledgeable in parallel or distributed computing.

A comprehensive review of existing techniques to overcome the challenges of ABS on hardware accelerators is found in [28]. One important method is to abstract from hardware specifics to simplify porting to hardware accelerators. FLAME GPU [23] is an extension to FLAME that provides a template-driven framework for agent-based modelling targeting GPUs based on a state machine model called X-machine. The Many-Core Multi-Agent System (MCMAS) [12] provides a Java-based tool-kit 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. Several previous works focus on the generation of performance-portable code targeting heterogeneous hardware by pattern-matching parallelisable C snippets [9], relying on code templates [13], or using domain-specific languages [24]. Some works perform pattern-matching procedures on intermediate representations instead of high-level code [26, 25]. Unlike OpenABL, which can exploit the parallelisable structure shared by most ABS, the above works focus on automatically detecting parallelisable computations such as nested loops with predictable control flows.

In parallel ABS, conflicting actions may occur, e.g., 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 [16] or on enumerating the agents involved in conflicts once an update cycle has completed [22]. 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. [29]. 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.

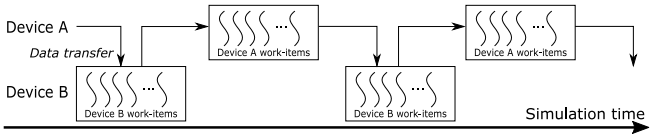


Fig. 2: Co-execution on devices A and B. Each work-item of device A processes the step functions assigned to A. After that, the data is transferred to Device B (via host) for processing the step functions assigned to B.

2.1 OpenABL

OpenABL is a domain-specific language to describe the behaviour of agent-based simulation and a framework to generate code targeting multiple execution platforms. It acts as an intermediate layer to generate parallel or distributed time-stepped ABS, given sequential simulation code written in the C-like OpenABL language. An overview of the OpenABL framework is depicted 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.1-2), constants (keyword `param`, L.4-6), simulation environments (keyword `environment`, L.8), step functions (keyword `step`, L.10), and a main function (keyword `main()`, L.12-14).

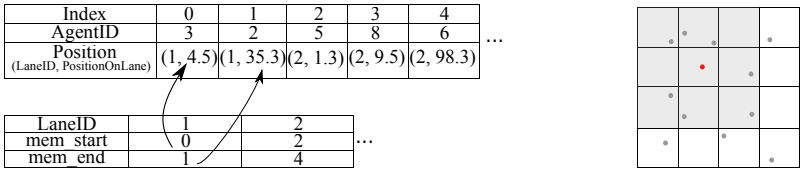
The OpenABL compiler parses OpenABL code and compiles it to an Intermediate Representation (IR) called Abstract Syntax Tree (AST). The AST IR is then further relayed to one of the available backends. The backend reconstructs simulation code from the AST IR and parallelises the step functions targeted for CPUs, GPUs, clusters or cloud environments. OpenABL supports the following backends: C, FLAME [11], FLAME GPU [23], MASON [15], and D-MASON [5].

2.2 OpenCL

The Open Computing Language (OpenCL) is a framework that allows users to write parallel programs in a C-like syntax without considering low-level hardware specifics. An OpenCL execution environment is comprised of a host (usually CPUs) and one or multiple devices (e.g., CPUs, GPUs). A host program initialises the environment, control, memory, and computational resources for the devices. A device program consists mainly of so-called kernels that implement the computational tasks. Threads that process the tasks are referred to as work-items. Parallelism is achieved by processing many work-items in parallel. OpenCL is supported by a wide range of hardware including CPUs, GPUs, APUs, and FPGAs, allowing it to target heterogeneous hardware environments.

3 Extending OpenABL

In this section, we propose extensions to the OpenABL language and framework to support a wider range of simulation models as well as additional types of



(a) Agents are sorted by their `position` (e.g. `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 neighbour search of the red agent loads itself and adjacent cells.

Fig. 3: Coalesced memory access in the generated OpenCL code.

hardware. We first provide an OpenCL backend to support code generation targeting heterogeneous hardware environments, enabling the execution on a variety of devices as well as a multi-device co-execution, e.g., combining a CPU and an FPGA. Further, we extend the OpenABL language to allow for the definition of graph-based simulation spaces. Finally, we introduce a mechanism for automated resolution of inter-agent conflicts based on user-defined rules.

3.1 Code Generation for Heterogeneous Hardware

OpenABL enables the addition of new backends without modifying the frontend, which allows us to target heterogeneous hardware by adding an OpenCL backend. The existing backends only allow for the execution on a single platform, e.g., a GPU. In contrast, OpenCL enables *co-execution* across multiple devices of different types. Our aim is to allow modellers to fully utilise the available hardware without specifying simulation code for each device manually.

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 for each available device. 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 initialises the devices, allocates the required memory, and initialises the agent state variables as well as the environment. In a co-execution setting, the host program also orchestrates the data exchange between devices. After each simulation iteration, data processed by different devices is transferred back to the host. In the `simulate` statement, each step function is annotated with the identifier of the OpenCL device on which the step function should execute, e.g.: `simulate(sim_steps) {stepFunc1(0), stepFunc2(1)}`.

One `compute_kernel` function is created in the device program for each device, where the designated step functions are called in sequence. On each device, the work-items execute in parallel with each one processing one step function.

The main loop of the simulation calls the `compute_kernel` of each device iteratively until the step count defined in the parameter of `simulation()` has been reached. As shown in Figure 2, the execution of subsequent `compute_kernels`

across devices is serialised to guarantee that data dependencies across kernels are respected. In the future, additional merging steps based on the model-specific dependencies could allow kernels to execute in parallel across devices.

3.2 User-Specified Environments

The original OpenABL limits the simulation environment to continuous 2D or 3D spaces, parametrised by the `max, min`, and `granularity` attributes in the `environment` declaration. Furthermore, user-defined types can only be used within agents, and not in function bodies or the environment, complicating the 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 follows:

```
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 memory accessible to groups or individual work-items. Due to the high latency of global memory accesses, data locality is an important consideration in ABS development [6]: accesses of adjacent work-items to adjacent memory addresses can frequently be coalesced, i.e., translated to a single memory transaction, allowing for peak memory performance on OpenCL devices such as GPUs. In common ABS models, agents tend to interact only with agents within a certain radius or on the same edge in a graph environment. To achieve data locality during execution, we implemented the efficient neighbour search method by [14]. 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. In the original OpenABL, data locality is achieved in 2D or 3D space by specifying a radius using the following neighbour search query:

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

We extend the language to allow for a similar neighbour search query for graph-based models: `for (AgentType neighbours : on (env))`

In the example of a traffic simulation with graph edges representing road lanes, the following query retrieves all agents on a lane:

```
for (Vehicles neighbours : 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 3a, 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 neighbour search query is called, instead of iterating through global memory, only a chunk of memory is loaded. 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 chunks of memory holding the agents in current partition and all the neighbouring partitions (cf. Figure 3b).

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, e.g., a position on a road or consumables [7]. Unlike desired spatial collisions, e.g., in particle collision models, conflicts introduced purely by the parallel execution must be resolved to achieve results consistent with a sequential execution.

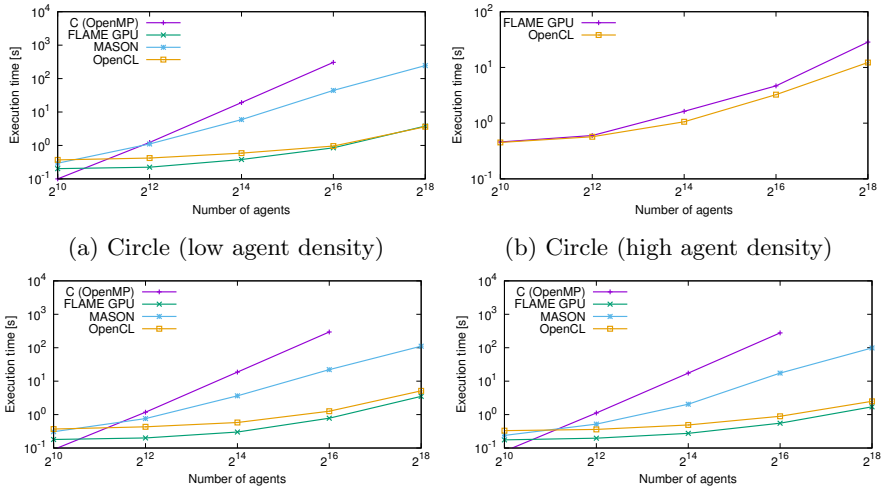
Conceptually, conflict resolution involves two steps: First, *conflict detection* determines pairs of conflicting agents, and second, *tie-breaking* determines the agent that acquires the resource. The loser of a conflict can be rolled back to its previous state. Since roll-backs may introduce additional conflicts, the process repeats until no further conflicts occur. A number of approaches for conflict resolution on parallel hardware have been proposed in [29]. Here, we propose a generic interface to specify a spatial range for conflict detection and a policy for tie-breaking, from which low-level implementations are generated.

The conflict resolution is specified 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. When considering 2D and 3D environments, the environment array is comprised of the internally generated partitions of the simulation space, with `search_radius` specifying the search radius. `tie_breaking` is a binary predicate that, given two agents A and B as arguments, returns `true` if A should be rolled back. If the agents are not in conflict or agent B should be rolled back, `tie_breaking` returns `false`.

As an example, in a traffic simulation scenario, the `env` is the environment array `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 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 3.2. 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 (cf. Section 3.2). Currently, the conflict resolution code is based on a user-specified tie-breaking rule. Our future work includes the automatic generation of model-agnostic resolution code [29].



(a) Circle (low agent density)

(b) Circle (high agent density)

(c) Boids

(d) Game-of-Life

Fig. 4: Performance of the C, FLAME GPU, MASON and OpenCL code.

4 Experiments

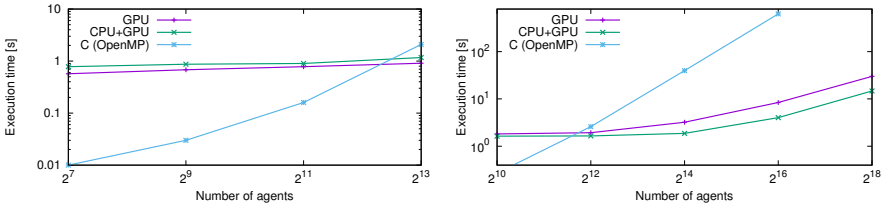
We evaluate the extended OpenABL on a system equipped with a 4-core Intel Core i7-4770 CPU, 16 GB of RAM and an NVIDIA GTX 1060 graphics card with 6 GB of RAM. We rely on GCC version 5.4, OpenCL version 1.2, and NVIDIA CUDA 10.0.292. We compare the performance of the OpenCL backend and the other backends: C with OpenMP, FLAME GPU, MASON. We consider three existing models: Circle, a benchmark for accessing neighbours within a certain radius provided in [3]; Conway's Game of Life [8]; and Boids [21], which simulates the flocking behaviour of birds. We based our implementation on the code provided in the OpenABL repository[2]. During preliminary experiments, we observed that FLAME GPU's performance is severely affected by file system I/O to store simulation statistics, which we disabled in our measurements. We run the simulations in two scenarios: "low agent density" generates agents evenly throughout the simulation space, whereas "high agent density" generates agents only in the upper left quadrant. We run all simulations for 100 time steps to allow for comparison with the existing results in [6].

As illustrated by Fig. 4a, 4c, 4d, the C variant is slow in all cases. This is because it iterates through all agents to search neighbours while the other backends rely on grid-based approaches to limit the search space. The performance of the OpenCL backend is on par with the FLAME GPU backend and outperforms the other backends for low agent densities. As shown in the first four rows of Table 1, this is mainly owing to the massive parallelism on the GPU and the efficient neighbour search implemented by both backends. Despite the relatively

[2] https://github.com/OpenABL/OpenABL

Table 1: Breakdown of simulation runtime [s] for Circle (2^{16} agents)

Backend	Agent updates	Neighb. search	Other	Total
C	308.5 (100.0%)		0.00 (0.0%)	308.5
OpenCL	0.32 (33.3%)	0.61 (63.5%)	0.03 (3.2%)	0.96
FLAME GPU	0.24 (28.9%)	0.17 (20.5%)	0.42 (50.6%)	0.83
MASON	45.00 (100.0%)	0.00 (0.0%)	0.08 (0.0%)	45.08
OpenCL, high density	2.64 (81.2%)	0.58 (17.8%)	0.03 (1.0%)	3.25
FLAME GPU, high density	2.52 (54.1%)	1.74 (37.3%)	0.40 (8.6%)	4.66



(a) Traffic (b) Crowd

Fig. 5: Performance of the CPU-GPU co-execution.

long initialisation time as shown in Table 1 (the 'Other' column), FLAME GPU performs the best in low agent density scenarios. This is owing to FLAME GPU's message passing mechanism that generates one message per agent in the current cell. However, the performance is sensitive to the agent density. In contrast, the OpenCL backend sorts all agents in global memory after each simulation iteration to ensure their correct assignment to cells. The performance of sorting is barely affected by the density of agents. Thus, with high agent density, the OpenCL backend outperforms FLAME GPU in all cases, as depicted in Fig. 4b and the last two rows of Table 1. The other two models follow the same trend.

Our extensions to the OpenABL enable modellers to generate graph-based ABSs. As a proof of concept, we developed a traffic simulation akin to a previous manual implementation [27]. The agent behaviour is governed by two models: the Intelligent Driver Model determines the agents' longitudinal movement, whereas Ahmed's lane-changing model [1] determines the lateral movement. The generation of the conflict resolution code is enabled, the winner of each conflict being the agent further ahead on the same lane. We evaluate two execution schemes: executing purely on a GPU as well as CPU-GPU co-execution. In the co-execution scheme, the car-following model is offloaded to the CPU, while the lane-changing model and conflict resolution remain on the GPU. In all tested cases, the time spent on conflict resolution occupies less than 0.1% of the overall runtime with on average 0.86 rollbacks per agent in 100 simulation steps. As shown in Figure 5a, with a small number of agents, the C variant outperforms the others due to the conflict resolution overhead and the data transfers between the CPU and the GPU in the co-execution case. As the number of agents increases,

the GPU and co-execution variants produce better results than the C variant. The absolute runtime of the pure GPU and co-execution variants is similar. The co-execution achieves a maximum speedup of 1.78x over the C variant while the purely GPU-based execution achieves a maximum speedup of 2.29x. To further demonstrate the benefit of co-execution, we developed a crowd simulation based on the building evacuation behaviour described in [20]. Agents are divided into two groups based on their high-level behaviour: *Leaders*, which are assumed to have a floor plan of the building, conduct path finding to search for the exits. *Followers* flock to the nearest leader and follow the leader's movement. If there is no leader within a defined radius, the followers move in a random direction. All agents follow the Social Force Model [10] as their low-level behaviour. In the co-execution scheme, the memory-intensive path finding based on Dijkstra's algorithm is executed on the CPU, while the computationally intensive Social Force Model is executed on the GPU. Similar to the traffic simulation, with a small number of agents, the C backend variant outperforms the others, as illustrated in Fig. 5b. As the number of agents increases, the co-execution variant outperforms the other variants. A maximum speedup of 3.5x over the C variant and 2.03x over pure GPU is achieved through co-execution.

Finally, the OpenCL backend also opens up the possibility of executing on OpenCL-enabled FPGA devices. For instance, Intel offers an SDK to compile OpenCL code for FPGAs[3]. While in preliminary experiments, the considered models exceeded the hardware resources of a Terasic DE10-Standard, we plan to explore the area of FPGA-based acceleration in future work.

5 Conclusion and Future Work

In this paper, we presented our work towards automatic code-generation of agent-based simulations for heterogeneous hardware environments. We extended the OpenABL framework to overcome limitations in terms of the supported hardware platforms and the representation of the simulation space to support portable high-performance ABS for various model types. Our extensions are fully open-source and available online. Furthermore, we presented a semi-automated conflict resolution mechanism required to maintain the correctness of the parallelised simulation. Our addition of an OpenCL backend to the OpenABL framework not only enables the execution on CPUs and accelerators such as GPUs and FPGAs, but also opens up new possibilities such as multi-device co-execution.

We evaluated the performance of the OpenCL backend using three existing simulation models. It was observed that on a GPU, our approach outperformed the existing C and MASON backends, mainly due to a more efficient neighbour search method. In a high agent density scenario, our backend also delivers better performance than the FLAME GPU backend. In addition, we demonstrated our approach by developing two proof-of-concept traffic and crowd simulations, showing the performance benefits of a CPU-GPU co-execution.

[3] http://fpgasoftware.intel.com/opencl/

Our future work will focus on the automated assignment of computational tasks to the available hardware devices.

Acknowledgement. This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

References

1. Ahmed, K.I.: Modeling Drivers' Acceleration and Lane Changing Behavior. Ph.D. thesis, Massachusetts Institute of Technology (1999)
2. Belviranli, M.E., Bhuyan, L.N., Gupta, R.: A Dynamic Self-Scheduling Scheme for Heterogeneous Multiprocessor Architectures. ACM Trans. Archit. Code Optim. **9**(4), 57 (2013)
3. Chisholm, R., Richmond, P., Maddock, S.: A Standardised Benchmark for Assessing the Performance of Fixed Radius Near Neighbours. In: Desprez F. et al. (eds) Euro-Par 2016. LNCS, vol. 10104, pp. 311–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-58943-5_253
4. Collier, N., North, M.: Repast HPC: A Platform for Large-Scale Agent-Based Modeling (2011)
5. Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., Spagnuolo, C.: A Framework for Distributing Agent-based Simulations. In: Alexander M. et al. (eds) Euro-Par 2011. LNCS, vol. 7155, pp. 460–470. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-29737-3_51
6. Cosenza, B., Popov, N., Juurlink, B., Richmond, P., Chimeh, M.K., Spagnuolo, C., Cordasco, G., Scarano, V.: OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations. In: Aldinucci M., Padovani L., Torquati M. (eds) Euro-Par 2018. LNCS, vol. 11014, pp. 505–518. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_36
7. Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institution Press (1996)
8. Gardner, M.: Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life". Sci. Am. **223**(4), 120–123 (1970)
9. Grewe, D., Wang, Z., O'Boyle, M.F.: Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 1–10. IEEE (2013)
10. Helbing, D., Molnar, P.: Social Force Model for Pedestrian Dynamics. Phys. Rev. E **51**(5), 4282 (1995)
11. Kiran, M., Richmond, P., Holcombe, M., Chin, L.S., Worth, D., Greenough, C.: FLAME: Simulating Large Populations of Agents on Parallel Hardware Architectures. In: Proceedings of the International Conference on Autonomous Agents and Multiagent Systems. pp. 1633–1636. IFAAMAS (2010)
12. 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. In: an Mey D. et al. (eds) Euro-Par 2013. LNCS, vol. 8374, pp. 544–554. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-54420-0_53

13. Li, P., Brunet, E., Trahay, F., Parrot, C., Thomas, G., Namyst, R.: Automatic OpenCL Code Generation for Multi-device Heterogeneous Architectures. In: Proceedings of the International Conference on Parallel Processing. pp. 959–968. IEEE (2015)
14. Li, X., Cai, W., Turner, S.J.: Efficient Neighbor Searching for Agent-Based Simulation on GPU. In: Proceedings of the International Symposium on Distributed Simulation and Real Time Applications. pp. 87–96. IEEE (2014)
15. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: A Multi-agent Simulation Environment. Simulation **81**(7), 517–527 (2005)
16. Lysenko, M., D'Souza, R.M., et al.: A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. J. Artif. Soc. Soc. Simul. **11**(4), 10 (2008)
17. Macal, C.M., North, M.J.: Tutorial on Agent-based Modelling and Simulation. In: Proceedings of the Winter Simulation Conference. pp. 2–15. IEEE (2005)
18. Minar, N., Burkhart, R., Langton, C., Askenazi, M., et al.: The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations. Tech. rep. (1996)
19. North, M.J., Collier, N.T., Vos, J.R.: Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit. ACM Trans. Model. Comput. Simul. **16**(1), 1–25 (2006)
20. Pelechano, N., Badler, N.I.: Modeling Crowd and Trained Leader Behavior during Building Evacuation. IEEE Comput. Graphics Appl. **26**(6), 80–86 (2006)
21. Reynolds, C.W.: Flocks, Herds, and Schools: A Distributed Behavioral Model. In: Proceedings of the ACM SIGGRAPH. pp. 25–34. ACM (1987)
22. Richmond, P.: Resolving Conflicts between Multiple Competing Agents in Parallel Simulations. In: Lopes L. et al. (eds) Euro-Par 2014. LNCS, vol. 8805, pp. 383–394. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14325-5_33
23. Richmond, P., Walker, D., Coakley, S., Romano, D.: High Performance Cellular Level Agent-based Simulation with FLAME for the GPU. Briefings Bioinf. **11**(3), 334–347 (2010)
24. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. ACM SIGPLAN Notices **50**(9), 205–217 (2015)
25. Steuwer, M., Remmelg, T., Dubach, C.: LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 74–85. IEEE (2017)
26. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A Compiler Architecture for Performance-oriented Embedded Domain-specific Languages. ACM Trans. Embedded Comput. Syst. **13**(4s), 134 (2014)
27. Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., Knoll, A.: Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In: Proceedings of the International Symposium on Distributed Simulation and Real Time Applications. pp. 1–10. IEEE (2018)
28. Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., Knoll, A.: A Survey on Agent-based Simulation Using Hardware Accelerators. ACM Comput. Surv. **51**(6), 131:1–131:35 (2019)
29. Yang, M., Andelfinger, P., Cai, W., Knoll, A.: Evaluation of Conflict Resolution Methods for Agent-Based Simulations on the GPU. In: Proceedings of the Conference on Principles of Advanced Discrete Simulation. pp. 129–132. ACM (2018)