TUM School of Computation, Information and Technology
Technische Universität München

A Framework to Generate High-Performance Time-stepped Agent-based Simulations on Heterogeneous Hardware

Jiajian Xiao

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr. Florian Matthes

Prüfende der Dissertation:

1. Prof. Dr.-Ing. habil. Alois Christian Knoll
2. Prof. Wentong Cai, Ph.D.,
   Nanyang Technological University

Die Dissertation wurde am 11.04.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.08.2022 angenommen.

Abstract

Agent-Based Simulation (ABS) is a modelling approach where simulated entities i.e., agents, perform actions autonomously and interact with other agents based on a set of rules. ABSs have demonstrated their usefulness in various domains such as transportation, social science, or biology. Agent-based simulators commonly rely vastly on Central Processing Unit (CPU)-based sequential execution. As a result, they often suffer from long execution times, especially when the simulation model is complex and the scale is large.

One approach to circumvent this problem is to employ heterogeneous hardware, e.g., systems equipped with both CPUs and Graphics Processing Units (GPUs). Although existing works have shown this to be a promising solution, simulators that support heterogeneous hardware are still not widely utilized. As this thesis demonstrates, blame can be laid on the difficulties to find a viable partition of simulation components to run on different hardware devices as well as steep programming requirements to enable ABSs on heterogeneous hardware.

To ease the aforementioned issues, in this thesis, we present a full framework for automated generation of high-performance simulation code targeting heterogeneous hardware. The framework consists of two parts: OpenABLext and OptCL (short for Optimise performance targetting high-performance domain-specifiC Languages). OpenABLext significantly extends the existing OpenABL, a tool designed for generating high-performance ABS programs from sequential representations by parallelising the execution of so-called step functions which implement the agent-based models. We first extend its supported hardware to include multi-core CPUs, GPUs, and Field-Programmable Gate Arrays (FPGAs) by designing an OpenCL backend, allowing the incorporation of a wide range of hardware devices. We address OpenABL's limitation of 2-dimensional and 3-dimensional simulation environments by supporting graph-based simulation spaces in OpenABLext, opening it up for simulation types such as transport simulation which commonly use graph-based networks. A conflict resolution mechanism is introduced to ensure the simulation results of a parallel simulation run do not deviate from a sequential run, significantly improving reproducibility. Eventually, an online dispatcher is proposed as part of OpenABLext to enable automatic best suitable hardware selection for different simulation parts. In a comprehensive simulation study, we show OpenABLext not only drastically simplifies the programming effort

but also produces simulation code that outperforms other similar state-of-the-art approaches. With OptCL, we extend the idea of the online dispatcher to further boost the performance of the generated code on top of OpenABLext by enabling a collaborative execution (co-execution) mode where multiple hardware devices work simultaneously on different simulation parts. Through static analysis of data dependencies among compute-intensive code regions and performance predictions, the tool selects the best hardware for each simulation part as well as the execution schemes. The OptCL middleware operates at a language independent layer, a so-called Intermediate Representation. Therefore, it can not only benefit OpenABLext but also many other C-like Domain Specific Languages (DSLs). We demonstrate the power and versatility of OptCL by applying it to OpenABLext as well as another commonly used DSL called SYCL. In both scenarios, up to 21x speed-ups can be achieved.

The full framework proposed in this thesis significantly reduces the effort of employing heterogeneous hardware environments for agent-based simulation and thereby paves the way for the high-performance execution for a large variety of simulation models.

Zusammenfassung

Agentenbasierte Simulation ist ein Modellierungsansatz. In der agentenbasierten Simulation, viele kleine Einheiten d.h. Agenten führen autonom Aktionen aus und interagieren mit anderen Agenten . Es ist ein Ansatz zur Systembewertung, der in verschiedensten Bereichen wie z.B. den Sozialwissenschaften, der Biologie, oder im Transportwesen Anwendung gefunden hat. Gängige agentenbasierte Simulatoren verlassen sich immer noch stark auf hauptsächlich CPU-gestützte Ausführung, was dazu führte, dass sie oft unter langen Ausführungszeiten leiden, insbesondere wenn der Simulationsumfang groß oder komplex ist.

Durch den Einsatz heterogener Hardware, wie z.B. Systemen, die sowohl mit CPUs als auch mit GPUs ausgestattet sind, lässt sich, wie in vielen Arbeiten bereits vielversprechend gezeigt, dieses Problem reduzieren. Was der Akzeptanz und Anwendung heterogener Systeme im Simulationsbereich jedoch noch entgegensteht, ist die Tatsache, dass die Programmierung heterogener Hardware fundierte und auch spezialisierte Kenntnisse erfordert. Dies wird in dieser Arbeit anhand zweier Machbarkeitsstudien demonstriert, die zeigen, dass es einen erheblichen Aufwand erfordert, um eine optimierte Performanz für agentenbasierte Simulationen in heterogenen Hardwareumgebungen zu erreichen: Die erste Studie veranschaulicht dies an einer Plattform bestehend aus GPU and CPU, die zweite an einem System mit einem FPGA und einer CPU. Um die Programmierung von agentenbasierten Simulationen für heterogene Hardwareumgebungen zu vereinfachen, wird in dieser Arbeit das Framework OpenABLext entwickelt. OpenABLext basiert auf OpenABL, welches darauf ausgelegt ist, hoch-performante agentenbasierte Simulationen aus sequentiellen Repräsentationen zu generieren und die Ausführung sogenannter Stufen-Funktionen (welche die eigentlichen agentenbasierten Modelle implementieren) zu parallelisieren. OpenABLext erweitert diese Plattform um die Unterstützung von Multicore-CPUs, GPUs und FPGA durch die Einführung eines OpenCL-Backends. Da OpenABL nur 2D- und 3D-Simulationsräume unterstützt, wird die Plattform in dieser Arbeit außerdem um die Fähigkeit ergänzt, graphenbasierte Simulationsräume zu ermöglichen. Des Weiteren wird ein halbautomatischer Konfliktlösungsmechanismus entwickelt, um sicherzustellen, dass die Simulationsergebnisse in der parallelen Ausführung nicht von der einer sequenziellen abweichen. Schließlich wird ein Online-Dispatcher als Teil von OpenABLext vorgestellt, welcher automatisch die am besten geeignete Hardware für die verschiedenen Teile einer Simulation auswählt. In

detaillierten Leistungsbewertungen zeigt sich die verbesserte Leistung von OpenABLext sowie das Potenzial von FPGAs im Kontext der agentenbasierten Simulation. Als weitere Leistungsverbesserung wird der Online-Dispatcher um einen kollaborativen Ausführungsmodus erweitert, der es erlaubt, verschiedene Simulationsteile auf mehreren Hardwareumgebungen gleichzeitig auszuführen. Dies wird durch eine neu-entwickelte Middleware namens OptCL unterstützt, welche durch statische Analyse von Datenabhängigkeiten zwischen rechenintensiven Coderegionen und Leistungsvorhersagen die beste Hardware für jeden Simulationsteil auswählt und festlegt, ob diese Teile auf homogener Hardware oder in kollaborativer Ausführung abgearbeitet werden sollen. Die Middleware selbst arbeitet auf einer IR-Ebene, die es erlaubt, dass sie nicht nur mit OpenABLext, sondern auch mit anderen C-ähnlichen DLSs genutzt werden kann. Diese Vielseitigkeit wird durch die Anwendung mit SYCL, ein verbreitetes DSL, demonstriert.

Das in dieser Arbeit vorgestellte Framework bestehend aus OpenABLext und OptCL vereinfacht die Entwicklung von hoch-performanten agentenbasierten Simulationen erheblich.

Acknowledgements

I would like to thank my supervisor Prof. Dr. Alois Knoll for giving me the opportunity to pursue my Ph.D. under his supervision and his inspiration in helping me find my research directions. I would also like to express my sincere gratitude towards Prof. Dr. Wentong Cai from the Nanyang Technological University for his mentorship and guidance. His plentiful knowledge in the field of high-performance computing has helped me a lot.

Special thanks go to Dr. Philipp Andelfinger and Dr. David Eckhoff for their incredible support and mentorship. Without them this thesis would not be possible. A special mention to Görkem Kılınç for giving me the unique chance to supervise his master's thesis. His work plays an important part in my research. I would also like to thank Dr. Heiko Aydt and Dr. Suraj Nair for offering me the opportunity to work at TUMCREATE.

Further, I would like to thank all my colleagues at TUMCREATE who have been providing me with a fantastic research environment. Of all my colleagues, I would like to give special thanks to Dr. Daniel Zehe and Dr. Jordan Ivanchev for their inspiration and support in many of my research works.

A very special gratitude goes out to all down at Singapore National Research Fund for helping and providing the funding for the work.

Lastly, I would like to thank my family for their support with all matters of the heart. A special thank goes to my wife Ms. Wang Yanting for her exceptional support.

Contents

List of Figures

List of Tables

Acronyms

ABMS	Agent-Based Modelling and Simulation.
ABS	Agent-Based Simulation.
ABTS	Agent-Based Traffic Simulation.
AMD	Advanced Micro Devices.
APU	Accelerated Processing Unit.
ASIC	Application-Specific Integrated Circuit.
AST	Abstract Syntax Tree.
CA	Cellular Automata.
CityMoS	City Mobility Simulator.
CLB	Configurable Logic Block.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DES	Discrete-Event Simulation.
DSL	Domain Specific Language.
EDA	Electronic Design Automation.
FPGA	Field-Programmable Gate Array.
FU	Function Unit.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
HLS	High-Level Synthesis.
HPDSL	High-Performance Domain Specific Language.
IDM	Intelligent Driver Model.

II	Initiation Interval.
IR	Intermediate Representation.
LUT	Look-Up Table.
MOBIL	Minimizing Overall Braking Induced by Lane change.
MPI	Message Passing Interface.
MPPA	Massively Parallel Processor Array.
OpenCL	Open Computing Language.
OpenMP	Open Multi-Processing.
OptCL	Optimise performance targetting high-performance domain-specifiC Languages.
PTX	Parallel Thread Execution.
SFU	Special Function Unit.
SIMD	Single Instruction Multiple Data.
SM	Streaming Multiprocessor.
SoC	System on Chip.
SP	Streaming Processor.
SPIR	Standard Portable Intermediate Representation.
TPU	Tensor Processing Unit.
TSS	Time-Stepped Simulation.

1 Introduction

Contents

Some parts of this chapter are taken from a work published in the ACM Computing Surveys (CSUR) Volume 51 Issue 6 [2].

1.1 Motivation

Due to the breakdown of Dennard scaling, clock frequencies of single Central Processing Units (CPUs) are no longer increasing exponentially, even though transistor counts are still growing [3]. Large CPU vendors such as Intel or Advanced Micro Devices (AMD) tend to focus on developing multi-core processors. This in turn calls for parallel computing techniques, as compute-intensive programs such as simulations that do not run in parallel can no longer be easily sped up by incorporating a newer and faster CPU. The other trend is that today's hardware has become more and more heterogeneous. A multi-core CPU with an integrated or commodity Graphics Processing Unit (GPU) is the de facto standard for a modern personal computer. Cloud service giants such as Amazon or Microsoft also offer instances equipped with gpu or even Field-Programmable Gate Arrays (FPGAs). Intel introduced a hybrid architecture on its latest generation CPU, with one die having both power efficient cores and high-performance cores. This reveals a new era of heterogeneity on one piece of hardware.

However, choosing the best suitable hardware for a given computational task is not trivial. Some types of hardware are better suited for certain types of computations than others. For example, tasks with large amounts of fine-grained parallelism can benefit greatly from the massively parallel architecture of modern GPUs with their thousands of cores. Tasks that are largely sequential or characterised by unpredictable data access

and control flow lend themselves better to CPUs with out-of-order execution, long pipelines and large caches. Similarly, if offloading a task to a GPU requires copying large amounts of data to and from graphics memory, execution on a CPU may be preferable even if substantial parallelism is available. This issue can be addressed by an Accelerated Processing Unit (APU), where CPU and an integrated graphics core (of lower performance compared to stand-alone GPUs) share the same memory. Lastly, compute-intensive and memory-light tasks can be outsourced to FPGAs which can be programmed to carry out specific computations in hardware.

One field that has always more demand for better performance is simulation. Faster computers allow an increase in complexity of the incorporated simulation models, allowing researchers to obtain more accurate results in less time. Agent-Based Simulation (ABS)s have received broad attention as they can be employed to study various domains, such as road traffic [4], social networks [5], pedestrian movement [6], military [7], biology [8], economics [9], privacy[10], and so on. The main characteristic of ABS is that autonomous agents (e.g., individuals or entities) act and interact to create effects of emergence on the entire system. The complex decision-making of agents and the huge scale of many simulated systems can lead to enormous execution times, motivating the need for high-performance computing platforms.

ABSs are a promising target for parallel computing techniques as agents are autonomous and in many cases carry out independent computations. In mobility simulations, for instance, interactions between agents usually only take place between close-by agents in a somewhat regular 2-dimension environment, allowing researchers to employ space partitioning without inducing too much synchronisation overhead. Moreover, many ABSs are time-stepped and agents are often updated at the same logical time, providing inherent independence and thus potentials for parallelised execution. However, being able to partition a problem and execute it in parallel is not a guarantee that it can be accelerated using heterogeneous hardware. Improper mapping of simulation parts to hardware could lead to little increase or even decrease in performance which we will show through two feasibility studies presented in Chapter 3. The studies will also reveal that tuning the hardware mapping manually requires in-depth knowledge of the underlying hardware as well as significant programming efforts. This creates burdens, especially for simulationists or domain experts, who might not be very familiar with hardware. Hence, there is an urgent need for a tool to abstract away the implementation and hardware details. This thesis aims to close this gap.

1.2 Problem Statement

To create such tool to accelerate ABSs on heterogeneous hardware, the following challenges have to be overcome:

- First, simulations have to be partitioned with heterogeneity in mind to decide which part of the program lends itself best to a specific hardware device, considering the resulting overhead from data transfers between the different devices. To solve this challenge requires a deep dive into the architectural characteristics of different hardware.
- Second, depending on the available hardware, the mapping of simulation parts to hardware devices will likely be different. Complex simulations typically also exhibit scattered and unpredictable memory access and control flow as the model state develops dynamically over time. This further complicates an efficient distribution to heterogeneous hardware. This implies that the partitioning problem raised in the first point needs to be solved again, if the underlying hardware has changed.
- Last but not least, in order to make heterogeneous accelerators available to simulationists and modellers without having thorough knowledge of the specific hardware platforms, the framework should abstract away from hardware specifics or automate the key or the entire adoption process.

Prior to this thesis, there are frameworks aiming to achieve a similar goal but focusing on one type of accelerator e.g., MASON [11], Repast-HPC [12], and EcoLab [13] targeting traditional CPU-based environments. Other frameworks such as FLAME GPU [14] and Many-Core Multi-Agent System (MCMAS) [15] support graphics cards. There also exist a handful of Domain Specific Language (DSL)s such as SYCL [16] and Liszt [17] for general high-performance computing which targets multiple hardware platforms. In particular, OpenABL [18] is a novel DSL for ABSs targeting CPUs, GPUs and a cloud environment. However, for most of the DSLs like OpenABL, users have to specify which type of accelerator to use at each compilation. This may yield another two issues: First, choosing an accelerator that leads to the best performance again requires knowledge of the hardware or trial-and-error attempts. Second, the outputted programs target different hardware can not be executed collaboratively. This leaves the power of the idle hardware untapped.

This thesis aims to build a framework based on OpenABL by overcoming its current limitation. The framework abstracts away hardware details, allowing users to write

sequential ABS code which is later translated by the framework to run on accelerators. It also takes care of the hardware assignment, enabling collaborative execution (co-execution) where multiple accelerators can work simultaneously, with little intervention from the user.

1.3 Key Contributions

In addition to the framework, there are intermediate results and achievements along the way which can benefit researchers, simulationists and modellers.

- (**Chapter 2**) A comprehensive survey of agent-based simulations on hardware accelerators is given. We provide an overview and categorisation of the literature according to the applied techniques. It targets simulationists and modellers seeking an overview of suitable hardware platforms and execution techniques for a specific simulation model, as well as methodology researchers interested in potential research gaps requiring further exploration.
- (**Chapter 3**) In order to understand the efforts, challenges, and oracle gains to enable ABSs on heterogeneous hardware, two feasibility studies are conducted where a typical Agent-Based Traffic Simulation (ABTS) is manually ported to run on different accelerators. In the first study, starting from a pure CPU-based execution mode, we evaluate the performance gains by varying different parallelisation schemes and offloading them onto GPUs. We believe it is the first work to systemically study the impact of simulation partition strategies on performance of ABTSs. The second study drafts a design of accelerating ABS on FPGAs using high-level synthesis. To the best of our knowledge, it is also the first FPGA-accelerated ABTS to rely on models from the traffic engineering literature, and the first to rely on high-level synthesis. The studies can be inspirational for researchers who intend to accelerate their own applications on the given hardware platforms.
- (**Chapter 4**) For simulationists intend to build their own ABS applications from scratch and avoid diving deep into the hardware details, we provide the OpenABLext framework which extends an existing framework, OpenABL [18], to automatically generate high-performance ABS code from sequential representations targeting heterogeneous hardware. The extension enriches the syntax of OpenABL to allow defining the simulation space as graphs as well as providing a conflict resolution mechanism through user-specified code. Further, the original

framework is also enhanced to output device-aware code in Open Computing Language (OpenCL), enabling the co-execution of ABS on heterogeneous hardware platforms consisting of CPUs, GPUs and FPGAs. OpenABLext can be seen as an enabler to tap the computing power of heterogeneous hardware platforms for ABS.

- (**Chapter 5**) To further optimise the performance, we propose a middleware called Optimise performance targetting high-performance domain-specifiC Languages (OptCL) that complements OpenABLext by enhancing its co-execution ability. Through a static analysis of data dependencies among compute-intensive code regions and performance predictions, OptCL selects the best execution schemes out of purely CPU/accelerator execution or co-execution in an automated manner. Notably, the middleware operates on Intermediate Representations (IRs) and, therefore, is completely independent of OpenABLext. That implies many other existing DSLs similar to OpenABLext can also benefit from employing OptCL.

1.4 Outline

The remainder of this thesis is structured as follows:

In Chapter 2, background knowledge of general ABS techniques as well as basics of the hardware platforms is given, followed by a comprehensive review of techniques for enabling ABSs on heterogeneous hardware.

In Chapter 3, two feasibility studies are conducted in which we try to port a CPU-based ABS onto GPUs and FPGAs. The studies demonstrate the manual efforts required to enable a CPU-based ABS on heterogeneous hardware. Furthermore, by evaluating different simulation parallelisation schemes, we reveal the best parallelisation scheme, i.e. how to partition and execute an ABS on available hardware.

To help reduce the aforementioned efforts for modellers and simulationists, Chapter 4 introduces the OpenABLext framework which extends the existing OpenABL framework.

The OpenABLext framework can be further automated and the performance of the generated ABS code can be improved by enabling co-execution. In Chapter 5, we describe and evaluate an extension plug-in to OpenABLext named OptCL to achieve such goal.

With the framework comprised of OpenABLext and OptCL formed, Chapter 6 concludes this thesis and sketches the future research directions.

2 State of the Art

Contents

Substantial parts of this chapter have been published in the ACM Computing Surveys (CSUR) Volume 51 Issue 6 [2].

Prior to introducing the proposed framework, this chapter gives an overview of the background knowledge and the state-of-the-art techniques.

2.1 Agent-based Simulation

Agent-Based Modelling and Simulation (ABMS) is a widely used approach [19] to evaluate complex systems in various domains such as traffic, crowds, economics, information propagation, and biology. The field of ABMS is extensive, leading to a large number of tools and frameworks (e.g., MASON [11], Repast [12], NetLogo [20], Swarm [21], MATSim [22]), some of them general purpose, others tailored to specific applications or domains. A 2010 survey by Allen discusses a selection of ABMS frameworks and their applications in a wide range of domains [23]. More recently, Abar et al. [24] provides a comprehensive overview of more than 70 agent-based simulation tools in terms of programming languages, software and hardware requirements, and agent interaction types, as well as the target domains and the difficulty in model development.

```
1  class Agent: Coord position;
2
3  while (sim_time not end) {
4    model_1();
5    model_2();
6    ...
7    sim_time advances;
8  }
9
10 void model_1() {
11   /** sense_begin **/
12   List agents = getNeighbouringAgents(position);
13   /** sense_end **/
14
15   /** think_begin **/
16   Coord velocity = computeVelocity(agents);
17   /** think_end **/
18
19   /** act_begin **/
20   position = position + velocity*time;
21   /** act_end **/
22 }
23
```

```
24 void model_2() {...}
```

Listing 2.1: Example of an agent-based crowd simulation.

In ABSs, the simulated entities are agents that perform actions autonomously and interact with other agents based on certain rules. Listing 2.1 gives an example pseudo code of a typical ABS simulating the movement of crowd. Each agent-based model ABS typically follows a *Sense-Think-Act* cycle (e.g., [25]): in the *Sense* stage, an agent detects and analyses its neighbours as well as the environment in which it resides. In the *Think* stage, an agent makes judgement based on the information collected during the Sense stage. The update of states takes place in the *Act* stage. The simulation time is typically advanced in fixed time steps following the Time-Stepped Simulation (TSS) approach at which all agents update their states.

2.1.1 Constraints for the studied ABSs

Many agent-based simulators and models roughly follow a common set of constraints which can be leveraged to simplify our extraction of parallelism. We identify the following constraints which are held throughout the entire thesis:

1. *Time-stepped execution*: usually, the model time is advanced in fixed increments. At each time step, all agents update their states.

2. *Two states per agent*: to decouple the simulation results from the order in which agent updates are performed, simulators commonly support storing each agent's old state at $t-1$ and the new state at t separately. During an update from $t-1$ to t, only read accesses are performed to the agents' states and the environment state at $t-1$, and only write accesses to the states at t. Thus, within an update, there are no read-after-write dependencies across agents.

3. *Agent-based models as function calls*: the actions and interactions of agents are implemented by executing a series of agent-based models. Conceptually, one model can be represented by one function call. Notably, the models can carry interdependencies. Therefore they may required to be executed in certain order. A blind parallel execution of all models can possibly result in wrong simulation results.

4. *Sense-Think-Act cycle*: we assume that each agent-based model follows the well-known Sense-Think-Act cycle, with one such cycle per model.

2.1.2 Computational Aspects of ABS

The literature on executing ABS using heterogeneous hardware can be organised according to the challenges addressed by the individual works. In this literature review, we identified five such challenges, which are consequences of features shared by most ABS models.

1. **Hardware assignment**: A well-known challenge in parallel and distributed simulation lies in partitioning the simulation workload among the processing elements. Generally, there are two dimensions according to which a simulation can be partitioned [26]: *domain decomposition* partitions according to the simulation space (e.g., different roads in a traffic simulation), while *functional decomposition* partitions according to different models (e.g., different layers of the network stack in a computer network simulation). In ABS on heterogeneous hardware, the hardware assignment is further complicated by the shifting workload due to the agents' autonomy and mobility, and by the heterogeneity of the hardware platform, in which devices may differ in their suitability for certain types of computations. Existing techniques to approach this challenge either attempt to determine static boundaries within the simulation that allow for an efficient partitioning without further adaptation, or dynamic hardware assignments that are updated at simulation runtime.

2. **Data transfer overhead**: As a result of the agents' mobility and interactivity, even under an efficient static or dynamic partitioning, frequent data transfers are usually necessary among the hardware devices to migrate agents or to reflect inter-agent communications. Techniques have been proposed to exploit the limited agent velocity and interaction range to reduce the impact of the data transfers on the simulation performance.

3. **Scattered memory accesses**: The dynamic and largely unpredictable agent movement and interaction translates to memory access patterns that are in conflict with the regular, i.e., linear, accesses preferred by common accelerators. The literature proposes representations of irregular structures using regular memory layouts and caching heuristics to improve the efficiency when accessing the simulation data.

4. **Maximisation of parallelism**: A defining characteristic of simulations is the notion of simulated time, which puts constraints on the simulation progress during parallel execution. Although there may be a substantial amount of computation

scheduled, processing elements may frequently be blocked waiting in order to maintain synchronisation with the other processing elements. The field of parallel and distributed simulation proposes a wide range of algorithms to extract as much parallelism as possible while maintaining the synchronisation of simulated time [27]. In recent years, a number of works have re-evaluated and adapted these approaches targeting the execution of ABS on accelerators.

5. **Abstraction from hardware specifics**: Given the frequent adaptations and extensions commonly made during the development of an ABS model and during the verification and validation process, it is necessary to provide frameworks to simplify the modellers' implementation work while still maintaining high performance. In past years, a number of frameworks have been presented in the literature that abstract from the hardware details as well as libraries for unified access to the memory of different devices.

2.2 Hardware Platforms

In this section, the technical characteristics of hardware platforms that have been used to accelerate agent-based simulations are given.

2.2.1 Many-Core CPU

Architecture: A many-core (or many integrated core, MIC) CPU contains a group of CPU cores on a single chip. A many-core CPU can be connected to the host machine via PCI-E or can be a standalone CPU with direct access to the system memory.

Benefits: A notable advantage of some many-core CPUs over GPUs and FPGAs is their capability to execute largely unmodified parallelised code written for regular CPUs [28]. Since the individual cores support out-of-order execution, employ deep instruction pipelines, and have access to comparatively large caches, the need to adapt a program's control flow to the hardware is less pressing than with, e.g., GPUs [29]. Further parallelism may be extracted using a Single Instruction Multiple Data (SIMD) style of programming using instruction set extensions such as AVX-512 [30].

Recent work showed that many-core CPUs can substantially accelerate Discrete-Event Simulation (DES) [31, 32]. A number of authors also evaluated the acceleration of various types of simulations such as fluid dynamics and seismic wave propagation using non-x86 many-core CPUs [33, 34].

Limitations: In light of the relatively high cost of recent many-core CPUs (≈ US$3368.00 as of 03/2018 for an Intel Xeon Phi Processor 7290F) compared to other accelerators, the performance gains compared to traditional multi-core CPUs have frequently been relatively low (e.g., [35]). Since the performance depends strongly on parameters such as the number of threads and on the use of the different available types of memory, parameter tuning may be necessary [36].

2.2.2 GPU

Architecture: GPUs utilise a massively parallel architecture, which makes them more efficient than CPUs when large volumes of data can be processed using the same instructions in parallel. This type of parallelisation is called SIMD. Their original purpose was to accelerate the processing of three-dimensional scenes to be displayed on two-dimensional screens. However, modern GPUs have evolved to support a wide range of computational tasks using programming frameworks such as Compute Unified Device Architecture (CUDA), OpenACC [37], and OpenCL [38]. CUDA and OpenCL follow a similar programming model, with some differences in terminology.

We sketch the GPU architecture and programming model using NVIDIA's terminology. AMD hardware follows a similar design. A modern GPU consists of a scalable number of Streaming Multiprocessors (SMs), which contain a number of Streaming Processors (SPs) that carry out most of the computations, Special Function Units (SFUs) for special operations such as trigonometric functions, and low-latency on-chip memory. Off-chip RAM is shared among all SMs [39].

GPU computations are organised hierarchically: at the lowest level, there are threads representing a sequential control flow. Threads are grouped into *warps* of a hardware-specific size (32 threads on current NVIDIA hardware). Threads within a warp are executed in lockstep, i.e., if the control flow diverges, the branches are serialised. Thus, it is important to minimise intra-warp divergence. A configurable number of warps form a block, within which efficient memory synchronisation is possible. Per-SM warp schedulers dynamically assign runnable warps to the available SP to minimise stalling on high-latency memory accesses. Typically, programs schedule many more threads than there are physical SP to support this type of memory latency hiding [40].

Memory access overheads can be reduced by adhering to memory access patterns that allow for *coalescing*, i.e., aggregated execution of memory accesses by multiple threads [40]. Generally, the number of memory requests is minimised when adjacent threads access adjacent memory locations. Achieving memory coalescing is a common focus of research on GPU acceleration (e.g., [41, 42]).

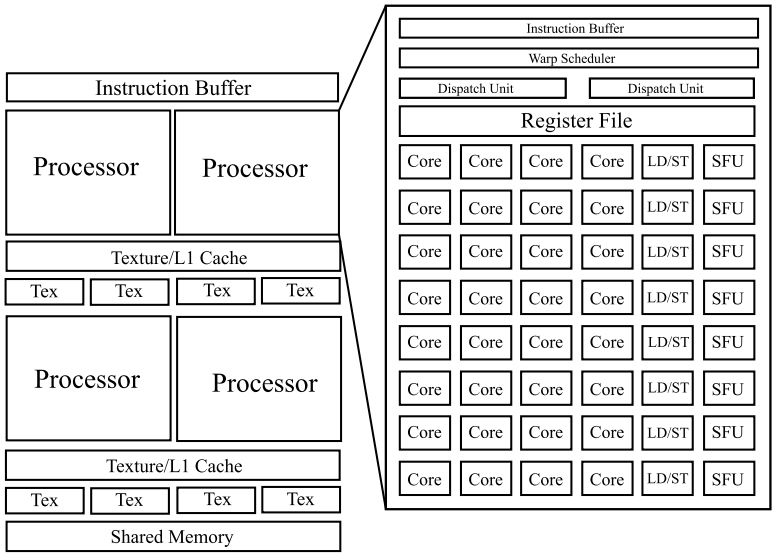


Figure 2.1: GPU architecture

Benefits: GPUs lend themselves best to problems that can be formulated so that large numbers of similar code segments are executed on different data. Often, GPUs accelerate such data-parallel tasks by one to two orders of magnitude compared to implementations on multi-core CPUs.

Frameworks such as CUDA, OpenCL, and OpenACC, as well as libraries such as Thrust [43] and CUBLAS [44], enable relatively simple development compared to platforms such as FPGAs [45]. Programming frameworks are available even for more specialised tasks such as ABS [14].

Limitations: The main requirements for high performance GPU code are a large degree of parallelism, the possibility to achieve coalesced memory access, and a largely common control flow among the threads within a warp. Thus, memory-intensive tasks with complex data dependencies are typically difficult to execute efficiently on GPUs [46, 47].

Further, since dedicated graphics cards are connected to the host CPU via the PCI-E bus, overhead is introduced by the data transfers between CPU and GPU. For instance, a PCI-E 3.0 x16 link allows an NVIDIA Titan X card to transfer data between host and graphics memory at up to 16 GB/s, while the GPU can access its off-chip RAM at up to 336.5 GB/s. However, the impact of data transfers may be lower when relying on recent architectures' interconnects such as NVIDIA's NVLink [48] and AMD's Infinity Fabric [49], which achieve throughputs of up to 300 GB/s.

Compared to many-core CPUs, programming for GPUs still requires profound knowledge of the GPU architecture [50]. As with many-core CPUs, the large number of configurable parameters render the performance tuning of GPU programs an important but challenging task [51].

2.2.3 APU

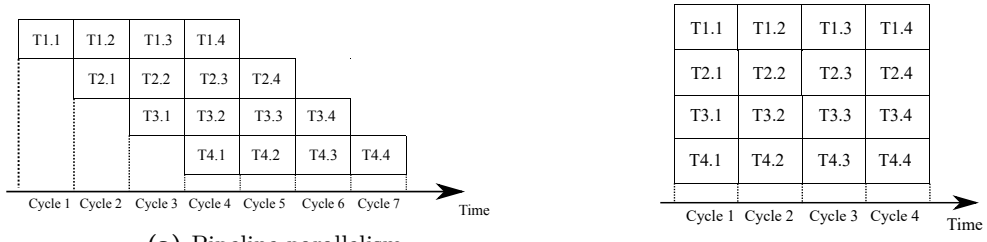
Architecture: APUs integrate CPU and GPU on a single chip. Although the term APU was coined by AMD, Intel CPUs with Intel HD Graphics follow a similar architecture. Unlike stand-alone GPUs, the fused GPU of an APU has direct access to the host memory through a low-latency and high-bandwidth bus.

Benefits: The main benefit of APUs is the opportunity for zero-copy memory access: since all memory is accessible both from the CPU and the GPU, costly data transfers over a relatively low-bandwidth bus like PCI-E can be avoided. Zero-copy memory access also provides memory savings, as only one copy of an object in memory is required. Since memory access is shared, tasks can efficiently be assigned according to their suitability for the CPU or GPU portion of the device.

Limitations: Existing APU products focus more on energy efficiency than high performance. They typically contain fewer processing units than stand-alone CPUs and GPUs of the same hardware generation. For example, the Ryzen 5 2400G APU by AMD has 704 Vega-based stream processors, while the dedicated graphics card AMD RX Vega 64 has 4096 stream processors. As a consequence, compared to high-end stand-alone CPUs and GPUs, their computational power is relatively low. Still, as will be discussed in Section 2.4, some works have considered APUs for accelerating agent-based simulations.

2.2.4 FPGA

Architecture: An FPGA is an integrated circuit made of an array of interconnected Configurable Logic Blocks (CLBs). FPGAs often provide various communication interfaces such as PCI-E, UART, and Ethernet. A CLB consists of several slices (sometimes also called logic cells), each slice containing a set of storage elements and Look-Up Table (LUT). A LUT has a number of inputs and outputs as well as flip flops that store a mapping between possible inputs and outputs. The mapping between inputs and outputs is defined by the users [52]. In addition, the FPGA may have access to off-chip Dynamic Random Access Memory (DRAM).



(a) Pipeline parallelism

(b) SIMD parallelism

Figure 2.2: Pipeline parallelism versus SIMD parallelism. We assume four tasks, each task being divided into four operations (e.g., T1.1 - T1.4).

The logic to be placed on an FPGA is typically specified in a Hardware Description Language (HDL) such as VHDL [53] or Verilog [54]. In recent years, there have been intensive efforts to enable High-Level Synthesis, i.e., to generate FGPA layouts directly from high-level programming languages such as C, C++, or Java. Recently, big FPGA vendors such as Intel and Xilinx released dedicated SDKs to support FPGA programming using OpenCL [55].

Benefits: Due to the flexibility and high energy efficiency of FPGAs, they are frequently used for computationally intensive and highly parallelisable tasks. For instance, FPGAs can be three orders of magnitude faster than GPUs when conducting specialised tasks such as encrypting a single 64-bit block by the DES [56]. In contrast to CPUs or GPUs, on which data paths are fixed, FPGAs provide flexible and customised data paths [57]. In past years, FPGA have received more attention in the field of simulation, particularly in Electronic Design Automation (EDA), since hardware designs can be naturally expressed as FPGA layouts.

Limitations: FPGAs are usually connected to a host CPU without direct access to system memory. The resulting need for data transfers can reduce the potential for performance gains.

FPGAs are regarded as lacking in programmability when compared to CPUs and GPUs [45, 56]. Although recent efforts towards high-level synthesis alleviate this limitation, manual tuning is still necessary to achieve the best performance [58, 59].

Finally, FPGA are configured for a specific task. Since reconfiguration can take multiple hours [60], FPGAs do not facilitate development processes that require fast iteration. This may limit the applicability of FPGAs in early phases of simulation model development, where changes to the simulation model frequently occur and require immediate feedback for evaluation.

2.2.5 Other Hardware Platforms

Application-Specific Integrated Circuits (ASICs) are integrated circuits fabricated to support a particular application. System on Chip (SoC) devices, which integrate components such as microprocessors, memory, and input/output, on a single chip are commonly fabricated as ASICs. To our knowledge, ASIC and SoC devices have not yet been explored as platforms to accelerate ABS. However, in the field of DES, some works have considered offloading to ASICs [61, 62, 63, 64, 65]. Notably, some of the envisioned components were fabricated physically [64].

Recent ASIC and SoC devices include Google's Tensor Processing Units (TPUs) and NVIDIA's Xavier. We briefly sketch potential uses of these hardware platforms in the context of ABS. The core of a TPU is a matrix-multiply unit designed to accelerate machine learning applications based on neural networks. At such tasks, TPUs can outperform recent CPUs or GPUs by a factor of up to 30 [66]. A potential use case for TPUs in ABS lies in the acceleration of agent models relying on neural networks (e.g., [67]). When exploring the input parameter space of an ABS (e.g., [68]), TPUs could also accelerate machine learning algorithms used to steer the exploration. Xavier could be used to accelerate applications with feedback between an ABS and a real-world system, e.g., for dynamic road traffic control based on simulation-based predictions (e.g., [69]). The processing of sensor data and the execution of the simulation could be assigned to the different processing elements of Xavier.

Although there are promising directions for future work based on these emerging platforms, we are not aware of existing literature on the acceleration of ABS using ASICs and SoCs. As our main focus is common accelerators, these types of special hardware will be excluded in the rest of the thesis.

2.3 Open Computing Language (OpenCL)

A large part of this thesis will rely on the Open Computing Language (OpenCL) as the programming language. We, therefore, give a brief introduction here. OpenCL is a framework that allows users to write parallel programs in C-99 standard. It abstracts away low-level hardware specifics. OpenCL is supported by a wide range of hardware including CPUs, GPUs, APUs, and FPGAs, allowing it to target heterogeneous hardware environments.

An OpenCL execution environment is comprised of a host (usually a CPU) and one or multiple devices (e.g., CPUs, GPUs) (cf. Figure 2.3). A host program initialises the

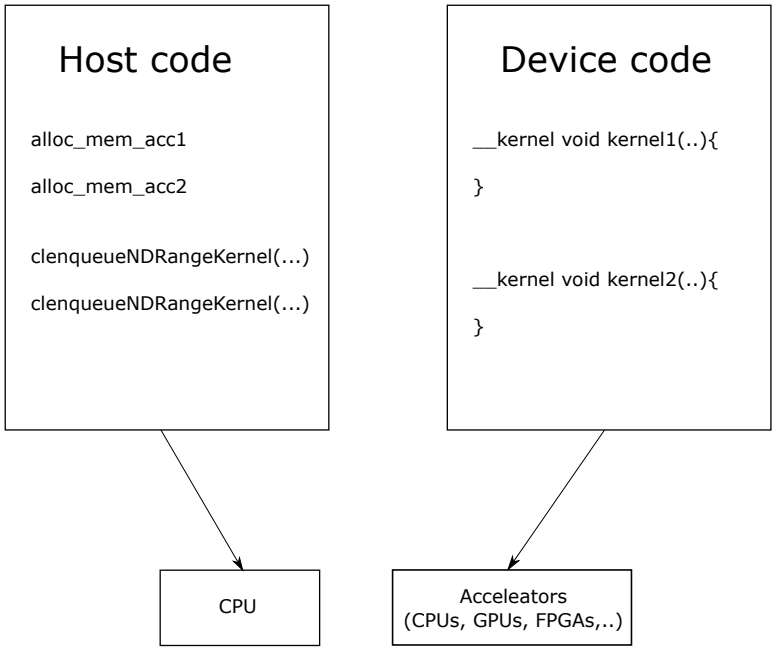


Figure 2.3: OpenCL programming paradigm

environment, control, memory, and computational resources for the devices. A device program consists mainly of so-called kernels that implement the computational tasks.

In OpenCL, threads that process the tasks are referred to as work-items. A configurable number of work-items form a *work-group*. Work-items belonging to the same work-group have access to a certain amount of shared memory and can be synchronised efficiently.

OpenCL offers a two-layer memory hierarchy. Low-latency local memory usually maps to the on-chip memory, shared among work-items in the same work-group. Global memory often binds to the massive but latency-prone off-chip memory to which all work-items have access. In some research, the memory hierarchy is sometimes described as a four-layer architecture which includes, other than the two layers introduced above, a read-only memory layer for constants and a private memory layer which is usually registers.

2.4 Addressing the Challenges of Agent-Based Simulation on Accelerators

Agent-based simulation on hardware accelerators started to receive attention from the research community from the early 2000s onwards. The vast majority of these works

Domain/Hardware	Many-Core CPU	GPU	APU	FPGA
Mobility		[70, 71, 72, 73, 74, 75, 76, 77, 69, 78]	[79]	[80]
Biology	[15]	[81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92]		[93]
Ecology		[94, 95]		[96]
Social	[28]	[97, 98, 99, 100, 101, 50, 102]		[103]
Physics and Chemistry		[104, 105, 106, 107, 108, 109, 95]		[110]
Networks		[111, 112, 1, 113, 114, 115, 87, 116]		
Domain-independent Simulation Framework	[15]	[85, 91, 15, 117, 118, 119, 120, 18, 121]		

Table 2.1: Simulation model domains considered in the works covered in the chapter.

Challenge	Technique	Publications
Hardware assignment	Static assignment by type of computation	Many-Core [28], GPU [69, 122, 1, 123, 112, 75, 76, 124, 106, 121] [88, 89, 125], APU [79], FPGA [80, 93, 96, 103]
	Dynamic assignment based on runtime measurements	GPU [126, 127, 128, 129, 125, 130], FPGA [126]
Data transfer overheads	Overlapping of communication and computation	GPU [111, 131, 107]
	Computation replication at partition boundaries	GPU [83, 86]
Scattered memory accesses	Manual caching in shared memory	GPU [85, 86, 92]
	Heuristics for agent update order	GPU [78, 132, 97, 98]
	Representation of irregular data structures by arrays and grids	APU [79], GPU [109, 133, 84, 70, 71, 104, 81, 72, 100, 82] [134, 99, 105, 90, 135, 116, 136, 114, 137, 138], FPGA [57, 110]
Maximisation of parallelism	Multiple replications in parallel	GPU [122, 73, 94, 111, 101, 95]
	Window-based event execution	GPU [108, 139, 113, 114, 115, 116, 99, 87]
	Speculative execution	GPU [102, 136], FPGA [110]
	Computation sorting	GPU [111, 87, 90]
Abstraction from hardware specifics	Frameworks to support simulation development	Many-Core [15], GPU [85, 91, 15, 133, 77, 18]
	Unified memory access	GPU [118, 117, 119, 120]

Table 2.2: A classification of the challenges in ABS on accelerators along the relevant works addressing them.

focus on GPUs, mainly because they are comparatively inexpensive, and because, in recent years, the ease of programming of GPUs has slowly been approaching that of CPUs. Furthermore, well-established programming frameworks such as OpenCL enable the formulation of models in a less hardware-specific manner. For publications that considered specific simulation models, Table 2.1 shows the simulation domains and hardware platforms, providing researchers with pointers to relevant works in their respective domain.

The literature is organised along the key challenges we identified in Section 2.1, that is, hardware assignment, data transfer overheads, scattered memory accesses, maximisation of parallelism, and abstraction from hardware specifics. In the following, we discuss the techniques from the literature applicable to these key challenges in agent-based simulation. Table 2.2 summarises the systematisation of knowledge presented in this chapter. It contains our classification of challenges, techniques, publications, and types of accelerators.

2.4.1 Hardware Assignment

One of the main challenges in parallel and distributed computations in heterogeneous hardware environments lies in finding a suitable partitioning, i.e., assignment of a given problem to the available hardware [140]. We discuss techniques that have been used to address this problem according to two different, yet interrelated, aspects: first, we consider techniques to select suitable hardware for sub-tasks according to their ability to efficiently execute certain types of computations. The minimisation of data transfers among the partitions running on separate devices will be considered in the next subsection.

The existing approaches can be roughly categorised as follows:

1. **Static assignment**: if the simulation model involves different types of computations that clearly suggest a certain hardware mapping, it may be sufficient to partition the model prior to a simulation run without any adaptation during runtime. For instance, model segments involving large numbers of independent floating point operations may be well-suited for execution on a GPU, whereas segments with highly data-dependent control flow suggest the execution on a CPU.

2. **Dynamic assignment**: frequently, the dynamic behaviour of a simulated system at runtime translates to unpredictable computational patterns. In such cases, maintaining high performance may require an adaptation of the hardware mapping based on performance measurements at runtime. An inherent challenge of dynamic assignment is the trade-off between the performance increase through an improved assignment and the costs of runtime measurements and re-assignment.

An ample body of research has considered the parallelisation of general programs onto heterogeneous platforms, which is an enormous challenge due to the arbitrary control flows and memory access patterns that can be present in general programs. Thus, typically, the approaches limit themselves to program portions that are particularly amenable to parallelisation on accelerators. In the case of ABS, constraints such as the separation of data into a per-agent state and the limited sensing range of agents somewhat simplify the problem of parallelisation, potentially enabling a higher degree of automation in the hardware mapping. In Section 2.5, we outline the vision of an automated approach.

2.4.1.1 Static Assignment

The simplest hardware assignment is to execute the entire simulation on a single device. This approach is common in the existing work on FPGA-based ABS. For instance, Vourkas and Sirakoulis [96], who implement an environmental model simulation based on Cellular Automata (CA). The authors note the structural similarity between a two-dimensional cellular automaton and an FPGA and assign one cell to eachCLB. If the number of cells exceeds the number of CLBs, the simulation lattice is partitioned into several layers, which are processed one after the other. Similarly, Cui et al. [93] and Georgoudas et al. [103] show high performance when assigning ABS models operating on cellular grids to a single FPGA. A number of works on GPU-based ABS take the same approach of assigning the entire simulation to the accelerator [69, 83, 84, 70, 71, 77]. In these works, the computations associated with the Sense-Think-Act cycle of the individual agents are often assigned to one GPU thread each.

Often, the available hardware devices lend themselves to specific types of computations, or the memory consumption of the simulation exceeds the capacities of an individual device. Then, it is necessary to find a partitioning for the ABS, which may follow either a domain decomposition or a functional decomposition [26].

Domain decomposition: When using domain decomposition, the simulation space is partitioned and each partition is assigned to a separate processing element. An example is given by the work by Lai et al. [28], who implement Game of Life [141] and a simulation of urban sprawl processes using cellular automata [41]. The authors compare the performance achieved when using one CPU per execution node, one GPU per node and 60 cores per CPU-based many-core accelerator, using MPI for inter-node communication in each instance. The authors conclude that the use of accelerators provides a performance benefit over the purely CPU-based execution. Given a sufficiently large number of assigned processors, using the CPU-based many-core accelerator with fully device-based simulation achieves similar performance as the GPU-based acceleration.

Generally, a domain decomposition of an ABS targeting hardware accelerators is suited to assign different parts of the simulation to devices of the *same* type. Since different types of hardware device typically favour certain types of computations, when assigning computations to *different* types of hardware, functional decomposition is commonly applied instead. It is thus natural that most of the literature on hardware assignment of ABS to heterogeneous hardware has focused on functional decomposition.

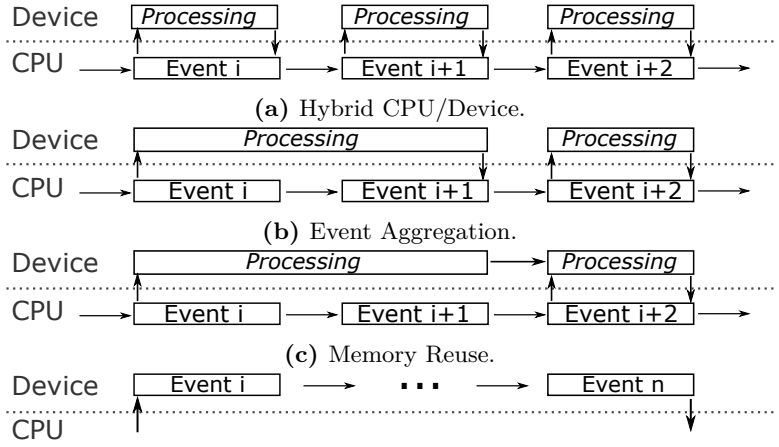
Functional decomposition: An efficient functional decomposition identifies static functional boundaries in the considered simulation to maximise the computational performance on each device, while minimising data transfers. Pavlov and Müller [122]

discuss approaches for the hardware assignment of ABS and conclude that an approach in which the CPU and the GPU hold duplicated or partial agent and environment data is the most promising. An overall development process for a GPU-accelerated ABS starting from a CPU-based implementation is proposed in [106, 124]. After the decomposition of the simulation into small task modules, modules suitable for execution on a GPU such as loops are identified heuristically and manually replaced with GPU-executable counterparts. Several case studies [106, 88, 89] show substantial speedup when employing this method.

Some works have focused on isolating simulation code that heavily relies on floating point arithmetic to be executed on a GPU. Bauer et al. [123] assign the discrete part of a combined continuous-discrete simulation to the CPU and the continuous part, which relies on floating point arithmetic, to the GPU. The authors conclude that while keeping the GPU fully utilised poses a challenge, models with large numbers of floating point operations can benefit from GPU acceleration. Andelfinger et al. [1] compare different GPU/CPU simulator architectures aiming to offload events associated with floating point operations to a GPU. In a basic CPU/GPU hybrid scheme (cf. Figure 2.4a), the CPU offloads each event to the GPU individually. The required data transfers can be reduced by aggregating independent events to execute them in a single step (cf. Figure 2.4b) and by leaving computation results required by subsequent events in graphics memory (cf. Figure 2.4c). Finally, if the entire simulation is ported to the GPU, data transfers are only required at the start of the simulation and once the simulation terminates (cf. Figure 2.4d). While the simulation performance increases with each of the above optimisations, the developer is burdened with some additional complexity.

Since floating point arithmetic is a natural fit to the GPU's capabilities, models heavily relying on such operations are likely to benefit from a functional decomposition focusing on this aspect. Examples of such models include kinematic equations as in microscopic traffic simulation or crowd simulation, as well as models of wireless communication and of biological or chemical processes.

In another approach to functional decomposition, agent behaviour is left to the CPU while environment dynamics are handled by the GPU [106]. With this approach, the author aims to reduce the impact of the GPU acceleration on the maintainability of the simulator code. To increase performance, portions of the agent behaviour that do not depend on the agent state, e.g., perception of the environment, are carried out on the GPU independently of individual agents for all locations.



(a) Hybrid CPU/Device.

(b) Event Aggregation.

(c) Memory Reuse.

(d) Fully Device-based Simulation.

Figure 2.4: Four CPU-device simulation schemes [1]. Devices can be GPUs or many-core CPUs.

In contrast to the above works, a number of approaches partition the simulation along functional boundaries specific to the given simulation models. For instance, when the underlying simulation can be clearly separated into model computation and management tasks, a master-worker scheduling approach can be applied, as shown by Bilel et al. [112] in the context of large-scale mobile networks simulation. In the proposed design, the model is executed on the GPU, while the CPU orchestrates the event scheduling, simulation status monitoring, and memory allocation. Nguyen et al. [121] propose a general approach to enable spiking neural network simulators on heterogeneous hardware. The core of the simulator's functionality called static parts, are ported manually to run on accelerators such as GPUs, while the dynamic parts, i.e. the neuron models, are transformed to OpenCL code automatically.

Finally, the nature of traffic simulation allows for a relatively straight-forward functional decomposition according to different simulation aspects. Xu et al. [75] and Song et al. [76] assign agent mobility in a mesoscopic traffic simulation to the GPU, whereas the route calculation, agent generation, and file reading and writing remain on the CPU. The two parts run asynchronously to avoid data transfer latencies. In the traffic simulation on an APU presented by Wang et al. [79], sorting of agent states is required to locate each agent's neighbours. To reduce synchronisation overheads on the GPU portion of the APU, the GPU portion only performs state updates and local sorting, whereas the sorting across GPU blocks is handled by the CPU resources. The work separation can be carried out efficiently using zero-copy memory access. Considering FPGAs, Tripp et al. [80] show how the movement of agents on individual lanes can be

computed on an FPGA, while the agents' transitions from one road to another as well as the behaviour at intersections are computed on the CPU.

In summary, most approaches relying on static hardware assignment split the simulation workload into coarse-grained functional tasks so that some tasks are clearly suited for a certain hardware device. To minimise trial-and-error, heuristics may be applied to identify a suitable mapping of tasks to the hardware. Tasks involving large numbers of parallel floating point operations are among the most common portions of simulations offloaded to accelerators.

2.4.1.2 Dynamic Assignment

While a wide range of literature has considered the problem of dynamically adapting a partitioning of agent-based simulations to multiple CPUs (e.g., [142, 143, 144]), we are not aware of such works that specifically target heterogeneous hardware environments. In the following, we outline recent works on dynamic assignment of general computational workloads to heterogeneous hardware. Since these works are generic, they cannot rely on knowledge of the general structure of ABS simulators or on model knowledge. Still, the proposed methods to determine suitable hardware platforms for given segments of code can be applied to ABS as well.

Belviranli et al. [126] propose a self-scheduling scheme for partitioning generic application workloads into blocks and assigning them to CPUs, GPUs, and FPGAs. The proposed system consists of two phases: in the first phase, the system performs an online training with a small amount of data to estimate the maximum workload capacity of each hardware device. Fast convergence is achieved by fitting four sampled data points to a logarithmic function. Once the capacity is determined, the processing units' performance can be inferred from the same data. When the change of processing speed between two samples drops below a threshold, it is used as the final estimated value. In the second phase, the remaining workload is partitioned based on the relative processing speeds of the available processing units, assigning a larger portion of the workload to faster processing units.

Some authors use machine learning techniques such as support vector machines, artificial neural networks, and decision trees to distribute the workload of OpenCL programs to CPUs and GPUs. For example, Grasso et al. [128, 129] and Zhang et al. [125] translate a single-device OpenCL program to a multiple-device program, while Wen et al. [127] focus on scheduling multiple OpenCL functions to run in parallel on CPU/GPU. They train a machine learning algorithm according to a set of typical OpenCL programs and benchmarks. The prediction generated by the machine learning

algorithm guides the assignment of a portion of the computation to CPU or GPU. Their results show that the above three machine learning approaches outperform purely CPU- or GPU-based approaches. The scheduling scheme by Wen et al. achieves a performance improvement compared to a first-come, first-served scheme and a scheme where computation-heavy tasks are handled by the GPU.

To automate the compilation of sequential programs for parallelised execution on heterogeneous hardware, Grosser and Hoefler [130] present a compiler that generates CPU and GPU code. Regions with mostly static control flow and sufficient computational intensity are detected and transformed to a formal representation to facilitate program transformations [145]. After optimisations have been performed to increase memory access locality and parallelism, CUDA code for GPUs is generated from the formal representation. A runtime library eliminates repeated memory allocations and unnecessary data transfers between CPU and GPU. The decision whether a region is compute-intensive enough for execution on the GPU is made either statically or at runtime, using heuristics based on metrics such as the number of instructions. The authors conclude that the compiler is able to translate CPU code into cross-platform code with no performance penalty. For some computations, such as the correlation benchmark from polybench [146], significant speedup of up to two orders of magnitude can be achieved.

The main difficulty in automated hardware mapping lies in determining the control flow and data dependencies of the original program. Current approaches either rely on the program code being formulated in languages such as OpenCL that express independent control flows explicitly, or only consider specific portions of programs such as loops with largely static control flow. In ABS, however, most of the available parallelism may exist across the update routines of separate agents. Thus, without semantic information describing the code structure, automatic detection of the parallelism is challenging. In Section 2.5, we sketch how the common structure of many ABS may be utilised to support the extraction of parallelism.

2.4.2 Minimisation of Data Transfer Overheads

Since most hardware accelerators are equipped with their own memory, simulations making use of accelerators typically require data transfers between host and accelerator memory. Even with a high-quality domain decomposition or functional decomposition of the simulation, agent mobility and communication or the data dependencies between the different functional aspects make data transfers unavoidable. In this section, we

survey works that focus on minimising the cost of such data transfers. The existing approaches can be categorised according to the following techniques:

1. **Overlapping of communication and computation:** some authors proposed techniques to hide communication overhead by transferring data while independent computations are performed. This technique has sometimes been referred to as *latency hiding* (e.g., [147]).

2. **Computation replication at partition boundaries:** another technique to address communication overhead is to increase the amount of computation performed before synchronisation among processing elements is required. This is achieved by duplicating some computations on multiple processing elements, thus delaying the need to resolve data dependencies across processing elements.

2.4.2.1 Overlapping of Communication and Computation

One way of mitigating the overhead from data transfers between the host and an accelerator is to execute computations at the same time as data is being transferred. In the approach described by Kunz et al. [111], event computations are overlapped with data transfers across the CPU-GPU boundary, thus hiding data transfer latencies in a pipelined fashion. Since events from multiple simulation instances are considered concurrently, there are substantial opportunities for overlapping these steps. While their approach is applied to a discrete-event simulation, it can be applied to time-stepped ABS by initiating the transfer of output data of some agents' state updates at a given time step, while computations for other agents are still in progress.

Bauer et al. [131, 107] propose a generic API to optimise the data transfer between global memory and shared memory of CUDA GPUs using so-called warp specialisation. The warps within one cooperative thread array are split into two groups: *dedicated memory warps* are in charge of data transfer between the on-chip and off-chip memory, while *compute warps* process the data. The approach improves performance over thread-level separation between communication and computation since separate warps can follow divergent control flows without any performance penalty. While their general idea can be applied to other types of independent processing elements, the warp-based implementation is specific to GPUs.

2.4.2.2 Computation Replication at Partition Boundaries

In time-stepped ABS, at model time t each agent updates its state based on the states of its neighbours and itself at time $t-1$. If the simulation is distributed across multiple

processing elements, synchronisation and data transfers are required to provide this information at each time step. The associated overhead may make up a substantial portion of the total simulation runtime. Thus, some authors have proposed methods to reduce synchronisation by replicating some computations on multiple processing elements, similarly to performance optimisations in numerical computing [148]. The main challenge when applying this approach to ABS is the consideration of the model-specific sensing range of agents and the speed according to which the effect of an agent's actions can propagate throughout the simulation space.

Aaby et al. [83] present a multi-level data partitioning scheme for cellular simulations on multi-CPU/GPU clusters. The simulation state is partitioned into blocks and each block is executed by a thread, a core, or a node, depending on the configured granularity. In contrast to the traditional data partitioning into blocks of $B \times B$ cells and synchronisation at each time step, their approach partitions the data into several overlapping $(B + 2R) \times (B + 2R)$ blocks where $((B + 2R)^2 - B^2)$ cells form the overlapping area. The computation in the overlapping area is performed redundantly by multiple processing units. Thus, assuming that at each time step, a cell can only affect its immediate neighbours, R time steps are required for a cell in the inner block to be affected by cells in another processing element. Therefore, synchronisation is only required every R time steps. Between synchronisation points, an error propagates inwards within the overlapping areas, but does not affect the inner $B \times B$ cells before a new synchronisation occurs. This partitioning approach is further employed in multi-GPU clusters on the node-, GPU-, block-, and thread-level, and for multi-CPU clusters at the node-, socket-, core-, and thread-level.

While Aaby et al. illustrate the idea based on cellular grids, the approach applies to general ABS. The sensing range of agents is generally limited and provides an upper bound on the propagation of the effects of an agent's actions within a time step. As long as overlapping segments of the simulation space can be distributed to the processing elements in a manner so that an effect requires at least $R > 1$ time steps, some synchronisation can be avoided. The generality of the approach is illustrated by Zou et al. [86], who extend the idea of computation replication to graph-based topologies in a GPU-accelerated epidemic ABS.

2.4.3 Scattered Memory Accesses

Throughout the past decades, the increase in computational performance has outpaced the decrease in memory access latencies, leading to modern hardware designs tending towards large caches and deep memory hierarchies. In the context of ABS,

the issue of memory access latencies is particularly pressing. Due to the autonomous decision-making of agents, the runtime interactions between agents and their environment cannot easily be predicted before executing the simulation, significantly limiting the opportunities for a priori optimisation of data access patterns. However, commonalities between different simulation models can be exploited to propose data structures supporting efficient simulation of an entire range of models on a specific type of accelerator.

Since dynamic memory allocation on GPUs is costly [149], most GPU-based simulators allocate memory statically for data such as the agent states (e.g., [92]). Another approach is to determine after each time step the required amount of memory and perform allocations accordingly [114].

We categorise the existing approaches to address scattered memory accesses as follows:

1. **Manual caching in shared memory:** although the support for transparent caching has improved in recent years, achieving highest performance frequently still requires manual caching in low-latency memory. In ABS, agents often influence and are influenced by their direct neighbours. This fact can be exploited when arranging the simulation data in memory, reducing high-latency memory accesses during state updates.

2. **Heuristics for agent update order:** since the data dependencies between agent state updates are typically not known prior to the execution of the simulation, minimising cache misses during the state updates is non-trivial. Heuristics have been proposed, which favour sequences of computations acting on the same agent data.

3. **Representation of irregular data structures by arrays and grids:** the hardware architecture of GPUs and FPGAs is designed so that highest performance is achieved when acting on regular data structures such as arrays and grids. Thus, efforts are taken to represent highly irregular data structures in a regular fashion. When covering the techniques from the literature, we first cover *model-specific data structures* such as graph representations of a simulated road network. Subsequently, we discuss works covering two generic building blocks commonly required as part of ABS engines: *priority queues and sorting*.

2.4.3.1 Manual Caching in Shared Memory

Richmond et al. [85] propose utilising the shared memory of the GPU as a manual cache. In their agent-based simulation framework for cellular models in biology based on FLAME GPU [14], they copy sets of messages to be transferred between agents into shared memory. Each thread within a block can then efficiently iterate through the messages and identify those pertaining to the local agent. Once all threads have iterated through the messages, the next sets of messages are loaded into shared memory. Recently, Heywood et al. [150] specialise their messaging method for traffic simulations on graph-based road networks. Messages are sorted by edge (road segment) or vertex (intersection) so that each agent only considers messages that pertain to its immediate neighbourhood in the road network.

Similarly, Zou et al. [86] implement a manual software cache in shared memory to increase the performance of their graph-based epidemic simulation on GPU clusters. Before the simulation commences on the GPU, the CPU sorts the edges of the directed graph by the source vertex. Each thread block's shared memory stores edges originating from one specific node. Since each block processes only edges originating from this node, a cache hit rate of at least 50% is ensured.

In the GPU-based ABS by Li et al. [92], assuming a constant number of agents, each agent is assigned to a GPU thread and its state data is permanently kept in global memory. The simulation space is partitioned into a grid of rectangles. When an agent needs to search for its immediate neighbours, a search rectangle that encloses the searching circle is created, so that only agents inside the search rectangle have to be considered. Two approaches to utilise the GPU's shared memory are proposed: in the first approach, one block manages the searching process for a chunk C of close-by agents. Per-block shared-memory loads the data of the agent and the agent's neighbours. Each agent in C has a high probability of being in the other agents' neighbourhoods, so that these agents can frequently be accessed through the current block's low-latency shared memory. However, since the limited shared memory capacity allows only for small numbers of agents to be stored, it is still likely that some neighbours are managed by another block and thus have to be accessed through global memory. In the second approach, the shared memory loads the data of all agents located in the union of search rectangles of the agents handled by the current block. If the shared memory is not sufficient to hold all agents' data, the data is loaded as a sequence of chunks. Of course, the increase in the search space given by the union of search rectangles leads to a higher number of unnecessary agent accesses through shared memory. To address

this problem, the union rectangle can be constructed on the warp level instead of the block level.

2.4.3.2 Heuristics for Agent Update Order

The order in which agent updates are performed must adhere to the causal dependencies between the agent states and behaviours, e.g., in road traffic simulation, vehicles in direct proximity must be at the same point in simulated time to be able to interact according to the model specification.

Typically, this is achieved by a strictly time-stepped scheme in which agents always reside at the same time step, after which conflicts in the resulting agent states are resolved [151]. However, since in a typical simulation not all agents interact at each point in time, some agents may be updated further into the simulated future than others without affecting the simulation results [78]. Harris and Scheutz have shown that distributed agent-based simulations can be accelerated by favouring agent updates that resolve dependencies across multiple processing elements [132]. This way, processing elements waiting for others to proceed can be unblocked, decreasing the amount of idle time. Their approach can be applied independently of the underlying hardware platform, but requires bounds on the sensing range and the agent movement per time step.

Jin et al. [97] present an information propagation simulation supporting execution on HPC systems and single GPUs and extend it to run on multiple GPUs [98]. Their focus lies on maximising the cache hit rate when traversing a graph according to rules defined by the simulation models. Two categories of approaches are developed for the cascade model [152] and the threshold model [153], which both simulate the propagation of information among nodes in a graph: vertex-oriented processing and edge-oriented processing. For the vertex-oriented approach, the authors further describe two agent update orders: one iterates starting from active vertices, i.e., those that already have the information, and the other from inactive vertices. Since the costs depend on the portion of active nodes, the simulation can switch dynamically between the two vertex-oriented approaches. Finally, the edge-oriented approach iterates over the connecting edges between two vertices. Since the number of edges is constant over a simulation run, the cost of the edge-oriented approach is less variable than that of the vertex-oriented approaches. The authors achieved the highest performance when dynamically switching between the two vertex-oriented approaches.

2.4.3.3 Representation of Irregular Data Structures by Arrays and Grids

GPUs and FPGAs are particularly suited for operations on regularly structured data. However, many model types specify topologies that are more naturally expressed in terms of irregular structures such as graphs. Further, execution of the simulator core itself may require operations on irregular data structures.

A basic optimisation commonly applied in works on GPU-based computing to improve memory access patterns is the transformation of the data layout in memory from arrays of structures (AoS) to structures of arrays (SoA) (e.g., [85, 72]). Commonly, sequential programs store data in an AoS representation. Since AoS bundles the properties associated with each object in object-oriented programming, or the states of agents in agent-based simulations, it is a natural way to represent data within these paradigms. However, with an AoS data layout, parallel operations on the same property across many objects results in scattered memory accesses. An SoA data layout bundles the same property across all objects, which can increase cache hits rates and opportunities for memory access coalescing, thus improving performance substantially.

Beyond this simple optimisation, the data representation can be specialised for a given model to further improve performance. In the following, we give an overview of methods applicable to ABS to achieve high performance by translating irregular data structures to a more regular form.

Model-specific data structures

Early works on executing ABS using GPUs frequently focused on cellular grids and translated the required computations into the graphics processing domain. In a pioneering work by Harris et al. [109], GPU shaders are used for implementing computations on the RGBA values in a texture that holds the agents' states. The same idea is employed by Lysenko et al. [133], Perumalla and Aaby [84], and Kolb et al. [104].

Perumalla et al. [84] evaluate the performance of running agent-based simulation entirely on a GPU. They ported the cellular models Mood Diffusion [154, 155], Game of Life [156] and Schelling Segregation [157]. Through the Open Graphics Library (OpenGL), individual agent states are mapped to pixel colour values. The authors report a speedup of 15 to 40 compared to CPU-based sequential execution. Kolb et al. [104] develop a particle simulation and a GPU-based collision detection mechanism built on the authors' previous work [158]. Similarly, Richmond et al. [81] utilise the GPU's texture processing ability and map agent states onto texture data. To accelerate neighbourhood detection, the simulation space is partitioned dynamically according to the agents' current states. The algorithm to generate partitions is borrowed from the particle pinning problem in rigid body particles physics [159, 160]. Identification of the

start and end of the partition boundary is performed similarly to the method described in [161]. Textures are used to represent the agents' states and vertex texture fetching enables the search for the start and end of the partition boundary by comparing the partition value to the previous agent's state.

To enable traffic simulations on GPUs, Perumalla [70] (and Perumalla et Aaby [71]) proposes to model the road network as a grid made up of cells. A road network in Cartesian coordinates is translated to a grid representation overlaying the network: a cell in the grid is marked as occupied when an edge of the original road network starts in the cell, passes the cell, or ends in the cell. In graphics memory, the cells' properties such as turning probabilities and length are stored in texture buffers. Simulation is carried out by performing operations on the texture buffers.

A different method for traffic simulation on GPUs is presented by Strippgen and Nagel [72], who propose a queue-based approach using CUDA. Each road is represented as a single first-in, first-out (FIFO) queue stored in memory in the form of a ring buffer. With the ring buffer, insertion of a vehicle entering a road and removal of a vehicle exiting a road is achieved with constant time complexity. Coalesced memory access can be achieved by processing adjacent roads using adjacent threads. Since the vehicles' mobility is modelled by a fixed per-link velocity, their approach can be considered mesoscopic. The representation of lanes as ring buffers relies on changes of the relative position of vehicles being rare. Behaviours such as overtaking or lane-changing are not modelled and would require random insertions and removals from the ring buffers, which are associated with linear time complexity.

Other domains in which agent-based simulations have been successfully ported to GPUs using model-specific data structures include collision detection [100] and a simulation study of tuberculosis [82]. In the former, a grid is split into tiles and data at the boundary of the tiles is replicated so that a consecutive space is occupied in the global memory of the GPU. In the latter, the authors propose to use a sorted array according to the liveness status of agents, so that the state of a new agent can be stored in a memory location previously occupied by one of the dead agents.

Sorting and priority queues

Full or partial sorting is frequently required in agent-based simulations, e.g., for neighbourhood discovery or to implement Priority Queues (PQs) if time advancement is performed in a discrete-event manner. These operations can involve large amounts of data-dependent and scattered memory access and are therefore challenging to implement efficiently on hardware accelerators. Since this operation can occupy a substantial

portion of the simulation runtime [162], a number of works have focused on memory layouts and algorithms for sorting and priority queues on accelerators.

As building blocks for time advancement in a discrete-event fashion, parallel reduction and bitonic sorting are commonly used in GPU- and FPGA-based simulation [134, 79, 99, 105, 90]. We discuss these two operations jointly due to their structural similarities. In both cases, an input array is split into chunks, each chunk being handled by one thread. At each cycle, the sorted arrays/minimum values of two threads are then merged to form a new input array. Thus, at each cycle, the number of chunks and active threads is cut into half. The algorithm iterates until only one thread is active, leaving a sorted array or the global minimum value, respectively.

He et al. [163] propose a parallel heap-based PQ on GPU based on a previous CPU-based design [164]. The data structure resembles a binary min-heap, but stores r items per heap node. Items are inserted and extracted in a joint bulk operation that inserts up to $k \leq 2r$ and extracts up to r elements. At any time the root node is guaranteed to hold the highest-priority elements, while elements of lower priority are gradually inserted into deeper levels of the tree over the course of multiple insert-extract operations. Parallelism can be exploited across the sorting operations on the items within a tree node, across the nodes on one level of the tree, and by processing all even-numbered and odd-numbered levels of the tree in parallel. The costs of the queue operations can be hidden by performing them in parallel with the processing of extracted items.

Similarly, the FPGA-based DES simulator by Rahman et al. [57] relies on a pipelined heap [165] for storing events. In contrast to the parallel heap by He et al., the pipelined heap is designed to achieve near-constant access times, but does not provide bulk operations.

A number of works avoid the need for a global PQ holding all future events. Instead, the set of events is considered jointly in an unsorted fashion [135], split by model segment [105] or simulated entity [99, 116, 136], split according to a fixed policy [114, 137], or split randomly [110]. To determine the events that can be executed without violating the simulation correctness, a parallel reduction is performed to determine the minimum timestamp among the events.

Baudis et al. [138] evaluate the performance of PQs on a GPU implemented as a single parallel heap or as a set of ring buffers, implicit binary heaps, and splay trees [166] in the context of DES and path finding on grids. Their results indicate that for up to about 500 elements per PQ, ring buffers achieve the highest performance. At larger element counts, implicit heaps outperform the other approaches in their study. Their

results suggest that higher performance is achieved by relying on multiple PQs, one for each agent or set of agents, compared to a single PQ holding all events.

2.4.4 Maximisation of Parallelism

The autonomous decision-making and mobility of agents can limit the exploitable parallelism in a simulation in two ways. First, variations in the computational intensity among the model segments may leave some processing elements idle. Second, the single-instruction multiple-thread execution model of GPUs requires divergent operations within a warp to be serialised.

The existing techniques to maximise the parallelism of ABS using accelerators can be roughly categorised as follows:

1. **Multiple replications in parallel:** full utilisation of a massively parallel accelerator requires large numbers of computations that are independent and can thus be executed in parallel. If a simulation involves a sequence of mostly dependent computations, the overheads for communication may outweigh the gains from parallelisation. Thus, techniques have been proposed to perform computations from multiple simulation runs in parallel.

2. **Window-based event execution:** in simulations involving a discrete-event mechanism, only a proper subset of the simulated entities may require an update at a certain point in simulation time. Multiple authors have proposed gathering events across a window in simulated time, and executing these events in parallel. In effect, this approach forces a discrete-event approach into a time-stepped execution. A key difference among the window-based techniques lies in whether the simulation correctness is strictly maintained.

3. **Speculative execution:** as in general optimistic parallel and distributed simulation [27], computations may be performed speculatively to improve hardware utilisation. A rollback mechanism is required to revert to a correct simulation state after erroneous computations.

4. **Computation sorting:** on a GPU, threads within a warp following divergent branches of the control flow are serialised. Since each agent performs actions based on its attributes, its current state as well as its environment and neighbouring agents, the computations at each simulation step are often diverse across agents. Some authors have proposed sorting of computations to minimise the serialisation resulting from branch divergence.

2.4.4.1 Multiple Replications in Parallel

If an individual simulation run does not provide sufficient parallelism to fully utilise the available hardware, a Multiple Replications in Parallel (MRIP) approach [122] can be applied, as shown by Shen et al. [73]: in their approach, multiple replications of a traffic simulation [74] are executed in parallel on a GPU. Thus, both parallelism among agents and the parallelism across replications can be exploited. Laville et al. [94] implement a multi-agent simulation of microorganisms in soil for CPU/GPU in OpenCL. Each GPU thread manages one agent and each block is responsible for one simulation instance so that multiple simulation instances can run concurrently on one graphics card. This idea is applied to discrete-event simulations by Kunz et al. [111], focusing on executing parameter studies comprised of multiple replications on a GPU.

In addition to exploiting the parallelism across replications, Li et al. [101] aim to avoid unnecessary redundant computations common to multiple replications. They propose a cloning mechanism for ABS on the GPU: in an ensemble simulation run comprised of multiple simulation instances, the computations that are common to multiple instances are only performed once. When the behaviour of an agent diverges between two simulation instances, a clone of the agent is created. Since the agent may affect other agents, cloning is performed according to the propagation of the effects of the original change in agent behaviour. Similarly to the technique "computation replication at partition boundaries" (cf. Section 2.4.2.2), cloning exploits the limited propagation speed of agent updates due to the limited sensing ranges and movement speeds of agents. If agents move and communicate arbitrarily across the entire simulation space, the required number of clones is too large to achieve a performance benefit. Across cloned simulation instances, neighbour detection can be aggregated to improve the utilisation of the GPU resources. The benefit of cloning is limited when simulation runs diverge strongly, e.g., across multiple runs of a stochastic simulation using different seeds for random number generation. Recently, the cloning approach has been applied to large-scale cellular simulations on GPU clusters [95].

2.4.4.2 Window-based Event Execution

On a GPU, all threads in a warp execute the same sequence of instructions on different elements of data. If no input data is available for some of the threads within a warp, hardware utilisation is reduced. In ABS, this issue is particularly obvious when time advancement is performed in a discrete-event fashion to accommodate varying state update intervals among the agents. Then, the probability that many events share the

same timestamp may be low. Thus, a simple parallelisation across the events at a certain point in model time may be insufficient. An approach to address this problem is to execute DES models in a time-stepped fashion: all events within a certain time interval are executed in parallel. The lower bound of this time interval is usually referred to as Lower Bound on Time Stamp (LBTS), which is similar to Global Virtual Time in optimistically synchronised parallel and distributed simulation [167]. With a sufficiently large time step size, hardware utilisation is increased. However, since dependencies between events are not considered, the simulation results may differ from a sequential execution.

A study comparing the performance of time advancement mechanisms for simulations on the CPU and the GPU is presented by Perumalla [108]. They study diffusion simulations running in a time-stepped, discrete-event, and hybrid fashion. The GPU variant is implemented in the GPU programming language Brook [139]. While the GPU outperforms the CPU in the time-stepped variant, it does not perform as well as the discrete-event implementation on the CPU. However, high speedup is achieved using the hybrid approach, where at each cycle, the minimum gap between two events is used as a time step. The simulation time then advances according to this time step.

Park and Fishwick [113, 114] present a method for queuing network simulation that executes a DES model in a time-stepped fashion. The simulation time advances according to a fixed time step size, but skips periods where no events occur. All events within the current time step are executed without considering potential dependencies. Although the results are affected by their approach, the authors show that for a queueing network simulation, error bounds can be given. Other works assume a minimum time delta between an event and its creation (*lookahead*) to guarantee the correctness of the simulation results [115, 116, 99]. If lookahead is available, a window can be determined within which events are independent, allowing for parallel execution without affecting the results.

The current time window is extended dynamically in work by Tang and Yao [87] to allow more events to be executed in parallel. After executing all events within the current window, their algorithm evaluates the first event in the event queue with a timestamp larger than the LBTS that can still safely be executed according to the lookahead.

2.4.4.3 Speculative Execution

To maintain the correctness of the simulation results when executing in parallel on an accelerator, the simulator must consider dependencies between state updates. In some

of the approaches described above, a time window is determined where state updates cannot affect each other. If it is difficult to determine a time window of sufficient size to extract substantial parallelism, a speculative (or optimistic) approach can be employed: state updates are performed without regard for correctness, and rolled back if errors are detected.

Speculative execution of simulations on FPGAs has been first demonstrated by Model and Herbordt [110]. They make use of predictions of the interaction between particles, generating new events accordingly. Events may later be cancelled as a consequence of a false prediction.

Targeting GPUs, Li et al. [102] present an execution model that achieves high parallelism by speculative event execution. In an initial step, all events that may occur in the simulation are created. Subsequently, all events are executed in parallel. A scanning process detects and revokes causally invalid event executions: if an event leaves the simulation in an incorrect state according to a model-specific criterion, the erroneous event and all events created by it are revoked recursively.

A more general approach for GPU-based discrete-event simulation is presented by Liu and Andelfinger [136]. An optimistic execution scheme based on the Time Warp algorithm [167] implemented in CUDA is shown to be beneficial at low event density in simulated time. To support rollbacks in case of erroneous computations, the authors show how the default random number generator in CUDA can be reversed computationally without storing additional data.

2.4.4.4 Computation Sorting

The individual decision-making of the autonomous and heterogeneous agents often leads to diverse computations being executed at the same simulation step. Some approaches attempt to arrange the assignment of computations to the available threads on a GPU so that the serialisation caused by branch divergence within a warp is minimised.

In their DES engine on the GPU, Tang and Yao [87] sort events by type before execution, i.e., by the code associated with the event.

The idea is applied to GPU-based execution of multiple simulation instances at the same time by Kunz et al. [111] (cf. Section 2.4.4.1). If the simulation instances do not diverge too strongly, many events of the same type are available across multiple instances, enabling efficient parallel execution.

Kofler et al. apply computation sorting to their ABS of mosquitoes [90]. In their simulator, a one-to-one mapping between agents and threads is used. Depending on their current state, agents may perform different operations, which can result in taking

different control flow branches during the state updates. Thus, to reduce divergence among threads within a warp, agents are sorted by their current state, so that the state updates of adjacent agents share the same control flow.

In a recent work, Chimeh et al. [168] provide guidelines on formulating models to be executed in FLAME GPU so that branch divergence is minimised. They suggest modifying the state machines defining the agent behaviour to eliminate conditional branches by creating a new state or even a new agent type for each branch. The state updates of agents currently in the same state can then be executed without divergence.

2.4.5 Abstraction from Hardware Specifics

Compared to model development in CPU-based environments, development for accelerators can be cumbersome and error-prone. To avoid the need for modellers to gain deep expertise in programming for specific accelerators, several frameworks have been proposed that enable the specification of parts of the model structure and behaviour in a hardware-agnostic fashion. Since ABS models are commonly developed, modified and extended in an iterative process, it is critical to avoid the need for modellers to consider low-level aspects of accelerators. The following works address the abstraction from hardware specifics:

1. **Frameworks to support simulation development:** some authors have proposed generating partial model code to be executed on accelerators from domain-specific languages or the reliance on a library of pre-defined implementations of common simulation tasks and models. However, in these approaches, developing a full ABS will typically still require manual implementation work using comparatively low-level languages such as CUDA. Further, workload partitioning and assignment to different hardware devices is currently not considered by these approaches.

2. **Unified memory access:** since in most cases, the CPU and hardware accelerators involved in a simulation operate on separate memory, resolving data dependencies may involve cumbersome explicit data transfers. A number of authors have proposed techniques to transparently access data in programs executed on heterogeneous hardware.

2.4.5.1 Frameworks to Support Simulation Development

In the Flexible Large Scale Agent Modelling Environment (FLAME GPU) [85, 91], agent states are specified using the state machine model X-Machine [169, 170]. Mod-

ellers define agent states in an XML-based format, while state transitions, i.e., the code segments describing the state updates, have to be manually specified as CUDA code. Generic facilities for exchanging messages between agents are provided by the framework. Traffic simulation has been presented as one of the use cases of FLAME GPU [77].

The domain-specific language OpenABL [18] enables the specification of ABS models in a C-like language in compact and platform-independent fashion. The OpenABL code is translated to an intermediate representation, from which code is generated targeting different backends such as CPU, GPU or cloud.

Another framework for GPUs and other many-core architectures is called Many-Core Multi-Agent System (MCMAS) [15]. MCMAS provides a high-level Java interface to OpenCL code as well as a set of pre-defined data structures and functions called plugins. To implement agent models, users either rely on plugins or define their own plugins as OpenCL code that can be called from Java code. The authors state that unlike FLAME GPU, in which models are targeted exclusively at the framework, the models defined in MCMAS can be reused by other agent-based simulators.

While FLAME and MCMAS both reduce the implementation work required to develop agent-based simulations targeting accelerators, these frameworks do not provide guidance or automation in distributing the simulation workload to the available hardware. Thus, manual experimentation is required to determine a suitable hardware mapping.

2.4.5.2 Unified Memory Access

GPGPU frameworks such as OpenCL or CUDA require the user to either explicitly trigger data transfers between host and device memory, to explicitly select certain variables or memory regions for access from both CPU and GPU code [40], or to annotate the program to manage data transfers [117, 118]. These manual steps complicate the development of agent-based simulations in heterogeneous environments. Some works aim to improve on this situation by transparently transferring required data between host and graphics memory. However, in languages based on C or C++, static alias analysis, i.e., determining which pointers refer to the same memory regions, is known to be undecidable [119].

Jablin et al. [119, 120] present the first fully automated data management system based on compilation steps and a runtime library. The developer formulates his program and GPU code as if all data resides in host memory and can be accessed both from the CPU and GPU. The proposed approach instruments the code to track accesses to

different memory regions using code instrumentation and trapping of system calls. To avoid the need for static pointer analysis, memory accesses through pointers are tracked by the runtime library. In addition to transparently handling data transfers, CPU-GPU communication is optimised during compile time by re-ordering the program flow to reduce the alternation between computations and data transfers. Unnecessary data transfers are avoided by leaving data in the GPU memory until it is accessed from the host.

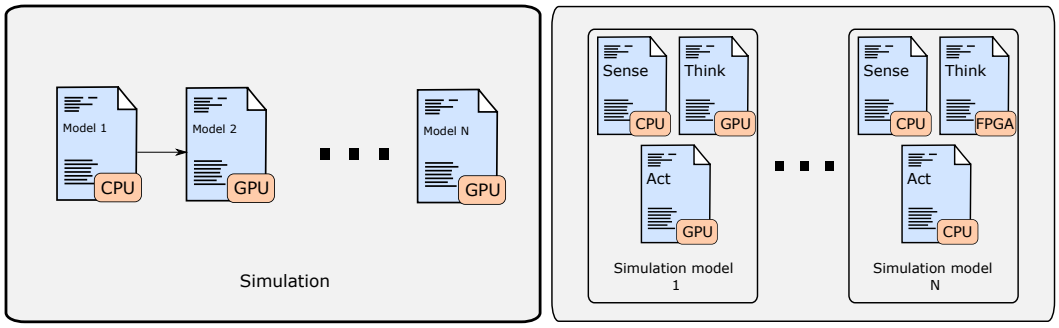
While the work of Jablin et al. could be applied to automate data transfers in heterogeneous ABS, the detection of parallelism is not covered. In Section 2.5, we sketch research directions towards automation in porting ABS to accelerators.

2.5 Towards an automated parallelisation framework for ABS

From the observations in the previous sections, we can see that there is a vast range of techniques covering the main challenges of high-performance ABS on hardware accelerators. However, there exist only a few ABS frameworks that support such accelerators. Since existing agent-based simulation and model implementations typically target purely CPU-based environments, there is a clear need for processes and tools to support the transition to an execution on accelerators. More specifically, modellers and simulationists should be supported in the parallelisation and hardware mapping as much as possible. While methodologies have been proposed to systematise the steps of porting a simulation to a GPU [106, 89] and more for general computing [130, 172, 173, 174], there are few automated tools to support this process in the field of ABS.

Out of all the techniques reviewed, OpenABL seems to be one of the best candidates to support the adoption of ABSs on heterogeneous hardware. However, there are limitations in the current form of OpenABL: Firstly, the outputted code can only target one specific type of hardware platform. A combined used of e.g. CPUs and GPUs is not possible. FPGA, a promising hardware platform as can be seen from the literature review, is not yet supported. Secondly, so far OpenABL only allows simulation environment declared as a 2D or 3D space, limiting its usage in the domain of e.g. traffic where the simulation environment is usually represented by a graph. An extension of OpenABL is necessary, to addresses the aforementioned limitations.

To facilitate all hardware, the simulation needs to be first partitioned so that different parts can be executed on available hardware. A design question also arises: which parallelisation scheme should the envisioned tool employ? Current OpenABL requires



(a) Model-level **(b)** Sense-Think-Act-level

Figure 2.5: Model-level vs Sense-Think-Act-level parallelisation schemes.

the simulation to be partitioned (cf. Figure 2.5a) at a model level (the smallest parallelisable unit is then one simulation model). Each agent-based model represented by a function call is offloaded to an accelerator.

Since each model is structured as a Sense-Think-Act cycle, the individual stages of this cycle can also be offloaded to an accelerator (cf. Figure 2.5b). However, such partitioning can bring both positive and negative impacts. On one hand, breaking down models into smaller pieces would result in more potentially independent components that can be executed simultaneously (e.g. the Think stage often involves only read operations and therefore multiple Think stages from different agent-based model can very likely be executed in parallel). On the other hand, more components may also lead to more synchronisation efforts, counteracting the gains from parallel execution.

To answer the design question as well as to identify whether there are other unforeseen challenges, two feasibility studies are conducted in the next chapter. In the first feasibility study, a system equipped with a multi-core CPU and GPU or an APU is employed to compare the two parallelisation schemes shown in Figure 2.5. In the second study, we try to extend the knowledge gained on the GPU platform to draft a design for ABSs to run on FPGAs. To the best of our knowledge, automated generation of high-performance ABS code on FPGAs has never been addressed in the previous works. Therefore, through this study, we also intend to learn what extra facilitates are required to enable ABSs on FPGAs. Through these two studies, we can also get a taste of the oracle gains by employing hardware accelerators as well as showcase the amount of manual efforts required without any help from existing tools.

3 Feasibility Studies: Accelerating Agent-based Simulations on Heterogeneous Hardware

Contents

Substantial parts of this chapter have been published in the proceedings of the 2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT 18") [171] and the proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS 20") [175].

3.1 Introduction

This chapter constitutes of two feasibility studies. In both studies, an ABTS named City Mobility Simulator (CityMoS), originally designed for CPU-based systems, is manually ported to run on different accelerators. As mentioned in Chapter 2, the goal of the studies is twofold. Firstly, we try to answer the design question by evaluating different simulation parallelisation schemes presented in Section 2.5. The second goal is to demonstrate the amount of programming efforts needed and the performance gain achieved.

The first feasibility study mainly focuses on the evaluation of parallelisation schemes. Starting from pure CPU-based execution, we accelerate using CPU-GPU systems (dedicated CPU-GPU or APU). Two parallelisation schemes described in Section 2.5 are applied, yielding two simulation variants. In the first variant, each simulation model is decomposed according to the Sense-Think-Act cycle. We will consider offloading parts of the Sense, Think or Act stages onto the GPUs (therefore called partial offloading). In the second variant, we offload the entire model to run on the GPUs (full offloading). Their respective performance will be evaluated.

The second study focuses on porting CityMoS to run on FPGAs using high-level synthesis. The aim of the study is to learn what are the essential steps to enable a typical ABS to run on FPGAs. Then, we try to generalise the methodology to all types of ABSs.

As OpenABL currently supports neither ABTSs, of which the simulation space is represented by a graph, nor FPGA as a target platform, the knowledge built up in both studies can also vastly help us identify what are currently lacking in the framework.

The reminder of this chapter is organised as follows: In Section 3.2, we give an overview of related work. In Section 3.3, the set of models that constitute CityMoS are introduced. In Section 3.4, we demonstrate the first feasibility study where we accelerate CityMoS using CPU and GPU systems. In Section 3.5, the second feasibility study is provided to port CityMoS onto FPGA platforms. Section 3.6 concludes this chapter.

3.2 Related Work

Considering the existing works on ABTS using many-core devices, two general approaches can be differentiated: offloading approaches using a host CPU and a many-core device, and purely GPU-based approaches.

While offloading approaches have been widely explored in the context of discrete-event simulation [176, 177], most existing works on traffic simulation using many-core devices have focused on purely GPU-based execution. In purely GPU-based approaches, the entire simulation is executed on the GPU so that significant communication with the host CPU is only required at the start and end of the simulation. However, this approach makes it necessary to adapt the data structures and the control flow to the hardware properties of the GPU. Further, debugging and extending the simulator may require expert knowledge in parallel computing and the consideration of hardware-specific details.

Perumalla [178] proposes to map a graph-based network onto a grid in a GPU-based traffic simulation. Since this representation is highly suited to a GPU architecture, the approach enables simulations at the scale of road networks covering entire states of the USA. The considered model is field-based, i.e., vehicles probabilistically move in a certain direction at each cell in the grid, each cell storing the number of agents currently residing at the cell.

In the approach proposed by Strippgen and Nagel [72], each road is represented by a first-in, first-out queue stored as a ring buffer, one GPU thread processing one road. Since a vehicle's mobility to the end of each road is determined directly from the speed limit and road length, their simulation can be considered mesoscopic instead of microscopic.

Hirabayashi et al. [69] compare two approaches to purely GPU-based ABTS on a single-lane road based on the Optimal Velocity model [179]: as in most other works, in the first approach, the CPU calls GPU kernels to execute the agent updates at each step in model time. In the second approach, the entire simulation is performed within a single kernel call. Since synchronisation across thread blocks is not supported within a kernel, the authors quantify the error incurred by the lack of synchronisation.

Wang et al. [79] execute road traffic simulations of an infinite-length two-lane road on a dedicated GPU or the GPU portion of an APU. The main focus of their work is an efficient neighbour discovery algorithm on an APU. While all main parts of the simulation run on the GPU, a merging step required when agents enter a lane can be performed on the CPU. As in our work, the Intelligent Driver Model [180] is employed to simulate car-following. For lane-changing, authors rely on the Minimizing Overall Braking Induced by Lane change (MOBIL) model [181]. Of the existing works, this is the closest to our present work, since it shares our intention to compare execution approaches on heterogeneous hardware and employs common driver behaviour models. The main difference to our work is the reliance on a single road instead of a road

network. Thus, Wang et al. rely on bulk GPU operations on two large arrays holding all vehicles, whereas our simulation using a graph-based road network requires fine-grained operations on hundreds of thousands of small arrays representing one lane each.

Heywood et al. [182] use their FLAME GPU framework to execute a traffic simulation running entirely on a GPU. Vehicles accelerate according to Gipps' car-following model [183] on grid road networks. Neighbouring agents are not stored in a joint data structure associated with each lane but communicate each update in position and velocity using a messaging system. In contrast to our work, their focus lies more on different messaging systems than on exploring the possibilities for offloading to many-core devices.

Finally, an approach to accelerate parameter studies is proposed by Shen et al. [73]. They consider simulations of a small road network of six intersections, executing multiple replications of the traffic simulation in parallel on a GPU. Their work relies on the GM car-following model [184], but disregards lane-changing. The focus is on exploring the possibility to execute large numbers of simulations in parallel.

While many options for GPU-based simulations have been investigated by the existing works, we are not aware of any previous work evaluating offloading opportunities under the constraint of maintaining a graph-based road network representation and well-known models for car-following and lane-changing as in common CPU-based simulators. Thus, our results can support simulationists in the parallelisation of their simulation systems without the need to rely on models tailored to the GPU platform.

Challenges for speeding up ABTSs using FPGAs deviates from using GPUs due to the manifold architectural differences between an FPGA and a GPU, such as smaller memory bandwidths and lower frequencies.

Cong et al. [185] compare the performance of running Rodinia, a widely used GPU benchmark suite, on a GPU and on an FPGA, respectively. They conclude that modern FPGAs can achieve comparable or even better performance for certain tasks. A maximum speedup of 7x over GPU is achieved running a bioinformatics application.

A general FPGA-based discrete event simulation accelerator is presented by Rahman et al. [186]. Their work focuses on the efficient maintaining of the event queues on FPGAs. However, event queues are not necessarily needed in our case.

In previous works, FPGAs are mostly applied to CA-based models owing to the natural mapping of cells to logic blocks [2]. FPGAs have demonstrated performance benefits when applied to CA-based models such as an environment simulation [96], Game of Life [93] and a crowd evacuation model [103]. Schäck et al. [187] implement

a traffic simulation on a single lane road based on so-called Global CA model in which a set of cells change their states simultaneously. Tripp et al. [80] develop a CA-based traffic simulation running on a grid of roads. The movement of agents on individual lanes is computed on an FPGA while the agents' transitions from one road to another and at intersections are computed on the CPU. Our implementation does not rely on transforming models to a CA-based format, and thus can be extended to traffic models beyond those discussed in this chapter.

3.3 City Mobility Simulator(CityMoS)

City Mobility Simulator (CityMoS) [188] is a holistic agent-based, time-stepped, city-scale mobility simulator, designed for CPU-based platforms. It aims to study the whole mobility system which involves traffic, power, vehicle from a holistic perspective, allowing to investigate the whole complex interaction and dependencies between different simulation participants. In addition to private vehicles, it is also able to simulate many elements of urban mobility systems such as public buses or railway-based system. It has lent itself to many successful use cases in investigating various traffic-related problems such as charging station placement in city areas [189], ride-sharing problems [190], etc.

Each vehicle is an agent moving in a road network represented as a graph. An agent is represented by a so-called *Driver-Vehicle-Unit* consisting of driver behaviour and vehicle component models. Driver behaviour models include e.g. car-following model, lane-changing model or charging models. Vehicle component models describe the properties of a vehicle such as battery or air-conditioner, etc. Among all these models, the movement of the agents, which is the core part of the traffic simulation, is governed by two essential models: a car-following model and a lane-changing model.

Each car follows the vehicle ahead and avoids collisions by adjusting its acceleration and deceleration at every time step. The Intelligent Driver Model (IDM) [180] is one of the most popular car-following behavioural models. The input parameters of IDM include the current velocity, a desired velocity, and the distance and speed of the vehicle in front. IDM returns the acceleration for the next time step limited by a maximum possible acceleration.

While the car-following model controls the longitudinal movement of the vehicle, lateral movement is handled by the lane-changing model. Typically, the simulated vehicle evaluates whether changing the lane could allow it to accelerate (Discretionary Lane Change, DLC) or whether a lane change is necessary to continue its route (Mandatory Lane Change, MLC). Usually, lane changing decision is modelled as an incentive-based

Table 3.1: Average and peak number of agents on the road depending on the agent generation rate.

Generation Rate	Average	Peak
2	4 557	6 257
5	7 257	20 745
10	9 788	35 423
20	11 291	44 922
50	12 218	48 964
100	12 717	49 652

decision that evaluates potential lane changing manoeuvres and generates a decision that balances the current vehicles acceleration with the required braking by other vehicles. Widely used lane changing models include Ahmed's lane-changing model[191, 192] and MOBIL [181].

3.4 Feasibility Study 1: Model-level vs Sense-Think-Act-level Parallelisation

In the first study, we parallelise CityMoS on CPU-GPU systems using the parallelisation schemes described in Section 2.5. We also discuss the required changes to an existing simulator architecture and assess the performance gains compared to an optimised CPU-based execution.

3.4.1 Simulation Settings

We consider a scenario where CityMoS runs on an arbitrary road network represented as a graph. Edges represent road segments while nodes represent intersections. Each road segment may have multiple lanes.

The traffic intensity is varied by configuring the *generation rate*, which is the number of agents generated at each time step of 250ms, creating different levels of congestion (cf. Table 3.1). In total, around 50 000 agents are generated.

The considered traffic simulation is executed on a model of the road network of the city-state of Singapore. The agents' routes are pre-calculated based on the shortest path given a origin-destination pair drawn uniformly at random.

The performance measurements are performed on a system equipped with an Intel Core i7-4770, 16 GB of RAM and a dedicated NVIDIA GTX 1060 graphics card with 6 GB of RAM. An execution scheme for partial offloading is evaluated on an APU

platform with a dual-core Intel Core i5-4278U, an integrated Intel Iris Graphics 5100 GPU and 16 GB of RAM.

3.4.2 Overview

As a starting point, we briefly discuss how CityMoS originally works.

```
while (termination criterion not satisfied) {
  for each (agent) {
    /*Model 1*/
    agent.model1.sense();
    agent.model1.think();
    agent.model1.act();

    /*Model 2*/
    agent.model2.sense();
    agent.model2.think();
    agent.model2.act();

    ...
  }
  advance simulation time
}
```

Listing 3.1: Pseudocode of the core execution loop of an agent-based simulation. The agent interactions during the Act stage prohibit trivial parallelisation of the loop iterations.

Each vehicle is represented by an agent, whose behaviour is determined by several models, each following the Sense-Think-Act cycle. Pseudocode of the core simulation loop is given in Listing 3.1.

To identify potentials for parallelisation, we consider the dependencies among the stages and within each stage (cf. top of Figure 3.1). We can observe that the Sense and Think stages are independent across agents and are thus candidates for parallelisation. During the Act stage, agents may affect their neighbours, e.g., by attempting to enter the same position on a lane as another agent. Since these interactions are difficult to predict, parallelisation of the Act stage is non-trivial. Thus, in our CPU-based execution scheme (OMP-SENSE-THINK-CPU), used as a baseline in our experiments, we parallelise only the Sense and Think stages.

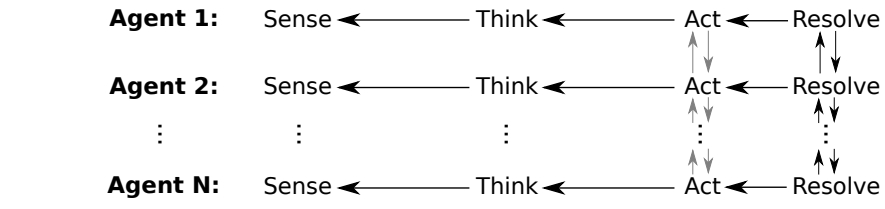
```
while (termination criterion not satisfied){
```

```
2   for each (agent) in parallel {
3     agent.model1.sense();
4     agent.model2.sense();
5     ...
6   }
7
8   for each (agent) in parallel {
9     agent.model1.think();
10    agent.model2.think();
11    ...
12  }
13
14  for each (agent) in parallel {
15    agent.model1.act();
16    agent.model2.act();
17    ...
18  }
19
20  advance simulation time
21 }
```

Listing 3.2: Pseudocode of the core execution loop of an agent-based simulation after applying loop fission. Since the Sense and Think stages are independent across agents, the first two for-loops can both be trivially parallelised.

Considering opportunities for offloading the stages to a GPU, we note that for each agent, the Sense stage requires access to a portion of the road network and agent states to gather the relevant neighbour states. In effect, the Sense stage requires access to most or all of the simulation state and is thus not well-suited for offloading. Similarly, updating the agent states in the Act stage requires access to most or all of the agent states. Thus, we propose two variants of an offloading scheme, corresponding to the Sense-Think-Act-level execution scheme, that executes the Think stage on a GPU: in OCL-THINK-GPU, we execute the Think stage on a dedicated GPU, which requires data transfers over the PCI-E bus before and after the computations. In OCL-THINK-AGPU, we offload to the GPU portion of an APU. Although the limited computing resources of the integrated GPU constrain the performance benefits in terms of pure computation, the shared access to main memory by the CPU and the GPU portion allows us to eliminate data transfer delays.

Finally, we explore a fully GPU-based execution scheme i.e. model-level execution scheme (OCL-ALL-GPU), in which all stages (the entire model) are executed on a dedicated GPU. This scheme eliminates all major data transfers during the main simu-



***-CPU**	*CPU*	-	*CPU*	-	*CPU*	-	-
OCL-THINK-GPU	*CPU*	*PCI-E*	*GPU*	*PCI-E*	*CPU*	-	-
OCL-THINK-AGPU	*CPU*	*Zero-Copy*	*GPU*	*Zero-Copy*	*CPU*	-	-
OCL-ALL-GPU	*GPU*	-	*GPU*	-	*GPU*	-	*GPU*

Figure 3.1: Top: dependencies among the stages in a simulation time step. If inter-agent dependencies are ignored during the Act stage, a separate conflict resolution stage is required. Bottom: the execution schemes considered in our experiments and the means of data transfer from one stage to the next.

lation loop but requires porting the entire simulator engine to the GPU. Since properties of a graph-based road network representation are exploited, OCL-ALL-GPU is specific to the considered models, whereas the other execution schemes are applicable to other agent-based models that follow a Sense-Think-Act cycle.

Since our OpenCL implementation also allows for execution on a CPU, we compare the execution schemes with the same implementations running in parallel on a CPU (OCL-THINK-CPU), the CPU portion of an APU (OCL-THINK-ACPU), and a fully parallelised CPU-based execution (OCL-ALL-CPU).

The bottom of Figure 3.1 lists the execution schemes together with the means to carry out the required data transfers. In the remainder of this section, we describe each execution scheme in detail. For each scheme, we discuss the overall design and implementation concerns as well as the results of our performance measurements.

3.4.3 Baseline: CPU-Based Execution

The starting point is a sequential CPU-based implementation. To acquire a fair baseline for performance comparisons, we parallelise the portions of the simulation that do not require substantial changes to the existing architecture.

3.4.3.1 Architecture

In the CPU-based execution scheme, we execute the entire simulation on the CPU. As discussed above, the Sense and Think stages can be executed independently for each agent. Parallelisation of the Act stage requires efficient synchronisation and conflict

resolution and thus profound changes to the simulator architecture, which a variety of existing works have explored [193, 194]. Since our focus is on the transition from a CPU-based simulator to a heterogeneous execution, we restrict the parallelisation of our initial CPU-based implementation to the Sense and Think stage. A fully parallelised execution scheme will be explored in Section 3.4.5. We focus on parallelisation within a single execution node.

Considering the pseudocode of Listing 3.1, there is one loop iteration for each agent, each iteration covering all models for the current agent. Since the agent states are only updated in the Act stage, Sense and Think can be parallelised across all agents. However, the data dependencies given by the interactions among agents in the Act stage prohibit a straightforward parallelisation. Thus, we adapt the control flow to separate the Sense and Think stage from the Act stage.

The required transformation of splitting a loop into multiple loops is known as *loop fission* [195]. Listing 3.2 gives the pseudocode after loop fission. Although the computational steps are unchanged on a conceptual level, loop fission distributes the accesses to each agent's state variables across multiple loops. Due to the decrease in memory access locality, a performance decrease must be expected. We study the effect of loop fission on the performance in Section 3.4.3.3.

3.4.3.2 Implementation

We restructured the simulator code by applying loop fission. The parallelisation is performed using OpenMP by annotating the for-loops of the Sense and Think stages. To limit contention for the workload among the CPU threads, we configured a chunk size of 1000 agents. During the Sense stage of the car-following and lane-changing models, the respective output is stored in a per-agent element of an array. The two resulting arrays form the input to the Think stage of the two models.

For lane-changing behaviour, we rely on Ahmed's model, which requires drawing two uniformly distributed random numbers in $[0, 1]$ for each agent at each time step. To achieve a fair comparison with the GPU-based implementations, we use the MWC64X generator[1], for which efficient implementations exist both in plain C and OpenCL.

3.4.3.3 Performance Evaluation

The experiment is conducted on a platform equipped with a dedicated GPU. As illustrated in Figure 3.2, loop fission slightly decreases performance. The runtime was

[1] `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`

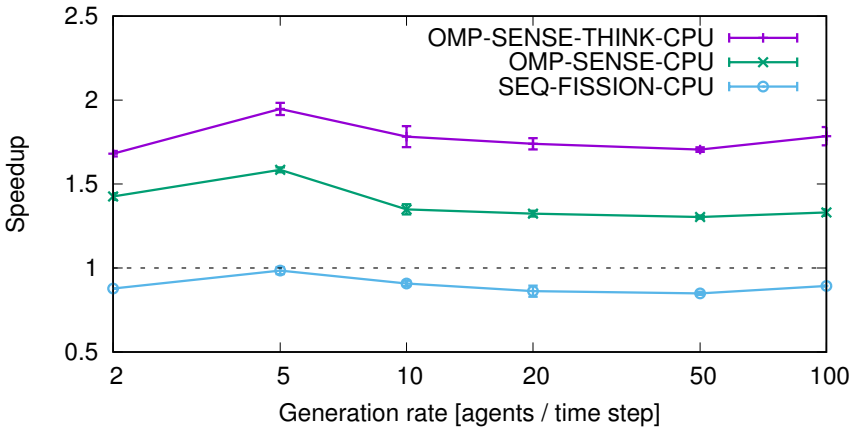


Figure 3.2: Speedup with standard errors when parallelising Sense (OMP-SENSE-CPU) or Sense and Think (OMP-SENSE-THINK-CPU) over sequential execution (SEQ-CPU). Loop fission (SEQ-FISSION-CPU) results in a slight slowdown.

increased by 15% in the worst case. When applying the OMP-SENSE-CPU scheme, i.e., with only the Sense stage parallelised by OpenMP, a performance gain of at least 1.30x is observed. The speedup is increased by also parallelising the Think stage (OMP-SENSE-THINK-CPU), achieving at least 1.68x for all cases. The highest speedup is achieved when the agent generation rate is 5 per time step for both the OMP-SENSE-CPU and OMP-SENSE-THINK-CPU schemes, with a respective speedup of 1.58x and 1.95x.

3.4.4 Sense-Think-Act-level parallelisation (Partial Offloading)

In the following, we aim at accelerating the simulation in a heterogeneous CPU/GPU environment using the Sense-Think-Act-level execution scheme described in Section 2.5. We first explore an offloading approach [196, 140], where computationally intensive portions of the simulation are offloaded to the GPU.

3.4.4.1 Architecture

The offloading approach follows the idea of the parallelised CPU-based execution: we exploit the independence of per-agent stages. However, since the GPU does not have direct access to the host memory, data transfers over the PCI-E bus are introduced. The Think stage is a natural candidate for offloading, since it relies only on the output of the Sense stage. In contrast to this, the Sense stage accesses both the static environment, i.e., the lengths and speed limits of nearby road segments, and the states of

nearby agents. Similarly, an agent's Act stage relies on the static environment and may interfere with the agent's neighbours. Thus, offloading the Sense or Act stage would require transferring substantial parts of the simulation state to the GPU at each time step, which instead suggests porting the entire simulation to the GPU. Based on this reasoning, we offload only the Think stage. As in the CPU-based scheme, the Sense stage is executed in parallel on the CPU. The Act stage is executed sequentially on the CPU.

At the start of each Think stage, the CPU transfers the data needed for the computation from the host memory to the graphics memory. The GPU processes the data and transfers the results back to the host memory. Further, we explore the offloading scheme using an APU, allowing us to avoid data transfers over the PCI-E bus.

3.4.4.2 Implementation

The implementation of the scheme follows the general approach used for the Think stage in the CPU-based scheme, replacing the for-loop of the Think stage with a call to GPU code that executes the stage in parallel for all agents. Our implementation is based on OpenCL, which allows us to execute the same code on a CPU, GPU, or APU. OpenCL API calls are used to transfer the model input data to the GPU, to call an OpenCL kernel executing the model, and to transfer the model output data back to the host memory. On the GPU, each thread executes the model for one agent. The OpenCL implementation of the models is nearly identical with the plain C++ implementation, apart from the indexing based on GPU threads required in OpenCL. Additional arrays in graphics memory are used to store the model input and output for each agent. The input array is filled by a data transfer from host memory prior to the OpenCL call. After the OpenCL call has finished, the content of the output array are transferred to host memory. On the CPU, each agent reads the respective output values from the output array during the subsequent Act stage. Quantitatively, the input data comprises 16 bytes per agent for the car-following model and 52 bytes for the lane-changing model. The output for each models comprises 4 bytes per agent. The implementation targeting APUs follows the same general process. However, to utilise the zero-copy technique, the input and output arrays accessed by the OpenCL kernels are created using the OpenCL memory flag *CL_MEM_USE_HOST_PTR* [197], which allocates memory in a shared space that can be accessed by the CPU and the GPU portion, avoiding data transfers.

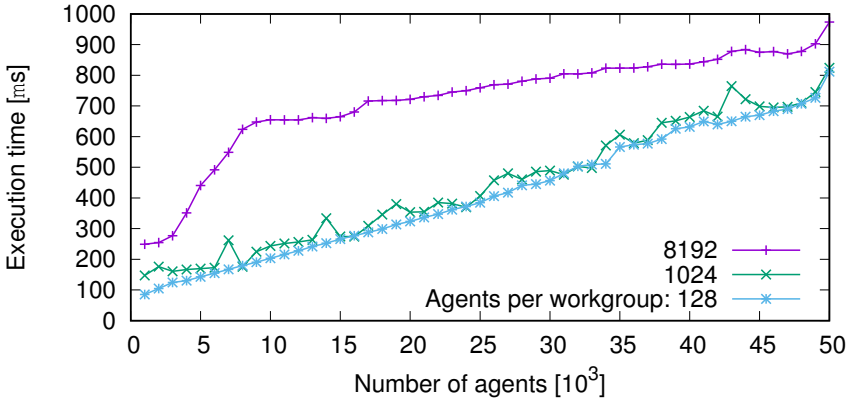


Figure 3.3: Total execution time on the dedicated CPU for the Think stage of the car-following model and lane-changing model when varying the workgroup size.

3.4.4.3 Performance Evaluation

We evaluate partial offloading with respect to purely CPU-based execution and offloading of the Think stage to the GPU, both for the platform with a dedicated GPU (OCL-THINK-CPU and OCL-THINK-GPU) and for the APU platform (OCL-THINK-ACPU and OCL-THINK-AGPU).

The configured workgroup size in OpenCL can have a substantial impact on the overall performance. The best-performing workgroup size depends on properties of the hardware and the computation to be performed [198]. To find the best configuration, we vary the workgroup size from 128 to 8192 for the CPU and 128 to 1024 for the GPU both for the car-following and the lane-changing kernels. As depicted in Figure 3.3, the smallest workgroup size 128 always leads to best performance on the CPU. In contrast, we observe that the workgroup size configurations we studied do not have an obvious impact on the GPU performance. We set the GPU's workgroup size to 128. The same value of 128 was identified to achieve the best performance on both the CPU and GPU portion of the APU platform. In the remainder of the section, we will use the best configurations in all measurements.

Figure 3.4 shows a comparison of partial offloading execution schemes. A speedup of 1.8x over sequential execution on the CPU (SEQ-CPU) is achieved for OCL-THINK-GPU, i.e., offloading the Think stage to the dedicated GPU. OCL-THINK-CPU produces a slightly better result, achieving a speedup up to 2.0x, due to the relatively lightweight computations in relation to the data transfer overheads introduced by OCL-THINK-GPU.

However, on the APU, on which the data transfer overhead for offloading is eliminated, OCL-THINK-AGPU achieves an overall speedup of up to 1.7x over CPU-SEQ on the CPU portion of the APU, outperforming both the OCL-THINK-ACPU and OMP-SENSE-THINK-ACPU schemes (cf. Figure 3.5).

Although the Think stage can be accelerated substantially by offloading, the overall performance gain is limited by Amdahl's law: since the Think stage constitutes about 18.6% of the runtime on the APU platform given the Sense stage has already been parallelised, an upper bound on the speedup is given by $1/(1 - 0.186) = 1.23$. Thus, our aforementioned speedup of 1.18x is close to the theoretical maximum if offloading is limited to the Think stage.

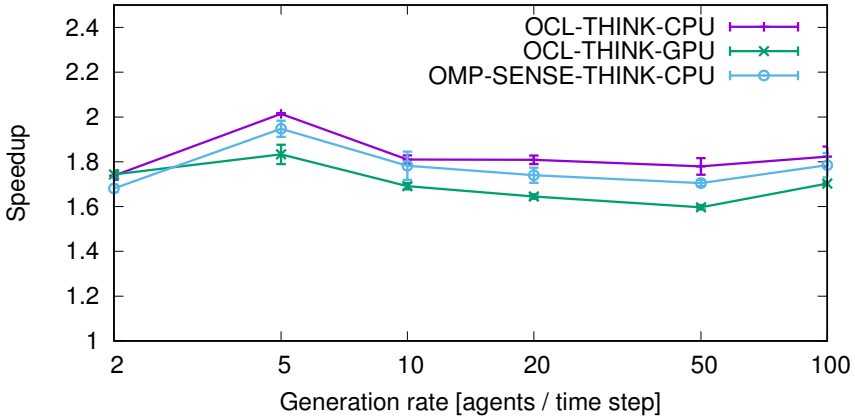


Figure 3.4: Speedup with error bars showing standard errors over sequential execution when parallelising Sense by OpenMP and Think by OpenCL (OCL-THINK-CPU) and when offloading Think to the dedicated GPU (OCL-THINK-GPU).

3.4.5 Model-level parallelisation (Full Offloading)

In the previous section, we observed that offloading the Think stage can already give some performance gains. However, even if we avoid data transfers using zero-copy memory access on an APU, the performance gains are limited by Amdahl's law. As discussed in Section 3.4.4, offloading Sense and Act requires access to nearly all of the simulation state and static environment data. This situation suggests using a model-level execution scheme i.e. full offloading to accelerators instead. In the following, we present and evaluate a fully offloaded traffic simulator running entirely on a many-core GPU. Thanks to the implementation in OpenCL, the simulator also supports a parallelised execution on a CPU.

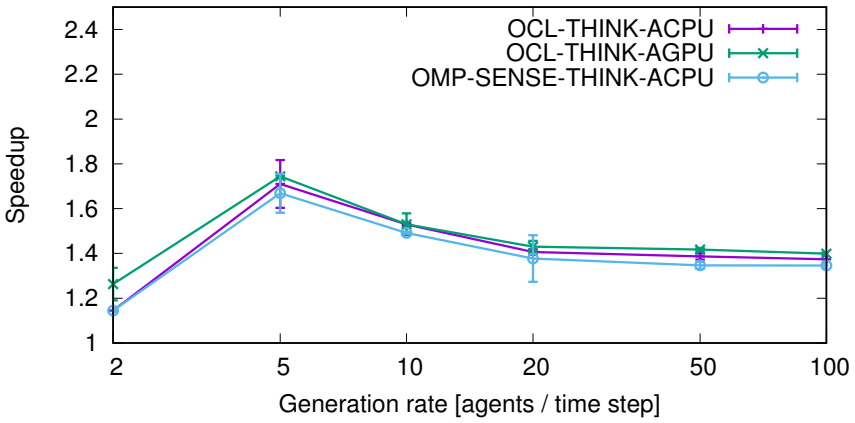


Figure 3.5: Speedup with error bars showing standard errors over sequential execution when parallelising Sense by OpenMP and Think by OpenCL on the CPU portion of an APU (OCL-THINK-ACPU) and when offloading Think to the GPU portion of an APU (OCL-THINK-AGPU).

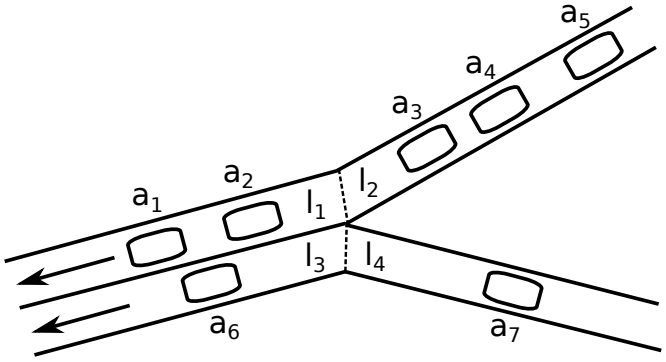
3.4.5.1 Architecture

In this execution scheme, major data transfers are only required during initialization to set up the static environment and to provide the simulator with the initial scenario parameters. Subsequently, the simulation proceeds as a sequence of OpenCL kernel calls, with minor data transfers to signal the termination of the simulation. Output of simulation statistics may be performed either by data transfers during the simulation or once the termination criterion is met.

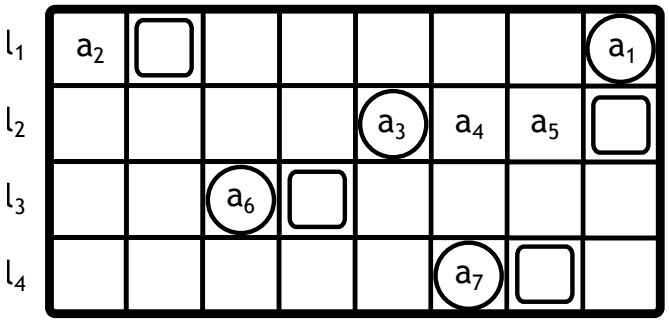
Considering the dependencies between the agent stages, the Sense and Think stages are now both trivially parallelisable. However, when parallelising the Act stage, the potential interactions among agents must be considered: when moving, multiple agents may enter overlapping positions in the simulation space. While a sequential execution with one-by-one position updates can avoid such situations, a parallelised execution requires a conflict resolution mechanism to achieve consistent, i.e., collision-free, agent states. We perform conflict resolution after the Act stage.

3.4.5.2 Implementation

In our implementation of the fully GPU-based execution scheme, each agent is a structure composed of a numerical identifier as well as the current and desired lane, position, and velocity. Each lane in the road network is represented by a ring buffer holding the agents currently on the lane, ordered by their positions (cf. Figure 3.6). As in the offloading version, apart from the indices used to access the input and output data, the



(a) Example of a portion of the graph-based road network, showing 3 links with a total of 4 lanes (l_1 to l_4) and 7 vehicles (a_1 to a_7).



(b) Representation of the road network in memory. Each lane l_i is a ring buffer holding vehicles a_j ordered by their position on the lane, with associated indexes for the head (circle) and the initial index of vehicles entering from other lanes (rectangles).

Figure 3.6: Road network representation in OCL-ALL-CPU and OCL-ALL-GPU.

OpenCL code for the car-following and lane-changing model are nearly identical with the CPU implementation.

Six OpenCL kernels are called at each time step:

1. SpawnVehicles: in this kernel, a single thread creates and initialises a configurable number of new vehicles.

2. CollectActiveLanes: as preparation for the Sense and Think stages, lanes with at least one vehicle are gathered in an array. Each GPU thread evaluates one lane.

3. SenseAndThink: this kernel combines the Sense and Think stage of both the car-following and lane-changing model. Each workgroup operates on one active lane, with each thread handling one agent at a time.

4. Act: this kernel actualises the desired lane, position and velocity determined in SenseAndThink. Since the Act stage may affect the number and indexes of

agents on each lane, we avoid synchronisation by executing only a single thread per lane. However, when an agent enters a lane, synchronisation is still needed to avoid inconsistencies when multiple agents enter the same lane at the same time step. To this end, an atomic operation increments the index of the insertion position at the tail of the target ring buffer (rectangles in Figure 3.6b), retrieving the old index. The agent can then safely be stored at the old index. The insertion into the sorted ring buffer is performed in the next kernel. We mark agents that leave the current lane so they can be removed by the next kernel.

5. SortLanes: during the Act stage, agents may change lanes, which may affect the relative positions of agents on the target lane. To restore the ordering on each lane, we sort the vehicles by position, including new agents entering the lane. Agents that leave a lane are removed from the ring buffer. Since each lane typically holds at most a few dozen agents, we apply sequential quicksort using one thread per lane.

6. ResolveConflicts: overlaps between the agents are resolved by moving a vehicle that has performed a lane change or advanced to the next link back to the original lane. If more than one vehicle involved in a conflict has entered a new link, the vehicle that is further behind is moved. Each conflict resolution round may affect the ordering at each lane, and may also create new conflicts. Thus, SortLanes and ResolveConflicts are executed until no further conflicts occur.

The implementation based on ring buffers and the synchronisation based on atomic operations closely resemble GPU-based discrete-event simulations, which have been shown to achieve high speedup over a CPU-based execution [199, 138]. Our approach to conflict resolution postpones the conflict resolution to after the Act stage and iterates until all conflicts have been resolved based on the relative position of agents. Alternative approaches to conflict resolution for agent-based simulations on GPUs, as well as considerations of determinism and bias, have been studied in [151].

Since the CPU-based and partially offloaded versions execute the Act stage sequentially, conflicts can be avoided entirely. However, our conflict resolution approach affects the results only marginally compared to SEQ-CPU, with deviations in average travel times smaller than 3% for all tested parameter combinations.

3.4.5.3 Performance Evaluation

Initial performance evaluation runs showed that the performance of OCL-ALL-CPU and OCL-ALL-GPU was only marginally affected by the workgroup size. The mea-

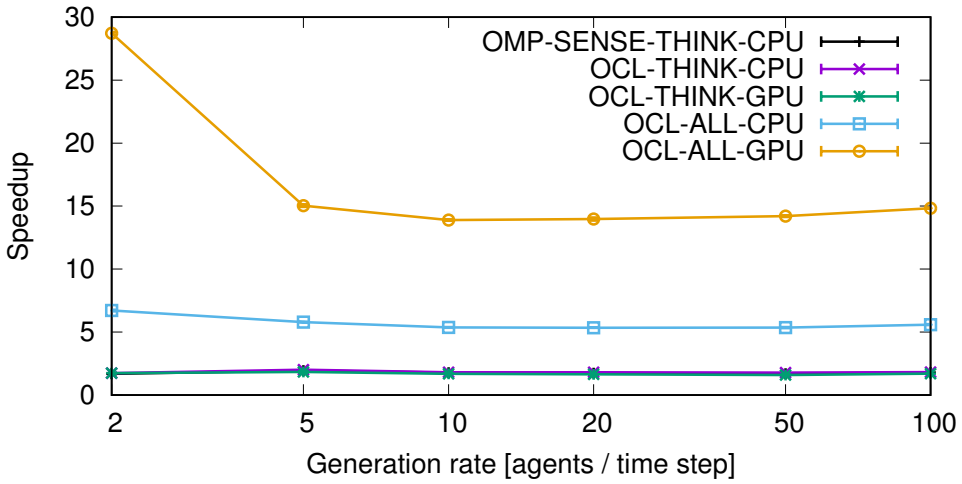


Figure 3.7: Overall comparison of the execution schemes over sequential execution with error bars showing standard errors.

surements described in the following were executed with a workgroup size of 64 for all kernels.

As shown in Figure 3.7, the speedup achieved when parallelising all stages on the CPU (OCL-ALL-CPU) is up to 6.7x. A maximum speedup of 28.7x is achieved for OCL-ALL-GPU at low traffic density (agent generation rate of 2). In a more congested traffic scenario with an agent generation rate of 100, the performance gain by the parallelised execution is counteracted by the increasing overhead for conflict resolution, leading to a smaller overall performance gain. However, even in the most congested scenario, a speedup of 14.8x is achieved.

3.4.6 Discussion

Figure 3.7 gives an overview of the speedup gained by the different schemes on the platform with the dedicated GPU. Table 3.2 (generation rate = 2) and Table 3.3 (generation rate = 100) show the percentages of runtime per time step spent on the Sense, Think and Act stages. For the OCL-ALL-CPU and the OCL-ALL-GPU schemes, CollectActiveLanes and SenseAndThink are considered jointly as a single "Sense and Think" stage. The time spent on Act, SortLanes and ResolveConflicts is counted towards the Act stage.

OpenMP parallelisation in OMP-S-T-CPU (OMP-SENSE-THINK-CPU) halves the runtime for the Sense stage compared with SEQ-CPU, whereas the runtime spent on

Table 3.2: Absolute and relative time spent on one iteration of each stage. Agent generation rate: 2 per time step.

	Absolute [ms]			**Relative [%]**		
	Sense	**Think**	**Act**	**Sense**	**Think**	**Act**
SEQ-CPU	7.94	2.41	4.92	52.0	15.8	32.2
OMP-S-T-CPU	3.64	0.71	4.74	40.0	7.8	52.2
OCL-THINK-CPU	3.69	0.20	4.90	42.0	2.3	55.7
OCL-THINK-GPU	3.78	0.39	4.58	43.2	4.5	52.3
OCL-ALL-CPU	0.60		1.67	26.5		73.5
OCL-ALL-GPU	0.32		0.21	59.8		40.2

Table 3.3: Absolute and relative time spent on one iteration of each stage. Agent generation rate: 100 per time step.

	Absolute [ms]			**Relative [%]**		
	Sense	**Think**	**Act**	**Sense**	**Think**	**Act**
SEQ-CPU	15.29	5.95	9.77	49.3	19.2	31.5
OMP-S-T-CPU	6.65	1.58	9.14	38.3	9.1	52.6
OCL-THINK-CPU	7.03	0.41	9.58	41.3	2.4	56.3
OCL-THINK-GPU	7.37	0.53	10.31	40.5	2.9	56.6
OCL-ALL-CPU	1.50		4.05	27.0		73.0
OCL-ALL-GPU	0.43		1.66	20.4		79.6

Think is reduced to less than a third. Offloading reduces the runtime spent on the Think stage by a factor of up to 16.

Due to the small contribution of the Think stage to the runtime, the speedup through partial offloading is modest, while a maximum speedup achieved through offloading is only up to 19%. On the hardware used in our experiments, the performance gains from parallelisation using OpenMP and OpenCL on the platform with the dedicated GPU are close (cf. Figure 3.7). Substantial speedup is achieved by the OCL-ALL-GPU scheme, i.e., executing the entire simulation on the GPU. The jointly considered Sense and Think stages are accelerated by a factor of 34.3 and 49.4 with an agent generation rate of 2 and 100, respectively. With the more congested scenario, a considerable amount of time is spent on the conflict resolution, with 79.6% spent on the Act stage.

When considering the pure OpenCL simulator implementation, there are still obvious opportunities for runtime reduction. At high traffic density, more than three quarters of the simulation runtime is spent in the Act stage, which includes conflict resolution. Currently, each round of conflict resolution checks for conflicts with respect to every agent on every lane of the road network. However, rules could be formulated to limit

the set of agents that may be involved in a conflict: for instance, depending on the considered models, conflicts may only result from agents either performing a lane change or advancing to the next lane. In such a case, considering only those agents that have entered a new lane may substantially reduce the cost of each round of conflict resolution.

From the performance evaluation results, we can make a number of general observations: firstly, in the considered models, the Think stage takes up a relatively small portion of the overall runtime. Even in a scenario with a peak of about 50 000 vehicles concurrently on the road, Think only contributes about 20% to the runtime. Hence, the gains achievable by offloading the Think stage are inherently limited: we achieved up to about 18% speed-up by offloading to an APU, only slightly exceeding the speed-up by a simple parallelisation of the Sense and Think stage on a CPU. For simplicity, a purely CPU-based parallelisation of these two stages may be preferable.

Secondly, substantial speed-up can be achieved when parallelising all stages i.e. the entire model execution on one accelerator. The achieved runtime reduction by a factor of more than 28 on a GPU may put some large-scale parameter studies into reach that would be considered overly time-consuming using a sequential simulator. Since even on a multi-core CPU our OpenCL implementation achieves a speedup of more than 6, our results show the suitability of common traffic simulation models for fine-grained parallelisation. Since only small amounts of computation are required at each time step of the simulation, retaining all simulation data within a single device contributes positively to the performance.

Our measurements are specific to the selected driver behaviour models: the Intelligent Driver Model for car-following, and Ahmed's model for lane-changing. However, we expect other microscopic traffic simulation models or even some other agent-based models in various domains to exhibit similar demands in terms of input data. Thus, for offloading approaches to achieve significant speed-up even with the cost of the added data transfers, the computational demands of the models would need to be substantially larger than those of the models considered in this chapter. This implies that although the Sense-Think-Act-level parallelisation seems to provide more opportunities to parallelise, it would not lead to sustainable performance benefits. Performance gains are larger when the entire model is executed on the accelerators.

Notably, OCL-ALL-CPU and OCL-ALL-GPU make use of properties specific to traffic simulation on a graph-based road network and require a model-specific conflict resolution step. Compared to the other execution schemes, programming efforts increase significantly.

In summary, from the first feasibility study, two conclusions can be drawn. First, the model-level parallelisation scheme is preferable and therefore will be employed in the rest of this thesis. Second, programming efforts grow almost proportionally to the performance gain. To achieve the best performance as indicated by OCL-ALL-CPU and OCL-ALL-GPU, in-depth knowledge of the hardware and significant amount of programming efforts are required.

3.5 Feasibility Study 2: ABTS on FPGAs

We explore design options to port CityMoS to run on FPGAs. In the last decade, a growing interest can be observed in FPGAs as high-performance and energy-efficient accelerators for compute-intensive tasks. Previous works [80, 187] also show the feasibility of accelerating ABTSs using FPGAs. Those works often rely on converting common traffic models to models based on CA, which naturally map to the FPGA's hardware building blocks. In these works, the FPGA applications were expressed in HDL, which describes the required functionality at a low level in terms of logical operations and data transferred among hardware registers. The emergence of high-level synthesis toolchains for FPGAs from vendors such as Intel and Xilinx enables development of FPGA programs in C-like languages, reducing the development effort [200] and enabling portability to and from other hardware platforms.

3.5.1 Simulation Settings

Due to limited hardware resources available on FPGAs as mentioned in Section 2.2, especially owing to the lack of massive high-speed memory to store e.g. a large road network, we simplify the traffic simulation scenario to be a proof-of-concept but which still exercises both car-following and lane-changing models. For the simulation environment, we consider a single road with a configurable number of parallel lanes.

We still focus on the heterogeneous hardware scenario, i.e., a PC or a cluster equipped with an FPGA accelerator connected via PCI-E. The traffic simulation is run on a road with four lanes for 1,000 time steps, varying the total number of agents from 2^4 to 2^{14}. Initially, agents are spawned at the leftmost two lanes of the road. For acceleration using FPGAs, we will only consider a full FPGA-based execution scheme owing to two reasons. The first reason is that, as concluded in Section 3.4.6, for the simulation models we studied, performance benefits are larger with a full offloading approach. The second reason is, although commonly both GPU and FPGA are connected to the host via PCI-E channel, FPGA typically has a slow data transfer speed, when

compared to GPU [201]. This would further increase the data transfer overhead, causing a downgrade in the partial offloading approach.

3.5.2 FPGA-based Execution

3.5.2.1 Architecture

We present our design of an FPGA-accelerated ABTS generated from OpenCL code using the SDAccel tool by Xilinx. OpenCL offers a two-layer memory hierarchy for FPGAs: *Local memory* maps to the Block RAM (BRAM) of an FPGA, which is shared among work-items in the same work-group. *Global memory* binds to the massive but latency-prone off-chip memory to which all work-items have access.

Unlike GPUs which utilise Single Instruction Multiple Data (SIMD) parallelisation, an FPGA's parallelisation is two-fold. Each operation (store/load, arithmetic operation, etc.) is compiled to a small circuit called a Function Unit (FU). An FPGA can generate many FUs for the same operation so that multiple data elements can be processed simultaneously. Further, many FUs are wired together to form a *pipeline*. A work-item steps over one FU at a time while many work-items start successively to fill the different stages of the pipeline (Fig. 2.2a). The start interval between two consecutive work-items is called Initiation Interval (II). The performance of an FPGA is influenced mainly by two factors: the initiation interval and the operating frequency. Dependencies across work-items may cause a bigger II. The operating frequency is guided by the cycles spent on the slowest pipeline stage.

SDAccel is a High-Level Synthesis (HLS) tool from Xilinx for programming FPGAs for x86 systems. It allows users to express an OpenCL kernel using HDL, HLS (high-level synthesis from C or C++), or OpenCL. We explore designs with the latter two approaches. The C or C++ used for HLS is close to sequential code in terms of style, with extra syntax provided for FPGA-specific data mappings and optimisation, making porting code from existing sequential code trivial. The resulting program presents as a single work-item to the host. The body of an HLS kernel is typically a loop where one data point is dealt with per iteration. When parallelised on an FPGA, many iterations start a number of cycles apart (as defined by the II), yielding pipelined parallelism. Two types of OpenCL kernels are supported when programming in OpenCL [202]. *Single-work-item* kernels are intrinsically equivalent to HLS with extra use of OpenCL primitives such as *__kernel*, *__global*. *NDRange* kernels are a type of kernel widely used for programming GPUs, and thus offer good portability to and from GPU code. When executed on FPGAs, *NDRange* kernels can either yield SIMD parallelism as

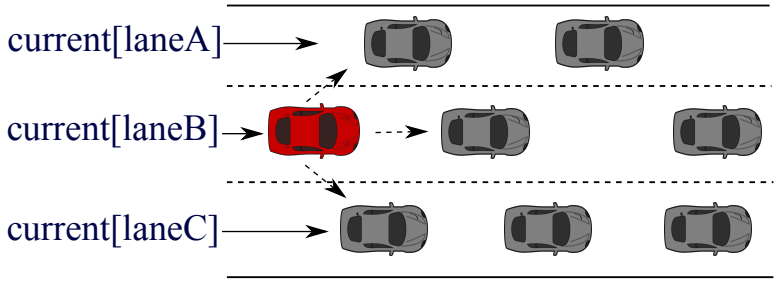
GPUs or pipelined parallelism if dependencies exist across work-items. Due to the similarity between kernels written as *Single-work-item* and in HLS, we will omit *Single-work-item* and focus on HLS and *NDRange* for the remainder of the chapter.

3.5.2.2 Implementation

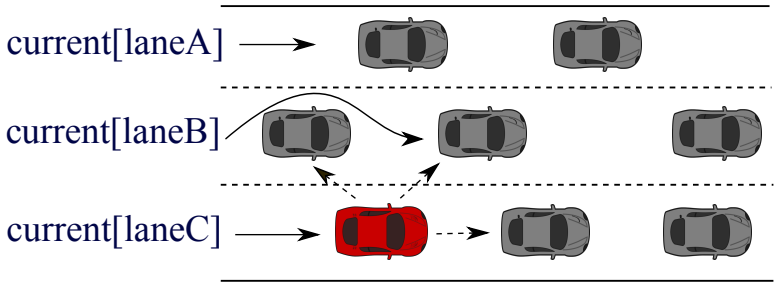
In a typical ABTS, an agent autonomously performs neighbour searches, makes decisions based on its neighbours' states and updates its own state in each time step. Neighbour search is a crucial step and potentially consumes a significant amount of execution time. Therefore, we start this section with a simple neighbour search algorithm that can be executed efficiently on FPGAs. Then, we discuss two design considerations that can further improve the performance. Lastly, we assemble everything and present our overall design.

Agent Update and Neighbour Search As our simulation scenario, we consider a long road with multiple parallel lanes. Initially, agents are assumed to be stored in a per-lane array, sorted by ascending agent positions in driving direction. After a single step of the car-following model or lane-changing model is executed, the agents remain sorted: 1) The car-following model drives the agent to follow but not to overtake the agent in the front; 2) To carry out a lane change, the lane-changing agent is removed from its original lane and inserted between its leader and follower on the target lane. We present a simple algorithm that relies on the per-lane agent order to efficiently determine the neighbours of all agents on the road. The basic idea is to iterate through the agents in the order of their position considering all lanes, updating per-lane pointers to efficiently identify their neighbours.

Figure 3.8a illustrates the neighbour search: For each lane l, a pointer $current[l]$ is maintained, initially pointing to the rearmost agent. The neighbour search iterates through the agents by their position across all lanes, until the foremost agent has been reached. Once an agent's neighbours have been identified on lane l, the pointer $current[l]$ is updated to point to the agent's leader on lane l (cf. Figure 3.8b). During the iterations, the invariant holds that the neighbours of each current agent are identified by the *current* pointers as follows: the leader and the follower, if any, on the current lane l are located by $current[l]+1$ and $current[l]-1$. The leaders and followers on neighbouring lanes l_i, if any, are identified by $current[l_i]$ and $current[l_i]-1$.



(a) Neighbour search of the first agent (coloured in red). Neighbours are indicated by the dash arrowed lines.



(b) Neighbour search of the second agent (coloured in red). Neighbours are indicated by the dash arrowed lines.

Figure 3.8: Neighbour search.

Design Considerations To maximise the parallelisation opportunities and tailor the computing workload to the underlying FPGA hardware, we consider the following two design concerns:

- **Single and Double Buffering**

  Single and double buffering (SB and DB) are two design principles for ABSs, both of which have implications on parallelized execution, model behaviour and the potential for state conflicts [151]. With SB, each agent overwrites its state immediately after the agent behaviours have been executed, whereas with DB, the new agent state is first written to an intermediate buffer and consequently applied for all agents at once. The SB design may cause write-read or write-write dependencies, resulting in larger initiation interval when pipelined on FPGA. Further, with SB, inconsistent agent states may be read as the agent updates gradually progress throughout the pipeline. To avoid these complications, we rely on DB in our implementation.

- **Number Representation**

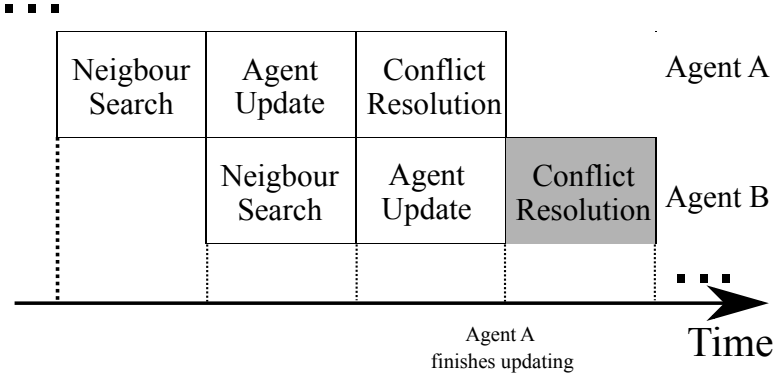


Figure 3.9: Illustration of pipelined processing of agents

As the high dynamic range of floating point numbers is not required to represent positions and velocities on individual roads, we rely on fixed-point arithmetic to achieve better power and space efficiency [203].

We use 24 bits to represent the integer part and 8 bits for the fractional part, which enables a minimum number increment of 2^{-8}, i.e., a precision of 0.4cm or cm/s.

Main Simulation Loop In the HLS implementation, the main body of the kernel function is a nested loop. The outer loop iterates through the simulation steps. The inner loop, which is pipelined, iterates through the agents to carry out the neighbour search and agent update.

In the NDRange kernel implementation, the loop that counts the simulation steps resides in the host code. The loop that updates the agents is replaced by an NDRange function call. Due to the incremental updating of the *current* pointers in the neighbour search algorithm, SIMD processing of multiple agents is infeasible. However, pipeline parallelization across work-items is possible.

Unintended vehicle collisions may occur due to the employment of DB design [204]: as illustrated in Fig. 3.10, the red vehicle and the yellow vehicle are close in longitudinal positions and both intend to change lanes to the middle lane. As the vehicles are two lanes apart, they are unaware of each other. As a result, both of them may decide to change to the middle lane, causing a collision. To resolve such conflicts, before writing to the buffer holding the new state, we check whether the last stored agent on the target lane collides with the current agent. The pipelined execution on the FPGA ensures that at this point, the last agent has finished updating its state (cf. Figure 3.9).

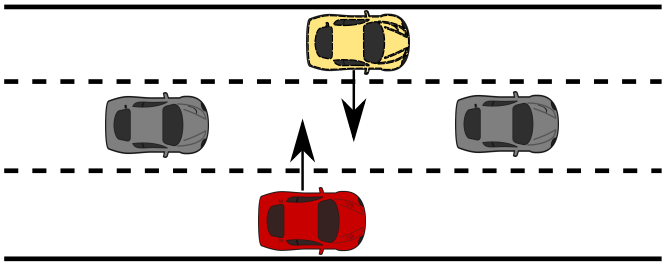


Figure 3.10: Illustration of a possible collision due to our double-buffering design.

3.5.2.3 Performance Evaluation

We evaluate our HLS and NDRange kernels on an Amazon f1.2xlarge instance equipped with an 8-core Intel Xeon E5-2686, 122GB RAM and an Xilinx Virtex UltraScale+ XCVU13P FPGA. The FPGA has a maximum frequency of 500MHz, 3,780K Logic Cells, 12,288 DSPs and 455Mb RAM. The version of the Xilinx OpenCL Compiler is 2018.2 with GCC version 4.8.

The run time is compared to that of a sequential CPU implementation on an Intel Xeon-E5 CPU and a SIMD implementation on an NVIDIA GTX 1060 GPU. However, the proposed neighbour search and conflict resolution implementations rely on the pipelined execution and strict update order guaranteed on the FPGA, as GPUs process many agents in an undefined order. The GPU implementation is based on the one described in Section 3.4.5. When a conflict is detected, the agent that is further behind on the lane is rolled back to its original lane. As shown in Fig. 3.11, FPGA-HLS is slower than the CPU at small scales due to the initialisation and data transfer overheads. However, FPGA-HLS outperforms the CPU for 256 and more agents. Interestingly, the GPU performs similarly to the CPU and is slower than FPGA-HLS. This is due to the substantial overheads in the conflict resolution step on the GPU, which constitutes 60-90% of the total execution time. FPGA-HLS achieves a 24.35x speed up over the CPU and 8.9x over the GPU when simulating 16,384 agents. Notably, FPGA-NDRange is the slowest implementation in all cases. One reason for this is the dependencies among work-item on the `current` pointers, which prevent parallel processing of the agents. Moreover, FPGA-NDRange cannot rely on pre-fetching: in the HLS implementation, agent data is pre-fetched from the latency-prone off-chip memory to the fast BRAM before the main simulation loop is entered. The pre-fetching contributes to the initialisation overhead, resulting in a slowdown at small scales, but a performance increase at larger scales. However, in the NDRange implementation, where multiple work-groups are presented and the BRAM address space can only be

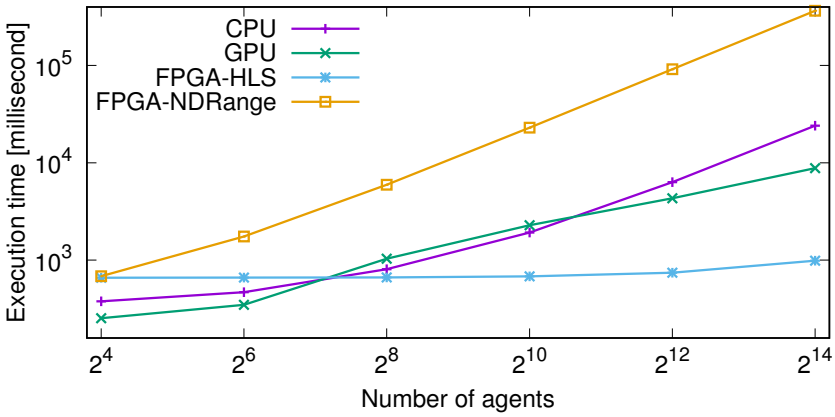


Figure 3.11: Performance comparison between CPU, GPU, FPGA-HLS and FPGA-NDRange

shared within the same work-group, the memory required to pre-fetch all agent data for each work-group exceeds the available BRAM at large scales. Pre-fetching a chunk of data per work-group is infeasible as it is only beneficial when the neighbouring agents are pre-fetched, which naturally cannot be done before the neighbour search.

Instead, each work-item fetches agent data from the slow off-chip memory, causing a decrease in performance.

Our kernels consume less than 6% of the FPGA's resources. This indicates a good potential to extend our approach to full road network topologies by generating multiple pipelines on the same FPGA, each pipeline dealing with one road. Extra logic might be required in this case for transmitting agents between pipelines.

3.6 Summary

Two feasibility studies were demonstrated in this chapter. In the first study, we evaluated two parallelisation schemes proposed in Section 2.5 on a CPU-GPU as well as an APU system. In the second study, an ABTS was ported onto FPGAs using high-level synthesis.

There are several essential messages that can be taken away:

First, as shown in Section 3.4.6, although partitioning the simulation using Sense-Think-Act-level paradigm appears to provide more potentially parallelisable components, it also increases the data transfer overheads. While a performance gain could still be achieved, the benefit was larger when offloading the entire model on one accelerator. Therefore, when extending OpenABL, we will keep the design of model-level parallelsation.

Another important message is that, while accelerating ABS using FPGAs is feasible, the challenges are different from e.g. using a CPU or a GPU. As revealed in Section 3.5.2.3, the code that tailored for an FPGA architecture does not perform equally well on a SIMD architecture e.g. on a GPU and vice versa. This is due to the substantial architectural differences in terms of memory hierarchy and the computing power of a single processing unit (in FPGA a processing unit is a FU). Therefore, when considering OpenCL-based parallelisation for the aforementioned accelerators, CPU and GPU can be treated similarly, whereas FPGA should be dealt with separately.

As can be seen, parallelising an ABS on heterogeneous hardware require model- and hardware-specific implementations. Many common constructs are not easily available for accelerators and will typically need to be substituted with implementations developed from scratch. Such constructs include the container types from the C++ standard template library, libraries for computing output statistics, facilities for input and output or for coupling with other simulation tools. Thorough knowledge of the hardware is needed to achieve peak performance. This puts burdens on the simulationists, who may not be an expert in parallel programming. Users should be provided by a framework like OpenABL with easy-to-use facilities for steering the simulation. However, as discussed in the previous chapters, the current form of the OpenABL framework has its limitations, preventing it from being widely applied to all types of ABSs. In next chapter, we start to build our framework by first extending OpenABL to fill the gaps with the lessons learnt in this chapter in mind.

4 Generating High-Performance Code for Agent-Based Simulations on Heterogeneous Platforms Using OpenABLext

Contents

Substantial parts of this chapter have been published in the Concurrency and Computation: Practice and Experience (CCPE) [205] and the proceedings of the Euro-Par 2019 Parallel Processing Workshops [206].

4.1 Introduction

As shown in the last chapter, a prevalent way to speed up ABS to meet the increasing performance needs is to make use of *accelerators*. However, the studies in Chapter 3 illustrate two challenges to optimising simulation code on the target hardware platforms. Firstly, programming for accelerators requires in-depth knowledge of the hardware. Although employing programming languages such as OpenCL or CUDA abstracts away some hardware details, the programming efforts can still be significant, as displayed in the first feasibility study. The second challenge is that code tailored for one specific hardware type may not be simply used by another type and achieve a similar optimal performance (the second feasibility study). This degrades maintainability as well as portability of the code [59]. The introduction of a new abstraction layer in the form of a domain-specific language can help alleviate this problem. In the context of agent-based simulations, OpenABL [18] 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 parallelised code for CPUs, GPUs, clusters, or cloud environments.

The original OpenABL targeted one specific type of hardware platform, i.e., 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 a high level of hardware utilisation using co-execution [126]. However, once enabled, co-execution poses the additional challenge of determining a suitable combination of hardware devices for execution to maximise performance. Further, 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. Lastly, through the studies in Chapter 3, we identified the necessity of conflict resolution in a heterogeneous setting. However OpenABL does not provide a mechanism for conflict resolution, requiring modellers to manually provide code to detect and resolve situations where multiple agents request the same resources.

In this chapter, we address these limitations by introducing OpenABLext, an extension to the OpenABL language. We improve the OpenABL framework in the following aspects:

- We extend the declaration of simulation environments to feature user-defined types, the possibility to support graph-based simulation spaces, and an efficient neighbour 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 optimise the hardware assignment during co-execution.
- We define an interface that enables the generation of parallel conflict resolution code based on user-specified rules.

The remainder of the chapter is organised as follows: In Section 4.2, we introduce OpenABL fundamentals and discuss related work in the respective fields. We present OpenABLext in detail in Section 4.3 and evaluate its performance in Section 4.4. Section 4.5 gives a summary.

4.2 Background

4.2.1 Related Work

The acceleration of ABS through parallelisation has received wide attention from the research community. A number of CPU-based frameworks such as MASON [11], Repast [207], Swarm [21], or FLAME [208] simplify the process of developing parallel ABS. Variants that exploit CPU-based parallelisation or distributed execution include D-MASON [209] and Repast-HPC [210]. However, these frameworks target CPU-based hardware and require modellers to be knowledgeable in parallel or distributed computing.

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 [211], which is an extension of FLAME. It 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) [15] 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 parallelisable C snippets [212] and the use of code templates [213] have been proposed. Most of these approaches focus on detecting localized sections of the source code that can be parallelised, e.g., 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 [214] and [215]. 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 utilised for better performance. Open Accelerators (OpenACC) [216] and Open Multi-Processing (OpenMP, version 4.0 and above) provide directive-based application programming interfaces to parallelise general sequential code to run on heterogeneous systems. There are also efforts to translate OpenACC code to OpenCL targeting FPGAs [217]. Unlike OpenCL, both OpenACC and OpenMP provide only limited control over low-level performance-critical aspects such as memory access patterns and control flow. Further, as general programming standards, they do not include domain-specific optimisations as our framework does for ABS. DSLs are another group of methods to simplify development for high-performance code generation on heterogeneous hardware platforms [218, 17]. 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 [219, 220, 221] have been proposed. However, these methods usually require substantial offline training. The runtime behaviour 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 [222, 223, 224, 225]. Some works [226, 227, 228] tackle the problem of accelerating numerical algorithms such as matrix multiplication and the fast Fourier transform on heterogeneous systems using data partitioning approaches. In agent-based simulations, however, we observe a strong locality of dependencies, as agents primarily interact with nearby agents, i.e., their neighbours. This can cause additional application-specific overhead when generic data-level partitioning methods are applied to ABS, e.g., by regularly repartitioning data geographically [229]. Therefore, we propose a generalisable 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 [230], or they need to profile the hardware and program offline [231].

To reduce overhead, we propose a heuristic to dynamically trigger re-evaluation of the hardware assignment. Further, 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, 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 [133] or on enumerating the agents involved in conflicts once an update cycle has completed [232]. 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. [151]. 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.

4.2.2 OpenABL

OpenABL is a domain-specific language to describe the behaviour of agent-based simulations and a framework to generate code targeting various execution platforms [18]. 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 4.1. The framework consists of a *frontend* and a *backend*. Listing 4.1 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), agent-based model defined as step functions (keyword `step`, L.9-13), and a main function (keyword `main()`, L.17-19).

```
1 agent Agent { position float2 coordinate; }
2
3 param int num_of_agents = 1000;
4 param int sim_steps = 100;
5 param float env_size = 100;
6
7 environment { max : float2 (env_size) }
8
```

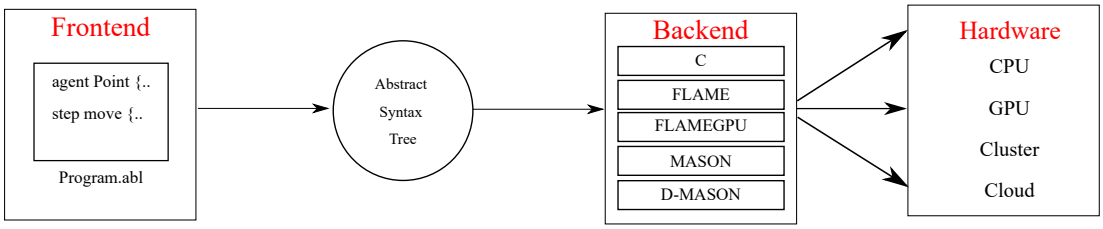


Figure 4.1: An overview of the OpenABL framework.

```
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 4.1: Example OpenABL simulation definition.

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 [208], FLAME GPU [211], MASON [11], and D-MASON [209].

4.3 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: 1) The support of a more extensive environment declaration, featuring user-defined types, the possibility to support graph-based simulation space, and a more efficient neighbour search to achieve efficient memory access. 2) The introduction of an OpenCL backend with specific compilation rules for GPUs and FPGAs as well as support for multi-device co-execution. 3) An online dispatcher that profiles step functions and

assigns them to the most suitable hardware device. 4) A conflict resolution mechanism which automatically detects conflicts and resolves them using a tie-breaking function.

4.3.1 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 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, e.g., 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 [18]: 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 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 [233]. 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 neighbours, 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 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))
```

Given traffic simulation as an example where edges in a graph represent 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 4.2, 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 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 neighbouring cells (Figure 4.3).

Index	0	1	2	3	4	5	6
AgentID	5	3	2	7	1	6	4
Position (LaneID, PositionOnLane)	(1, 4.5)	(1, 15.8)	(1, 35.3)	(2, 9.1)	(2, 38.3)	(3, 0.5)	(3, 62.5)

LaneID	1	2	3
mem_start	0	3	5
mem_end	2	4	6

Figure 4.2: 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.

4.3.2 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

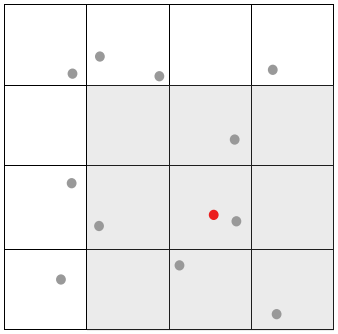


Figure 4.3: In a grid with cell width at least the search radius, the neighbour search of the red agent loads itself and adjacent cells.

of different types, while the existing backends only support single-platform execution. With this new backend, we aim to allow modellers to fully utilise 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, e.g.: `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 Sec. 4.3.2.3.

In the following, we will discuss how the structure of the generated code changes depending on the target device (GPU, FPGA, Co-execution). To this end, we assume Listing 4.1 on page 73 as the input of OpenABLext.

4.3.2.1 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 4.2 and Listing 4.3 show pseudo code for the host and device programs (extraneous details omitted; assumed input OpenABLext syntax as shown in Listing 4.1). 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 initialises the device, allocates the required memory, and initialises the agent state variables as well as the environment.

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 Sec. 4.3.1. 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.

```
1 ... /* function and variable declarations */
2 int main()
3 {
4   initialise(agentArray[LENGTH]);
5   initialise(clEnvironment, clDeviceBuffers);
6   clWriteBuffer(agentArray[LENGTH], clDeviceBuffers);
7
8   for (int i=0;i<NUM_STEPS;i++) {
9     clExecuteNDRangeKernel(compute_kernel_1);
10    clExecuteNDRangeKernel(compute_kernel_2);
11  }
12
13  clReadBuffer(clDeviceBuffers, agentArray[LENGTH]);
14 }
```

Listing 4.2: Host code.

```
1 ... /* function and variable declarations */
2 void stepFunc1(Agent *agent) {
3   for( agents inside the nine surrounding cells ) {...}
4 }
5
6 void stepFunc2(Agent *agent) {...}
7
8 __kernel void compute_kernel_1(__global Agent *agentArray)
9 {
10  if ( get_global_id(0) >= LENGTH ) return;
11  stepFunc1(&agentArray[id]);
```

```
12 }
13
14 __kernel void compute_kernel_2(__global Agent *agentArray)
15 {
16   if ( get_global_id(0) >= LENGTH ) return;
17   stepFunc2(&agentArray[id]);
18 }
```

Listing 4.3: Device caode.

4.3.2.2 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 4.4) for initiating the OpenCL and simulation environment and a device program (Listing 4.5) with one `compute_kernel` function are 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 2.2.4. 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 [18]. The read and write buffers are therefore swapped after each step function as shown in Listing 4.5. The `on` or `near` statement is replaced by the efficient neighbour search method. Radix sort, rather than bitonic sort is used to sort the agents, which has been shown to perform well on FPGAs [234].

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, e.g., looping through agents to accumulate a value, can still cause a large II. Further, 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.

```
1 ... /* function and variable declarations */
2 int main()
3 {
4   initialise(agentArray[LENGTH]);
5   initialise(clEnvironment, clDeviceBuffers);
6   clWriteBuffer(agentArray[LENGTH], clDeviceBuffers);
7   clExecuteKernel(compute_kernel);
8   clReadBuffer(clDeviceBuffers, agentArray[LENGTH]);
9 }
```

Listing 4.4: Host code.

```
1 ... /* function and variable declarations */
2 void stepFunc1(Agent *agentRead, Agent *agentWrite) {
3   #pragma pipeline_loop
4   for( agents inside the nine surrounding cells )
5   {...}
6 }
7 void stepFunc2(Agent *agent) {...}
8
9 __kernel void compute_kernel(__global Agent *agentArrayReadBuffer,
      __global Agent *agentArrayWriteBuffer)
10 {
11   for (int i = 0 ; i < NUM_STEPS ; i++)
12   {
13     for (int j = 0 ; j < NUM_AGENTS ; j++)
14       stepFunc1(&agentArrayReadBuffer[j], &agentArrayWriteBuffer[j]);
15     agentArrayReadBuffer = agentArrayWriteBuffer;
16     for (int j = 0 ; j < NUM_AGENTS ; j++)
17       stepFunc2(&agentArrayReadBuffer[j], &agentArrayWriteBuffer[j]);
18   }
19   agentArrayReadBuffer = agentArrayWriteBuffer;
20 }
```

Listing 4.5: Host code.

4.3.2.3 Code Generation for Co-execution

In a co-execution setting, the output of the OpenCL backend consists of a host program (Listing 4.6) and multiple device programs, one for each available device (Listing 4.7 and Listing 4.8). 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

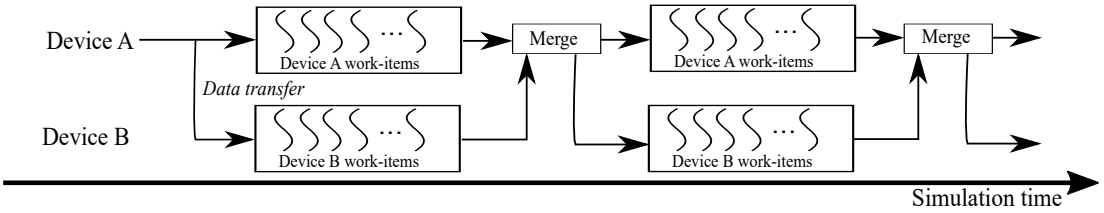


Figure 4.4: Co-execution on device A (acts as host) and B. Each work-item of device A and device B processes the step functions assigned to them in parallel. After that, the data is merged at the host and assigned to device A and B respectively for the next step.

one agent's step function. This means that parallelisation 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 4.4).

The host program orchestrates the data exchange between devices. After each simulation iteration, data processed by different devices is 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.

```
1 ... /* function and variable declarations */
2 int main()
3 {
4   initialise(agentArray[LENGTH]);
5   initialise(clEnvironment, clDeviceBuffersDev0, clDeviceBuffersDev1)
      ;
6   clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev0);
```

```
7   clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev1);
8   for (int i = 0 ; i < NUM_STEPS ; i++) {
9     clExecuteNDRangeKernel(compute_kernel_dev0);
10    clExecuteNDRangeKernel(compute_kernel_dev1);
11    clReadBuffer(clDeviceBuffersDev0, agentArray0[LENGTH]);
12    clReadBuffer(clDeviceBuffersDev1, agentArray1[LENGTH]);
13    agentArray[LENGTH] = merge(agentArray0[LENGTH], agentArray1[
      LENGTH]);
14    clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev0);
15    clWriteBuffer(agentArray[LENGTH], clDeviceBuffersDev1);
16  }
17 }
```

Listing 4.6: Host code.

```
1 ... /* function and variable declarations */
2 void stepFunc1(Agent *agent) {
3   for(agents inside the 9 surrounding cells) {...}
4 }
5
6 __kernel void compute_kernel_dev0(__global Agent *agentArray)
7 {
8   if ( get_global_id(0) >= LENGTH ) return;
9   stepFunc1(&agentArray[id]);
10 }
```

Listing 4.7: Code for device ID 0.

```
1 ...  /* function and variable declarations */
2 void stepFunc2(Agent *agent) {...}
3
4 __kernel void compute_kernel_dev1(__global Agent *agentArray)
5 {
6   if ( get_global_id(0) >= LENGTH ) return;
7   stepFunc2(&agentArray[id]);
8 }
```

Listing 4.8: Code for device ID 1.

4.3.2.4 Hardware Selection with Online Dispatcher

While co-execution can lead to better hardware utilisation, 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 (i.e., 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 agent-based simulation, most of the workload occurs in the main simulation body which iteratively calls the step functions. In these functions, an agent's behaviour and its interaction with near-by agents is defined. The computing intensity of step functions therefore depends on two factors: the complexity of the behaviour models and the amount of agent interactions. While the complexity of a behavioural model is fixed, the distribution of agents in the simulation space varies over the course of a simulation, resulting in a non-constant 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 Fig. 4.5.

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 is 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,

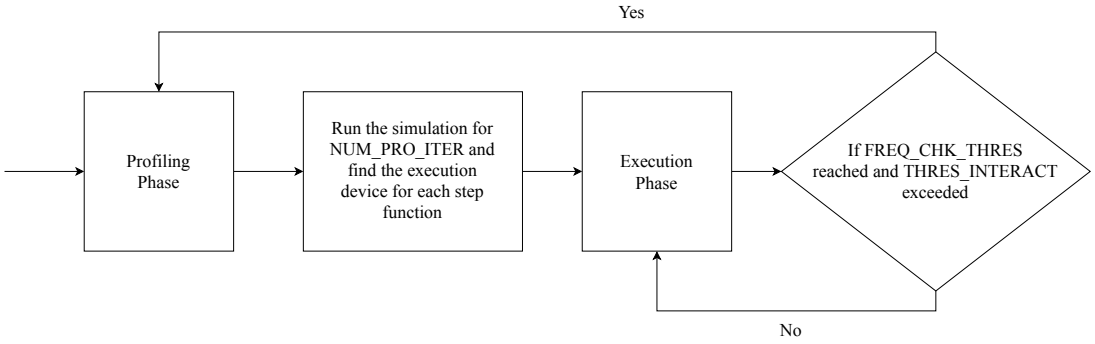


Figure 4.5: Workflow of the online dispatcher.

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` is distributed among all available devices and the profiling starts for another `NUM_PRO_ITER` iterations. Again, the hardware device which 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.

4.3.3 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 [235]. 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, this process repeats until no further conflicts occur. A number of approaches for conflict resolution on parallel hardware have been studied in [151]. 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 4.3.1, with `search_radius` specifying the search radius. `tie_breaking` has different meanings for graph-based simulations and for 2D/3D 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 and returns `false` if there is 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 4.3.1. 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 4.3.1). 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,

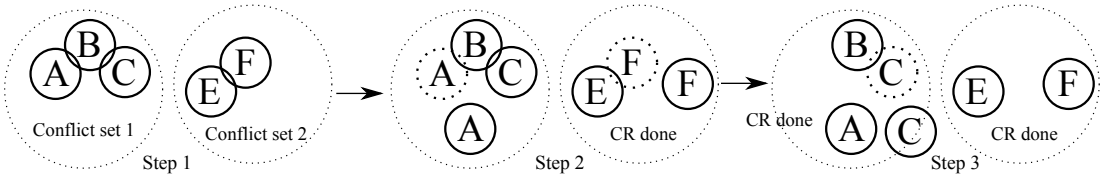
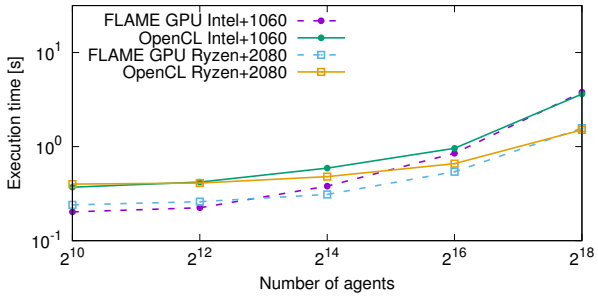
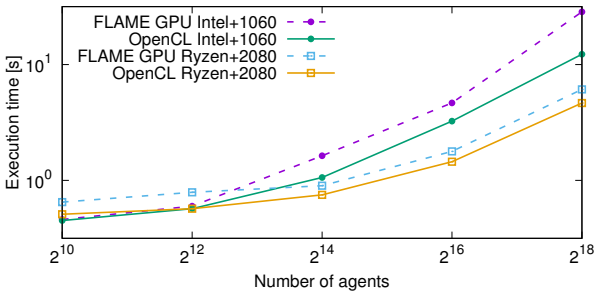


Figure 4.6: Illustration of Conflict Resolution (CR) for 2D/3D simulations.



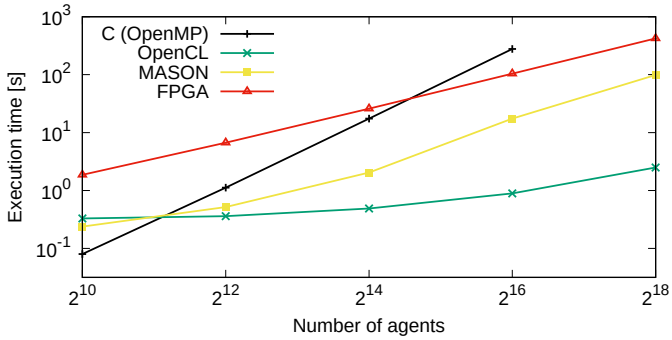
(a) Circle (evenly distributed agents).



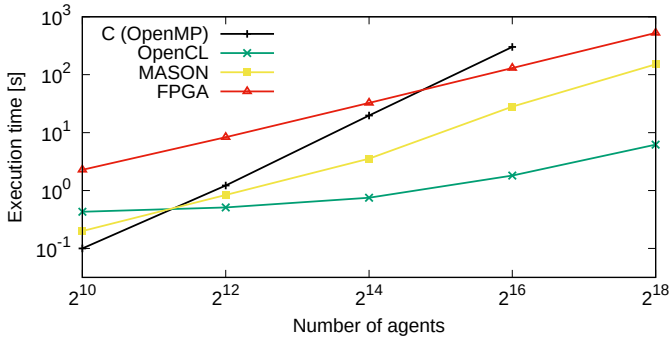
(b) Circle (high agent density).

Figure 4.7: Comparison of the OpenCL backend with the FLAME GPU using the Circle application.

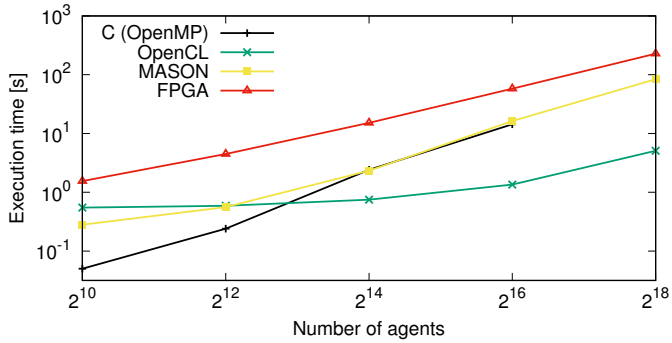
agent C is also added to the set. The result of the conflict detection step is a list of non-overlapping conflict sets (Fig. 4.6, Step 1) on each of which the conflict resolution can be carried out simultaneously. In each set, a randomly selected agent is rolled back to its previous state and the remaining agents inside the set are again checked for conflicts (Fig. 4.6, Step 2). If no more conflicts exist, the processing of this conflict set ends, otherwise, another randomly selected agent is rolled back (Fig. 4.6, Step 3). This repeats until there is only one agent left in this conflict set.



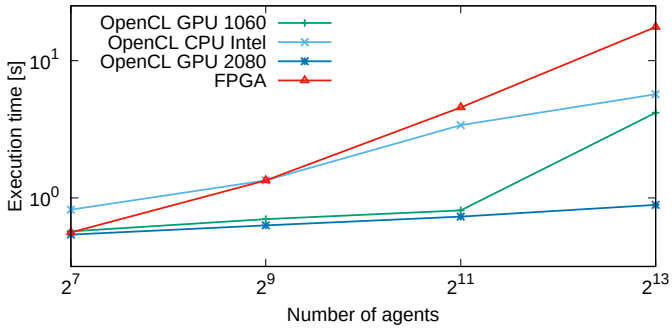
(a) Game-of-Life



(b) Sugarscape



(c) Ants



(d) Traffic

Figure 4.8: Performance of the OpenCL backend, compared with the C, MASON, FPGA (where applicable).

Table 4.1: Configuration of the tested platforms

	Platform 1	Platform 2	Platform 3
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 Ultra-Scale+ 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

4.4 Evaluation

In this section, we use three typical hardware platforms (Table 4.1) 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, i.e., C with OpenMP, FLAME GPU, and MASON. For this, we evaluate a range of different simulation models: *Circle*, a benchmark for accessing neighbours within a certain radius provided in [236]; Conway's *Game of Life* [156], a cellular-automata based simulation of the evolution of agents based on the neighbouring agents' states; *Sugarscape* [237], a social model where each agent searches for nearby resources (a piece of sugar) to metabolise; and *Ants* [238], which simulates the foraging behaviour 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 utilise two additional simulation models: *Traffic*, a traffic simulation based on the implementation in [171] and *Crowd*, which simulates the flocking behaviour of people following leaders (fire wardens) during a building evacuation [239]. 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 neighbour search queries, indicated by the keywords **on** or **near** in the step functions, are replaced with the respective efficient neighbour 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.

4.4.1 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 288MHz. For the other 2D/3D simulations 6-17% of LUTs and at most 6% of other resources were utilised and the FPGA ran at a frequency of 310-318MHz. We ran all simulations at least 5 times for 100 time steps to enable comparison with the existing results in [18]. 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 4.7. 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 4.7a), 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 points to be closer together (Figure 4.7b). 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 backend as well as the MASON backend. Furthermore, we show how feasibly agent-based simulations can be executed

on an FPGA using OpenABLext. Our findings are summarised in Figure 4.8. First, we demonstrate the performance of three simulation models with a 2D/3D environment (Fig. 4.8a to 4.8c). Unsurprisingly, the GPU performed best for higher agent numbers, as the relative initialisation overhead decreased and the workload was sufficient to benefit from the massive parallelisation. We confirm the findings from [18] 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 (Fig. 4.8a to 4.8b). This is caused by the efficient neighbour search and the pipelining parallelism the FPGA implements. As mentioned in Section 4.3.2.2, we employ a double-buffering design where we swap read/write buffers after each step function. The three step functions in the Ants application (Fig. 4.8c) are computationally lightweight so that the FPGA is not fully utilised while they cause more memory-intensive buffer swaps, slowing down the performance of the FPGA. Since to our knowledge these measurements represent the first results on FPGA-accelerated agent-based simulations from high-level model specifications, the performance results demonstrate the promise of FPGAs in this context.

As described above, OpenABLext enables modellers to utilise graph-based simulation environments. As a proof of concept, we developed a traffic simulation based on our previous implementation [171]. Our results are shown in Figure 4.8d. 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 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.

4.4.2 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 specify 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% of the overall runtime on a CPU and 9% of the overall runtime on a GPU 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.

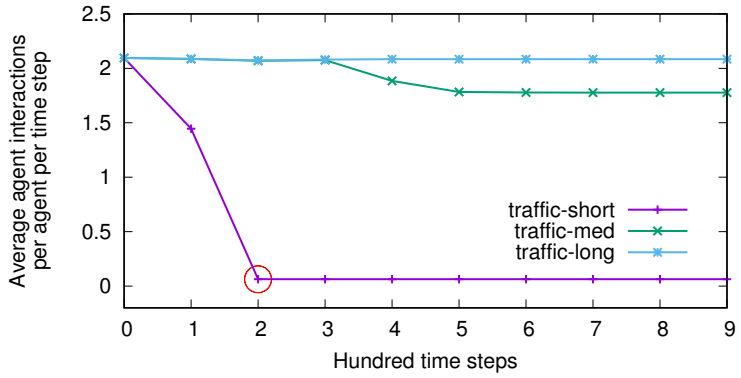
4.4.3 Online dispatcher

Lastly, 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` 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 Fig. 4.9a, 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 plater in the simulation. 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 1,000 steps.

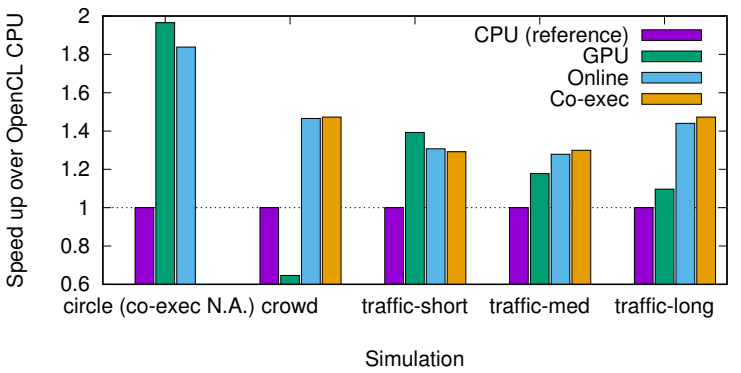
Naturally, co-execution requires at least two step functions to be distributed among different devices. However, the online dispatcher can still be employed in cases where there is only one step function in order to find the most suitable device. Fig. 4.9b shows the speed up over running the generated OpenCL code on CPU. For the circle application which 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 to 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 parallelisation 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 recognises the decrease of agent interactions, it switches from co-execution to a pure GPU execution (the switch point is marked in red in Figure 4.9a), resulting in a total overhead of about 7%.



(a) Average agent interactions of the lane changing model over the simulation course in different trip-length settings. The red cycle marks the switch point in the traffic-short setting.



(b) Speed-ups of different execution schemes normalised to the performance of CPU OpenCL. **Co-exec** shows the manually picked hardware assignment achieves the best performance among all assignment combinations.

Figure 4.9: Evaluation of the proposed online dispatcher on Platform 1.

4.5 Summary

In this chapter, 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 utilises an efficient neighbour 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 agent-based simulation is a promising approach as our experiments exhibited shorter execution times for scenarios with a larger agent count compared to the OpenMP backend. The main reason for this is the more efficient neighbour search and the pipeline parallelism of the FPGA. Lastly, 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.

The online dispatcher assigns step functions to hardware. However, it assumes no dependencies among step functions. Two data-dependent step functions could be assigned to different hardware and executed at the same time, causing simulation outputs to deviate from a sequential run. To prevent this situation, the proposed framework should be able to identify such data dependencies while assigning the hardware. Ideally, to not put burdens back on the users, this detection process should be carried out without much user invention. Therefore, in the next chapter, we propose OptCL, a middleware that can be directly plugged into OpenABLext to automatically detect data dependencies.

5 Squeezing more Performance - Enhancing Co-execution Capability with OptCL

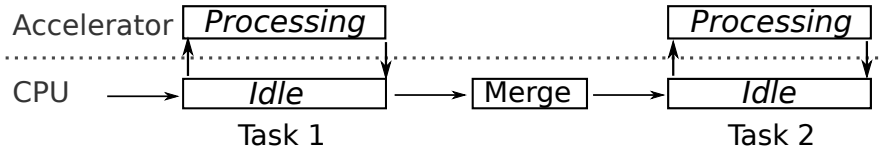
Contents

Substantial parts of this chapter have been published in the proceedings of the 2021 International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 21").

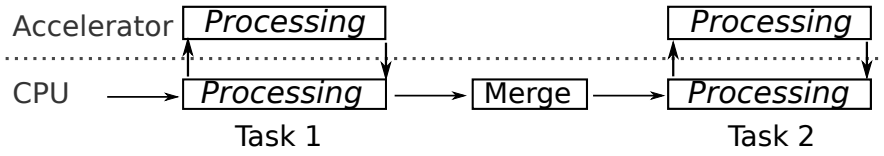
5.1 Introduction

With OpenABLext introduced in Chapter 4, simulationists are able to generate high-performance ABS code targeting heterogeneous hardware settings. However, the problem remains of how to efficiently map simulation models, which can display versatile workload patterns, to their best suitable hardware. Such mapping is not trivial as demonstrated in the studies in Chapter 3. Thorough knowledge of the hardware is again required to make optimal mapping of workloads to accelerators, limiting the benefits of OpenABLext. The online dispatcher introduced in the OpenABLext framework partially solves this problem by automatically choosing the best suitable hardware for each step function at run time and enabling co-execution. However, the dispatcher does not consider possible data dependencies among step functions.

To tackle this problem, we propose a middleware called OptCL (Optimise performance targeting high-performance domain-specifiC Languages) to enhance the co-execution ability of OpenABLext. The middleware enables co-execution determined through data dependency analysis and performance predictions on the available hardware. OptCL is based on Clang, operating on Abstract Syntax Tree (AST) IR level where the OpenABLext backend works on as well, making it possible to be seamlessly integrated with OpenABLext. Further, since AST IR is language independent, the middleware can also benefit a wide range of C-based High-Performance Domain Specific Language (HPDSL) similar to OpenABLext. It can complement an original HPDSL compiler, enabling it to also work for closed-source HPDSLs, provided that the HPDSL can output IR of OpenCL kernels in the shape of Standard Portable Intermediate Representation (SPIR), a binary OpenCL IR for CPUs and AMD GPUs or Parallel Thread Execution (PTX) for NVIDIA GPUs.



(a) Offload mode. CPU is idle while accelerator is processing.



(b) Co-execution mode. CPU and accelerator are processing simultaneously.

Figure 5.1: Offload mode vs co-execution mode.

The remainder of this chapter is organised as follow: In Section 5.2, we present background and an overview of related work. In Section 5.3, we describe the OptCL middleware in detail. We present our case studies and evaluate the performance of OptCL in Section 5.4. Section 5.5 concludes the chapter.

5.2 Background and Related Work

5.2.1 OpenABLext

The workflow of OpenABLext overlaps with the middleware. The middleware can be completely integrated into the OpenABLext compilation flow to generate high-performance OpenCL code.

5.2.2 SYCL

```
1  gpu_selector device_selector;
2  queue deviceQueue(device_selector);
3
4  buffer<float, 1> dev_a(a, range<1>(N));
5  buffer<float, 1> dev_b(b, range<1>(N));
6  buffer<float, 1> dev_c(c, range<1>(N));
7
8  deviceQueue.submit([&] (handler& ch) {
9    accessor a = dev_a.get_access<mode::read>(ch);
10   accessor b = dev_b.get_access<mode::read>(ch);
11   accessor c = dev_c.get_access<mode::write>(ch);
12   auto kernel = [=](id<1> wid) {
13     c[wid] = a[wid] + b[wid]; }
14 });
```

Listing 5.1: An example of SYCL code

Other than OpenABLext, we illustrate the versatility of OptCL by plugging it into another DSL called SYCL. SYCL is a specification developed by Khronos targeting heterogeneous hardware platforms. It allows code portability to enable the same code running on versatile accelerators such as GPUs, FPGAs as well as CPUs. SYCL abstracts away from hardware details, enabling developers to code parallel programmes using normal C++ style. The backend of SYCL varies from implementations which is usually but not limited to OpenCL.

SYCL utilises a so-called single-source concept, meaning one source file consolidates both host and device programs. Listing 5.1 illustrates an example to sum up two N-dimensional vectors in parallel using SYCL. Users are required to first select a device to run the kernels on (L.1-2), followed by allocating memory space on the device-side (L.4-6). L.8-14 is a special function marked by the key word *submit* defining a kernel in the form of a lambda expression by first declaring the inputs and outputs using a data type called *accessor*. L.12-13 implements the logic of a kernel. Many SYCL implementations also support heterogeneous platforms by providing facilities such as asynchronous scheduling of kernels by overlapping data transfer and computation.

ComputeCpp is a commercial implementation of SYCL (a free community edition is also available) [240]. Following the SYCL specification, ComputeCpp consists of two parts: a device compiler and a run-time library, both of which are closed source. So far, only AMD GPUs and x86/ARM CPUs (as accelerators) are fully supported. The support for NVIDIA GPUs was experimental (and was superseded in the most recent version).

By inputting SYCL code, ComputeCpp can output SPIR (CPU and AMD GPU) or PTX (NVIDIA GPU), and an integration file to be loaded by the ComputeCpp run-time. During compilation, users have to choose one IR (SPIR or PTX) to output. Therefore, NVIDIA GPUs cannot be used together with other accelerators in a multiple-device environment.

Although ComputeCpp supports offloading to multiple devices, it leaves the assignment of individual kernels to hardware to the user. Further, as ComputeCpp focuses more on portability than performance, it was reported that ComputeCpp did not fully utilise the power of OpenCL [241]. In this chapter, we will demonstrate that by plugging OptCL into ComputeCpp, the performance of the generated OpenCL code can indeed be improved, both through reducing the kernel invocation overhead and co-execution. OptCL reconstructs an OpenCL program that supports a mix use of PTX and SPIR binaries. As a side effect, we also re-enable using NVIDIA GPUs with ComputeCpp, and more importantly enable using NVIDIA GPUs along with other accelerators which was not supported in the original ComputeCpp.

5.2.3 Abstract Syntax Tree (AST)

AST are a widely used tree representation to display the syntactic structure of a program. An example of the AST generated from an if-else clause is depicted in Figure 5.2. Compared to raw code, an AST presents a clean structure omitting non-essential syntactic elements such as blank spaces or punctuation. An AST records the position of

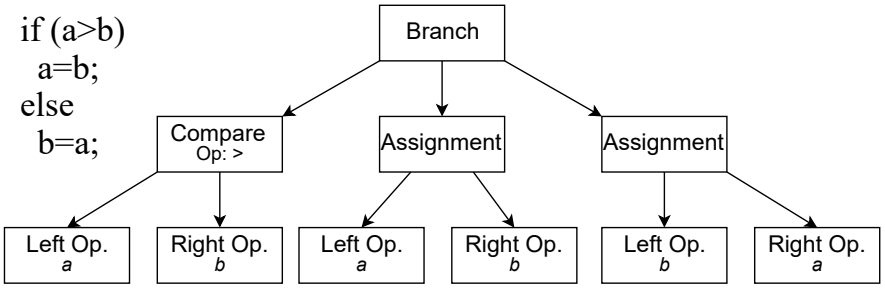


Figure 5.2: An example AST generated from an if-else clause.

each syntactic element, which allows analysis tools to determine, e.g., the scope and the read and write dependencies of variables.

We are mainly interested in data dependencies between High Performance Regions (HPRs), defined as the code regions compiled to OpenCL kernels. Therefore, instead of generating an AST for the entire program, we generate ASTs only for those HPRs. Data dependencies are then identified by analysing these ASTs. HPRs possessing no data dependencies are candidates for co-execution.

5.2.4 Related work

Previous efforts parallelise sequential code using a set of pre-defined rewrite rules [243], code templates [244] or special syntax for loops [225, 245], enabling translation to programs in OpenCL, CUDA, or Threading Building Blocks code. These frameworks pursue a goal of generating parallel programs from sequential representations similar to HPDSLs. Necessary structural amendments have to be made for existing HPDSL programs to use these frameworks, while our OptCL aims to parallelise existing programs without any changes required.

Several works [130, 246, 247] automatically detect parallelisable loops in sequential representations and translate the loops to OpenCL kernels. The existing approaches targeting OpenCL follow a similar workflow as OptCL: a sequential program is first converted into an IR. Data parallel loops or static control regions are detected in the IR and translated to OpenCL kernels. These existing works assume that parallelisable regions are explicitly stated as loops in the sequential code. However, this assumption may not hold for HPDSL programs. As HPDSLs abstract away implementation details, they usually only require users to code loop bodies, i.e., the HPRs. The iterative behaviour is handled internally by the HPDSL runtime (and eventually by the OpenCL runtime). Further, none of these approaches consider co-execution opportunities. In contrast to the existing works on automatic parallelisation, our approach

relies on HPDSL-level code segments to identify regions for co-execution. The combination of existing automatic parallelisation methods with our approach for mapping computations to heterogeneous hardware is an interesting avenue for future work.

There exist a handful of DSLs providing different levels of native co-execution support. Habanero-C (HC) [248] features shared virtual memory and smart data layout to achieve performance portability on CPU-GPU systems. CnC-HC [249], which maps the Concurrent Collections (CnC) model to the HC runtime, extends the supported hardware to include FPGAs. HC and CnC-HC both use work-stealing approaches to achieve load-balancing between CPU and accelerators. Unlike our work, which automatically determines data dependencies, in HC, the data dependencies are ensured by the users based on *async* and *finish* constructs. Performance on the individual piece of hardware is also estimated based on a user-specified machine description. The work reported in [250] extends the PetaBricks language which allows users to specify multiple algorithmic paths for the same input and output. The compiler then chooses the path leading to the best performance given hardware settings determined using an evolutionary mechanism. However, users still need to produce parallel code explicitly.

Two co-execution schemes are extensively used in the literature: data partitioning and task partitioning. Task partitioning has been carried out offline using performance analytical models [251] or using machine learning-based approaches [219, 220, 252, 253]. In the evaluation section, we will conduct a comparison study between a representative offline machine learning-based approach and a sampling-based approach similar to [205] but applicable beyond the ABS domain. Other works [225, 254, 255] propose novel adaptive scheduling mechanisms to partition the workload at the data level aiming at achieving load-balancing or power-saving. Many frameworks built based on these designs [224, 222, 223] can operate directly on OpenCL kernels. They typically start with a small portion of workload assigned to CPUs and the remainder to the accelerators (or vice versa). A balancing phase then incrementally balances the workload assigned to the CPUs and to the accelerators until convergence is reached. If the workload is irregular, this balancing phase can be re-triggered if certain imbalance criteria are met. Data layout and transfer among different devices are also dealt with by the frameworks automatically.

A study comparing data partitioning and task partitioning schemes on a CPU-FPGA system is carried out in [256]. The authors concluded that both schemes can be beneficial. OptCL partitions the workload at the task level, which is a natural choice following the specification of HPDSLs, as each HPR will be typically translated to one OpenCL kernel. Data partitioning will only be used in a special case as an additional

optimisation (cf. Sec. 5.3.4.3). Future work could explore the combination of task and data partitioning in OptCL by employing one of the aforementioned designs.

5.3 The OptCL Middleware

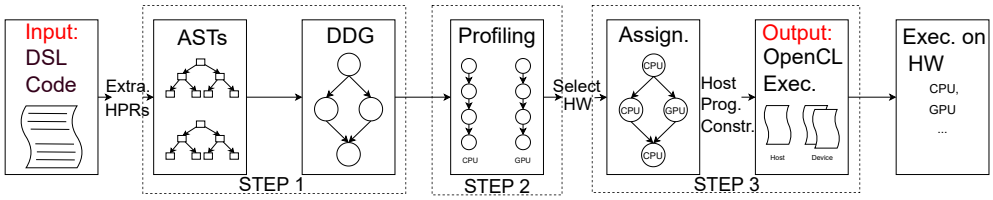


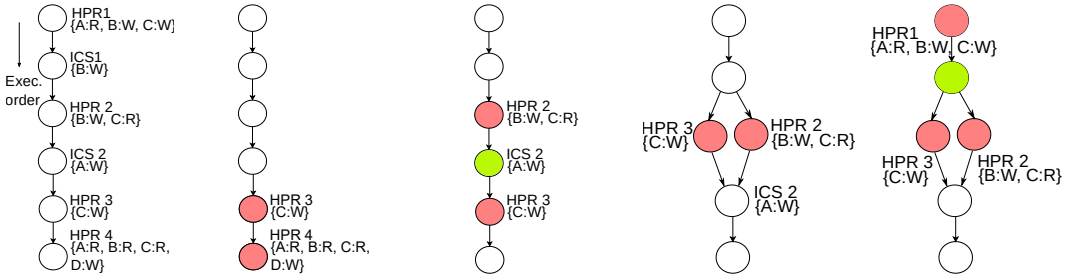
Figure 5.3: An overview of OptCL.

An overview of OptCL is given in Figure 5.3. OptCL generates OpenCL code in three steps: 1) data dependency analysis, 2) profiling, 3) hardware assignment, and code reconstruction. In the first step, OptCL identifies the sub-AST containing HPRs from the AST generated for the entire program. The sub-AST is further split up into smaller ASTs, each representing an HPR or the code region between two HPRs. A Data Dependency Graph (DDG) derived from those small ASTs identifies the data dependencies and distinguishes HPRs that are free of interdependencies, which can thus be co-executed. The second step profiles the kernels generated out of the HPRs on the available hardware. Based on the profiling results, the third step decides on the execution scheme and reconstructs the program accordingly.

Users are required to specify two inputs to OptCL at installation time: the keyword used to annotate the start of HPRs (HPR keyword), and the data type used to define in- and output (in other words, to allocate memory on the devices) of those HPRs (e.g., *accessor* in SYCL or *agent* in OpenABLext), which is referred to as device variable keyword. Notably, this setup is done once per language and is application-agnostic. After this initial set up, OptCL can run in a fully automated way. In what follows, we will discuss each step in detail.

5.3.1 Step 1: Data Dependency Analysis

Common compilers also produce ASTs during compilation, from which data dependencies for the entire code base are deduced. OptCL only focuses on a portion of the entire AST i.e. the sub-AST around the HPRs, and eliminates code snippets that are not targeted for parallel execution, e.g., reading inputs, initialisation of variables, etc. This can reduce evaluation complexity, tailored to the structure of HPDSL code.



(a) Original DDG **(b)** Processing HPR3&4 **(c)** Processing HPR2&3 w. ICS2 **(d)** Parallelising HPR2&3 **(e)** Processing HPR 1, 2&3 w. ICS1

Figure 5.4: DAG generation.

Firstly, OptCL triggers a partial compilation of the input HPDSL code. This step assembles code if multiple source files are available and sorts HPRs according to the required execution order. The partial compilation stops when the AST for the entire program is generated. At a glance, this step seems to be redundant with the HPDSL's own compilation step. However, it is essential to enable OptCL on closed-source HPDSLs, since OptCL does not have access to the intermediate compiling stages of a closed-source compiler. For open source compilers, this partial compilation can be embedded into the real compilation process.

```
1  HPR1(global_id i){
2    B[i] = A[i] + 1;
3    C[i] = A[i] * 5;
4  }//ϵ = {A:R, B:W, C:W}
5  //ICS1
6  for (each element B[i] in B) {
7    B[i]++;
8  } //ϵ = {B:W}
9  HPR2(global_id i){
10   B[i] = B[i] + C[i];
11 }//ϵ = {B:W, C:R}
12 //ICS2
13 for (each element A[i] in A) {
14   A[i]++;
15 } //ϵ = {A:W}
16 HPR3(global_id i){
17   C[i]++;
18 }//ϵ = {C:W}
19 HPR4(global_id i){
20   D[i] = A[i] + B[i] + C[i];
```

```
21    }//ϵ = {A:R, B:R, C:R, D:W}
```

Listing 5.2: Pseudo-code of a 4-HPR HPDSL program

The AST of the entire program is first traversed to locate HPRs and usage of device variables. The traversal is implemented based on the *RecursiveASTVisitor* class of Clang. HPRs are identified in the raw code using the HPR keyword inputted by the users. The scope of each HPR is to the end of the function (e.g., for OpenABLext) or lambda expression (e.g., for SYCL) following the keyword. Device variables are identified by the device variable keyword (e.g. the Device_Var keyword in Listing 5.2) for both their device part (variables which are mapped to the device's memory space) and their host counterpart (variables declared on the host to initialise device variables or to store the results read from the device) as well as their aliases determined by the clang alias analysis. There can be code snippets between two consequent HPRs where the host counterpart of a device variable can be amended. Data dependency can thus also occur, preventing the neighbouring HPRs from parallel execution. Therefore, during data dependency analysis, those code snippets, further referred to as In-between Code Snippets (ICSs), should also be taken into consideration.

While traversing the AST of the entire program, it can be identified whether an HPR or an ICS is within a loop. This information is recorded to be applied later in Step 1 as well as in Step 3. With all HPRs detected, the key code regions encapsulating all HPRs and ICSs can be identified, and its corresponding sub-AST is extracted from the overall AST. The sub-AST is further split into small ASTs, each AST representing an HPR or an ICS.

Within one small AST, we trace read and write operations on the device variable for both their device part and host counterpart. Each AST is traversed to check for three basic patterns and combinations thereof that may indicate a read or write operation:

- Assignment statement ($A = B$ or $A = \text{func}(B, C)$): the former case entails a write operation on A and a read operation on B. The latter entails a write operation on A. As we do not analyse the function func(), we conservatively assume that func() may cause both read and write on B and C.

- Binary operation ($A = B + C$): a binary operation incurs a write operation on A and read operations on B and C respectively.

- Unary operation (A++ or !A): for unary operations, we distinguish between increment (A++ or ++A) and decrement (A-- or --A) on the one hand, which

incur both read and write operations on A; and other unary operations which incur only read operations on A on the other hand.

The results of the above checks are summarised by an expression: $\epsilon = \{A : R|W, B : R|W, C : R|W, ..\}$ where $A, B, C..$ are the device variables touched in this HPR and $R|W$ indicates either Read(R), if all operations on this device variable in this HPR are read operations, or Write(W) if at least one write operation on this device variable exists in this HPR with W always overrides R.

Listing 5.2 illustrates an example HPDSL program with four HPRs and two ICSs. HPRs or ICSs within a loop are treated as a single node in DDG, as the data dependency remains unchanged in each iteration. The generation of DDG starts with assuming dependencies everywhere, resulting in a DDG representing sequential execution (Figure 5.4a). OptCL then tries to parallelise as many HPR nodes as possible. ICS nodes are not considered for parallelisation, as they by user's design have to be executed sequentially on the host. However, they are taken into count when evaluating their surrounding HPRs, as they play an essential role to determine the data dependencies and can influence the co-execution possibility. The attempt begins with the last HPR node (e.g., HPR4 in Figure 5.4a) by evaluating the dependencies between its preceding node and itself. Two consecutive HPR nodes can touch the same device variable, for example, HPR1 and HPR2 touch variable A, B and C, yielding four types of dependencies: Read-After-Read (RAR) , Write-After-Read (WAR), Read-After-Write (RAW) and Write-After-Write (WAW). This can be determined as the element-wise union of the ϵ expressions. For example, an element-wise union of the *epsilon* expressions of HPR1 ($\epsilon = \{A : R, B : W, C : W\}$) and HPR2 ($\epsilon = \{B : W, C : R\}$) is $\{B : WAW, C : RAW\}$.

Two or more consecutive HPR nodes can be parallelised if: 1) There is no ICS node in between carrying write dependencies of the device variables that are used in the latter node in the DDG graph; *and* 2) They touch a disjunct set of device variables *or* all device variables they touch have either RAR or WAR dependency.

While RAR intuitively causes no dependency, WAR also results in no dependency. This is because when executing in parallel on different devices, each device keeps a local copy of the variable. Modifying the local copy on one device has no effect on other devices. For instance, as shown in Listing 5.2, when co-executing HPR2 and HPR3 on different devices, HPR2 and HPR3 each keep a local copy of C which reflects the values after HPR1. The write operations on C (e.g., $C[i]{+}{+}$) in HPR3 do not change the values of C in HPR2.

As illustrated in Figure 5.4b, HPR4 cannot be parallelised with HPR3 due to RAW dependency on C (HPR3 writes to C and then HPR4 reads from C). Node HPR3 can be parallelised with node HPR2 (Figure 5.4c and 5.4d), because only WAR dependency exists (HPR2 reads from C and HPR3 writes to C) and ICS2 carrying write-dependency to A while A is not used in the latter node i.e. node HPR3. Node HPR1 cannot be parallelised with node HPR2 and node HPR3 (Figure 5.4e) for two reasons. First, there is RAW dependency on device variable B and C. Second, ICS1 modifies variable B which is overwritten in HPR2.

This parallelisation process traverses a DDG iteratively until no more nodes can be parallelised.

5.3.2 Step 2:Profiling

We propose two design options for the profiling stage: a sampling-based profiling approach executing a small portion of the application to estimate the performance of the whole program and an offline mechanism using a machine-learning based approach to predict the performance. The usability and accuracy of two different approaches will be compared in Section 5.4.

Sampling-based Profiling Device programs encapsulating OpenCL kernels (e.g. PTX or SPIR) are generated using HPDSL's own compiler. OptCL produces one temporary host program per device (including CPUs), assuming a sequential execution order and a single device environment. These temporary host programs are functionally similar to the ones generated by the original HPDSL compiler with extra utilities to measure the execution time of individual kernels, including both kernel invocation and data transfers (for using CPUs as accelerators, only kernel invocation time is counted, as there is no data transfer). Further, after each kernel invocation, the data is transferred back to the host in order to measure the data transfer overhead.

Each kernel is profiled with the real data with a timer or the time used for one full kernel invocation, whichever takes longer. In the case that multiple rounds of kernel invocations can be done within the time limit set by the timer, the throughput is recorded. If one kernel invocation takes longer than the timer, the execution time is used to indicate the performance.

Offline Profiling The offline approach implements an established performance prediction model using so-called architecture-independent features introduced in work [219]. Each OpenCL kernel is abstracted as a series of architecture-independent features ranging from opcode counts to branch deviation entropies. The Architecture Independent Workload Characterization (AIWC) tool [219] developed based on the idea is employed

to characterise the OpenCL kernels generated by the HPDSL's compiler. The AIWC tool acts as a plugin to an OpenCL simulator, Oclgrind, which has been widely used to debug OpenCL kernels [257]. Oclgrind simulates the execution of OpenCL kernels on the IR level, and therefore it is hardware independent. During the simulation, Oclgrind generates events, e.g., on a conditional branch, based on the encountered IR instructions. AIWC acts as an event handler which counts the appearance of certain events. When a full kernel invocation is completed, AIWC conducts a statistical summary of the counters and produces metrics. These features are later fed into a prediction model based on a Random Forest [258] to predict the execution time on a CPU, a GPU or any other accelerator. By iteratively partitioning the data, the algorithm builds decision trees. Then, the forest, which is an ensemble of decision trees, provides the prediction based on the mean among the trees, providing a more reliable and robust prediction result. The experiments conducted by the authors in [219] showed an average of 1.2% deviation between the predicted execution time and measured time.

5.3.3 Step 3: Hardware Assignment and Program Reconstruction

In this step, a hardware assignment is determined with the aim of maximising performance. It is based on the results from the profiling stage, following these rules:

- If the throughput/execution time of a kernel on one device outperforms other devices, the kernel is assigned to this winning device.
- When co-execution is possible as indicated by the DDG: 1) If the co-executed kernels have different winning devices, they are assigned to their respective winning devices. 2) If some of them share a winning device 1, the second best device 2 of a kernel is chosen, given the total execution time of these kernels on device 1 is larger than co-execution on device 1 and 2 (cf. Figure 5.5).

Based on the hardware assignment, OptCL reconstructs the HPDSL program. OptCL employs the Clang rewrite method to replace the HPRs with their OpenCL kernel invocations, together with the necessary initialisation and data transfer. Other parts such as ICSs and I/Os are copied over from the original HPDSL program. During the reconstruction, a couple of measures are taken to reduce the kernel invocation overhead: firstly, all OpenCL kernels are compiled only once prior to the start of the first HPR. The compiled binaries stay in memory and are used when needed. Secondly, we optimise for the situation where HPRs reside in a loop. This can cause data transfer redundancies if every single call to the HPR in a loop iteration is treated as a new

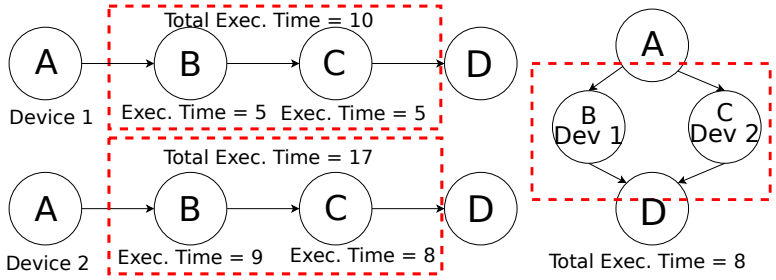


Figure 5.5: An example showing that co-execution may still lead to a performance gain even if some kernels are assigned to sub-optimal hardware. T: execution time.

OpenCL kernel, which entails bi-directional data transfer to/from the host. The information of whether an HPR is within a loop is collected in Step 1. In case all HPRs residing in the same loop are assigned to the same device and there is no ICS in the loop, the data transfer between host and device is extracted and executed outside the loop to eliminate redundant data transfer.

In a multi-device execution environment, different devices usually do not share the same memory space. OptCL also inserts code that is responsible for allocating memory on the respective devices. In the case where a shift of devices is required between two consecutive kernels, a data path is built in between. There are special cases where data exchange can be avoided, e.g. when an Accelerated Processing Unit (APU) is used. We will showcase this in the evaluation section. OptCL smartly decides which data should be copied over to the other devices and performs the copies only when necessary based on the data dependency information collected in Step 1 as well as the hardware type.

After the co-execution, the host may receive inconsistent outputs from different devices. The correct output is then restored using the dependency information recorded in Step 1. A *merge_function* is inserted into the host program in two scenarios: In a WAR dependency scenario, the *merge_function* picks the output from the device conducting the write operation. In the scenario where co-executed kernels write to different device variables, the *merge_function* gathers the individual outputs from the devices that carried out the write operation and merges them together on the host. For example, HPR2 and HPR3 illustrated in Listing 5.2 can be co-executed on different devices. The *merge_function* then merges device variable B outputted by HPR2 and device variable C by HPR3 on the host. The merged data is re-distributed to the devices if the next kernel is not executed on the host (CPUs).

A piece of clean-up code concludes the host program, freeing buffers and kernels as well as outputting the results if required. Eventually, OptCL lets Clang compile the reconstructed code and generate the final executables.

5.3.4 Optimisation

5.3.4.1 Enhanced Dependency Detection

Device variables can be declared using user-defined structures. For example, when using OpenABLext to program a traffic ABS, a device variable can be an array of car agents where each car possesses its identifier, position, velocity, etc. Dependencies may be overestimated if OptCL were to treat the structure as a whole. In a given program, it may occur that two consecutive kernels write to disjoint sets of members of the same structure. In this case, these two kernels can potentially still be co-executed. However, given the dependency detection rules described in Section 5.3.1, they would be identified as having a WAW dependency.

To solve this issue, an enhanced dependency detection is introduced. Device variables declared as structures are broken down to the member level. Given device variable A defined using structure $T\{type\ member1, type\ member2, ...\}$,
the new ϵ expression of a kernel that operates on A will be $\{A.member1 : R|W, A.member2 : R|W, ..., B : R|W, ...\}$. With this new ϵ expression, the dependency detection rules can be applied in the same way as described in Section 5.3.1.

5.3.4.2 User-specified *merge_function*

If extra logic is provided to resolve dependency conflicts, parallelisation of HPRs with RAW or WAW dependencies is also possible. In some use cases, RAW or WAW dependency may even be tolerated, e.g., in stochastic ABSs [259].

To fulfill such needs, we allow users to define their own *merge_functions* following the naming convention *kernel1_kernel2_merge_function* in the respective HPDSL's syntax. Once a RAW or WAW dependencies are detected, OptCL will search for the existence of an optional *merge_function*. Users can also define an empty *merge_function* to allow RAW or WAW dependency to exist. This also implies that OptCL will not check if the dependency conflicts are resolved by applying the user-specified *merge_functions*. Users are then responsible for ensuring the logic of the program is still correct by providing the *merge_functions*. An example of a user-specified *merge_function* will be given in Sec. 5.4.

5.3.4.3 Single Kernel

When an HPDSL program contains only a single HPR, co-execution is still possible if the data partitioning scheme is used. It is safe to do so because the input data is processed in parallel even on one device. Each device receives a subset of the data

proportional to its computational power as profiled in Step 2. After processing, the output is transferred back to the host for merging.

To prevent discrepancies in the outcome of the HPDSL programs, this optimisation only applies to single kernels possessing no intra-read-and-write dependency. That means, e.g., if the kernel touches device variable A[i], there has to be no write to A[j] where $i \neq j$ in the same kernel. This is because A[i] and A[j] can be potentially processed on different devices where there is no guaranteed synchronisation. This rule can be imposed using an additional intra-dependency check during the data dependency analysis in Step 1.

5.4 Evaluation

We evaluate OptCL by plugging it into OpenABLext as well as SYCL. Both of the studied HPDSLs target CPU-GPU heterogeneous platforms. Our evaluation was, therefore, conducted on CPU-GPU systems. To cover possible hardware configurations, we include two types of CPU-GPU systems: a dedicated CPU-GPU (dCPU-GPU) platform equipped with an Intel Core i5-11400F CPU with 16 GB of RAM and an NVIDIA GTX 1070 graphics card with 8 GB of RAM; and an integrated CPU-GPU (iCPU-GPU) platform equipped with an Intel i5-7400 CPU with 16 GB of RAM and an integrated Intel HD 630 iGPU. Both systems run Ubuntu 18.04. The key difference between the two platforms is that while data transfer is often required between the CPU and the GPU in the dCPU-GPU setting, physical memory is shared between the CPU and the iGPU in the iCPU-GPU setting. Thus, the CPU and the iGPU can directly access each other's data, eliminating the data transfer overhead.

For SYCL, six applications covering domains ranging from physics simulation to machine learning were selected to evaluate the performance of OptCL. The applications are:

Back Propagation (BP) is an algorithm used for training neural networks in supervised machine learning tasks. By determining the gradient of the error function with respect to the model parameters, BP enables the use of gradient-based optimization methods to search local minima of the error function.

K-Means (KM) is a clustering algorithm. The algorithm partitions N nodes into K clusters with each node assigned to a cluster with the shortest distance to the mean defined as the centroid of this cluster. The algorithm initially assigns nodes randomly to clusters and then iteratively re-computes the new mean of each cluster and re-assigns nodes to clusters until convergence.

Speckle Reducing anisotropic diffusion (SR) is a widely used image processing algorithm to remove noise from ultrasonic and radar images. Based on partial differential equations, the algorithm iteratively diffuses the value of each pixel using the value of its four neighbours.

Hot Spot (HS) is a simulation to model the thermal dynamics of a processor based on its floor plan. Inputting the power and initial temperatures, HS simulates the heat dissipation and propagation among the blocks on the chip.

CirCle (CC) is a molecular simulation that models the potential and relocation of molecules driven by mutual forces in a 2-D space. Molecules only interact with their neighbours within a cut-off radius when computing the forces.

hierarchical-Matrix-Vector multiplication (MV), is a method proposed in [260]. We employ this idea to implement a large matrix-vector multiplication which completes after several iterations. In each iteration, two independent matrix-vector multiplications are performed and merged.

The tested applications all follow a pattern where HPRs are executed iteratively. We were not aware of any off-the-shelf SYCL implementations of these applications. Therefore, we implemented them from scratch, applying our best efforts to optimise for a general SIMD architecture such as those of GPUs.

5.4.1 Profiling Approaches Comparison

In the first evaluation, we compare the sampling-based approach and the offline approach introduced in Sec. 5.3.2. The goal is to select the approach with higher accuracy while also evaluating the usability. The evaluation is done on the dCPU-GPU system.

No setup is needed for the sampling-based approach. For the offline approach, the Random Forest model must be pre-trained with an extensive number of kernel patterns. We trained the model using the OpenDwarfs Extended Benchmark Suite[1], the same suite used for training in the original AIWC work [219]. The suite covers applications ranging from solvers to mathematical problems to image processing algorithms, demonstrating versatile kernel patterns. By varying the input sizes, the suite provided ~5,000 kernel patterns and their execution times.

The Random Forest model intakes three major parameters. *num.trees*: the number of trees in a random forest; *mtry*: number of possible independent features; and *min.node.size*: the minimal node size per tree. We employed the values suggested by the AIWC authors which are $num.trees = 505$, $mtry = 30$ and $min.node.size = 9$. For the sampling-based approach, we profile the applications with a timer set to 1 second.

[1] https://github.com/BeauJoh/OpenDwarfs

The profiling step is to guide the hardware assignment. Therefore, it is more important to learn the relative performance comparison between different hardware types rather than the individual execution time. Further, for the sake of creating an application-independent metric to quantify the two profiling approaches, we employ a *performance ratio* metric $R_{C/G}$ defined as the execution time on the CPU divided by the time on the GPU. In case throughput is taken, $R_{C/G}$ is defined as the throughput on the GPU divided by the one on the CPU.

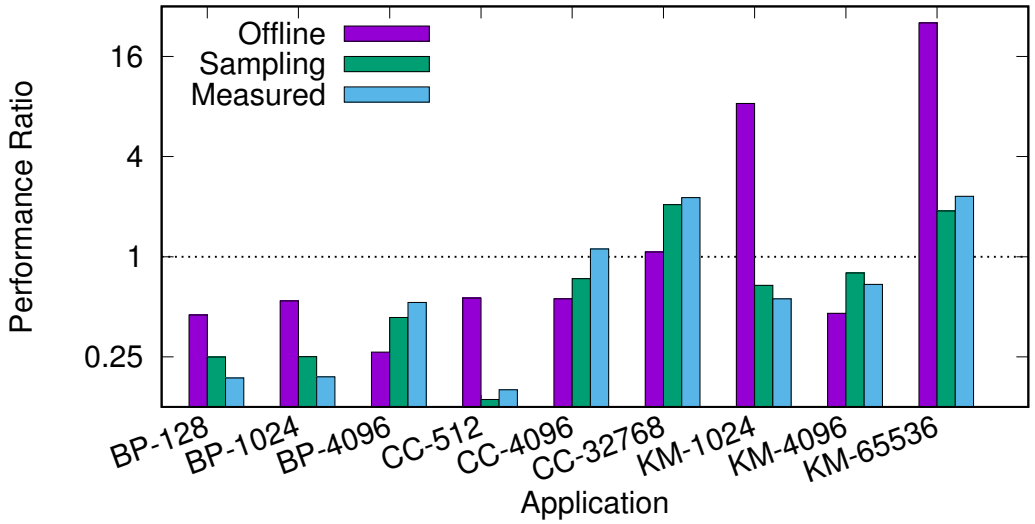


Figure 5.6: Comparison of different $R_{C/G}$ results.

Figure 5.6 illustrates $R_{C/G}$ outputted by the sampling-based approach, the offline approach as well as the measured $R_{C/G}$ on the CPU and the GPU. The x-axis labels are of the form *NAME-SCALE*, where *NAME* refers to the name of the application and *SCALE* is the input size. Due to limit space, here we demonstrate three applications: BP, CC, and KM, but varying different input scales. As depicted in Figure 5.6, a base line $R_{C/G} = 1$ (meaning the performance on the CPU and the GPU is equal) splits the space into two sides. The sampling-based approach managed to identify correctly the better-performing hardware between the CPU and the GPU in all cases but one (CC-4096), albeit with a certain estimation error (defined as the estimated ratio divided by the measured ratio). It failed in the CC-4096 case because the workload in each iteration of CC depends strongly on the changing positions of the molecules, and, therefore, it changes from iteration to iteration. Using the first few iterations to estimate the full execution time thus leads to deviations. Although the offline approach also succeeded in estimating the performance deviation in most of the cases (7 out of 9), it came with

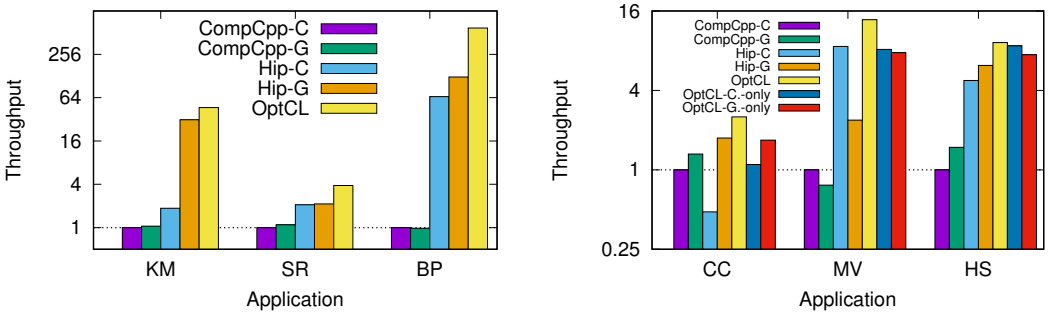
much bigger estimation errors (82% on average versus 22% with the sampling-based approach). Significant errors are observed in KM-1024 as well as in KM-65536, as the training data lacked of such kernel pattern or input size.

In summary, the sampling-based approach led to better accuracy in all tested applications with zero setup effort, compared to a moderately trained (training data size/test data size = 555.6) offline machine-learning based approach. Hence, we choose to use the sampling-based approach in the rest of this chapter. However, the offline approach could still produce valid results when trained to more comprehensively cover the possible kernel patterns and input sizes, which we defer to future work.

5.4.2 Case Study 1: SYCL

We relied on ComputeCpp version 2.21, which still supports NVIDIA GPUs. All the tested applications were compiled with `-O3` optimisation.

Six applications are categorised into two groups based on the observed speed up types: **Group 1: BP, KM and SR.** These applications consist of several kernels. Owing to the dependencies between kernels, co-execution is not feasible. However, performance improvements are still possible through kernel invocation overhead reduction introduced in Sec. 5.3.3 as well as by executing them on the best suitable hardware. **Group 2: HS, CC and MV.** Co-execution is feasible. In addition to the performance benefits achieved by less invocation overhead, further performance improvements are observed due to co-execution. We report the performance of these two groups separately.



(a) Group 1 applications

(b) Group 2 applications

Figure 5.7: Speed up of SYCL applications on the dedicated CPU-GPU system. CompCpp-C/G: performance of the SYCL code compiled by ComputeCpp and executes on CPU/GPU. Hip-C/G: performance of the same code compiled by HipSYCL on CPU/GPU. Each application is normalised to the throughput of its respective CompCpp-C.

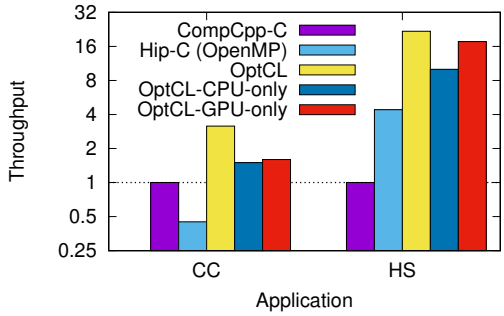


Figure 5.8: Throughput of SYCL applications on the iCPU-GPU system.

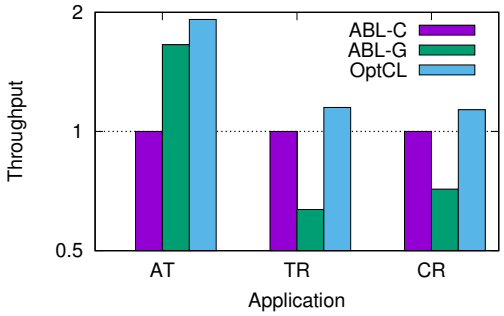


Figure 5.9: Throughput of OpenABLext applications.

To put the performance of OptCL into perspective, we provide another baseline. The performance of the same SYCL code compiled by hipSYCL, an open-source SYCL implementation using OpenMP (for CPU) and CUDA (for GPU) as the backends. hipSYCL supports NVIDIA graphics cards (via clang-CUDA), and for CPUs it uses OpenMP, which incurs only little invocation overhead.

Figure 5.7a and 5.7b show the throughput of CompCpp-C/G (the throughput of the SYCL code compiled by ComputeCpp and executed on CPU/GPU), Hip-C/G (the same code compiled by hipSYCL and run on CPU/GPU) with each application normalised to the throughput of its respective CompCpp-C. As depicted in the two figures, OptCL achieved the best performance in all settings. The performance was not optimal for the ComputeCpp variants as shown in Figure 5.7a, due to the incomplete support of NVIDIA GPUs and the kernel invocation overheads as the HPR in each iteration was treated as a new kernel. The runtime freshly compiles the kernel and redundantly transfers data to accelerators each time an HPR is invoked, which can be verified by running the executables in the Oclgrind simulation. By compiling the same SYCL code using hipSYCL, substantial speed-ups up to 120x were achieved. The performance was improved further by employing the OptCL middleware, thanks to the kernel invocation overhead reduction and using the most suitable hardware.

To better illustrate the power of co-execution, in Figure 5.7b we also show the 'otherwise' scenarios, where we suppose OptCL would assign the kernels to only CPU (OptCL-CPU-only) or GPU (OptCL-GPU-only). Notably, Hip-C (OpenMP) performed worse than the ComputeCpp executables in the CC application. This is because in CC, each molecule traverses the global memory for other nearby molecules, causing large numbers of cache misses. As can be seen in the 'otherwise' scenarios, while running the OptCL variants on a single accelerator (OptCL-CPU-only and OptCL-GPU-only) already produced similar or even better performance than other variants,

co-execution further boosted the throughputs. A maximum speed-up of 1.67x over the 'otherwise' scenarios and 13x over the baseline CompCpp-C was observed for the MV application.

The performance on the iCPU-GPU system is displayed in Figure 5.8. As both ComputeCpp and hipSYCL do not well support iGPU (the support is experimental in hipSYCL), we exclude them from the evaluation. The iGPUs are usually not as powerful as the dedicated ones due to fewer cores and thermal concerns. As a consequence, co-execution for the MV application was not feasible in the iCPU-GPU setting, because of the large throughput deviation between the CPU and the iGPU, causing long execution time on the iGPU even with a small amount of data. However, for the applications (CC and HS) that are eligible for co-execution, more significant performance benefits were obtained owing to the zero-copy technology reducing the data transfer overhead. We achieved a speed-up of 7x and 5x over the Hip-C executables and 3x and 21x over the CompCpp-C executables in CC and HS, respectively.

5.4.3 Case Study 2: OpenABLext

In this study, we apply OptCL on OpenABLext. Three ABS models were selected from three domains: social science, transportation, and biology.

CRowd (CR) simulates the flocking behaviour of people following fire wardens to escape from a single-entrance room in case of a fire accident. The fire wardens perform a path finding every few steps, and the other agents follow the closest fire warden within their sight. The path finding of fire wardens and the flocking of people are independent.

TRaffic (TR), a traffic simulation comprised of a car following model, which controls the agent to follow the vehicle in the front and avoid collisions, and a lane changing model for the agent to switch to less crowded lanes. Both model may modify the current lane of an agent. A user-specified *merge_function* is hence needed in order to co-execute these two models. For experimentation purposes, we set the *merge_function* such that the car following model always overwrites the output of the lane changing model.

AnTs (AT) is a model simulating the foraging behaviour of ants by leaving pheromones between their home and the food. Ants follow the pheromone trail towards food while pheromones diffuse over time. The movement of ants and the diffusion of old pheromones are independent, and can thus be co-executed.

Despite the fact that data transfer overhead was avoided, co-execution was not feasible for the tested applications on the iCPU-GPU system. Similarly to the MV application, this is caused by the large performance deviation between the CPU and the iGPU.

Therefore, in what follows, we only report the performance using the dCPU-GPU system. As shown in Figure 5.9, compared to running on a single device, co-execution led to the best performance in all three applications. In the CR application, the path finding is slow on the GPU owing to the heavy memory operations, causing an overall performance reduction on the GPU as indicated by the ABL-G bar in Figure 5.9. Similarly, for the TR application, the car following model requires memory-intensive search for nearby vehicle which is again slow on the GPU. Using co-execution, i.e., assigning the memory-intensive kernel to the CPU and the rest to the GPU, a speed-up of 1.15x and 1.13x over running on the CPU was achieved by OptCL for the TR and CR application.

Although all kernels run faster on the GPU than the CPU in the AT applications due to GPU's massive parallelism, overlapping execution of kernels on the CPU and the GPU can still lead to performance benefits. This is because of the reason explained in Fig. 5.5. For the AT applications, co-execution was also 1.15x faster than running on the GPU.

5.5 Summary

In this chapter, we presented OptCL, a middleware to complement the co-execution ability of OpenABLext. The middleware works at an IR level. Therefore, it can not only benefit the OpenABLext framework but also a wide range of HPDSLs like OpenABLext. Through a data dependency analysis among High Performance Regions (HPRs) and performance predictions, OptCL assigns each kernel to the most suitable hardware device and selects the best execution strategy out of purely CPU-based execution, offloading to an accelerator, or co-execution. Kernel invocation and data transfer overheads are also minimised in the generated code.

The workflow of OptCL consists of three steps. Starting from HPDSL code, OptCL triggers a partial compilation, where an Abstract Syntax Tree (AST) is generated. By identifying all High Performance Regions (HPRs), the sub-ASTs containing all HPRs are extracted. A data dependency graph is built using the information gathered from analysing the sub-ASTs, revealing the dependencies between HPRs and thus possibilities for co-execution. In Step 2, kernels converted from HPRs are profiled on the available hardware devices. The profiling results are used to assign each kernel to its best suitable hardware and to enable co-execution where possible. Two profiling approaches are studied to estimate the power of each device: a sampling-based approach executing the applications with a small amount of data and an offline approach using

a prediction model. In our comparison study, the sampling-based approach enabled higher performance than the offline approach. Finally, in Step 3, OpenCL executables are generated reflecting the hardware assignment.

We demonstrated the versatility of OptCL by plugging it to OpenABLext as well as another HPDSL named SYCL. Two different hardware settings were used: a system using a CPU and a discrete GPU as well as an integrated CPU-GPU system. In an extensive study using various applications at different scales, OptCL outperformed existing solutions and exhibited significant speed ups. We showed that OptCL can be used to enable high-performance execution on heterogeneous hardware environments without in-depth knowledge of programming paradigms for the underlying hardware. Maximum speed-ups of 13x and 21x over the original compiler were achieved on the dCPU-GPU system and iCPU-GPU system respectively.

With OptCL, the workflow to generate high-performance ABS code is simple First, OpenABLext is used to write ABS code using the sequential OpenABL specifications. The OpenABLext compiler then generates OpenCL kernels optimised for the available hardware architectures. Eventually, the OptCL middleware decides the optimal distribution of kernels to available hardware and choose the best suitable execution scheme.

6 Conclusion & Outlook

Contents

6.1 Summary

Agent-based Simulation (ABS) has been widely employed to conduct system analysis and answer what-if questions in a wide variety of domains. However, the increasingly growing model complexity and simulation scale can lead to substantial execution time. Anxiety over performance arises. While exploiting the power of accelerators such as Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) in a heterogeneous hardware setting helps to improve the performance, we demonstrated that thorough knowledge of the hardware and substantial programming efforts were required in order to achieve an optimal performance. Further, the code tailored for one specific type of hardware can not be trivially ported to another as the performance may suffer. Significant burdens are put on the simulationists to keep the performance optimal.

To tackle this issue, we present OpenABLext, an extension to OpenABL, a model specification language for agent-based simulations. By providing a device-aware OpenCL backend, OpenABLext enables generating high-performance agent-based simulation code targetting heterogeneous hardware platforms consisting of Central Processing Units (CPUs), GPUs and FPGAs. We present a novel online dispatching method which efficiently profiles partitions of the simulation during run-time to optimise 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 utilises an efficient neighbour 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.

While the online dispatcher deals with hardware assignment, it does not take into account the possible data dependencies between the different parts of the simulation. A middleware named Optimise performance targetting high-performance domain-specifiC Languages (OptCL) is proposed to complement OpenABLext. The middleware requires little set up effort, and it can run in a fully automated way afterwards. Through a static analysis of the source code, the middleware first detects data dependencies between step functions. Based on performance predictions, the middleware assigns OpenCL kernels generated by the OpenABLext framework to the most suitable hardware device and selects the best execution mode out of pure CPU-based execution, offloading to an accelerator, or co-execution. The OptCL is language independent. Hence, it can benefit a wide range of Domain Specific Languages (DSLs) like OpenABLext. We show that substantial performance gains can be achieved by plugging OptCL to OpenABLext as well as other widely used DSLs such as SYCL.

The framework consisting of OpenABLext and OptCL is an ideal tool for the simulationists to program high-performance ABS code on heterogeneous hardware. It requires no profound knowledge of the hardware. This frees the simulationists from exhaustive efforts on improving the performance so that they can focus on the modelling aspect instead.

6.2 Outlook

The framework is made up of two major parts: OpenABLext and the OptCL middleware. Both parts could enable interesting future research avenues:

OpenABLext Future work includes optimising the FPGA code outputted by OpenABLext by e.g. generating more computing units per simulation instance. This may require a redesign of the simulation structure, e.g., dealing with limited on-chip memory bandwidth of FPGAs when multiple computing units are present.

So far, we do not consider a heterogeneous cluster setting e.g. a machine equipped with a CPU cluster and a GPU cluster. Following our current workflow, each accelerator is treated as an individual device. This could limit the performance benefits, as some homogeneity aspects can potentially be exploited to further speed up the simulation e.g. rapid data transfer in between GPU clusters via NVLink.

OptCL The performance prediction accuracy can be further improved by introducing e.g. some heuristics modelling the computational power of the hardware. Currently, OptCL selects hardware devices solely based on the performance. Other criteria such as energy efficiency also constitute possible future research directions.

Bibliography

[1] P. Andelfinger, J. Mittag, and H. Hartenstein. GPU-Based Architectures and Their Benefit for Accurate and Efficient Wireless Network Simulations. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '11)*, pages 421–424, Singapore, July 2011. IEEE.

[2] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. A survey on agent-based simulation using hardware accelerators. *ACM Computing Surveys (CSUR)*, 51(6):131, 2019.

[3] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, March 2005.

[4] A. Doniec, R. Mandiau, S. Piechowiak, and S. Espié. A Behavioral Multi-Agent Model for Road Traffic Simulation. *Elsevier Journal of Engineering Applications of Artificial Intelligence*, 21(8):1443–1454, December 2008.

[5] J. M. Epstein. Agent-Based Computational Models and Generative Social Science. *Wiley Complexity*, 4(5):41–60, May 1999.

[6] K. Teknomo, Y. Takeyama, and H. Inamura. Review on microscopic pedestrian simulation model. *CoRR*, abs/1609.01808, 2016. `arXiv:1609.01808`.

[7] T. M. Cioppa, T. W. Lucas, and S. M. Sanchez. Military Applications of Agent-Based Simulations. In *Proceedings of the Winter Simulation Conference (WSC '04)*, pages 171–180, Washington, DC, USA, December 2004. IEEE.

[8] G. An, Q. Mi, J. Dutta-Moscato, and Y. Vodovotz. Agent-Based Models in Translational Systems Biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 1(2):159–171, September 2009.

[9] L. Tesfatsion. Agent-Based Computational Economics: A Constructive Approach to Economic Theory. *Elsevier Handbook of Computational Economics*, 2:831–880, May 2006.

[10] I. Wagner and D. Eckhoff. Privacy Assessment in Vehicular Networks Using Simulation. In *Winter Simulation Conference (WSC '14)*, pages 3155–3166, Savannah, GA, December 2014. IEEE.

[11] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: A Multiagent Simulation Environment. *SCSI Journal of Simulation*, 81(7):517–527, July 2005.

[12] N. Collier and M. North. Repast sc++: A platform for large-scale agent-based modeling. In W. Dubitzky, K. Kurowski, and B. Schott, editors, *Large-Scale Computing Techniques for Complex System Simulations*. Wiley, Hoboken, NJ, USA, 2011.

[13] R. K. Standish and R. Leow. Ecolab: Agent based modeling for C++ programmers. *CoRR*, cs.MA/0401026, January 2004.

[14] S. Coakley, P. Richmond, M. Gheorghe, S. Chin, D. Worth, M. Holcombe, and C. Greenough. Large-scale simulations with flame. In J. Kołodziej, L. Correia, and J. Manuel Molina, editors, *Intelligent Agents in Data-intensive Computing*, pages 123–142. Springer International Publishing, New York City, NY, USA, 2016.

[15] G. Laville, K. Mazouzi, C. Lang, N. Marilleau, B. Herrmann, and L. Philippe. MCMAS: A Toolkit to Benefit From Many-Core Architecure in Agent-Based Simulation. In *Proceedings of the European Conference on Parallel Processing (Euro-Par '13)*, pages 544–554, Aachen, Germany, August 2013. Springer.

[16] L. Howes and M. Rovatsou. SYCL Specification, SYCL integrates OpenCL devices with modern C++ version 1.2. *Khronos OpenCL Working Group*, 2016.

[17] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. IEEE, 2011.

[18] B. Cosenza, N. Popov, B. Juurlink, P. Richmond, M. K. Chimeh, C. Spagnuolo, G. Cordasco, and V. Scarano. OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations. In *Proceedings of the European Conference on Parallel Processing (Euro-Par '18)*, pages 505–518, Turin, Italy, August 2018. Springer.

[19] M. J. North and C. M. Macal. *Managing Business Complexity: Discovering Strategic Solutions With Agent-Based Modeling and Simulation.* Oxford University Press, Oxford, UK, 2007.

[20] S. Tisue and U. Wilensky. Netlogo: A Simple Environment for Modeling Complexity. In *Proceedings of the International Conference on Complex Systems (ICCS '04)*, volume 21, pages 16–21, Boston, MA, USA, June 2004. NECSI.

[21] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations. Technical Report 1996-06-042, Swarm Development Group, June 1996.

[22] A. Horni, K. Nagel, and K. W. Axhausen. *The Multi-Agent Transport Simulation MATSim.* Ubiquity Press, London, UK, 2016.

[23] R. Allan. Survey of Agent Based Modelling and Simulation Tools. Technical Report DL-TR-2010-007, Science and Technology Facilities Council, October 2010.

[24] S. Abar, G. K. Theodoropoulos, P. Lemarinier, and G. M. O'Hare. Agent Based Modelling and Simulation Tools: A Review of the State-of-Art Software. *Elsevier Computer Science Review*, 24:13–33, May 2017.

[25] P. F. Riley and G. F. Riley. Next Generation Modeling III-Agents: Spades—a Distributed Agent Simulation Environment With Software-in-the-Loop Execution. In *Proceedings of the Conference on Winter Simulation (WSC '03)*, pages 817–825, New Orleans, LA, USA, December 2003. IEEE.

[26] K. Nagel and M. Rickert. Parallel Implementation of the TRANSIMS Micro-Simulation. *Elsevier Journal of Parallel Computing*, 27(12):1611–1639, November 2001.

[27] R. M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley, New York City, NY, USA, 2000.

[28] C. Lai, M. Huang, X. Shi, and H. You. Accelerating Geospatial Applications on Hybrid Architectures. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications, IEEE International Conference on Embedded and Ubiquitous Computing (HPCC & EUC '13)*, pages 1545–1552, Singapore, November 2013. IEEE.

[29] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing. *Elsevier Journal of Parallel and Distributed Computing*, 73(1):4–13, January 2013.

[30] Intel Corporation. Intel® Architecture Instruction Set Extensions and Future Features – Programming Reference, January 2018. 319433-032.

[31] D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh. Characterizing and Understanding PDES Behavior on Tilera Architecture. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS '12)*, pages 53–62, Zhangjiajie, China, July 2012. IEEE.

[32] B. Williams, D. Ponomarev, N. Abu-Ghazaleh, and P. Wilsey. Performance Characterization of Parallel Discrete Event Simulation on Knights Landing Processor. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '17)*, pages 121–132, Singapore, May 2017. ACM.

[33] S. Raase and T. Nordstrm. On the Use of a Many-Core Processor for Computational Fluid Dynamics Simulations. In *Proceedings of the International Conference On Computational Science (ICCS '15)*, pages 1403–1412, Reykjavík, Iceland, June 2015. Elsevier.

[34] M. Castro, E. Francesquini, F. Dupros, H. Aochi, P. Navaux, and J.-F. Mehaut. Seismic Wave Propagation Simulations on Low-Power and Performance-Centric Manycores. *Elsevier Journal of Parallel Computing*, 54:108–120, May 2016.

[35] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. H. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, L. Oliker, A. Ovsyannikov, A. Sarje, J.-L. Vay, H. Vincenti, S. Williams, P. Carrier, N. Wichmann, M. Wagner, P. Kent, C. Kerr, and J. Dennis. Evaluating and Optimizing the NERSC Workload on Knights Landing. In *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '16)*, pages 43–53, Salt Lake City, UT, USA, November 2016. IEEE.

[36] X. Liu, L. Chen, J. S. Firoz, J. Qiu, and L. Jiang. Performance Characterization of Multi-Threaded Graph Processing Applications on Intel Many-Integrated-Core Architecture. *CoRR*, abs/1708.04701, August 2017. `arXiv:1708.04701`.

[37] OpenACC Working Group. The OpenACC Application Programming Interface, October 2015. Version 2.5.

[38] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Computing in Science & Engineering*, 12(3):66–73, May 2010.

[39] NVIDIA Corporation. Whitepaper – NVIDIA GeForce GTX 1080 – Gaming Perfected, 2016.

[40] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2018. Version 9.1.85, NVIDIA Corporation.

[41] F. Wu. Calibration of Stochastic Cellular Automata: The Application to Rural-Urban Land Conversions. *Taylor & Francis International Journal of Geographical Information Science*, 16(8):795–818, 2002.

[42] N. Fauzia, L.-N. Pouchet, and P. Sadayappan. Characterizing and Enhancing Global Memory Data Coalescing on GPUs. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*, pages 12–22, San Francisco, CA, USA, February 2015. IEEE.

[43] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, pages 359–371. Elsevier, Boston, MA, USA, 2011.

[44] NVIDIA Corporation. CUDA Toolkit 4.2 – CUBLAS Library, February 2012. PG-05326-041_v01.

[45] B. Falsafi, B. Dally, D. Singh, D. Chiou, J. Y. Joshua, and R. Sendag. FPGAs Versus GPUs in Data Centers. *IEEE Micro*, 37(1):60–72, January 2017.

[46] Y. Jiao, H. Lin, P. Balaji, and W. Feng. Power and Performance Characterization of Computational Kernels on the GPU. In *Proceedings of the IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM '10)*, pages 221–228, Hangzhou, China, December 2010. IEEE.

[47] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '12)*, pages 141–151, La Jolla, CA, USA, November 2012. IEEE.

[48] NVIDIA Corporation. Whitepaper – NVIDIA Tesla P100 – The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU, 2016.

[49] Advanced Micro Devices, Inc. Radeon's Next-Generation Vega architecture. Technical Report 061317_FINAL_V2, Radeon Technologies Group, June 2017.

[50] Wąs, Jarosław and Mróz, Hubert and Topa, Paweł. GPGPU Computing for Microscopic Simulations of Crowd Dynamics. *Slovak Academy of Sciences Computing and Informatics*, 34(6):1418–1434, February 2016.

[51] Y. Torres, A. Gonzalez-Escribano, and D. Llanos. Understanding the Impact of CUDA Tuning Techniques for Fermi. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS '11)*, pages 631–639, Istanbul, Turkey, July 2011. IEEE.

[52] S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, Burlington, MA, United States, 2010.

[53] U. Heinkel, M. Padeffke, W. Haas, T. Buerner, H. Braisz, T. Gentner, and A. Grassmann. *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (Including VHDL-AMS)*. Wiley, New York City, NY, USA, 2000.

[54] S. Palnitkar. *Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2003.

[55] Intel Corporation. Intel® FPGA SDK for OpenCL – Programming Guide, December 2017. UG-OCL002.

[56] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating Compute-Intensive Applications With GPUs and FPGAs. In *Proceedings of the Symposium on Application Specific Processors (SASP '08)*, pages 101–107, Anaheim, CA, USA, June 2008. IEEE.

[57] S. Rahman, N. Abu-Ghazaleh, and W. Najjar. PDES-A: A Parallel Discrete Event Simulation Accelerator for FPGAs. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '17)*, pages 133–144, Singapore, May 2017. ACM.

[58] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, October 2016.

[59] R. M. Fujimoto, C. Bock, W. Chen, E. Page, and J. H. Panchal. *Research Challenges in Modeling and Simulation for Engineering Complex Systems*. Springer, New York City, NY, USA, 2017.

[60] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing With FPGAs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, pages 35:1–35:12, Salt Lake City, UT, USA, November 2016. IEEE.

[61] P. F. Reynolds, C. M. Pancerella, and S. Srinivasan. Design and Performance Analysis of Hardware Support for Parallel Simulations. *Elsevier Journal of Parallel and Distributed Computing*, 18(4):435–453, August 1993.

[62] J. G. Cleary, M. Pearson, and H. Kinawi. The Architecture of an Optimistic CPU: The WarpEngine. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS '95)*, pages 163–172, Wailea, HI, USA, January 1995. IEEE.

[63] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan. Design and Performance of Special Purpose Hardware for Time Warp. In *Proceedings of the Annual International Symposium on Computer Architecture (SCA '88)*, pages 401–409, Honolulu, HI, USA, May 1988. IEEE.

[64] J. L. Berlin. Design of a Parallel Discrete Event Simulation Coprocessor. Master's thesis, Air Force Institute of Technology, School of Engineering, Wright-Patterson AFB, OH, USA, December 1993.

[65] E. W. Lynch. *Hardware Acceleration for Conservative Parallel Discrete Event Simulation on Multi-Core Systems*. PhD thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, February 2011.

[66] N. P. Jouppi, C. Young, N. Patil, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA '17)*, pages 1–12, Toronto, Canada, June 2017. IEEE.

[67] S. Panwai and H. Dia. A Reactive Agent-based Neural Network Car Following Model. In *Proceedings of the International IEEE Conference on Intelligent Transportation Systems (ITSC '05)*, pages 375–380, Vienna, Austria, October 2005. IEEE.

[68] E. Cabrera, M. Taboada, M. L. Iglesias, F. Epelde, and E. Luque. Optimization of Healthcare Emergency Departments by Agent-Based Simulation. In *Proceedings of the International Conference on Computational Science (ICCS '11)*, pages 1880–1889, Singapore, June 2011. Elsevier.

[69] M. Hirabayashi, S. Kato, M. Edahiro, and Y. Sugiyama. Toward GPU-Accelerated Traffic Simulation and Its Real-Time Challenge. In *Proceedings of the International Workshop on Real-time and Distributed Computing in Emerging Applications (REACTION '12)*, pages 45–50, San Juan, Puerto Rico, December 2012. Universidad Carlos III de Madrid.

[70] K. S. Perumalla. Efficient Execution on GPUs of Field-Based Vehicular Mobility Models. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS '08)*, pages 154–154, Roma, Italy, June 2008. IEEE.

[71] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. GPU-Based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 95–103, Lake Placid, NY, USA, June 2009. IEEE.

[72] D. Strippgen and K. Nagel. Using Common Graphics Hardware for Multi-Agent Traffic Simulation With CUDA. In *Proceedings of the International Conference on Simulation Tools and Techniques (Simutools '09)*, pages 62:1–62:8, Rome, Italy, March 2009. ICST.

[73] Z. Shen, K. Wang, and F. Zhu. Agent-Based Traffic Simulation and Traffic Signal Timing Optimization With GPU. In *Proceedings of the International IEEE Conference on Intelligent Transportation Systems (ITSC '11)*, pages 145–150, Washington, DC, USA, October 2011. IEEE.

[74] K. Wang and Z. Shen. A GPU Based Trafficparallel Simulation Module of Artificial Transportation Systems. In *Proceedings of the IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI '12)*, pages 160–165, Suzhou, China, July 2012. IEEE.

[75] Y. Xu, G. Tan, X. Li, and X. Song. Mesoscopic traffic simulation on CPU/GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '14)*, pages 39–50, Denver, CO, USA, May 2014. ACM.

[76] X. Song, Z. Xie, Y. Xu, G. Tan, W. Tang, J. Bi, and X. Li. Supporting Real-World Network-Oriented Mesoscopic Traffic Simulation on GPU. *Elsevier Journal of Simulation Modelling Practice and Theory*, 74:46–63, May 2017.

[77] P. Heywood, P. Richmond, and S. Maddock. Road Network Simulation Using FLAME GPU. In *Proceedings of the European Conference on Parallel Processing (Euro-Par '15)*, pages 430–441, Vienna, Austria, August 2015. Springer.

[78] P. Andelfinger, Y. Xu, D. Eckhoff, W. Cai, and A. Knoll. Fast-Forwarding Agent States to Accelerate Microscopic Traffic Simulations. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '18)*, pages 113–124, Rome, Italy, May 2018. ACM.

[79] J. Wang, N. Rubin, H. Wu, and S. Yalamanchili. Accelerating Simulation of Agent-Based Models on Heterogeneous Architectures. In *Proceedings of the Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU '06)*, pages 108–119, Houston, TX, USA, March 2013. ACM.

[80] J. L. Tripp, H. S. Mortveit, A. A. Hansson, and M. Gokhale. Metropolitan Road Traffic Simulation on FPGAs. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '05)*, pages 117–126, Napa, CA, USA, April 2005. IEEE.

[81] P. Richmond and D. Romano. Agent Based GPU, a Real-time 3D Simulation and Interactive Visualisation Framework for Massive Agent Based Modelling on the GPU. In *Proceedings of the International Workshop on Super Visualization (IWSV '08)*, Island of Kos, Greece, June 2008. ACM.

[82] R. M. D'Souza, M. Lysenko, S. Marino, and D. Kirschner. Data-Parallel Algorithms for Agent-Based Model Simulation of Tuberculosis on Graphics Processing Units. In *Proceedings of the Spring Simulation Multiconference (SpringSim '09)*, pages 21:1–21:12, San Diego, CA, USA, March 2009. SCSI.

[83] B. G. Aaby, K. S. Perumalla, and S. K. Seal. Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools '10)*, pages 29:1–29:10, Torremolinos, Spain, March 2010. ICST.

[84] K. S. Perumalla and B. G. Aaby. Data Parallel Execution Challenges and Runtime Performance of Agent Simulations on GPUs. In *Proceedings of the Spring Simulation Multiconference (SpringSim '08)*, pages 116–123, Ottawa, Canada, April 2008. SCSI.

[85] P. Richmond, S. Coakley, and D. Romano. Cellular Level Agent Based Modelling on the Graphics Processing Unit. In *Proceedings of the International Workshop on High Performance Computational Systems Biology (HIBI '09)*, pages 43–50, Trento, Italy, October 2009. IEEE.

[86] P. Zou, Y.-s. Lü, L.-l. Chen, and Y.-p. Yao. Epidemic Simulation of Large-Scale Social Contact Network on GPU Clusters. *Sage Publications Journal of Simulation*, 89(10):1154–1172, 2013.

[87] W. Tang and Y. Yao. A GPU-Based Discrete Event Simulation Kernel. *SCSI Journal of Simulation*, 89(11):1335–1354, October 2013.

[88] E. Hermellin and F. Michel. GPU Environmental Delegation of Agent Perceptions: Application to Reynolds's Boids. In *Proceedings of the International Workshop on Multi-Agent Systems and Agent-Based Simulation (MABS '15)*, pages 71–86, Istanbul, Turkey, May 2015. Springer.

[89] E. Hermellin and F. Michel. GPU Delegation: Toward a Generic Approach for Developping MABS using GPU Programming. In *Proceedings of the International Conference on Autonomous Agents & Multiagent Systems (AAMAS '16)*, pages 1249–1258, Singapore, May 2016. IFAAMAS.

[90] K. Kofler, G. Davis, and S. Gesing. Sampo: An Agent-Based Mosquito Point Model in OpenCL. In *Proceedings of the Symposium on Agent Directed Simulation (ADS '14)*, pages 5:1–5:10, Tampa, FL, USA, April 2014. SCSI.

[91] P. Richmond and D. Romano. Template-Driven Agent-Based Modeling and Simulation With CUDA. In *GPU Computing Gems Emerald Edition, Applications of GPU Computing Series*, pages 313–324. Elsevier, Amsterdam, Netherlands, February 2011.

[92] X. Li, W. Cai, and S. J. Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '14)*, pages 87–96, Toulouse, France, October 2014. IEEE.

[93] L. Cui, J. Chen, Y. Hu, J. Xiong, Z. Feng, and L. He. Acceleration of Multi-Agent Simulation on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '11)*, pages 470–473, Chania, Greece, September 2011. IEEE.

[94] G. Laville, K. Mazouzi, C. Lang, L. Philipppe, and N. Marilleau. Using GPU for Multi-Agent Soil Simulation. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '13)*, pages 392–399, Belfast, UK, February 2013. IEEE.

[95] S. B. Yoginath and K. S. Perumalla. Scalable Cloning on Large-Scale GPU Platforms With Application to Time-Stepped Simulations on Grids. *ACM Transactions on Modeling and Computer Simulation*, 28(1):5:11–5:26, January 2018.

[96] I. Vourkas and G. C. Sirakoulis. FPGA Based Cellular Automata for Environmental Modeling. In *Proceedings of the International Conference on Electronics, Circuits and Systems (ICECS '12)*, pages 93–96, Seville, Spain, December 2012. IEEE.

[97] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. HPC Simulations of Information Propagation Over Social Networks. In *Proceedings of the International Conference on Computational Science (ICCS '12)*, pages 292–301, Omaha, NE, USA, June 2012. Elsevier.

[98] J. Jin, S. J. Turner, B.-S. Lee, J. Zhong, and B. He. Simulation of Information Propagation Over Complex Networks: Performance Studies on Multi-GPU. In *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '13)*, pages 179–188, Delft, Netherlands, October 2013. IEEE.

[99] L. Zhen, Q. Gang, G. Gang, and C. Bin. A GPU-Based Simulation Kernel within Heterogeneous Collaborative Computation on Large-Scale Artificial Society. *IACSIT Press International Journal of Modeling and Optimization*, 4(3):205–210, Jun 2014.

[100] G. Vigueras, J. M. Orduña, M. Lozano, J. M. Cecilia, and J. M. García. Accelerating Collision Detection for Large-Scale Crowd Simulation on Multi-Core and Many-Core Architectures. *Sage Publications The International Journal of High Performance Computing Applications*, 28(1):33–49, February 2014.

[101] X. Li, W. Cai, and S. J. Turner. Cloning Agent-based Simulation on GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '15)*, pages 173–182, London, UK, June 2015. ACM.

[102] X. Li, W. Cai, and S. J. Turner. GPU Accelerated Three-Stage Execution Model for Event-Parallel Simulation. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '13)*, pages 57–66, Montreal, Canada, May 2013. ACM.

[103] I. G. Georgoudas, P. Kyriakos, G. C. Sirakoulis, and I. T. Andreadis. An FPGA Implemented Cellular Automaton Crowd Evacuation Model Inspired by the Electrostatic-Induced Potential Fields. *Elsevier Journal of Microprocessors and Microsystems*, 34(7):285–300, November 2010.

[104] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-Based Simulation and Collision Detection for Large Particle Systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '04)*, pages 123–131, Grenoble, France, August 2004. ACM.

[105] M. G. Seok and T. G. Kim. Parallel Discrete Event Simulation for DEVS Cellular Models Using a GPU. In *Proceedings of the Symposium on High Performance Computing (HPC '12)*, pages 11:1–11:7, Orlando, FL, USA, March 2012. SCSI.

[106] F. Michel. Translating Agent Perception Computations Into Environmental Processes in Multi-Agent-Based Simulations: A Means for Integrating Graphics Processing Unit Programming Within Usual Agent-Based Simulation Platforms. *Wiley Online Library Journal of Systems Research and Behavioral Science*, 30(6):703–715, November 2013.

[107] M. Bauer, S. Treichler, and A. Aiken. Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, pages 119–130, Orlando, FL, USA, February 2014. ACM.

[108] K. S. Perumalla. Discrete-Event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS '06)*, pages 74–81, Singapore, May 2006. IEEE.

[109] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-Based Visual Simulation on Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02)*, pages 109–118, Saarbrücken, Germany, September 2002. ACM.

[110] J. Model and M. C. Herbordt. Discrete Event Simulation of Molecular Dynamics With Configurable Logic. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pages 151–158, Amsterdam, Netherlands, August 2007. IEEE.

[111] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle. Multi-Level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on GPUs. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS '12)*, pages 23–32, Zhangjiajie, China, July 2012. IEEE.

[112] B. R. Bilel, N. Navid, and M. S. M. Bouksiaa. Hybrid CPU-GPU Distributed Framework for Large Scale Mobile Networks Simulation. In *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT '12)*, pages 44–53, Dublin, Ireland, October 2012. IEEE.

[113] H. Park and P. A. Fishwick. A Fast Hybrid Time-Synchronous/Event Approach to Parallel Discrete Event Simulation of Queuing Networks. In *Proceedings of the Winter Simulation Conference (WSC '08)*, pages 795–803, Miami, FL, USA, December 2008. IEEE.

[114] H. Park and P. A. Fishwick. An Analysis of Queuing Network Simulation Using GPU-Based Hardware Acceleration. *ACM Transactions on Modeling and Computer Simulation*, 21(3):18:1–18:22, March 2011.

[115] J. Sang, C.-R. Lee, V. Rego, and C.-T. King. A Fast Implementation of Parallel Discrete-Event Simulation on GPGPU. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '13)*, page 501, Las Vegas, NV, US, July 2013. WorldComp.

[116] P. Andelfinger and H. Hartenstein. Exploiting the Parallelism of Large-Scale Application-Layer Networks by Adaptive GPU-Based Simulation. In *Proceedings of the Winter Simulation Conference (WSC '14)*, pages 3471–3482, Savannah, GA, USA, December 2014. IEEE.

[117] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs With CUDA. In *Proceedings of the European Conference on Parallel Processing (Euro-Par '09)*, pages 887–899, Delft, Netherlands, August 2009. Springer.

[118] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*, pages 101–110, Raleigh, NC, USA, February 2009. ACM.

[119] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, pages 142–151, San Jose, CA, USA, June 2011. ACM.

[120] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '12)*, pages 165–174, San Jose, CA, USA, March 2012. ACM.

[121] Q. A. P. Nguyen, P. Andelfinger, W. Cai, and A. Knoll. Transitioning spiking neural network simulators to heterogeneous hardware. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 115–126, Chicago, IL, USA, 2019. ACM.

[122] R. Pavlov and J. P. Müller. Multi-Agent Systems Meet GPU: Deploying Agent-Based Architectures on Graphics Processors. In *Proceedings of the Doctoral Conference on Computing, Electrical and Industrial Systems (DoCEIS '13)*, pages 115–122, Costa de Caparica, Portugal, April 2013. Springer.

[123] D. W. Bauer, M. McMahon, and E. H. Page. An Approach for the Effective Utilization of GP-GPUs in Parallel Combined Simulation. In *Proceedings of the Conference on Winter Simulation (WSC '08)*, pages 695–702, Miami, FL, USA, December 2008. IEEE.

[124] E. Hermellin and F. Michel. Defining a Methodology Based on GPU Delegation for Developing MABS Using GPGPU. In *Proceedings of the International Workshop on Multi-Agent Systems and Agent-Based Simulation (MABS '16)*, pages 24–41, Singapore, May 2016. Springer.

[125] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):905–918, March 2017.

[126] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Transactions on Architecture and Code Optimization - Special Issue on High-Performance Embedded Architectures and Compilers*, 9(4):57:1–57:20, January 2013.

[127] Y. Wen, Z. Wang, and M. F. O'Boyle. Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms. In *Proceedings of the International Conference on High Performance Computing (HiPC '14)*, pages 1–10, Dona Paula, India, December 2014. IEEE.

[128] I. Grasso, K. Kofler, B. Cosenza, and T. Fahringer. Automatic Problem Size Sensitive Task Partitioning on Heterogeneous Parallel Systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, pages 281–282, Shenzhen, China, February 2013. ACM.

[129] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the International ACM Conference on International Conference on Supercomputing (ICS '13)*, pages 149–160, Eugene, OR, USA, June 2013. ACM.

[130] T. Grosser and T. Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the International Conference on Supercomputing (ICS '16)*, pages 1:1–1:13, Istanbul, Turkey, June 2016. ACM.

[131] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 12:1–12:11, Seattle, DC, USA, November 2011. ACM.

[132] J. Harris and M. Scheutz. New Advances in Asynchronous Agent-based Scheduling. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '12)*, pages 1–7, Las Vegas, NV, USA, July 2012. WorldComp.

[133] M. Lysenko and R. M. D'Souza. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *JASSS Journal of Artificial Societies and Social Simulation*, 11(4):10, October 2008.

[134] H. Park and P. A. Fishwick. A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. *SCSI Simulation*, 86(10):613–628, October 2010.

[135] B. P. Swenson. *Techniques to Improve the Performance of Large-Scale Discrete-Event Simulation*. Dissertation, Georgia Institute of Technology, May 2015.

[136] X. Liu and P. Andelfinger. Time Warp on the GPU: Design and Assessment. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '17)*, pages 109–120, Singapore, May 2017. ACM.

[137] T. Wenjie, Y. Yiping, and Z. Feng. An Expansion-Aided Synchronous Conservative Time Management Algorithm on GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '13)*, pages 367–372, Montreal, Canada, May 2013. ACM.

[138] N. Baudis, F. Jacob, and P. Andelfinger. Performance Evaluation of Priority Queues for Fine-Grained Parallel Tasks on GPUs. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '17)*, pages 1–11, Banff, Canada, September 2017. IEEE.

[139] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of the International Conference on Computer Graphics and Interactive Techiques (SIGGRAPH '04)*, pages 777–786, Los Angeles, CA, USA, August 2004. ACM.

[140] R. M. Fujimoto, C. Carothers, A. Ferscha, D. Jefferson, M. Loper, M. Marathe, and S. J. Taylor. Computational Challenges in Modeling & Simulation of Complex Systems. In *Proceedings of the Winter Simulation Conference (WSC '17)*, pages 431–445, Las Vegas, NV, USA, December 2017. IEEE.

[141] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[142] B. Cosenza, G. Cordasco, R. De Chiara, and V. Scarano. Distributed Load Balancing for Parallel Agent-Based Simulations. In *Proceedings of the International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '11)*, pages 62–69, Ayia Napa, Cyprus, February 2011. IEEE.

[143] Q. Long, J. Lin, and Z. Sun. Agent Scheduling Model for Adaptive Dynamic Load Balancing in Agent-Based Distributed Simulations. *Elsevier Journal of Simulation Modelling Practice and Theory*, 19(4):1021–1034, April 2011.

[144] Y. Xu, W. Cai, D. Eckhoff, S. Nair, and A. Knoll. A Graph Partitioning Algorithm for Parallel Agent-Based Road Traffic Simulation. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '17)*, pages 209–219, Singapore, May 2017. ACM.

[145] T. Grosser, A. Groesslinger, and C. Lengauer. Polly—Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *World Scientific Parallel Processing Letters*, 22(04):1250010, December 2012.

[146] L.-N. Pouchet. Polybench: the Polyhedral Benchmark suite, 2012. URL: `http://web.cs.ucla.edu/~pouchet/software/polybench/`.

[147] U. Brüning, W. K. Giloi, and W. Schroeder-Preikschat. Latency Hiding in Message-Passing Architectures. In *Proceedings of the International Parallel Processing Symposium (IPPS '94)*, pages 704–709, Cancun, Mexico, April 1994. IEEE.

[148] C. Ding and Y. He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '01)*, pages 55–55, Denver, CO, USA, November 2001. IEEE.

[149] R. M. Fujimoto. Research Challenges in Parallel and Distributed Simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(4):22:1–22:29, May 2016.

[150] P. Heywood, S. Maddock, J. Casas, D. Garcia, M. Brackstone, and P. Richmond. Data-parallel Agent-based Microscopic Road Network Simulation Using Graphics Processing Units. *Elsevier Simulation Modelling Practice and Theory*, 83:188–200, April 2018.

[151] M. Yang, P. Andelfinger, W. Cai, and A. Knoll. Evaluation of Conflict Resolution Methods for Agent-Based Simulation on the GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS '18)*, pages 129–132, Rome, Italy, May 2018. ACM.

[152] J. Goldenberg, B. Libai, and E. Muller. Talk of the Network: A Complex Systems Look at the Underlying Process of Word-of-Mouth. *Springer Marketing letters*, 12(3):211–223, August 2001.

[153] M. Granovetter. Threshold Models of Collective Behavior. *University of Chicago Press American Journal of Sociology*, 83(6):1420–1443, 1978.

[154] R. Neumann and F. Strack. 'Mood Contagion": The Automatic Transfer of Mood Between Persons. *American Psychological Association Journal of Personality and Social Psychology*, 79(2):211, August 2000.

[155] J. D. Hess, J. J. Kacen, and J. Kim. Mood-Management Dynamics: The Interrelationship Between Moods and Behaviours. *Wiley Online Library British Journal of Mathematical and Statistical Psychology*, 59(2):347–378, November 2006.

[156] M. Gardner. Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American*, 223(4):120–123, October 1970.

[157] T. C. Schelling. *Micromotives and Macrobehavior*. WW Norton & Company, New York City, NY, USA, 2006.

[158] A. Kolb and L. John. Volumetric Model Repair for Virtual Reality Applications. In *EUROGRAPHICS Short Presentation (2001)*, pages 249–256, Manchester, England, September 2001. The Eurographics Association.

[159] T. Harada. Real-Time Rigid Body Simulation on GPUs. *NVIDIA GPU Gems*, 3:123–148, December 2007.

[160] S. Green. Particle Simulation Using Cuda. *NVIDIA Whitepaper*, 6:121–128, 2010.

[161] C. Oat, J. Barczak, and J. Shopf. Efficient Spatial Binning on the GPU. Technical report, Advanced Micro Devices, Inc, February 2009.

[162] R. Rönngren and R. Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, April 1997.

[163] X. He, D. Agarwal, and S. K. Prasad. Design and Implementation of a Parallel Priority Queue on Many-Core Architectures. In *Proceedings of the International Conference on High Performance Computing (HiPC '12)*, pages 1–10, Pune, India, December 2012. IEEE.

[164] N. Deo and S. Prasad. Parallel Heap: An Optimal Parallel Priority Queue. *Springer Journal of Supercomputing*, 6(1):87–98, March 1992.

[165] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications (INFOCOM '00)*, pages 538–547, Tel Aviv, Israel, March 2000. IEEE.

[166] D. D. Sleator and R. E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[167] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[168] M. K. Chimeh and P. Richmond. Simulating Heterogeneous Behaviours in Complex Systems on GPUs. *Elsevier Simulation Modelling Practice and Theory*, 83:3–17, April 2018.

[169] S. Eilenberg. *Automata, Languages, and Machines*. Academic Press, Cambridge, MA, USA, 1974.

[170] M. Holcombe. X-Machines as a Basis for Dynamic System Specification. *IET Software Engineering Journal*, 3(2):69–76, March 1988.

[171] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In *Proceedings of the 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 243–252, Madrid, Spain, October 2018.

[172] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[173] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA, 2002.

[174] D. Majeti, K. S. Meel, R. Barik, and V. Sarkar. Automatic Data Layout Generation and Kernel Mapping for CPU+GPU Architectures. In *Proceedings of the International Conference on Compiler Construction (CC '16)*, pages 240–250, Barcelona, Spain, March 2016. ACM.

[175] J. Xiao, K. Görkem, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. Pedal to the Bare Metal: Road Traffic Simulation on FPGAs Using High-Level Synthesis.

In *Proceedings of the 20th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, page 117–121, Miami, FL, USA, June 2020. ACM.

[176] D. W. Bauer, M. McMahon, and E. H. Page. An Approach for the Effective Utilization of GP-GPUs in Parallel Combined Simulation. In *Proceedings of the Winter Simulation Conference (WSC '08)*, pages 695–702, Miami, FL, USA, December 2008. IEEE.

[177] P. Andelfinger, J. Mittag, and H. Hartenstein. GPU-Based Architectures and Their Benefit for Accurate and Efficient Wireless Network Simulations. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '11)*, pages 421–424, Singapore, July 2011. IEEE.

[178] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. GPU-Based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios. In *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 95–103, Lake Placid, NY, USA, June 2009. IEEE.

[179] M. Bando, K. Hasebe, K. Nakanishi, A. Nakayama, A. Shibata, and Y. Sugiyama. Phenomenological Study of Dynamical Model of Traffic Flow. *Journal de Physique I*, 5(11):1389–1399, November 1995.

[180] M. Treiber, A. Hennecke, and D. Helbing. Congested Traffic States in Empirical Observations and Microscopic Simulations. *Physical review E*, 62(2):1805, August 2000.

[181] M. Treiber and A. Kesting. An Open-source Microscopic Traffic Simulator. *IEEE Intelligent Transportation Systems Magazine*, 2(3):6–13, December 2010.

[182] P. Heywood, P. Richmond, and S. Maddock. Road Network Simulation using FLAME GPU. In *Proceedings of the European Conference on Parallel Processing (Euro-Par '15)*, pages 430–441, Vienna, Austria, August 2015. Springer.

[183] P. G. Gipps. A Behavioural Car-Following Model for Computer Simulation. *Transportation Research Part B: Methodological*, 15(2):105–111, April 1981.

[184] X. Ma and I. Andréasson. Driver Reaction Delay Estimation from Real Data and its Application in GM-type Model Evaluation. *Transportation Research Record*, (1965):130–141, December 2006.

[185] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang. Understanding performance differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, Boulder, CO, USA, 2018. IEEE.

[186] S. Rahman, N. Abu-Ghazaleh, and W. Najjar. Pdes-a: A parallel discrete event simulation accelerator for fpgas. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 133–144, Singapore, 2017. ACM.

[187] C. Schäck, R. Hoffmann, and W. Heenes. Efficient traffic simulation using the gca model. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–7, Atlanta, GA, USA, 2010. IEEE.

[188] D. Zehe, S. Nair, A. Knoll, and D. Eckhoff. Towards CityMoS: A Coupled City-Scale Mobility Simulation Framework. In *5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication*, Erlangen, Germany, April 2017. FAU Erlangen-Nuremberg.

[189] R. Bi, J. Xiao, V. Viswanathan, and A. Knoll. Influence of charging behaviour given charging infrastructure specification: A case study of singapore. *Journal of Computational Science*, 20:118–128, 2017.

[190] D. Pelzer, J. Xiao, D. Zehe, M. H. Lees, A. C. Knoll, and H. Aydt. A partition-based match making algorithm for dynamic ridesharing. *IEEE Transactions on Intelligent Transportation Systems*, 16(5):2587–2598, 2015.

[191] K. Ahmed, M. Ben-Akiva, H. Koutsopoulos, and R. Mishalani. Models of Freeway Lane Changing and Gap Acceptance Behavior. In *Proceedings of the International Symposium on Transportation and Traffic Theory (ISTTT '96)*, pages 501–515, Lyon, France, July 1996. Elsevier.

[192] K. I. Ahmed. *Modeling Drivers' Acceleration and Lane Changing Behavior*. PhD thesis, Massachusetts Institute of Technology, 1999.

[193] S. El hadouaj, A. Drogoul, and S. Espié. How to Combine Reactivity and Anticipation: The Case of Conflicts Resolution in a Simulated Road Traffic. In S. Moss and P. Davidsson, editors, *Proceedings of the International Workshop on Multi-Agent Systems and Agent-Based Simulation (MABS '00)*, pages 82–96, Boston, MA, USA, July 2000. Springer.

[194] Y. Xu, W. Cai, H. Aydt, M. Lees, and D. Zehe. An Asynchronous Synchronization Strategy for Parallel Large-scale Agent-based Traffic Simulations. In *Proceedings of the Conference on Principles of Advanced Discrete Simulation (PADS '15)*, pages 259–269, London, UK, June 2015. ACM.

[195] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers. ACM-Press Frontier Series.* Addison-Wesley, 1990.

[196] J. Liu, Y. Liu, Z. Du, and T. Li. GPU-Assisted Hybrid Network Traffic Model. In *Proceedings of the Conference on Principles of Advanced Discrete Simulation (PADS '14)*, pages 63–74, Denver, CO, USA, May 2014. ACM.

[197] A. Lake. Getting the Most from OpenCL™ 1.2: How to Increase Performance by Minimizing Buffer Copies on Intel® Processor Graphics, September 2014. URL: `https://software.intel.com/sites/default/files/managed/f1/25/opencl-zero-copy-in-opencl-1-2.pdf`.

[198] C. Cummins, P. Petoumenos, M. Steuwer, and H. Leather. Autotuning OpenCL Workgroup Size for Stencil Patterns. *arXiv preprint arXiv:1511.02490*, November 2015.

[199] X. Liu and P. Andelfinger. Time Warp on the GPU: Design and Assessment. In *Proceedings of the Conference on Principles of Advanced Discrete Simulation (PADS '17)*, pages 109–120, Singapore, May 2017. ACM.

[200] K. Rupnow, Y. Liang, Y. Li, and D. Chen. A study of high-level synthesis: Promises and challenges. In *2011 9th IEEE International Conference on ASIC*, pages 1102–1105, Xiamen, China, 2011. IEEE.

[201] D. Curd. Pci express for the 7 series fpgas, 2012.

[202] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama. *Design of FPGA-based computing systems with OpenCL.* Springer, 2018.

[203] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig. Deep learning with int8 optimization on xilinx devices. *White Paper*, 2016.

[204] P. Andelfinger, J. Ivanchev, D. Eckhoff, W. Cai, and A. Knoll. From effects to causes: Reversible simulation and reverse exploration of microscopic traffic models. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 173–184, Chicago, IL, USA, 2019. ACM.

[205] J. Xiao, P. Andelfinger, W. Cai, P. Richmond, A. Knoll, and D. Eckhoff. Openablext: An automatic code generation framework for agent-based simulations on cpu-gpu-fpga heterogeneous platforms. *Concurrency and Computation: Practice and Experience*, page e5807, 2020. doi:10.1002/CPE.5807.

[206] J. Xiao, P. Andelfinger, W. Cai, P. Richmond, A. Knoll, and D. Eckhoff. Advancing Automatic Code Generation for Agent-Based Simulations on Heterogeneous Hardware. In *Proceedings of the European Conference on Parallel Processing*, Göttingen, Germany, 2019. Springer.

[207] M. J. North, N. T. Collier, and J. R. Vos. Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit. *ACM Trans. Model. Comput. Simul.*, 16(1):1–25, 2006.

[208] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough. FLAME: Simulating Large Populations of Agents on Parallel Hardware Architectures. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pages 1633–1636. IFAAMAS, 2010.

[209] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. A Framework for Distributing Agent-based Simulations. In *Alexander M. et al. (eds) Euro-Par 2011*, volume 7155 of *LNCS*, pages 460–470, Heidelberg, 2011. Springer.

[210] N. Collier and M. North. *Large-Scale Computing Techniques for Complex System Simulations*. Wiley, Hoboken, NJ, USA, 2011.

[211] P. Richmond, D. Walker, S. Coakley, and D. Romano. High Performance Cellular Level Agent-based Simulation with FLAME for the GPU. *Briefings Bioinf.*, 11(3):334–347, 2010.

[212] D. Grewe, Z. Wang, and M. F. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 1–10. IEEE, 2013.

[213] P. Li, E. Brunet, F. Trahay, C. Parrot, G. Thomas, and R. Namyst. Automatic OpenCL Code Generation for Multi-device Heterogeneous Architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 959–968. IEEE, 2015.

[214] K. Krommydas, R. Sasanka, and W.-c. Feng. Bridging the fpga programmability-portability gap via automatic opencl code generation and tuning. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 213–218. IEEE, 2016.

[215] J. Rohde, M. Martinez-Peiro, and R. Gadea-Girones. Socao: Source-to-source opencl compiler for intel-altera fpgas. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 1–7. VDE, 2017.

[216] R. Farber. *Parallel programming with OpenACC*. Newnes, 2016.

[217] S. Lee, J. Kim, and J. S. Vetter. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554. IEEE, 2016.

[218] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A Compiler Architecture for Performance-oriented Embedded Domain-specific Languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134, 2014.

[219] B. Johnston, G. Falzon, and J. Milthorpe. Opencl performance prediction using architecture-independent features. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 561–569. IEEE, 2018.

[220] K. Moren and D. Göhringer. Automatic mapping for opencl-programs on cpu/gpu heterogeneous platforms. In *International Conference on Computational Science*, pages 301–314. Springer, 2018.

[221] M. Ahmad, H. Dogan, C. J. Michael, and O. Khan. Heteromap: A runtime performance predictor for efficient processing of graph analytics on heterogeneous multi-accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 268–281. IEEE, 2019.

[222] P. Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 273. ACM, 2014.

[223] B. Pérez, E. Stafford, J. L. Bosque, R. Beivide, S. Mateo, X. Teruel, X. Martorell, and E. Ayguadé. Auto-tuned opencl kernel co-execution in ompss for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 125:45–57, 2019.

[224] M. A. D. Guzman, R. Nozal, R. G. Tejero, M. Villarroya-Gaudó, D. S. Gracia, and J. L. Bosque. Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL. *The Journal of Supercomputing*, 75(3):1732–1746, 2019.

[225] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, and R. Asenjo. Heterogeneous parallel_for template for cpu–gpu chips. *International Journal of Parallel Programming*, 47(2):213–233, 2019.

[226] W. Liu and B. Vinter. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[227] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky. A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2176–2190, 2018.

[228] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.

[229] Y. Xu, W. Cai, H. Aydt, and M. Lees. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. In *Proceedings of the Winter Simulation Conference 2014*, pages 3483–3494. IEEE, 2014.

[230] P. Huchant, D. Barthou, and M.-C. Counilh. Adaptive partitioning for iterated sequences of irregular opencl kernels. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 262–265. IEEE, 2018.

[231] A. M. Aji, A. J. Pena, P. Balaji, and W.-c. Feng. Automatic command queue scheduling for task-parallel workloads in opencl. In *2015 IEEE International Conference on Cluster Computing*, pages 42–51. IEEE, 2015.

[232] P. Richmond. Resolving Conflicts between Multiple Competing Agents in Parallel Simulations. In *Lopes L. et al. (eds) Euro-Par 2014*, volume 8805 of *LNCS*, pages 383–394, Cham, 2014. Springer.

[233] X. Li, W. Cai, and S. J. Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of the International Symposium on Distributed Simulation and Real Time Applications*, pages 87–96. IEEE, 2014.

[234] A. O. Mahony and E. Popovici. Power analysis of sorting algorithms on fpga using opencl. In *2018 29th Irish Signals and Systems Conference (ISSC)*, pages 1–6. IEEE, 2018.

[235] J. M. Epstein and R. Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. Brookings Institution Press, 1996.

[236] R. Chisholm, P. Richmond, and S. Maddock. A Standardised Benchmark for Assessing the Performance of Fixed Radius Near Neighbours. In *Desprez F. et al. (eds) Euro-Par 2016*, volume 10104 of *LNCS*, pages 311–321, Cham, 2016. Springer.

[237] J. M. Epstein and R. Axtell. Artificial societies and generative social science. *Artificial Life and Robotics*, 1(1):33–34, 1997.

[238] L. Panait and S. Luke. A pheromone-based utility model for collaborative foraging. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, pages 36–43. IEEE, 2004.

[239] N. Pelechano and N. I. Badler. Modeling Crowd and Trained Leader Behavior during Building Evacuation. *IEEE Comput. Graphics Appl.*, 26(6):80–86, 2006.

[240] Codeplay. Codeplay:computecpp. `https://www.codeplay.com/products/computecpp/`. Accessed: 2020-07-30.

[241] A. Trigkas. Investigation of the OpenCL SYCL programming model. *Master's thesis, The University of Edinburgh, UK*, 2014.

[242] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 47(4):69:1–69:35, July 2015.

[243] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. 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.

[244] P. Tillet, K. Rupp, and S. Selberherr. An automatic OpenCL compute kernel generator for basic linear algebra operations. In *Proceedings of the 2012 Sym-*

posium on High Performance Computing, pages 1–2, Orlando, FL, USA, 2012. ACM.

[245] K. J. Brown, H. Lee, T. Romp, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 194–205, Barcelona, Spain, 2016. IEEE.

[246] H. Riebler, G. Vaz, T. Kenter, and C. Plessl. Transparent acceleration for heterogeneous platforms with compilation to opencl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–26, 2019.

[247] R. Sotomayor, L. M. Sanchez, J. G. Blas, J. Fernandez, and J. D. Garcia. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. *International Journal of Parallel Programming*, 45(2):262–282, 2017.

[248] D. Majeti and V. Sarkar. Heterogeneous Habanero-C (H2C): a portable programming model for heterogeneous processors. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 708–717, Hyderabad, India, 2015. IEEE.

[249] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar. Mapping a data-flow programming model onto heterogeneous platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, page 61–70, Beijing, China, 2012. ACM. doi:10.1145/2248418.2248428.

[250] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 431–444, Houston, Texas, USA, 2013. ACM.

[251] A. Chikin, J. N. Amaral, K. Ali, and E. Tiotto. Toward an analytical performance model to select between gpu and cpu execution. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 353–362, Rio de Janeiro, Brazil, 2019. IEEE.

[252] D. Grewe and M. F. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *International conference on compiler construction*, pages 286–305, Saarbrücken, Germany, 2011. Springer.

[253] A. D. Pereira, R. C. Rocha, L. Ramos, M. Castro, and L. F. Góes. Automatic partitioning of stencil computations on heterogeneous systems. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 43–48, Campinas, Brazil, 2017. IEEE.

[254] B. Pérez, E. Stafford, J. Bosque, and R. Beivide. Sigmoid: An auto-tuned load balancing algorithm for heterogeneous systems. *Journal of Parallel and Distributed Computing*, 157:30–42, Nov 2021.

[255] B. Pérez, J. L. Bosque, and R. Beivide. Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, page 42–51, Barcelona, Spain, 2016. ACM.

[256] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, et al. Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 79–90, Mumbai, India, 2019. ACM.

[257] J. Price and S. McIntosh-Smith. Oclgrind: An extensible opencl device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, Palo Alto, CA, USA, 2015. ACM.

[258] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[259] D. M. Rao, N. V. Thondugulam, R. Radhakrishnan, and P. A. Wilsey. Unsynchronized parallel discrete event simulation. In *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, volume 2, pages 1563–1570, Washington, USA, 1998. IEEE.

[260] S. Ohshima, I. Yamazaki, A. Ida, and R. Yokota. Optimization of hierarchical matrix computation on GPU. In *Asian Conference on Supercomputing Frontiers*, pages 274–292, Singapore, 2018. Springer.