A Survey on Agent-based Simulation Using Hardware Accelerators

JIAJIAN XIAO, TUM Create Ltd. & Technische Universität München
PHILIPP ANDELFINGER, TUM Create Ltd. & Nanyang Technological University
DAVID ECKHOFF, TUM Create Ltd. & Technische Universität München
WENTONG CAI, Nanyang Technological University
ALOIS KNOLL, Technische Universität München & Nanyang Technological University

Due to decelerating gains in single-core CPU performance, computationally expensive simulations are increasingly executed on highly parallel hardware platforms. Agent-based simulations, where simulated entities act with a certain degree of autonomy, frequently provide ample opportunities for parallelisation. Thus, a vast variety of approaches proposed in the literature demonstrated considerable performance gains using hardware platforms such as many-core CPUs and GPUs, merged CPU-GPU chips as well as Field Programmable Gate Arrays. Typically, a combination of techniques is required to achieve high performance for a given simulation model, putting substantial burden on modellers. To the best of our knowledge, no systematic overview of techniques for agent-based simulations on hardware accelerators has been given in the literature. To close this gap, we provide an overview and categorisation of the literature according to the applied techniques. Since, at the current state of research, challenges such as the partitioning of a model for execution on heterogeneous hardware are still addressed in a largely manual process, we sketch directions for future research towards automating the hardware mapping and execution. This survey targets 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.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Computing methodologies → Distributed simulation**; • **Hardware → Hardware accelerators**;

Additional Key Words and Phrases: Agent-based simulation, heterogeneous computing

ACM Reference format:
Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2019. A Survey on Agent-based Simulation Using Hardware Accelerators. *ACM Comput. Surv.* 51, 6, Article 131 (January 2019), 35 pages.
https://doi.org/10.1145/3291048

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.
Authors' addresses: J. Xiao, P. Andelfinger, and D. Eckhoff, TUM Create Ltd. & Technische Universität München, Area-Interlinking Design Analysis, Singapore; emails: {jiajian.xiao, philipp.andelfinger, david.eckhoff}@tum-create.edu.sg; W. Cai, Nanyang Technological University, School of Computer Science and Engineering, Singapore; email: david.eckhoff@tum-create.edu.sg; A. Knoll, Technische Universität München & Nanyang Technological University, Department of Informatics, Munich; email: knoll@in.tum.de.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0360-0300/2019/01-ART131 $15.00
https://doi.org/10.1145/3291048

1 INTRODUCTION

Since around 2005, it can be observed that, due to the breakdown of Dennard scaling, clock frequencies of single CPUs are no longer increasing significantly, even though the transistor counts are still growing [167]. Instead, CPU manufacturers have more and more focused on developing multi-core processors. This in turn calls for parallel computing techniques, as programs (including simulations) that cannot be run in parallel can no longer simply be sped up by incorporating a newer and faster CPU. Performance can be increased further when the workload of a programme is efficiently distributed to heterogeneous hardware such as Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs) [30].

Some types of hardware are better suited for certain tasks than others, for example, tasks with large amounts of fine-grained parallelism can benefit greatly from the massively parallel architecture of modern GPUs with its thousands of cores. Tasks that are largely sequential or characterised by unpredictable data accesses 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, then 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. Last, compute-intensive and memory-light tasks can be outsourced to FPGAs that can be programmed to carry out specific computations in hardware.

One field that has always sought after more performance is the field of simulation. Faster computers allow an increase in complexity of the incorporated simulation models, allowing researchers to obtain more accurate results in a faster manner. Agent-based simulations have received broad attention as they can be employed to study various domains, such as road traffic [46], social networks [49], pedestrian movement [170], military [33], biology [6], economics [171], privacy[179], and so on. The main characteristic of agent-based simulation 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 runtimes, motivating the need for employing high-performance computing platforms.

Agent-based simulations (ABS) are a promising target for parallel computing techniques as agents are autonomous and in some cases carry out independent computations. In mobility simulations, interactions between agents usually only take place between close-by agents in a somewhat regular two-dimensional (2D) or 3D environment, allowing researchers to employ space partitioning without inducing too much synchronisation overhead. Moreover, many ABS are timestepped and agents are often updated at the same logical time, providing inherent independence and thus potentials for parallelised execution. Unfortunately, being able to partition a problem and execute it in parallel is not a guarantee that it can be accelerated using heterogeneous hardware.

To enable ABS on heterogeneous hardware, some general challenges have to be overcome. First, the simulation has 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. From this it follows that, depending on the used 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. Last, to make heterogeneous accelerators available to modellers even without having in-depth knowledge of the specific hardware platforms, there is a need for frameworks that abstract away from hardware specifics. Some of the common frameworks provide variants supporting parallel and distributed execution, e.g., MASON [112], Repast-HPC [36], and EcoLab [164]. However, these frameworks only support traditional CPU-based environments. Some frameworks such as FLAME GPU [35] and MCMAS [103] have been proposed that focus on the execution on specific accelerators such as graphics cards.

In this survey, we structure the complex landscape of agent-based simulation on heterogeneous hardware. We give an overview of existing types of hardware that have been employed to accelerate agent-based simulations and discuss past developments and current trends. While some surveys exist that present generic high-performance computing techniques using heterogeneous hardware [50, 120, 176], we highlight the specific challenges of ABS on heterogeneous hardware and categorise an ample body of related work along these challenges. For each challenge, we discuss in detail how existing literature has contributed to solving them. This overview allows us to identify research gaps that need to be filled to establish heterogeneous accelerators in the simulation domain and make them applicable to a wider range of problems—ideally by providing an automated process to support the modeller.

The remainder of this survey is structured as follows: In Section 2, we characterise the main classes of hardware accelerators for general-purpose computations. Section 3 provides an overview of agent-based simulation concepts and outlines the computational challenges of executing agent-based simulations on hardware accelerators. In Section 4, we systematise and survey the existing works according to the identified challenges and according to the techniques used to do so. In Section 5, we discuss unresolved challenges and outline how a system tackling these challenges could look like, thus sketching avenues for future work. Section 6 summarises our findings and concludes the survey.

2 HARDWARE PLATFORMS

In this section, we describe the technical characteristics of hardware platforms that have been used to accelerate agent-based simulations, covering many-core CPUs, Graphics Processing Unit (GPUs), Accelerated Processing Units (APUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and System on Chips (SoCs). Readers familiar with these platforms may skip this section and continue to Section 3.

2.1 Many-Core CPUs

Architecture: A many-core (or many integrated core, MIC) CPU contains a group of CPU cores on a single chip. One of the well-known many-core CPUs, the Intel Xeon Phi, is equipped with up to 72 x86-compatible CPU cores communicating via an internal Network-on-Chip that enables fast and parallel data transfer between the cores. 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.

In the past years, a number of non-x86 many-core CPUs have emerged, such as the Parallella Board [4], the Epiphany-V [135], and the Kalray MPPA (Massively Parallel Processor Array) [43].

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 [101]. 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 [24]. Further parallelism may be extracted using a single-instruction, multiple-data (SIMD) style of programming using instruction set extensions such as AVX-512 [86].

Recent work showed that many-core CPUs can substantially accelerate Discrete Event Simulation (DES) [89, 184]. 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-cores [29, 148].

Limitations: In light of the comparatively 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., Reference [13]). 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 [110].

2.2 Graphics Processing Units (GPUs)

Architecture: GPUs utilise a massively parallel architecture, which makes them more efficient than CPUs when large volumes of data can be processed in parallel. 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 CUDA [133], OpenACC [136], and OpenCL [165]. 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 [130].

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 SPs to minimise stalling on high-latency memory accesses. Typically, programs schedule many more threads than there are physical SPs to support this type of memory latency hiding [133].

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 [133]. Generally, the number of memory requests is minimised when adjacent threads access adjacent memory locations. Achieving memory coalescing is a common focus of works on GPU acceleration (e.g., References [52, 186]).

Benefits: GPUs lend themselves to problems that can be expressed 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 [18] and CUBLAS [129] enable relatively simple development compared to platforms such as FPGAs [51]. Programming frameworks are available even for more specialised tasks such as ABS [35].

Beside the performance benefits of GPUs, Richmond and Romano [152] emphasise the opportunities for efficient visualisation of simulations. Since the simulation data are already stored in graphics memory, visualisation can be achieved easily, for instance, by writing to texture buffers.

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 [27, 91].

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 16GB/s, while the GPU can access its off-chip RAM at up to 336.5GB/s. However, the impact of data transfers may be lower when relying on recent architectures' interconnects such as NVIDIA's NVLink [131] and AMD's Infinity Fabric [3], which achieve throughputs of up to 300GB/s.

Compared to many-core CPUs, programming for GPUs still requires profound knowledge of the GPU architecture [185]. As with many-core CPUs, the large number of configurable parameters render the performance tuning of GPU programs an important but challenging task [174].

2.3 Accelerated Processing Units (APUs)

Architecture: APUs integrate CPU and GPU on a single chip. Although the term APUs has been coined by AMD, recent 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 have focused 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 4, some works have considered APUs for accelerating agent-based simulations.

2.4 Field-Programmable Gate Arrays (FPGAs)

Architecture: A Field-Programmable Gate Array 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 Tables (LUTs). 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 [70]. In addition, the FPGA may have access to several GB of off-chip DRAM.

The logic to be placed on an FPGA is typically specified in a Hardware Description Language (HDL) such as VHDL [72] or Verilog [137]. 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, Intel released a dedicated SDK to support FPGA programming using OpenCL [85].

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 Data Encryption Standard (DES) [30]. In contrast to CPUs or GPUs, on which data paths are fixed, FPGAs provide flexible and customised data paths [149]. In the past years, FPGAs have received more attention in the field of simulation, particularly in Electronic Design Automation, since hardware designs can be naturally expressed as FPGA layouts.

Limitations: As with GPUs, FPGAs are 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 [30, 51]. Although recent efforts towards high-level synthesis alleviate this limitation, manual tuning is still necessary to achieve the best performance [55, 123].

Finally, FPGAs are configured for a specific task. Since reconfiguration can take multiple hours [198], 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.

2.5 Other Hardware Platforms

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, ASICs and System on Chip devices have not yet been explored as platforms to accelerate ABS. However, in the field of DES, some works have considered offloading to ASICs [20, 34, 57, 113, 150]. Notably, some of the envisioned components were fabricated physically [20].

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 [94]. A potential use case for TPUs in ABS lies in the acceleration of agent models relying on neural networks (e.g., Reference [138]). When exploring the input parameter space of an ABS (e.g., Reference [28]), TPUs could also accelerate machine-learning algorithms used to steer the exploration.

NVIDIA's Xavier is a SoC targeted towards use in autonomous vehicles, focusing on deep learning and vision tasks. The system is equipped with 512 Stream Processors, an eight-core ARM CPU, and a processor designed for vision tasks [132]. 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., Reference [80]). 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. Thus, these platforms will not be considered in the remainder of the survey.

3 AGENT-BASED SIMULATION

Agent-based modelling and simulation (ABMS) is a widely used approach [128] 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 [112], Repast [36], NetLogo [173], Swarm [119], MATSim [83]), 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 [5]. More recently, Abar et al. [2] 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. There exists a number of surveys evaluating aspects such as performance, usability and extensibility [5, 23, 99, 126, 127, 156, 172]; other literature reviews put a particular focus on specific domains, including economics [124], energy [197], geology [117], and sociology [61]. Last, other works try to identify the fundamental elements and principles of ABMS to formulate basic simulator architectures [12, 84].

3.1 Modelling Approach

In ABS, the simulated entities are agents that perform actions autonomously and interact with other agents based on certain rules. ABS typically follows a *Sense-Think-Act* cycle (e.g., Reference

[154]): 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 at which all agents update their states. However, if a model requires agents to update their states at variable points in simulation time, time advancement using a *discrete-event simulation* (DES) approach may be more appropriate. In DES, state updates are performed through events scheduled for execution at discrete points in simulation time. The simulation proceeds by iteratively executing the earliest remaining event, potentially scheduling new events in the process.

Independent of the time advancement mechanism, a defining characteristic of ABS distinguishing it from other simulation techniques is the *autonomy* of agents, i.e., "agents are endowed with behaviours that allow them to make independent decisions" [115]. Since the focus of this survey paper is on ABS, we exclude simulation domains such as physics and chemistry, which usually consider sets of entities that are passively affected by their environment. However, we do discuss a number of methods proposed outside of the ABS domain with direct applications to ABS, e.g., GPU-based priority queues in the context of DES.

3.2 Computational Aspects

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

Crooks and Heppenstall summarise the literature on definitions of the term "agent" using the core features of autonomy (independent decision-making), heterogeneity (opportunity for different attributes per agent), and being active (independent influence on the simulation) [40]. The feature of "being active" is further described using a number of sub-features including mobility (ability to move in the simulation space) and interactivity (ability to communicate extensively).

From this definition, we identify a number of features with direct implications on the execution of agent-based simulations on parallel hardware as follows:

- **Autonomy, Heterogeneity, and Mobility**: The independent decision-making per agent provides opportunities for parallelisation across the computations involved in the individual agents' Sense-Think-Act cycle. However, since agents move and act according to their individual attributes as well as their internal states and current environment, the actions of an agent population are commonly diverse both in time and space. For instance, at certain points in simulated time there may be areas of the simulation space with little activity and thus low computational demands, while areas populated with highly active agents may require intense computations. Thus, the heterogeneity and mobility of agents make it challenging to balance the workload among the available processing elements. Further difficulties are given by the diversity in the type of action performed, since some hardware accelerators achieve highest performance when executing large numbers of identical computations in parallel.
- **Interactivity**: The communication among agents is typically reflected by memory accesses when the communicating agents are simulated on the same device, or by data transfers over a bus like PCI-E when crossing device boundaries. Since the agents' communication patterns are usually not fully predictable, exploiting regularities in the communication to increase performance, e.g., by storing messages in low-latency memory shared by two communicating agents, is non-trivial. Further, given autonomous and mobile agents, even under

a highly suitable and dynamic hardware assignment there will typically still be a need for substantial amounts of data transfers among processing elements.

When considering the modelling and simulation life cycle, another important property is given by the frequent changes to the model code in the iterative process commonly applied when developing simulation models. As a property of the agent-based modelling and simulation life cycle, it has been stated that modellers iteratively extend and refine models to "balance the expressive power of more details with the cost of additional complications" and that in fact "iterative model development and use is the ABMS paradigm" [128]. Given the resulting frequent changes to the models and the difficulty predicting the runtime behaviour resulting from the features above, there is insufficient information about the data dependencies, instruction mix or control flow to develop an overall simulation program optimised for a particular model. This is in contrast to well-defined tasks such as linear algebra operations, which allow for the development of highly optimised libraries targeting specific hardware platforms. Instead, there have been attempts to provide efficient facilities for tasks common to many agent-based simulations, allowing developers to specify models without the need to consider low-level specifics of the given hardware platform.

We summarise the above discussion by identifying five challenges for agent-based simulations on heterogeneous hardware platforms along which the remainder of the survey will be organised. While the challenges are shared by other types of workloads, the features discussed above make the challenges particularly common and pressing in agent-based simulations.

(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 [122]: *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 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

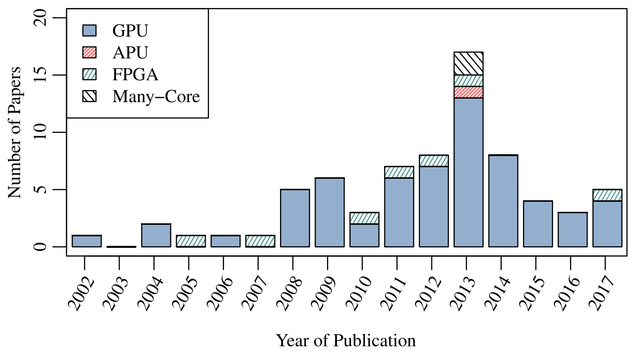


Fig. 1. Publications on agent-based simulation on heterogeneous hardware by year and hardware type.

Table 1. Simulation Model Domains Considered in the Works Covered in the Survey

Domain/Hardware	**Many-Core CPU**	**GPU**	**APU**	**FPGA**
Mobility		[9, 79, 80, 144, 146, 160, 163, 166, 181, 189]	[180]	[175]
Biology	[103]	[1, 47, 74, 76, 95, 107, 145, 151–153, 169, 199]		[41]
Ecology		[104, 192]		[178]
Social	[101]	[92, 93, 106, 108, 177, 185, 196]		[59]
Physics and Chemistry		[17, 69, 98, 118, 143, 159, 192]		[121]
Networks		[7, 8, 22, 100, 139, 141, 157, 169]		
Domain-independent Simulation Framework	[103]	[39, 87, 88, 103, 105, 151, 153, 190]		

the synchronisation of simulated time [53]. In the past 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 the past years, a number of frameworks that abstract from the hardware details as well as libraries for unified access to the memory of different devices have been presented in the literature.

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 focused on GPUs (see Figure 1 for an overview of the number of publications since 2002), mainly because they are comparatively inexpensive and because, in recent years, the ease of programming of GPUs is slowly 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 1 shows the simulation domains and hardware platforms, providing researchers with pointers to relevant works in their respective domain.

Our survey of the literature is organised along the key challenges we identified in Section 3, 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

Table 2. A Classification of the Challenges in ABS on Accelerators along the Relevant Works Addressing Them

Challenge	Technique	Publications
Hardware assignment	Static assignment by type of computation	Many-Core [101], GPU [8, 15, 22, 75, 80, 118, 142, 163, 189] [74, 76, 193], APU [180], FPGA [41, 59, 175, 178]
	Dynamic assignment based on runtime measurements	GPU [19, 63, 66, 96, 182, 193], FPGA [19]
Data transfer overheads	Overlapping of communication and computation	GPU [16, 17, 100]
	Computation replication at partition boundaries	GPU [1, 199]
Scattered memory accesses	Manual caching in shared memory	GPU [107, 151, 199]
	Heuristics for agent update order	GPU [9, 68, 92, 93]
	Representation of irregular data structures by arrays and grids	APU [180], GPU [47, 69, 98, 114, 144–146, 152, 166, 177] [7, 14, 95, 109, 140, 141, 159, 168, 183, 196], FPGA [121, 149]
Maximisation of parallelism	Multiple replications in parallel	GPU [100, 104, 108, 142, 160, 192]
	Window-based event execution	GPU [7, 26, 139, 141, 143, 157, 169, 196]
	Speculative execution	GPU [106, 109], FPGA [121]
	Computation sorting	GPU [95, 100, 169]
Abstraction from hardware specifics	Frameworks to support simulation development	Many-Core [103], GPU [39, 79, 103, 114, 151, 153]
	Unified memory access	GPU [87, 88, 105, 190]

the literature applicable to these key challenges in agent-based simulation. Table 2 summarises the systematisation of knowledge presented in this survey. It contains our classification of challenges, techniques, publications, and types of accelerators.

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 [56]. 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, then 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 5, we outline the vision of an automated approach and the required building blocks towards an automated hardware mapping for heterogeneous ABS.

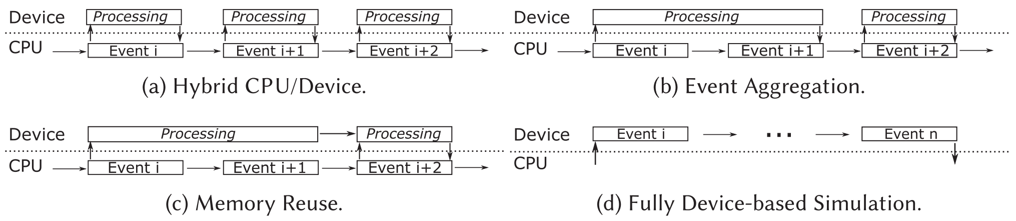
4.1.1 Static Assignment. The simplest hardware assignment is to execute the entire simulation on a single device. This approach is common in the existing work on FPGA-based ABS. For instance, Vourkas and Sirakoulis [178], who implement an environmental model simulation based on cellular automata (CA). The authors note the structural similarity between a two-dimensional cellular automaton and an FPGA and assign one cell to each Configurable Logic Block (CLB). If the number of cells exceeds the number of CLBs, then the simulation lattice is partitioned into several layers, which are processed one after the other. Similarly, Cui et al. [41] and Georgoudas et al. [59] 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 [1, 79, 80, 144–146]. 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 [122].

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. [101], who implement Game of Life [37] and a simulation of urban sprawl processes using cellular automata [186]. 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 [142] 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 [75, 118]. 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 [74, 76, 118] show substantial speedup when employing this method.



(a) Hybrid CPU/Device. (b) Event Aggregation. (c) Memory Reuse. (d) Fully Device-based Simulation.

Fig. 2. Four CPU-device simulation schemes [8]. Devices can be GPUs or many-core CPUs.

Some works have focused on isolating simulation code that heavily relies on floating point arithmetic to be executed on a GPU. Bauer et al. [15] 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. [8] 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(a)), 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(b)) and by leaving computation results required by subsequent events in graphics memory (cf. Figure 2(c)). 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(d)). 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, the agent behaviours are left to the CPU while environment dynamics are handled by the GPU [118]. 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.

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 as shown by Bilel et al. [22] in the context of large-scale mobile networks simulation can be applied. In the proposed design, the model is executed on the GPU, while the CPU orchestrates the event scheduling, simulation status monitoring, and memory allocation.

Finally, the nature of traffic simulation allows for a relatively straight-forward functional decomposition according to different simulation aspects. Xu et al. [189] and Song et al. [163] assign the 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 hide data transfer latencies. In the traffic simulation on an APU presented by Wang et al. [180], 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 accesses. Considering FPGAs, Tripp et al. [175] showed 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.

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., References [38, 111, 188]), 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. [19] 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 size 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 unit's 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 large 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. [63, 96] and Zhang et al. [193] translate a single-device OpenCL program to a multiple-device program, while Wen et al. [182] 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 Groesslinger [66] 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 [65]. 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 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 [147], 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 5, we sketch how the common structure of many ABS may be utilised to support the extraction of parallelism.

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 overheads by transferring data while independent computations are performed. Sometimes, the technique has been referred to as *latency hiding* (e.g., [25]).
2. **Computation replication at partition boundaries:** Another technique to address communication overheads 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.

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 are being transferred. In the approach described by Kunz et al. [100], 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 timestepped ABS by initiating the transfer of output data of some agents' state updates at a given timestep, while computations for other agents are still in progress.

Bauer et al. [16, 17] 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. *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.

4.2.2 Computation Replication at Partition Boundaries. In timestepped ABS, at model time t each agent updates its state based on the states of its neighbours at time $t-1$. If the simulation is distributed across multiple processing elements, then synchronisation and data transfers are required to provide this information at each timestep. The associated overhead may make up a substantial portion of the 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 [45]. 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. [1] 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 timestep, 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 timestep, a cell can only affect its immediate neighbours, R timesteps 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 timesteps. 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 timestep. 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 $R > 1$ timesteps, some synchronisations can be avoided. The generality of the approach is illustrated by Zou et al. [199], who extend the idea of computation replication to graph-based topologies in a GPU-accelerated epidemic ABS.

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 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 among 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 [54], most GPU-based simulators allocate memory for data such as the agent states statically (e.g., Reference [107]). Another approach is to determine after each timestep the required amount of memory and perform allocations accordingly [141].

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, aiming to 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*.

4.3.1 Manual Caching in Shared Memory. Richmond et al. [151] propose to utilise 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 [35], 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. [78] specialised their messaging method for traffic simulations on graph-based road networks. Messages are sorted by the 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. [199] 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. [107], assuming a constant number of agents, each agent is assigned to a GPU thread and its state data are permanently kept in global memory. The simulation space is partitioned into a grid of rectangles. Once a search for the neighbours within a circle around an agent is required, a search rectangle that encloses the searching circle is created, so 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 agents located in the union of all search rectangles of the agents' handled by the current block. If the shared memory is not sufficient to hold all agents' data, then the data are 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.

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 timestepped scheme in which agents always reside at the same timestep, after which conflicts in the resulting agent states are resolved [191]. 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 [9]. Harris and Scheutz have shown that distributed agent-based simulations can be accelerated by favouring agent updates that resolve dependencies across multiple processing elements [68]. 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 timestep.

Jin et al. [92] present an information propagation simulation supporting execution on HPC systems and single GPUs and extend it to run on multiple GPUs [93]. 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 [60] and the threshold model [62], 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.

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., References [151, 166]). 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 done by Harris et al. [69], 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. [114], Perumalla and Aaby [145], and Kolb et al. [98].

Perumalla et al. [145] evaluate the performance of running agent-based simulation entirely on a GPU. They ported the cellular models Mood Diffusion [77, 125], Game of Life [58] and Schelling Segregation [158]. 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. [98] develop a particle simulation and a GPU-based collision detection mechanism built on the authors' previous work [97]. Similarly, Richmond et al. [152] utilise the GPU's texture processing ability and map agent states onto texture data. To accelerate the 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 [64, 67]. Identification of the start and end of the partition boundary is performed similarly to the method described in Reference [134]. 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 [144] (and Perumalla and Aaby [146]) 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 [166], 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 [177] and a simulation study of tuberculosis [47]. 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 (PQ) if time advancement is performed in a discrete-event manner. These operations can involve large amounts of data-dependent and scattered memory accesses and are therefore challenging to implement efficiently on hardware accelerators. Since this operation can occupy a substantial portion of the simulation runtime [155], 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 [95, 140, 159, 180, 196]. 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. [71] propose a parallel heap-based PQ on GPU based on a previous CPU-based design [44]. 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. [149] relies on a pipelined heap [21] 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 [168], split by model segment [159] or simulated entity [7, 109, 196], split according to a fixed policy [141, 183], or split randomly [121]. 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. [14] 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 [162] 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.

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, then 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 timestepped execution. A key difference among the techniques lies in whether the simulation correctness is strictly maintained.
3. **Speculative execution:** As in general optimistic parallel and distributed simulation [53], 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.

4.4.1 Multiple Replications in Parallel. If an individual simulation run does not provide sufficient parallelism to fully utilise the available hardware, then a Multiple Replications in Parallel (MRIP) approach [142] can be applied, as shown by Shen et al. [160]: In their approach, multiple replications of a traffic simulation [181] are executed in parallel on a GPU. Thus, both the parallelism among agents as well as the parallelism across replications can be exploited. Laville et al. [104] 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. The idea is applied to discrete-event simulations by Kunz et al. [100], focusing on executing parameter studies comprised of multiple replications on a GPU.

In addition to exploiting the parallelism across replications, Li et al. [108] 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 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, then 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 [192].

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 are available for some of the threads within a warp, then the 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 timestepped 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 [90]. With a sufficiently large timestep 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 [143]. They study diffusion simulations running in a timestepped, discrete-event, and hybrid fashion. The GPU variant is implemented in the GPU programming language Brook [26]. While the GPU outperforms the CPU in the timestepped 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 timestep. The simulation time then advances according to this timestep.

Park and Fishwick [139, 141] present a method for queuing network simulation that executes a DES model in a timestepped fashion. The simulation time advances according to a fixed timestep size, but skips periods where no events occur. All events within the current timestep 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 [7, 157, 196]. If lookahead is available, then 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 [169] 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.

4.4.3 Speculative Execution. To maintain the correctness of the simulation results when executing in parallel on an accelerator, the simulator must consider the 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, then 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 [121]. 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. [106] 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, then 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 [109]. An optimistic execution scheme based on the Time Warp algorithm [90] 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.

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 [169] 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. [100] (cf. Section 4.4.1). If the simulation instances do not diverge too strongly, then 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 [95]. 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. [32] 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.

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.

4.5.1 Frameworks to Support Simulation Development. In the Flexible Large Scale Agent Modelling Environment (FLAME GPU) [151, 153], agent states are specified using the state machine model X-Machine [48, 82]. Modellers 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 [79].

The domain-specific language OpenABL [39] enables the specification of ABS models in a compact and platform-independent fashion. The OpenABL code is translated to an intermediate representation, from which code targeting different backends for sequential, parallelised, as well as GPU-based execution can be generated.

Another framework for GPUs and other many-core architectures is called Many-Core Multi-Agent System (MCMAS) [103]. 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.

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 [133], or to annotate the program to manage data transfers [105, 190]. 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 [88].

Jablin et al. [87, 88] presented 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 reside 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 they are 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 5, we sketch research directions towards automation in porting ABS to accelerators.

5 TOWARDS AN AUTOMATED OFFLOADING PROCEDURE

From the observations in the previous section, we can state that there is a vast range of techniques covering the main challenges of high-performance ABS on hardware accelerators. However, there exist only 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 [187]. 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 [76, 118], there is a lack of automated tools to support this process.

The problem of automatic parallelisation of general programs is a broad and active field of research [54]. Substantial successes have been achieved with respect to parallelisation of computationally intensive loops with predictable and mostly static control flow [66], whereas the extraction of parallelism across complex and irregular programs is still a largely manual process. Common approaches include specifying software systems using formalisms that express parallelism explicitly [81, 102, 116] or annotating programs with parallelisation hints [42]. In essence, these approaches provide the compiler or parallelisation middleware with a dependency graph of the statements or code blocks within the original program.

Fortunately, many agent-based simulators and models roughly follow a common set of properties that simplify the extraction of parallelism. We identify the following constraints that can be leveraged to support the parallelisation process:

(1) *Time-stepped execution*: Usually, the model time is advanced in fixed increments. At each timestep, 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

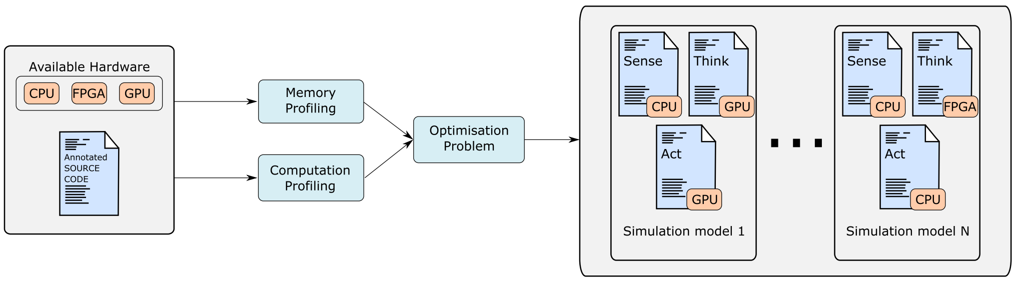


Fig. 3. Workflow of the envisioned automated offloading procedure.

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) *Sense-Think-Act cycle*: We assume that agent updates follow the well-known Sense-Think-Act cycle (cf. Section 3.1), with one such cycle per model.

With these constraints, a natural approach to parallelisation is to offload individual stages of a model's Sense-Think-Act cycle to an accelerator. For instance, in crowd simulations using the social force model, the Think stage comprised of the computation of the force affecting an agent may be performed by one thread of a GPU per agent.

In the following, we sketch an envisioned workflow and the required tools to support users in porting an existing CPU-based ABS to a system equipped with hardware accelerators. For the targeted simulator architecture, we assume a traditional *master-worker* scheme, with the host CPU acting as the master and assigning work to the available accelerators at each timestep.

5.1 Proposed Work Flow

The proposed semi-automated process is visualised in Figure 3. To facilitate the automatic partitioning of the simulation source code into segments that can be outsourced to various types of hardware, we suggest manually annotating the source code according to the Sense-Think-Act paradigm. From that it follows that the smallest unit that can be offloaded to a hardware accelerator in our proposed framework is one of these three stages. Each of the stages is profiled in terms of memory and computational requirements. According to the gathered requirements, an optimisation problem is solved to generate a hardware assignment (rightmost part of Figure 3).

For simplicity, we assume that all data required by the stage fits into one of the accelerator's memory entirely. Otherwise, agents could be distributed across multiple accelerators or processed in batches, both implying additional communication costs.

5.1.1 Input. The source code is annotated manually to signify the stages of the Sense-Think-Act cycle, e.g., in the form `#pragma sense_begin`, `#pragma sense_end`, and so forth. A simple example for a crowd simulation is given in Algorithm 1. In addition to the manual annotations, this clear separation may require refactoring of the simulation code. By parsing the annotated source code, the framework obtains a mapping between code and stages that will later be enriched with data from measurements. The second input is a specification of the available hardware. Each hardware device is characterised by its available memory, computational performance, and host-device data transfer overhead. The computational performance can be stated in terms of single-threaded performance on CPUs, many-core CPUs, GPUs, and APUs. We assume that for an FPGA,

ALGORITHM 1: Example for model code annotated with the stages of an agent update.

```
#pragma agent_begin
class Agent:
    Coord position;
    void executeOnTimeStep():
        #pragma sense_begin
        List agents = getNeighbouringAgents(position);
        #pragma sense_end
        #pragma think_begin
        Coord velocity = computeVelocity(agents);
        #pragma think_end
        #pragma act_begin
        position = position + velocity;
        #pragma act_end
#pragma agent_end
```

only model stages for which implementations already exist are eligible for offloading. Thus, the computational performance of an FPGA is given with respect to specific model stages.

5.1.2 *Memory Access Profiling.* Now that the source code is partitioned into offloadable stages and the capabilities of all the hardware components are known, the data dependencies of each stage are determined. Assuming a node in a graph represents one stage, then an edge in this graph represents a data dependency between these stages. The dependency can refer to either agent or environment data. The weight of the edge is the volume of the data that are accessed in the CPU-based simulator, i.e., that has to be transferred during offloading. Usually, the Think stage only has a dependency on the Sense stage within the same model and agent (intra-agent dependency), whereas the Sense stage might depend on the environment and on other agents' states (inter-agent dependency). Although we assume that an individual stage is not partitioned across multiple hardware devices, the amount of data gathered during the sense stage may vary over the course of the simulation. For instance, if agents form clusters in the simulation space, the number of neighbours per agent may increase over time. Thus, the data dependencies should be measured with respect to typical scenario conditions. To avoid exceeding the memory capacity of one of the considered hardware devices, the profiling can be repeated for a worst-case scenario.

Tools exist that are able to ascribe memory accesses performed during a program run to the source functions, data structures or threads [10]. For instance, the tool PinComm constructs a dynamic dataflow graph from instrumented program executions [73]. The annotations shown in Algorithm 1 allow us to map function names to the separate agent update stages. Thus, it is possible to obtain the amount of memory accessed within each stage. Once the graph describing the amount of memory accesses across stages is created, the implications in terms of memory copying of moving a certain stage to a hardware device can directly be evaluated. For example, if the Think stage is moved to the GPU and the Sense and Act stage remains on the host CPU, then the edges entering and leaving the Think node determine the data transfer overhead. The actual cost of this copy procedure can be obtained from the device specification or through measurements.

5.1.3 *Computational Profiling.* In addition to the memory requirements of each stage, information about the computational characteristics of each stage is required. The estimated runtime could be inferred from hardware performance models [11, 31, 161, 195]. Approaches as those described in Section 4.1.2 can be applied to estimate the suitability of different agent update stages for execution on a certain accelerator. By characterising the workload incurred by each stage in terms of instruction mix and memory accesses as well as the number of agents, the performance of executing the full-scale simulation can be estimated [63, 96, 182, 193]. Alternatively, if the runtime of a stage is dominated by a sub-task that can easily be ported to an accelerator, measurements with respect to this task can be performed directly on the accelerator [19].

5.1.4 Optimisation Problem. Building on the graph that represents data dependencies, an optimisation problem of assigning stages to hardware types can be formulated, similar to the approach targeting embedded systems by Zhang et al. [194]. In essence, constraints are formulated so that each stage is assigned to the host or a device, resulting in an overall simulation schedule. Importantly, the optimisation problem must reflect the data location after each stage or timestep (e.g., Reference [116]). For instance, to avoid data transfers, it may be more efficient to execute two subsequent stages on the same accelerator. The objective function of the optimisation problem is the overall runtime, i.e., the sum of all estimated execution times on the respective device and the incurred communication costs by distributing nodes of the dependency graph that are connected by an edge.

5.1.5 Output. The output of the optimisation steps is a recommendation of which stages should be executed on which hardware device. It is then the task of the user to port the code of each stage so it can be executed on the assigned device. This might require specific knowledge, e.g., programming in VHDL or OpenCL and can therefore be an obstacle to some researchers. Given that some established simulation models are used by many researchers (e.g., a CSMA/CA model in network simulation or different car-following models in traffic simulation), a public repository of common simulation models could be created, similarly to the plugin approach used in MCMAS [103]. Researchers could download these crowd-sourced simulation models to enable parts of their own simulations to be run on heterogeneous hardware environments, and contribute their own model implementations. Such a repository would also reduce the need to estimate execution times and improve the optimisation results by allowing direct measurements on the potential target devices. Similarly, after porting a specific model stage, new measurements may be performed to provide the optimisation process with more accurate performance data.

5.1.6 Discussion. In our approach, we take a pragmatic perspective: While the envisioned workflow is achievable based on existing building blocks, our assumptions may leave substantial performance potentials unexplored. In particular, by assuming that models and their stages are both executed as a series of dependent steps, we only exploit the inter-agent parallelism within each stage, while any parallelism across stages is not considered. In the following, we revisit the key challenges of ABS using hardware accelerators and sketch techniques from the literature that could be applied to maximise the performance benefits given our assumptions.

The hardware assignment (cf. Section 4.1) is the main focus of the proposed work flow. Above, we describe a static assignment using a functional decomposition. Still, the optimisation problem that determines the hardware mapping could be updated according to runtime measurements.

To minimise data transfer overheads that cannot be avoided (cf. Section 4.2), a bulk execution of multiple simulation runs would be feasible. The optimisation problem could be adapted so that the computational and memory requirements reflect those of each stage executed within multiple simulations runs at the same time. The output of the optimisation process would then be a schedule for an execution in a multiple replications in parallel (MRIP) fashion [100, 142] .

The technique of overlapping computations with data transfers seems challenging in our approach, since we assume a serialisation of the agent update stages. However, pre-fetching across stages may be performed by commencing data transfers once some agents have finished a stage.

Scattered memory accesses and the maximisation of parallelism (cf. Sections 4.3 and 4.4) could be addressed by providing a library of optimised functions and data structures for operations such as inter-agent communication or neighbour search (e.g., References [35, 103]).

A certain degree of abstraction from hardware specifics (cf. Section 4.5) is achieved by the automated profiling and hardware mapping of our proposed workflow. Since each stage is executed on a single accelerator, facilities for unified memory access across all devices are not required. Instead, all agent data are updated locally on the accelerator and transferred automatically according to the schedule determined in the optimisation process.

Overall, the envisioned workflow is intended to rely on existing tools and techniques to allow researchers to exploit the hardware at their disposal with reasonable performance gains, while avoiding the need for costly and time-consuming manual optimisation steps as much as possible.

6 CONCLUSIONS

We presented a survey of the literature on agent-based simulation using hardware accelerators. We categorised existing approaches according to the key challenges of hardware assignment, minimisation of data transfer overheads, scattered memory accesses, maximisation of parallelism, and the abstraction from hardware specifics. Our survey provides modellers with an overview of techniques to execute a certain class of models on the available hardware. Methodology researchers are given a summary of the existing work, pointing out research gaps where further exploration is required. Our main observations are twofold: First, most of the literature in the past years has focused on GPUs. We expect a significant amount of work exploring agent-based simulations on FPGAs to appear in the near future. Second, while a vast amount of work has proposed techniques that allow for efficient execution of agent-based simulations, only few techniques have found their way into a unified framework. Thus, the burden of developing a simulation executable in a heterogeneous environment is carried by the modeller. Aiming to reduce the need for expertise in the programming for accelerators, we sketched our vision of a framework for automated hardware mapping and performance optimisation based on building blocks from the literature.

REFERENCES

[1] Brandon G. Aaby, Kalyan S. Perumalla, and Sudip K. Seal. 2010. 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)*. ICST, Torremolinos, Spain, 29:1–29:10.

[2] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory MP O'Hare. 2017. Agent based modelling and simulation tools: A review of the state-of-art software. *Comput. Sci. Rev.* 24 (May 2017), 13–33.

[3] Advanced Micro Devices, Inc. 2017. *Radeon's Next-Generation Vega Architecture*. Technical Report 061317_FINAL_V2. Radeon Technologies Group.

[4] Spiros N. Agathos, Alexandros Papadogiannakis, and Vassilios V. Dimakopoulos. 2015. Targeting the parallella. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'15)*. Springer, 662–674.

[5] Rob Allan. 2010. *Survey of Agent Based Modelling and Simulation Tools*. Technical Report DL-TR-2010-007. Science and Technology Facilities Council.

[6] Gary An, Qi Mi, Joyeeta Dutta-Moscato, and Yoram Vodovotz. 2009. Agent-based models in translational systems biology. *Syst. Biol. Med.* 1, 2 (Sep. 2009), 159–171.

[7] Philipp Andelfinger and Hannes Hartenstein. 2014. Exploiting the parallelism of large-scale application-layer networks by adaptive GPU-based simulation. In *Proceedings of the Winter Simulation Conference (WSC'14)*. IEEE, 3471–3482.

[8] Philipp Andelfinger, Jens Mittag, and Hannes Hartenstein. 2011. 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)*. IEEE, 421–424.

[9] Philipp Andelfinger, Yadong Xu, David Eckhoff, Wentong Cai, and Alois Knoll. 2018. Fast-forwarding agent states to accelerate microscopic traffic simulations. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS'18)*. ACM, 113–124.

[10] Imran Ashraf, Mottaqiallah Taouil, and Koen Bertels. 2015. Memory profiling for intra-application data-communication quantification: A survey. In *Proceedings of the International Design & Test Symposium (IDT'15).* IEEE, 32–37.

[11] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10).* ACM, 105–114.

[12] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. 2009. Agent based modeling and simulation: An informatics perspective. *J. Artif. Soc. Soc. Simul.* 12, 4 (Oct. 2009), 4.

[13] Taylor Barnes, Brandon Cook, Jack Deslippe, Douglas Doerfler, Brian Friesen, Yun (Helen) He, Thorsten Kurth, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Abhinav Sarje, Jean-Luc Vay, Henri Vincenti, Samuel Williams, Pierre Carrier, Nathan Wichmann, Marcus Wagner, Paul Kent, Christopher Kerr, and John Dennis. 2016. 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).* IEEE, 43–53.

[14] Nikolai Baudis, Florian Jacob, and Philipp Andelfinger. 2017. 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).* IEEE, Banff, Canada, 1–11.

[15] David W. Bauer, Matthew McMahon, and Ernest H. Page. 2008. An approach for the effective utilization of GP-GPUs in parallel combined simulation. In *Proceedings of the Conference on Winter Simulation (WSC'08).* IEEE, 695–702.

[16] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11).* ACM, 12:1–12:11.

[17] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. 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).* ACM, 119–130.

[18] Nathan Bell and Jared Hoberock. 2011. Thrust: A productivity-oriented library for CUDA. In *GPU Computing.* Elsevier, Boston, MA, 359–371.

[19] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. 2013. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Arch. Code Optimiz.* 9, 4 (Jan. 2013), 57:1–57:20.

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

[21] Ranjita Bhagwan and Bill Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings of the IEEE INFOCOM Conference on Computer Communications (INFOCOM'00).* IEEE, 538–547.

[22] Ben Romdhanne Bilel, Nikaein Navid, and Mohamed Said Mosli Bouksiaa. 2012. 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).* IEEE, 44–53.

[23] Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni Seghrouchni, Jorge J. Gomez-Sanz, João Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. 2006. A survey of programming languages and platforms for multi-agent systems. *Drustvo Inf. Inf.* 30, 1 (Jan. 2006), 33–44.

[24] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. 2013. Graphics Processing Unit (GPU) programming strategies and trends in GPU computing. *J. Parallel Distrib. Comput.* 73, 1 (Jan. 2013), 4–13.

[25] Ulrich Brüning, Wolfgang K. Giloi, and Wolfgang Schroeder-Preikschat. 1994. Latency hiding in message-passing architectures. In *Proceedings of the International Parallel Processing Symposium (IPPS'94).* IEEE, 704–709.

[26] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of the International Conference on Computer Graphics and Interactive Techiques (SIGGRAPH'04).* ACM, 777–786.

[27] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'12).* IEEE, 141–151.

[28] Eduardo Cabrera, Manel Taboada, Ma Luisa Iglesias, Francisco Epelde, and Emilio Luque. 2011. Optimization of healthcare emergency departments by agent-based simulation. In *Proceedings of the International Conference on Computational Science (ICCS'11).* Elsevier, 1880–1889.

[29] Márcio Castro, Emilio Francesquini, Fabrice Dupros, Hideo Aochi, Philippe Navaux, and Jean-François Mehaut. 2016. Seismic wave propagation simulations on low-power and performance-centric manycores. *J. Parallel Comput.* 54, C (May 2016), 108–120.

[30] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the Symposium on Application Specific Processors (SASP'08).* IEEE, 101–107.

[31] Xi E. Chen and Tor M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 329–340.

[32] Mozhgan K. Chimeh and Paul Richmond. 2018. Simulating heterogeneous behaviours in complex systems on GPUs. *Simul. Model. Pract. Theory* 83 (April 2018), 3–17.

[33] Thomas M. Cioppa, Thomas W. Lucas, and Susan M. Sanchez. 2004. Military applications of agent-based simulations. In *Proceedings of the Winter Simulation Conference (WSC'04)*. IEEE, 171–180.

[34] John G. Cleary, Murray Pearson, and Husam Kinawi. 1995. The architecture of an optimistic CPU: The warpengine. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS'95)*. IEEE, 163–172.

[35] Simon Coakley, Paul Richmond, Marian Gheorghe, Shawn Chin, David Worth, Mike Holcombe, and Chris Greenough. 2016. Large-scale simulations with FLAME. In *Intelligent Agents in Data-intensive Computing*, Joanna Kołodziej, Luís Correia, and José Manuel Molina (Eds.). Springer International Publishing, New York, NY, 123–142.

[36] N. T. Collier and M. J. North. 2011. Repast SC++: A platform for large-scale agent-based modeling. In *Large-Scale Computing Techniques for Complex System Simulations*, W. Dubitzky, K. Kurowski, and B. Schott (Eds.). Wiley, Hoboken, NJ.

[37] John Conway. 1970. The game of life. *Sci. Am.* 223, 4 (1970), 4.

[38] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. 2011. Distributed load balancing for parallel agent-based simulations. In *Proceedings of the International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'11)*. IEEE, 62–69.

[39] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cordasco, and Vittorio Scarano. 2018. OpenABL: A domain-specific language for parallel and distributed agent-based simulations. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'18)*. Springer, Berlin, 505–518.

[40] Andrew T. Crooks and Alison J. Heppenstall. 2012. An introduction to agent-based modeling. In *Agent-Based Models of Geographical Systems*. Springer, New York, NY, 85–105.

[41] Lintao Cui, Jing Chen, Yu Hu, Jinjun Xiong, Zhe Feng, and Lei He. 2011. Acceleration of multi-agent simulation on FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'11)*. IEEE, 470–473.

[42] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE J. Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55.

[43] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'13)*. IEEE, 1–6.

[44] Narsingh Deo and Sushil Prasad. 1992. Parallel heap: An optimal parallel priority queue. *J. Supercomput.* 6, 1 (Mar. 1992), 87–98.

[45] Chris Ding and Yun He. 2001. A ghost cell expansion method for reducing communications in solving PDE problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'01)*. IEEE, 55–55.

[46] Arnaud Doniec, René Mandiau, Sylvain Piechowiak, and Stéphane Espié. 2008. A behavioral multi-agent model for road traffic simulation. *J. Eng. Appl. Artif. Intell.* 21, 8 (Dec. 2008), 1443–1454.

[47] Roshan M. D'Souza, Mikola Lysenko, Simeone Marino, and Denise Kirschner. 2009. Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the Spring Simulation Multiconference (SpringSim'09)*. SCSI, 21:1–21:12.

[48] Samuel Eilenberg. 1974. *Automata, Languages, and Machines*. Academic Press, Cambridge, MA.

[49] Joshua M. Epstein. 1999. Agent-based computational models and generative social science. *Complex.* 4, 5 (May 1999), 41–60.

[50] Fernando A. Escobar, Xin Chang, and Carlos Valderrama. 2016. Suitability analysis of FPGAs for heterogeneous platforms in HPC. *IEEE Trans. Parallel Distrib. Syst.* 27, 2 (Feb. 2016), 600–612.

[51] Babak Falsafi, Bill Dally, Desh Singh, Derek Chiou, J. Yi Joshua, and Resit Sendag. 2017. FPGAs versus GPUs in data centers. *IEEE Micro* 37, 1 (Jan. 2017), 60–72.

[52] Naznin Fauzia, Louis-Noël Pouchet, and P. Sadayappan. 2015. 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)*. IEEE, 12–22.

[53] Richard M. Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. Wiley, New York, NY.

[54] Richard M. Fujimoto. 2016. Research challenges in parallel and distributed simulation. *ACM Trans. Model. Comput. Simul.* 26, 4 (May 2016), 22:1–22:29.

[55] Richard M. Fujimoto, Conrad Bock, Wei Chen, Ernest Page, and Jitesh H. Panchal. 2017. *Research Challenges in Modeling and Simulation for Engineering Complex Systems*. Springer, New York, NY.

[56] Richard M. Fujimoto, Christopher Carothers, Alois Ferscha, David Jefferson, Margaret Loper, Madhav Marathe, and Simon J.E. Taylor. 2017. Computational challenges in modeling & simulation of complex systems. In *Proceedings of the Winter Simulation Conference (WSC'17)*. IEEE, 431–445.

[57] Richard M. Fujimoto, Jya-Jang. Tsai, and Ganesh Gopalakrishnan. 1988. Design and performance of special purpose hardware for time warp. In *Proceedings of the Annual International Symposium on Computer Architecture (SCA'88)*. IEEE, 401–409.

[58] Martin Gardner. 1970. Mathematical games: The fantastic combinations of john conway's new solitaire game "life."*Sci. Am.* 223, 4 (Oct. 1970), 120–123.

[59] Ioakeim G. Georgoudas, Panagiotis Kyriakos, G. Ch. Sirakoulis, and I. Th. Andreadis. 2010. An FPGA implemented cellular automaton crowd evacuation model inspired by the electrostatic-induced potential fields. *J. Microprocess. Microsyst.* 34, 7 (Nov. 2010), 285–300.

[60] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Market. Lett.* 12, 3 (Aug. 2001), 211–223.

[61] Nicholas Mark Gotts, J Gareth Polhill, and Alistair N. R. Law. 2003. Agent-based simulation in the study of social dilemmas. *Artif. Intell. Rev.* 19, 1 (Mar. 2003), 3–92.

[62] Mark Granovetter. 1978. Threshold models of collective behavior. *Am. J. Sociol.* 83, 6 (1978), 1420–1443.

[63] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. 2013. 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)*. ACM, 281–282.

[64] Simon Green. 2010. Particle simulation using cuda. *NVIDIA Whitepaper* 6 (2010), 121–128.

[65] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly–performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 04 (Dec. 2012), 1250010.

[66] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the International Conference on Supercomputing (ICS'16)*. ACM, 1:1–1:13.

[67] Takahiro Harada. 2007. Real-time rigid body simulation on GPUs. *NVIDIA GPU Gems* 3 (Dec. 2007), 123–148.

[68] Jack Harris and Matthias Scheutz. 2012. New advances in asynchronous agent-based scheduling. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*. WorldComp, 1–7.

[69] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. 2002. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'02)*. ACM, 109–118.

[70] Scott Hauck and Andre DeHon. 2010. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation.* Morgan Kaufmann, Burlington, MA.

[71] Xi He, Deborah Agarwal, and Sushil K. Prasad. 2012. Design and implementation of a parallel priority queue on many-core architectures. In *Proceedings of the International Conference on High Performance Computing (HiPC'12)*. IEEE, 1–10.

[72] Ulrich Heinkel, Martin Padeffke, Werner Haas, Thomas Buerner, Herbert Braisz, Thomas Gentner, and Alexander Grassmann. 2000. *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (Including VHDL-AMS).* Wiley, New York, NY.

[73] Wim Heirman, Dirk Stroobandt, Narasinga Rao Miniskar, Roel Wuyts, and Francky Catthoor. 2010. PinComm: Characterizing intra-application communication for the many-core era. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS'10)*. IEEE, 500–507.

[74] Emmanuel Hermellin and Fabien Michel. 2015. 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)*. Springer, 71–86.

[75] Emmanuel Hermellin and Fabien Michel. 2016. 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)*. Springer, 24–41.

[76] Emmanuel Hermellin and Fabien Michel. 2016. 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)*. IFAAMAS, 1249–1258.

[77] James D. Hess, Jacqueline J. Kacen, and Junyong Kim. 2006. Mood-management dynamics: The interrelationship between moods and behaviours. *Br. J. Math. Stat. Psychol.* 59, 2 (Nov. 2006), 347–378.

[78] Peter Heywood, Steve Maddock, Jordi Casas, David Garcia, Mark Brackstone, and Paul Richmond. 2018. Data-parallel agent-based microscopic road network simulation using graphics processing units. *Simul. Model. Pract. Theory* 83 (Apr. 2018), 188–200.

[79] Peter Heywood, Paul Richmond, and Steve Maddock. 2015. Road network simulation using FLAME GPU. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'15)*. Springer, 430–441.

[80] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, and Yuki Sugiyama. 2012. 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)*. 45–50.

[81] Charles Antony Richard Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677.

[82] Mike Holcombe. 1988. X-Machines as a basis for dynamic system specification. *IET Softw. Eng. J.* 3, 2 (Mar. 1988), 69–76.

[83] Andreas Horni, Kai Nagel, and Kay W. Axhausen. 2016. *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, London, UK.

[84] Maria Hybinette, Eileen Kraemer, Yin Xiong, Glenn Matthews, and Jaim Ahmed. 2006. SASSY: A design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In *Proceedings of the Winter Simulation Conference (WSC'06)*. IEEE, 926–933.

[85] Intel Corporation. 2017. *Intel FPGA SDK for OpenCL—Programming Guide*. UG-OCL002.

[86] Intel Corporation. 2018. Intel Architecture Instruction Set Extensions and Future Features—Programming Reference. 319433-032.

[87] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. 2012. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'12)*. ACM, 165–174.

[88] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU communication management and optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 142–151.

[89] Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2012. 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)*. IEEE, 53–62.

[90] David R. Jefferson. 1985. Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 3 (Jul. 1985), 404–425.

[91] Y. Jiao, H. Lin, P. Balaji, and W. Feng. 2010. 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)*. IEEE, 221–228.

[92] Jiangming Jin, Stephen John Turner, Bu-Sung Lee, Jianlong Zhong, and Bingsheng He. 2012. HPC simulations of information propagation over social networks. In *Proceedings of the International Conference on Computational Science (ICCS'12)*. Elsevier, 292–301.

[93] Jiangming Jin, Stephen John Turner, Bu-Sung Lee, Jianlong Zhong, and Bingsheng He. 2013. 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)*. IEEE, 179–188.

[94] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'17)*. IEEE, 1–12.

[95] Klaus Kofler, Gregory Davis, and Sandra Gesing. 2014. Sampo: An agent-based mosquito point model in OpenCL. In *Proceedings of the Symposium on Agent Directed Simulation (ADS'14)*. SCSI, 5:1–5:10.

[96] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An automatic input-sensitive approach for heterogeneous task partitioning. In *Proceedings of the International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, 149–160.

[97] Andreas Kolb and Lars John. 2001. Volumetric model repair for virtual reality applications. In *EUROGRAPHICS Short Presentation (2001)*. The Eurographics Association, Manchester, England, 249–256.

[98] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. 2004. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*. ACM, 123–131.

[99] Kalliopi Kravari and Nick Bassiliades. 2015. A survey of agent platforms. *J. Artif. Soc. Soc. Simul.* 18, 1 (Jan. 2015), 11.

[100] Georg Kunz, Daniel Schemmel, James Gross, and Klaus Wehrle. 2012. 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)*. IEEE, 23–32.

[101] Chenggang Lai, Miaoqing Huang, Xuan Shi, and Haihang You. 2013. 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 and EUC'13)*. IEEE, 1545–1552.

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

[103] Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, Bénédicte Herrmann, and Laurent Philippe. 2013. 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).* Springer, 544–554.

[104] Guillaume Laville, Kamel Mazouzi, Christophe Lang, Laurent Philipppe, and Nicolas Marilleau. 2013. Using GPU for multi-agent soil simulation. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'13).* IEEE, 392–399.

[105] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. 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).* ACM, 101–110.

[106] Xiaosong Li, Wentong Cai, and Stephen John Turner. 2013. 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).* ACM, 57–66.

[107] Xiaosong Li, Wentong Cai, and Stephen John Turner. 2014. 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).* IEEE, 87–96.

[108] Xiaosong Li, Wentong Cai, and Stephen John Turner. 2015. Cloning agent-based simulation on GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS'15).* ACM, 173–182.

[109] Xinhu Liu and Philipp Andelfinger. 2017. Time warp on the GPU: Design and assessment. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS'17).* ACM, 109–120.

[110] Xu Liu, Langshi Chen, Jesun S. Firoz, Judy Qiu, and Lei Jiang. 2018. Performance characterization of multi-threaded graph processing applications on intel many-integrated-core architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'18).* IEEE, 199–208.

[111] Qingqi Long, Jie Lin, and Zhixun Sun. 2011. Agent scheduling model for adaptive dynamic load balancing in agent-based distributed simulations. *J. Simul. Model. Pract. Theory* 19, 4 (Apr. 2011), 1021–1034.

[112] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. 2005. MASON: A multiagent simulation environment. *J. Simul.* 81, 7 (Jul. 2005), 517–527.

[113] Elizabeth Whitaker Lynch. 2011. *Hardware Acceleration for Conservative Parallel Discrete Event Simulation on Multi-Core Systems.* Ph.D. Dissertation. School of Electrical and Computer Engineering, Georgia Institute of Technology.

[114] Mikola Lysenko and Roshan M. D'Souza. 2008. A framework for megascale agent based model simulations on graphics processing units. *J. Artif. Soc. Soc. Simul.* 11, 4 (Oct. 2008), 10.

[115] Charles M. Macal and Michael J. North. 2010. Tutorial on agent-based modeling and simulation. *J. Simul.* 4, 3 (Sep. 2010), 151–162.

[116] Deepak Majeti, Kuldeep S. Meel, Rajkishore Barik, and Vivek Sarkar. 2016. Automatic data layout generation and kernel mapping for CPU+GPU architectures. In *Proceedings of the International Conference on Compiler Construction (CC'16).* ACM, 240–250.

[117] Robin B. Matthews, Nigel G. Gilbert, Alan Roach, J. Gary Polhill, and Nick M. Gotts. 2007. Agent-based land-use models: A review of applications. *Landsc. Ecol.* 22, 10 (Dec. 2007), 1447–1459.

[118] Fabien Michel. 2013. 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. *J. Syst. Res. Behav. Sci.* 30, 6 (Nov. 2013), 703–715.

[119] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. 1996. *The Swarm Simulation System: A Toolkit for Building Multi-agent Simulations.* Technical Report 1996-06-042. Swarm Development Group.

[120] Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.* 47, 4 (Jul. 2015), 69:1–69:35.

[121] Josh Model and Martin C. Herbordt. 2007. Discrete event simulation of molecular dynamics with configurable logic. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07).* IEEE, 151–158.

[122] Kai Nagel and Marcus Rickert. 2001. Parallel implementation of the TRANSIMS micro-simulation. *J. Parallel Comput.* 27, 12 (Nov. 2001), 1611–1639.

[123] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 35, 10 (Oct. 2016), 1591–1604.

[124] Ashkan Negahban and Levent Yilmaz. 2014. Agent-based simulation applications in marketing research: An integrated review. *J. Simul.* 8, 2 (May 2014), 129–142.

[125] Roland Neumann and Fritz Strack. 2000. "Mood Contagion": The automatic transfer of mood between persons. *J. Pers. Soc. Psychol.* 79, 2 (Aug. 2000), 211.

[126] G. Nguyen, T. Tung Dang, L. Hluchy, Z. Balogh, M. Laclavik, and I. Budinska. 2002. *Agent Platform Evaluation and Comparison.* Technical Report IST-2001-34519. Institute of Informatics, Bratislava, Slovakia.

[127] Cynthia Nikolai and Gregory Madey. 2009. Tools of the trade: A survey of various agent based modeling platforms. *J. Artif. Soc. Soc. Simul.* 12, 2 (Mar. 2009), 2.

[128] Michael J. North and Charles M. Macal. 2007. *Managing Business Complexity: Discovering Strategic Solutions with Agent-Based Modeling and Simulation*. Oxford University Press, Oxford, UK.

[129] NVIDIA Corporation. 2012. CUDA Toolkit 4.2—CUBLAS Library. PG-05326-041_v01.

[130] NVIDIA Corporation. 2016. Whitepaper. NVIDIA GeForce GTX 1080. Gaming Perfected.

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

[132] NVIDIA Corporation. 2018. Jetson Xavier FAQ.

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

[134] Christopher Oat, Joshua Barczak, and Jeremy Shopf. 2009. *Efficient Spatial Binning on the GPU*. Technical Report. Advanced Micro Devices, Inc.

[135] Andreas Olofsson. 2016. Epiphany-V: A 1024 processor 64-bit RISC system-on-chip. *CoRR* abs/1610.01832 (October 2016). arxiv:1610.01832

[136] OpenACC Working Group. 2015. The OpenACC Application Programming Interface. Version 2.5.

[137] Samir Palnitkar. 2003. *Verilog®Hdl: A Guide to Digital Design and Synthesis*, 2nd ed. Prentice Hall Press, Upper Saddle River, NJ.

[138] Sakda Panwai and Hussein Dia. 2005. A reactive agent-based neural network car following model. In *Proceedings of the International IEEE Conference on Intelligent Transportation Systems (ITSC'05)*. IEEE, 375–380.

[139] Hyungwook Park and Paul A. Fishwick. 2008. A fast hybrid time-synchronous/event approach to parallel discrete event simulation of queuing networks. In *Proceedings of the Winter Simulation Conference (WSC'08)*. IEEE, 795–803.

[140] Hyungwook Park and Paul A. Fishwick. 2010. A GPU-based application framework supporting fast discrete-event simulation. *J. Simul.* 86, 10 (Oct. 2010), 613–628.

[141] Hyungwook Park and Paul A. Fishwick. 2011. An analysis of queuing network simulation using GPU-based hardware acceleration. *ACM Trans. Model. Comput. Simul.* 21, 3 (Mar. 2011), 18:1–18:22.

[142] Roman Pavlov and Jörg P. Müller. 2013. 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)*. Springer, 115–122.

[143] Kalyan S. Perumalla. 2006. 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)*. IEEE, 74–81.

[144] Kalyan S. Perumalla. 2008. Efficient execution on GPUs of field-based vehicular mobility models. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS'08)*. IEEE, 154–154.

[145] Kalyan S. Perumalla and Brandon G. Aaby. 2008. Data parallel execution challenges and runtime performance of agent simulations on GPUs. In *Proceedings of the Spring Simulation Multiconference (SpringSim'08)*. SCSI, 116–123.

[146] Kalyan S. Perumalla, Brandon G. Aaby, Srikanth B. Yoginath, and Sudip K. Seal. 2009. 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)*. IEEE, 95–103.

[147] Louis-Noël Pouchet. 2012. Polybench: the Polyhedral Benchmark suite. Retrieved May 11, 2018 from http://web.cs.ucla.edu/ pouchet/software/polybench/.

[148] Sebastian Raase and Tomas Nordstrm. 2015. On the use of a many-core processor for computational fluid dynamics simulations. In *Proceedings of the International Conference On Computational Science (ICCS'15)*. Elsevier, 1403–1412.

[149] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2017. 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)*. ACM, 133–144.

[150] Paul F. Reynolds, Carmen M. Pancerella, and Sudhir Srinivasan. 1993. Design and performance analysis of hardware support for parallel simulations. *J. Parallel Distrib. Comput.* 18, 4 (Aug. 1993), 435–453.

[151] Paul Richmond, Simon Coakley, and Daniela Romano. 2009. Cellular level agent based modelling on the graphics processing unit. In *Proceedings of the International Workshop on High Performance Computational Systems Biology (HIBI'09)*. IEEE, 43–50.

[152] Paul Richmond and Daniela Romano. 2008. 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)*. ACM.

[153] Paul Richmond and Daniela Romano. 2011. Template-driven agent-based modeling and simulation with CUDA. In *GPU Computing Gems Emerald Edition, Applications of GPU Computing Series*. Elsevier, Amsterdam, 313–324.

[154] Patrick F. Riley and George F. Riley. 2003. 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)*. IEEE, 817–825.

[155] Robert Rönngren and Rassul Ayani. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul.* 7, 2 (Apr. 1997), 157–209.

[156] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. 2014. A survey on parallel and distributed multi-agent systems. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'14).* Springer, 371–382.

[157] Janche Sang, Che-Rung Lee, Vernon Rego, and Chung-Ta King. 2013. 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).* WorldComp, 501.

[158] Thomas C. Schelling. 2006. *Micromotives and Macrobehavior.* WW Norton & Company, New York City, NY.

[159] Moon Gi Seok and Tag Gon Kim. 2012. Parallel discrete event simulation for DEVS cellular models using a GPU. In *Proceedings of the Symposium on High Performance Computing (HPC'12).* SCSI, 11:1–11:7.

[160] Zhen Shen, Kai Wang, and Fenghua Zhu. 2011. Agent-based traffic simulation and traffic signal timing optimization with GPU. In *Proceedings of the International IEEE Conference on Intelligent Transportation Systems (ITSC'11).* IEEE, 145–150.

[161] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12).* ACM, 11–22.

[162] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *J. ACM* 32, 3 (Jul. 1985), 652–686.

[163] Xiao Song, Ziping Xie, Yan Xu, Gary Tan, Wenjie Tang, Jing Bi, and Xiaosong Li. 2017. Supporting real-world network-oriented mesoscopic traffic simulation on GPU. *J. Simul. Model. Pract. Theory* 74 (May 2017), 46–63.

[164] Russell K. Standish and Richard Leow. 2004. EcoLab: Agent based modeling for C++ programmers. *CoRR* cs.MA/0401026 (January 2004).

[165] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Comput. Sci. Eng.* 12, 3 (May 2010), 66–73.

[166] David Strippgen and Kai Nagel. 2009. Using common graphics hardware for multi-agent traffic simulation with CUDA. In *Proceedings of the International Conference on Simulation Tools and Techniques (Simutools'09).* ICST, 62:1–62:8.

[167] Herb Sutter. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's J.* 30, 3 (Mar. 2005), 202–210.

[168] Brian Paul Swenson. 2015. *Techniques to Improve the Performance of Large-Scale Discrete-Event Simulation.* Dissertation. Georgia Institute of Technology.

[169] Wenjie Tang and Yiping Yao. 2013. A GPU-based discrete event simulation kernel. *J. Simul.* 89, 11 (Oct. 2013), 1335–1354.

[170] Kardi Teknomo, Yasushi Takeyama, and Hajime Inamura. 2000. Review on microscopic pedestrian simulation model. In *Proceedings of the Japan Society of Civil Engineering Conference (JSCE).* 15–27.

[171] Leigh Tesfatsion. 2006. Agent-based computational economics: A constructive approach to economic theory. *Handb. Comput. Econ.* 2 (May 2006), 831–880.

[172] Georgios Theodoropoulos, Rob Minson, Roland Ewald, and Michael Lees. 2009. Simulation engines for multi-agent systems. In *Multi-Agent Systems.* CRC Press, Boca Raton, FL, 77–108.

[173] Seth Tisue and Uri Wilensky. 2004. Netlogo: A simple environment for modeling complexity. In *Proceedings of the International Conference on Complex Systems (ICCS'04)*, Vol. 21. NECSI, 16–21.

[174] Y. Torres, A. Gonzalez-Escribano, and D.R. Llanos. 2011. Understanding the impact of CUDA tuning techniques for fermi. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS'11).* IEEE, 631–639.

[175] Justin L. Tripp, Henning S. Mortveit, Anders A. Hansson, and Maya Gokhale. 2005. Metropolitan road traffic simulation on FPGAs. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05).* IEEE, 117–126.

[176] Mario Vestias and Horácio Neto. 2014. Trends of CPU, GPU and FPGA for high-performance computing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'14).* IEEE, 1–6.

[177] Guillermo Vigueras, Juan M. Orduña, Miguel Lozano, José M. Cecilia, and José M. García. 2014. Accelerating collision detection for large-scale crowd simulation on multi-core and many-core architectures. *Int. J. High Perf. Comput. Appl.* 28, 1 (Feb. 2014), 33–49.

[178] Ioannis Vourkas and Georgios Ch. Sirakoulis. 2012. FPGA based cellular automata for environmental modeling. In *Proceedings of the International Conference on Electronics, Circuits and Systems (ICECS'12).* IEEE, 93–96.

[179] Isabel Wagner and David Eckhoff. 2014. Privacy assessment in vehicular networks using simulation. In *Winter Simulation Conference (WSC'14).* IEEE, 3155–3166.

[180] Jin Wang, Norman Rubin, Haicheng Wu, and Sudhakar Yalamanchili. 2013. Accelerating simulation of agent-based models on heterogeneous architectures. In *Proceedings of the Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU'06).* ACM, 108–119.

[181] Kai Wang and Zhen Shen. 2012. 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).* IEEE, 160–165.

[182] Yuan Wen, Zheng Wang, and Michael F.P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *Proceedings of the International Conference on High Performance Computing (HiPC'14).* IEEE, 1–10.

[183] Tang Wenjie, Yao Yiping, and Zhu Feng. 2013. 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).* ACM, 367–372.

[184] Barry Williams, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Philip Wilsey. 2017. 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).* ACM, 121–132.

[185] Wąs, Jarosław and Mróz, Hubert and Topa, Paweł. 2016. GPGPU computing for microscopic simulations of crowd dynamics. *Slov. Acad. Sci. Comput. Inf.* 34, 6 (Feb. 2016), 1418–1434.

[186] Fulong Wu. 2002. Calibration of stochastic cellular automata: The application to rural-urban land conversions. *Int. J. Geogr. Inf. Sci.* 16, 8 (2002), 795–818.

[187] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2018. 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).* 243–252.

[188] Yadong Xu, Wentong Cai, David Eckhoff, Suraj Nair, and Alois Knoll. 2017. 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).* ACM, 209–219.

[189] Yan Xu, Gary Tan, Xiaosong Li, and Xiao Song. 2014. Mesoscopic traffic simulation on CPU/GPU. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS'14).* ACM, 39–50.

[190] Yonghong Yan, Max Grossman, and Vivek Sarkar. 2009. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'09).* Springer, 887–899.

[191] Mingyu Yang, Philipp Andelfinger, Wentong Cai, and Alois Knoll. 2018. 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).* ACM, 129–132.

[192] Srikanth B. Yoginath and Kalyan S. Perumalla. 2018. Scalable cloning on large-scale GPU platforms with application to time-stepped simulations on grids. *ACM Trans. Model. Comput. Simul.* 28, 1 (Jan. 2018), 5:11–5:26.

[193] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2017. Understanding co-running behaviors on integrated CPU/GPU architectures. *IEEE Trans. Parallel Distrib. Syst.* 28, 3 (Mar. 2017), 905–918.

[194] Tao Zhang, Xin Zhao, Xinqi An, Haojun Quan, and Zhichun Lei. 2017. Using blind optimization algorithm for hardware/software partitioning. *IEEE Access* 5 (Feb. 2017), 1353–1362.

[195] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 2011 IEEE International Symposium on High Performance Computer Architecture (HPCA'11).* IEEE, 382–393.

[196] Li Zhen, Qiuxiao Gang, Guo Gang, and Chen Bin. 2014. A GPU-based simulation kernel within heterogeneous collaborative computation on large-scale artificial society. *J. Model. Optimiz.* 4, 3 (Jun. 2014), 205–210.

[197] Zhi Zhou, Wai Kin Victor Chan, and Joe H. Chow. 2007. Agent-based simulation of electricity markets: A survey of tools. *Artif. Intell. Rev.* 28, 4 (Dec. 2007), 305–342.

[198] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. 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).* IEEE, 35:1–35:12.

[199] Peng Zou, Ya-shuai Lü, Li-li Chen, and Yi-ping Yao. 2013. Epidemic simulation of large-scale social contact network on GPU clusters. *J. Simul.* 89, 10 (2013), 1154–1172.

Received May 2018; revised August 2018; accepted October 2018