Transitioning Spiking Neural Network Simulators to Heterogeneous Hardware

QUANG ANH PHAM NGUYEN, PHILIPP ANDELFINGER, and WEN JUN TAN,
TUM Create Ltd. and Nanyang Technological University, Singapore
WENTONG CAI, Nanyang Technological University, Singapore
ALOIS KNOLL, Techn. Universität München, Germany and Nanyang Technological University, Singapore

Spiking neural networks (SNN) are among the most computationally intensive types of simulation models, with node counts on the order of up to 10^{11}. Currently, there is intensive research into hardware platforms suitable to support large-scale SNN simulations, whereas several of the most widely used simulators still rely purely on the execution on CPUs. Enabling the execution of these established simulators on heterogeneous hardware allows new studies to exploit the many-core hardware prevalent in modern supercomputing environments, while still being able to reproduce and compare with results from a vast body of existing literature. In this article, we propose a transition approach for CPU-based SNN simulators to enable the execution on heterogeneous hardware (e.g., CPUs, GPUs, and FPGAs), with only limited modifications to an existing simulator code base and without changes to model code. Our approach relies on manual porting of a small number of core simulator functionalities as found in common SNN simulators, whereas the unmodified model code is analyzed and transformed automatically. We apply our approach to the well-known simulator NEST and make a version executable on heterogeneous hardware available to the community. Our measurements show that at full utilization, a single GPU achieves the performance of about 9 CPU cores. A CPU-GPU co-execution with load balancing is also demonstrated, which shows better performance compared to CPU-only or GPU-only execution. Finally, an analytical performance model is proposed to heuristically determine the optimal parameters to execute the heterogeneous NEST.

CCS Concepts: • **Computing methodologies** → **Massively parallel and high-performance simulations**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Source code generation**; • **Applied computing** → Computational biology;

Additional Key Words and Phrases: Spiking neural network simulators, types of simulation: parallel & heterogeneous, automatic code transformation

ACM Reference format:
Quang anh pham Nguyen, Philipp Andelfinger, Wen jun Tan, Wentong Cai, and Alois Knoll. 2021. Transitioning Spiking Neural Network Simulators to Heterogeneous Hardware. *ACM Trans. Model. Comput. Simul.* 31, 2, Article 9 (April 2021), 26 pages.
https://doi.org/10.1145/3422389

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.
Authors' addresses: Q. A. P. Nguyen, P. Andelfinger, and W. J. Tan, TUMCREATE Limited, 1 Create Way #10-02 CREATE Tower, Singapore 138602; emails: {nqapham, pandelfinger, wjtan}@ntu.edu.sg; W. Cai, Nanyang Technological University, 50 Nanyang Ave, Singapore 639798; email: aswtcai@ntu.edu.sg; A. Knoll, Universität München, Arcisstraße 21, D-80333 Munich, Germany; 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 ACM 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.
© 2021 Association for Computing Machinery.
1049-3301/2021/04-ART9 $15.00
https://doi.org/10.1145/3422389

1 INTRODUCTION

Spiking neural networks (SNNs) are artificial neural networks that are used to model and understand the mammalian brain. Simulations of SNNs often require enormous amounts of computational resources and substantial running times due to the scale and complexity of these networks. For example, it takes more than 30 s to execute a single-threaded simulation for 250 ms of activity in a network of $11,250$ neurons and about 127 million synapses using the NEST simulator [17]. Meanwhile, a mammalian brain contains in the order of 10^7 to 10^{11} neurons, with thousands of synapses per neuron on average [22]. To tackle the computational demands of large-scale SNN simulations, many of the existing simulators are designed for parallel execution using multi-core CPUs in a single-node or multi-node environment. Due to the presence of accelerators, such as **graphics processing units (GPUs)**, in most computers ranging from individual workstations to the largest supercomputers, this leads to the shift in focus of works on high-performance SNN simulation towards accelerators in the past few years. Substantial performance improvements can be achieved in several GPU-enabled SNN simulators over a CPU-based execution [5, 10, 15, 24, 40, 46, 52]. Typically, relevant segments of the simulator codebase have been developed manually for separate CPU and GPU variants. While this approach enables better optimizations for the target hardware, there are code duplications in the simulator and model for each hardware platform, which poses significant challenges in terms of maintainability and extensibility. Further, code development on accelerators is widely considered to be more cumbersome and error-prone than developing only for CPU (e.g., Reference [40]). Thus, it is desirable to minimize the need to develop and modify the accelerator code directly. In contrast to the core simulator functionalities, neuron and synapse models may frequently be added, modified, or extended. Hence, the model development process should be abstracted from the target hardware as much as possible.

Newly developed simulators can achieve some hardware independence by using a template-based model specification [52]. However, a transition path is needed for the existing and widely used SNN simulators such as NEST [17], which targets only CPU execution. Adding support for the execution on heterogeneous hardware to these simulators allows researchers to conduct new simulation studies in a timelier manner, while still relying on widely studied and tested simulator and model implementations. In particular, new studies can easily reproduce previous results from the literature and benefit from the existing validation results, using the existing configuration files and tool flows. Moreover, it also enables researchers to better utilize the existing hardware resources, such as co-execution on both CPU and GPU.

This article presents a semi-automated approach for the transformation of SNN simulators to enable execution on heterogeneous hardware. While our experiments are executed on CPUs and GPUs, the transformed simulator can make use of other heterogeneous platforms, such as FPGAs, and DSPs. However, as demonstrated in Reference [49], more research on targeted algorithms and data representations might be required to provide good performance. Our main contributions are as follows:

(1) We analyze the common architecture of CPU-based SNN simulators, differentiating static components implementing basic simulator functionality, and the portions specifying neuron and synapse models.
(2) We present an approach based on an automated model code transformation to support the transition from a purely CPU-based simulator codebase to an implementation executable on heterogeneous hardware. We propose optimizations for synapse models to reduce their memory consumption.

(3) We demonstrate our approach on the NEST simulator and present performance measurement results when executing NEST on GPUs. Substantial performance gains are achieved over a purely CPU-based execution. Our transformation code is publicly available[1].
(4) We show CPU-GPU co-execution of NEST along with the performance results. It is possible to achieve better performance over a purely CPU-based or GPU-based execution by balancing the workload between CPU and GPU. To provide a heuristic to determine the optimal parameters for a well-balanced co-execution, we propose analytical performance models to represent GPU execution and CPU-GPU co-execution.

The present article is an extended version of our previous conference publication [42]. Beyond the previous results, the benchmark model is described in more detail. Further analysis of the correctness of the implementation has also been done to compare the spike output patterns between CPU and GPU executions. The previous work only featured CPU-only and GPU-only executions, which is extended to CPU-GPU co-execution in this article.

2 BACKGROUND AND RELATED WORK

In this section, we briefly introduce spiking neural networks as well as the simulators for this class of network. Further, we outline fundamentals and existing work on the execution of spiking neural network simulations on heterogeneous hardware.

2.1 Spiking Neural Network Models

SNNs [37] are artificial neural networks that are considered to be more biologically accurate representations of mammal brains than the more abstract neural network models used by common machine learning applications. An SNN is a directed graph with so-called integrate-and-fire neurons as nodes and synapses between two neurons as edges. Integrate-and-fire neurons generate rapid increases or decreases of their membrane potential, so-called spikes, according to the potential changes of the incoming synapses. *Excitatory* neurons trigger a positive change in the membrane potential while *inhibitory* neurons trigger a negative change. The spikes are transmitted to the neighboring neurons across the outgoing synapses, which may increase or decrease the spike's potential. The main application areas of SNN models are in computational neuroscience, which aims at understanding the nervous systems, and in machine learning, e.g., in the robotics field [7, 30].

In addition to the network topology, SNNs are characterized by the neuron and synapse models. Neuron models define neuron state variables and actions on incoming spikes, commonly using differential equations. Typically, these equations are solved numerically by time-stepped integration. Each time step updates a neuron's state according to the incoming spikes and its current state, potentially generating outgoing spikes. Synapse models define the way spikes are affected by the transmission through a synapse.

With *static* synapse types, each spike's potential is affected in a constant way as it travels to the target neuron. In contrast, ***Spike-timing-dependent plasticity*** **(STDP)** models dynamically vary the current spike's potential depending on the current synapse state, which may change with each transmitted spike [41]. An SNN may use several different types of neuron and synapse models.

2.2 SNN Simulators

SNN simulators are programs that execute the network's activities in the form of neuron and synapse behaviors over a period of time. Although the local state of each neuron leads to spike patterns that are usually not synchronous across the network, the synchronous execution of SNN

[1] https://github.com/opencl-nest/opencl-nest.

models permits a simple barrier-base parallelization. The simulation state is advanced by updating each neuron's state according to the chosen neuron model, which is typically reflected computationally by one time step of numerical integration. The number of integration steps before synchronization is required (*super step*), depends on the *lookahead*, which is a delta in time during which state changes of a neuron are guaranteed not to affect other neurons, as defined by the minimum time required for a spike to travel through a synapse. At the end of each super step, synchronization is achieved by exchanging newly generated spikes among the neurons. In parallel and distributed simulation terminology, this execution scheme is similar to synchronous conservative synchronization using the YAWNS algorithm [43].

A number of SNN simulators have been proposed in the literature, frequently focusing on scalability to large networks. The number of neurons in the brain of a mammal is on the order of 10^7 to 10^{11}, with about 1,000 to 10,000 synaptic connections per neuron on average [22]. Thus, the number of synapses is substantial even in small networks. To support large-scale SNN simulation, simulators have been developed targeting high-performance computing environments employing multi-core CPUs and GPUs.

Established simulators such as CARLsim 4 [10], Brian 2 [18], NEURON 7.5 [23], NCS 6 [24], Nemo 0.7 [15], Nengo 2.6 [5], NEST 2.14, HRLSim [40], and PCSIM 0.5 [44] support multi-threaded execution on CPUs, some of them with support for execution on multiple nodes. GeNN 3 [52] can be executed on a single GPU. Some other simulators additionally support the execution in GPU clusters [5, 10, 10, 15, 24, 40]. Among these simulators, CARLsim 4 and NCS 6 support a CPU-GPU co-execution. However, they have to write GPU code for the SNN models manually while our approach utilizes an automatic transition method that requires minimal manual work.

Several authors have explored the execution of SNN simulations on GPUs independently of the established simulators (e.g., References [6, 47]). These works share the approach of manual development and optimization of GPU code. While manually tuned GPU code can provide high performance, maintenance and extensibility are impacted by the presence of hardware-specific code or separate CPU and GPU implementations. This issue is exacerbated when considering the execution on further accelerator types such as FPGAs (e.g., Reference [38]). Among the simulators supporting GPU-based execution, to our knowledge, GeNN [52] and Brian2GeNN [46] are the only ones to support automatic GPU code generation. Brian2GeNN transforms Brian 2 models to GeNN models, and utilize GeNN for GPU code generation. For GeNN, users need to define neuron models using a C++ interface. The main inputs provided by the user are the model parameter names and the neuron update rules in the form of C++ statements. A drawback of both hand-tuned and generated GPU code is the lack of comparability of results to those generated using the well-tested and well-studied code and models of established simulators. The transformation approach proposed in our present article enables the use of unmodified models of an existing CPU-based simulator while reducing the simulation running time using hardware accelerators. While our experiments are performed on GPUs, the generated OpenCL code enables the execution on further hardware types such as FPGAs or DSPs.

2.3 Benchmark Network Model

To benchmark the performance of the simulator, existing literature used a balanced random network model with plastic connections [21, 27, 28, 32, 41], which is featured in the *hpc_benchmark* scenario. The network consists of two recurrently connected populations: one *excitatory* and one *inhibitory*. Neurons are modeled by single-compartment leaky-integrate-and-fire neurons with alpha-shaped postsynaptic currents (*iaf_psc_alpha* neuron model) using homogeneous parameters. Each neuron is randomly connected with a fixed number of incoming connections per neuron ($K = 11{,}250$). The excitatory-excitatory connections used the STDP model (*stdp_pl_synapse_hom* synapse model), while all other connections are static (*static_synapse* synapse model).

The network is driven by random spikes emitted by a Poisson generator (*poisson_generator*). Each simulation time step size is 0.1 ms, while the synaptic delay is 1.5 ms. Hence, each superstep consists of 15 time steps. There is a presimulation time of 50 ms (500 time steps), and the simulation terminates after 250 ms (2,500 time steps).

This model is scalable as the overall spikes patterns do not change with different network sizes due to the sparse connections of neurons [9]. In addition, the random network ensures that no local communication patterns can be exploited. The total number of neurons in the network is controlled by a scale factor S: $N = S \times 11{,}250$. A detailed network description and parameter values can be found in Reference [28].

2.4 Heterogeneous Computing and Graphics Processing Units

In the past two decades, high-performance computing has moved from mostly CPU-based platforms to heterogeneous architectures. While a range of accelerator types have emerged that provide performance benefits for certain classes of computational problems, GPUs are currently the dominant hardware type used to supplement CPU-based host systems with resources for massively data-parallel processing. Of the list of the top 500 supercomputers published in November 2019,[2] 154 of them are built with GPUs. The benefits of CPU-GPU co-processing or purely GPU-based execution have been shown for many scientific applications (e.g., References [14, 19, 35]), including several types of network simulations [1, 2, 50].

In NVIDIA's terminology, a GPU contains several ***streaming multiprocessors (SMs)*** with up to thousands of arithmetic units in total. This enables a GPU to execute thousands of lightweight threads in parallel to provide teraflops of computing power. However, the design of efficient GPU algorithms requires exploiting the GPU's unique architectural characteristics. The programs to be executed by the GPU's threads are referred to as *kernels*. On current NVIDIA GPUs, 32 consecutive GPU threads are grouped into a *warp*, in which threads run the same sequence of instructions in lockstep. If the program executed by a warp includes branches taken by only some of the threads, then the branches are serialized, decreasing performance. Another significant factor for GPU performance is the memory access pattern in the GPU's DRAM. If threads in a warp access 32 consecutive memory locations, then the accesses are served by a single memory transaction. Conversely, if the memory accesses by a warp do not follow this pattern, then multiple transactions are issued, which can severely reduce the overall performance. GPU programs are commonly implemented using frameworks such as CUDA or OpenCL. While the former only targets NVIDIA GPUs, OpenCL supports cross-platform development and deployment on a variety of hardware devices ranging from CPUs to accelerators such as GPUs, APUs, FPGAs, and Intel Xeon Phi. In recent years, these accelerators have shown promising performance results for various simulation problems [51]. To permit execution on a wide range of hardware platforms, the translation approach presented in our present article generates OpenCL code. In the remainder of the article, we refer to the portion of the hardware controlled directly by the CPU as the *host*, and portion executing OpenCL kernels as the *device* or *accelerator*. A CPU thread, also known as *GPU stream*, is used to enqueue the kernels and sleeps while waiting for kernel execution to finish. To maximize hardware utilization, the device can execute kernels from different streams, where each stream can operate independently on separate data.

[2]https://www.top500.org.

2.5 Code Transformation and Generation for Heterogeneous Computing

An automated transformation of a CPU program for efficient execution on heterogeneous platforms often requires the parallelization of suitable program portions. When transforming sequential into parallel code, loops are important sources of parallelism. In automated loop transformations, data dependencies between loop iterations are analyzed. Subsequently, the computations are reordered to maximize the amount of parallelizable operations, while satisfying all data dependencies. The code analysis frequently relies on knowledge of the problem class at hand. For instance, existing works propose tiling approaches for the successive over-relaxation method in linear algebra [13] and for stencil-based loop computations [8]. Da Li et al. [34] present a template-based approach to the parallelization of irregular nested loops and recursive computations for GPUs. Aside from maximizing the exploited parallelism, the second challenge in loop transformation targeting GPUs lies in achieving memory access locality [3]. Hou et al. [26] address the issue of memory access locality by reordering computational operations in wavefront loops without affecting the correctness of the results. Fully automated parallelization is possible for a certain class of nested loops using the so-called polyhedral (or polytope) model [33], which permits a compile-time analysis of data dependencies for affine programs. Several works propose approaches to transform affine programs for parallelized execution on GPUs [4, 48]. While the above approaches can be carried out without domain knowledge, their applicability is limited to programs of a certain structure, e.g., nested loops with predictable control flow.

Another development approach for heterogeneous computing environments is to formulate programs in a ***domain-specific language*** **(DSL)** and to generate code for the target platforms. By providing abstractions tailored to the given problem domain, DSLs are often more concise than general-purpose languages. Further, DSLs can be used to hide low-level or hardware-dependent implementation details. Many DSLs have been proposed to solve computational tasks such as graph problems [25], image processing [29, 39], and social network analysis [16]. A number of DSLs have been proposed to achieve heterogeneous execution for simulations. The DSL by Hawick et al. [20] permits the generation of simulation programs for many problems based on partial differential equations. Similarly, Devito et al. [12] propose a language to build portable mesh-based PDE solvers that can run on multiple platforms. In 2018, Cosenza et al. [11] presented the OpenABL language, which allows users to formulate agent-based simulation models in a hardware-independent manner. The generated code can be executed on CPUs or GPUs using a number of existing frameworks.

Since our goal is to provide support for heterogeneous execution of existing simulators, we do not define our own DSL. Instead, we rely on knowledge of the structure of SNN simulators to enable source-to-source translation from C++ to OpenCL.

3 ANALYSIS OF SPIKING NEURAL NETWORK SIMULATORS

Conceptually, SNN simulations follow a common structure that is reflected in the software architecture of simulators and the computations involved in the process of an SNN simulation run. In the following, we describe the general structure of SNN simulations as a basis for the transition approach presented in Section 4.

3.1 Architecture

An SNN simulation can be seen as a collection of simulations for individual neurons that interact with each other by the exchange of spikes. Often, each neuron is simulated in a time-stepped fashion by updating the neuron's internal state at each step. The changes in neuron state at a time step may trigger zero or more spikes that are then sent to neighboring neurons. Since the spikes must be considered in the target neurons' future state updates, a neuron's state can only be updated

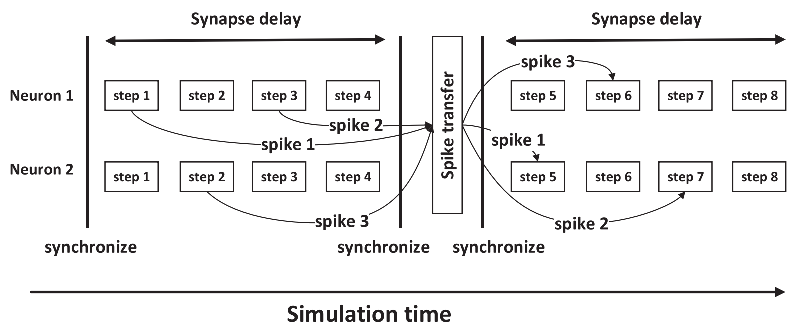


Fig. 1. A sequence of SNN simulation steps. The synaptic delay guarantees that neurons cannot interact within a super step. Thus, inter-neuron synchronization is required only after a super step has been processed.

once it has received all spikes with smaller timestamps. From this outline of an SNN simulation, we can see that the two main operations are the **neuron update** and the **spike delivery**, the facilities for which commonly form the main components of an SNN simulator: The **neuron update** component receives the current states and incoming spikes as input and performs a simulation step for each neuron. The output of this computation step is composed of the neuron's new state and any newly emitted spikes. The **spikes delivery** component is responsible for transferring the spikes among neurons in the network. The delivery may either occur locally, if the connected neurons are simulated in the same process, or remotely via inter-process communication or a physical network.

An important aspect of SNN simulations is the delay in spike transmission reflecting the synaptic delay, i.e., the number of time steps taken by the signal to arrive at and have an impact on the target neuron. Simulators such as NEST [17] exploit this characteristic in a synchronization scheme similar to the YAWNS algorithm [43] to avoid transferring all new spikes at every time step. Instead, the simulation time is divided into super steps with the size of the minimal delay in the neural network, with all spikes generated in a super step only being sent at the end of the super step (cf. Figure 1). With this approach, it is guaranteed that all neurons receive the incoming spikes in time, while the frequency of synchronization is reduced. The super steps also enable larger amounts of computation between synchronization points, which is beneficial for parallel computation on a GPU.

Since there are many different neuron and synapse models that can be used in SNN simulations, simulators typically provide interfaces to allow other simulator components to interact with the *neuron update* and *spike delivery* components regardless of the specifics of the underlying models. Since neuron and synapse models may be added, extended, or modified frequently, we consider the models the **dynamic** parts of a simulator.

The remaining simulator components are independent of the models and typically remain unchanged during modeling and when executing simulations using different combinations of models and parameters. We refer to these as **static** components. For instance, the static components in NEST are the *SimulationManager*, which orchestrates the overall simulation (e.g., by advancing time and triggering synchronization) and the *ConnectionManager*, which provides access to the topology defined by the synapses.

3.2 Neuron and Synapse Models

Neuron models are differentiated by two aspects:

(1) **Neuron state:** Each type of neuron uses a set of variables (e.g., the current membrane potentials) to represent the neuron's state at a given time. Further, some configurable

constants (e.g., the membrane capacitance and resting potential) may exist that affect the neuron's behavior.

(2) **Update function:** The update function defines how the state in the next time step is calculated from the current state and the incoming spikes. If certain conditions are met, then output spikes are generated to be transmitted to other neurons. Since neuron models are often specified in terms of differential equations, the neuron update function frequently involves performing one iteration of a numerical integration per super step.

As described in Section 2.1, synapse models can be either of *STDP* or *static* type. Similarly to neuron models, they can be differentiated according to their state and their update functions:

(1) **Synapse state:** Each synapse in the network may have state variables and parameters. The most important state variables are the weight, which affects a spike's weight on arrival at the target node, and the synaptic delay, which is the time required for a new spike to be received by the target neuron. *STDP* models reflect a synapse's plasticity, i.e., the synapse state variables may be modified each time a spike is transmitted. In contrast, *static* synapse models affect the spike weight in a fixed manner, independently of the state variables. Since STDP models rely on the current value of the state variables to determine each spike's weight, STPD synapses may require substantially larger amounts of memory for their internal data than static models.

(2) **Synapse update function:** In STDP models, each transmitted spike may modify the values of the synapse's state variables. To achieve correct updates of the variables, spikes must be processed according to timestamp order, which is an important constraint when considering parallelized execution. Static synapse models do not require updates of the state variables. Instead, a constant weight value is assigned to the spike before it is forwarded to the target neuron. Thus, while the transmit function for STPD models may require significant amounts of computation, the execution of the transmit function of static synapse models is reflected mainly by the memory accesses required to store new spikes at the target neurons.

The above descriptions imply that while the neuron updates within each super step can be trivially parallelized across neurons, it must be ensured that the spike transmission and update steps at a STDP synapse follow the temporal order of the spikes.

4 PROPOSED TRANSITION APPROACH

In this section, we propose an approach to generate a GPU implementation from a CPU-based SNN simulator in a semi-automated fashion. The approach follows the differentiation of static and dynamic simulator components from Section 3.1, which makes it applicable to several CPU-based SNN simulators: while some basic simulator functionalities are ported manually, neuron and synapse models are analyzed and translated to code executable on heterogeneous hardware in an automated fashion. In contrast, the computationally inexpensive portions of the simulator code remain on the CPU.

4.1 Static and Dynamic Components

As discussed in the previous section, SNN simulators can be regarded as being comprised of **static** and **dynamic** components. The *static* components constitute the simulator core of the simulator, carrying out the basic operations in an SNN simulation such as the creation of neurons and synapses, the triggering of neuron updates, and the exchange of spikes, synchronization among

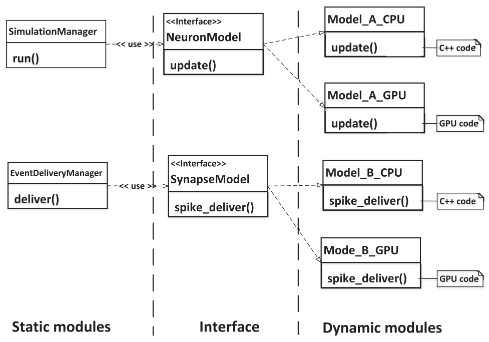


Fig. 2. Static and dynamic components in an SNN simulator.

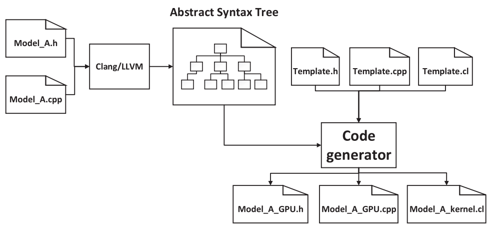


Fig. 3. Compilation workflow from C++ SNN simulator code to OpenCL.

processes, and the time advancement. These essential procedures are carried out by every SNN simulation, regardless of the neuron and synapse models used. Thus, the corresponding components are rarely modified between simulation experiments or in the development cycle of the simulator. The remaining part of an SNN simulator is comprised of the neuron and synapse models, which are used to construct networks. A simulation experiment may use any combination of models from a model library. Further, experiments may require changes or extensions to the existing neuron or synapse models, or the development of new models. Thus, the models constitute the *dynamic* part of the simulator codebase. A common interface for neuron and synapse models is often provided so that the static components can communicate with the dynamic components (cf. Figure 2). The resulting loose coupling among components simplifies the maintenance of the simulator codebase, as *static* components are not affected by changes to the *dynamic* components such as the addition, modification, or extension of models. The loose coupling also enables the development of multiple implementations, i.e., targeting CPUs and GPUs, for the same neuron or synapse model. The GPU-accelerated simulators NCS [24] and Nemo [15] as well as the purely CPU-based NEST simulator rely on such an architecture. In contrast, CARLsim [10], which provides support for GPU execution, does not follow this structure. Instead, it defines separate interfaces for CPU and GPU implementation and the simulator explicitly selects an implementation at runtime.

Since we assume that the *static* components of an SNN simulator are modified only rarely, the effort of a manual translation of code for execution on heterogeneous hardware can be justified. In addition, the static components are loosely coupled to the model implementations, allowing for the required code changes to be reasonably small and self-contained. For example, in our implementation described in Section 5, the key code modifications to add accelerator support were limited to only a few source files.

The same strategy cannot be applied to the *dynamic* components: first, the variety of neuron and synapse models makes a manual translation process cumbersome and error-prone. For instance, version 2.14 of the NEST simulator distribution includes more than 50 neuron and 10 synapse models. Further, it may frequently be necessary to add, modify, or extend models to carry out SNN studies. To handle the diverse and dynamic nature of the models, we propose a tool to automate the transformation of neuron and synapse models for execution on heterogeneous hardware. The transformation relies on the fact that most models share a similar basic structure, with major differences only in the specific computations performed. The transformation tool exploits the similarities among models by applying a template common to a class of models, which is then populated with the model-specific computations.

4.2 Transformation of Neuron Models

In this section, we present the steps required for porting neuron models to code executable on heterogeneous hardware. The overall workflow, which applies both to neuron and synapse models, is illustrated in Figure 3. In the first step, we parse and analyze the model's source code. We assume that each neuron model is defined by a class with the neuron variables being class members. The result of the code analysis is an **abstract syntax tree (AST)** describing the computations performed in the neuron model code, as well as their order. Relying on the fact that the models follow the same high-level structure, we can identify the code snippet carrying out the main computations that define the model behavior (*ModelBehavior* in Algorithm 1). The *ModelBehavior* routine will be translated to OpenCL. To do so, the neuron state variables used in the *ModelBehavior* routine must be identified. The AST allows us to determine several properties of the variables used in the routine: from the scope of a variable, we can determine whether it is local to the current function local data or a member variable of the neuron. If a variable represents a state variable of a neuron, then information on its type (e.g., integer or double-precision floating point) and dimensionality (scalar or array) allow us to allocate memory for the variable in the OpenCL code. To conserve memory, state variables that are not used in the *ModelBehavior* routine are not allocated in the OpenCL code. Finally, we identify function calls to later replace them with calls to equivalent OpenCL functions (cf. Section 5.2).

After the analysis, OpenCL code is generated based on predefined code templates (cf. Figure 3) for accelerator and host code. A critical part in the generation of the accelerator code is producing data access patterns suitable for the chosen hardware. Since our performance evaluations are performed using GPUs, we automatically apply a common memory access optimization applicable to all common GPU accelerators: due to the large number of neurons in a typical SNN, the neuron state variables reside in the GPU's high-capacity DRAM. Unfortunately, CPU code typically relies on an "array-of-struct" (AoS) representation, where neurons are stored as an array of objects, with the state variables as object members. On a GPU, the AoS representation is known to result in large numbers of memory transactions when operating on many array elements in a data-parallel fashion. Thus, we convert the code to follow a "struct-of-array" (SoA) access pattern by flattening out the neuron data into a one-dimensional array for each variable and accessing the entries using each neuron's numerical index. Since within a super step the state updates are independent across neurons, we assign one GPU thread to each neuron. In contrast to the neuron state variables, function-local variables rely on GPU registers by default and thus can be accessed efficiently without further code transformations.

Depending on the data used by the model, host-side OpenCL API calls are generated for GPU memory allocation and data transfer from and to the accelerator. Figure 4 provides pseudo-code of the original and the transformed host code. The result of the code generation step is a set of C++ files containing host and accelerator code. Further details on the transformation of the neuron models as well as the example code are provided in Section 5.2.

4.3 Transformation of Synapse Models and Spike Delivery

Since there are two synapse model types (*static* and *STDP*) that differ in their characteristics, we employ separate strategies for their transformation. One of the factors affecting this design decision is the high degree of nodes in neural networks. Motivated by the node degrees in mammals' brains, a neuron is commonly connected to about 10^4 other neurons. When relying on STDP models, the resulting large numbers of synapses in the network incur substantial requirements both in terms of memory capacity and in terms of computation to transfer spikes across the network. As we will see in Section 5.3, the spike delivery constitutes the largest portion of the simulation

Algorithm 1 A super step in a CPU-based SNN simulation.

```
1: function SimulationUpdate(from, to)
2:     for i ← number of neurons do
3:         neuron[i].Update(from, to)

1: function Update(from, to)
2:     for lag ← from; lag < to; lag ← lag + 1 do
3:         ModelBehavior()
```

Algorithm 2 A super step in a heterogeneous SNN simulator.

```
1: function SimulationUpdate(from, to)
2:     MassUpdate(neurons, from, to)

1: function MassUpdate(neurons, from, to)
2:     if first update then
3:         CopyDataHostToDevice(neurons)
4:     for lag ← from; lag < to; lag ← lag + 1 do
5:         ModelBehaviorGPU()
6:         CopyDataDevicetoHost(neurons)
```

Fig. 4. A super step in a CPU-based SNN simulator and in a simulator supporting heterogeneous hardware.

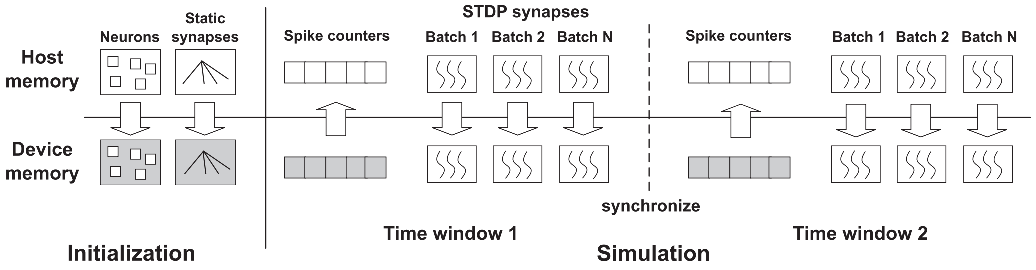


Fig. 5. Data transfer between host and device memory in the initialization step and during the simulation. The shaded shapes illustrate memory allocated permanently on the device.

workload. Therefore, it is critical to achieving high performance and low memory usage for STDP models.

In contrast, in the simple case of *static* synapse models, the computational and memory demands are relatively low. Since the network topology is generated at the start of the simulation, the states of all static synapses can be copied to device memory and remain there over the course of the entire simulation, reducing the overall amount of data transfer between the host and the device. Note that this also applies to the neuron data, as illustrated in Figure 5. The graph is stored in the **compressed sparse row (CSR)** format, connections being grouped by their source node. Each time we perform the spike delivery, the list of firing nodes is copied to the device. The CSR format allows us to quickly determine the outgoing connections from the source node indexes and perform the delivery. Because of the high degree of connectivity, on an NVIDIA GPU, we assign one entire warp (32 threads) to process the outgoing connections of each source node. In *static* synapse models, the order of transferred spikes can be ignored, since their weights are constant. Therefore, we can deliver spikes with different timestamps in parallel. At the target node, atomic operations are utilized to avoid race conditions with respect to multiple spikes received by a target node at the same time.

Since *STDP* synapses consume significantly more memory than *static* models, storing all the *STDP* synapses in device memory would severely limit the feasible network size. To achieve scalability, we instead carry out the spike delivery through STDP synapses one batch of spikes at a time. On a GPU, one thread is assigned to one spike. The spikes traveling through the same synapse must be processed sequentially in timestamp order. To avoid additional processing, we group only spikes with the same timestamp in each batch. When a batch of spikes is processed, the states of *STDP* synapses transmitting these spikes are transferred to the device memory. After a batch has been processed, the memory allocated for the STDP synapse data is freed (cf. Figure 5). The efficiency of this approach depends on the batch size, which is limited by the available graphics memory. The corresponding device kernels are generated automatically according to the same procedure as applied to the neuron models (cf. Figure 3).

5 HETEROGENEOUS NEST

We demonstrate our transition approach on the example of the well-known SNN simulator NEST [17]. NEST is an SNN simulator supporting various neuron and synapse models. In the following, we briefly sketch NEST's architecture and describe our modifications as well as the automatic transformation of neuron models.

The process to enable the automatic transformation of synapse models follows the same principles as the transformation of neuron models. Thus, we leave the implementation of the automatic transformation for synapses to future work. Our implementation currently includes manually ported versions of the synapse models required for the experiments.

5.1 NEST Overview

NEST is implemented using C++ and supports CPUs in shared and distributed-memory environments using OpenMP and MPI. The neural network is divided into a number of ***virtual processes* (VPs)**, where each VP runs as an OpenMP thread. Nodes are assigned to VPs in a round-robin fashion. A synapse is stored in the same VP as the target neuron, which guarantees thread-safety for the spike delivery, since only one thread can write data to a given neuron. The VPs can be assigned to multiple processes communicating via MPI in a distributed-memory environment.

A super step in NEST involves the following operations:

(1) Each VP executes a super step on its assigned neurons, which involves executing the state update function of each neuron according to the selected model. Since the super step is divided into multiple time steps, the internal state of each neuron is updated multiple times, potentially creating new spikes at each iteration. If an emitted spike has a target node in a remote MPI process, then it is stored in a local buffer.
(2) The stored spikes are inserted into an MPI buffer and issued using an MPI_Allgather call to provide all processes with the spike data.
(3) Each VP scans through its MPI receive buffer to obtain the list of source neurons of incoming spikes. If the VP holds one or more target neurons, then it delivers the spikes through the corresponding synapses according to the synapse model.

The division of work across VPs makes it convenient to extend NEST with support for heterogeneous hardware: Since each VP controls a device, multiple VPs can easily exploit the resources of a multi-device system. Further, the support for MPI communication enables a multi-device execution across multiple execution nodes without further development efforts.

The core component of NEST is the *SimulationManager*, which executes the operations of a super step as described above. For spike delivery, the *SimulationManager* relies on the *EventDeliveryManager*, which delivers spikes within and across VPs. The *ConnectionManager* stores the network topology of the current VP.

The above components constitute the core of the framework and do not require modification when varying the neuron or synapse models across simulation runs. Thus, since the code changes required to enable support for heterogeneous hardware are permanent and independent of the chosen models being simulated, we carry out these changes manually.

The main remaining components of the NEST framework are the neuron and synapse models. NEST 2.14 provides more than 50 neuron models and 10 synapse models. To perform a simulation run, the user prepares a script that describes the scenario, which is defined by the neuron and synapse models with their parameters, the number of neurons and connections, the simulated time, and so on. Based on the script, NEST constructs the network with the selected models and executes the simulation. These models are defined as C++ classes and share a common interface so that the core components, particularly the *SimulationManager*, can invoke the model behavior (e.g., as defined by the state update function) in a generic fashion. Using the techniques described in Sections 4.2 and 4.3, we developed a tool that is supplied with a model implementation in the form of C++ code as input and generates an OpenCL implementation executable on OpenCL-supported accelerators. The compilation workflow follows the description in Section 4.2.

5.2 Transformation of Neuron Models

Each neuron model defines the neuron's behavior when interacting with other neurons through incoming and outgoing spikes. In NEST, the behavior is implemented in the `Update` function of the neuron class. The model further defines the internal state variables, which are modified in the `Update` function. Algorithm 1 shows pseudo-code of the steps performed by the `Update` function: using the current state of the neuron and the incoming spikes from other neurons as input, the `Update` function executes state changes caused by the incoming spikes in a super step. The computation is divided into multiple small time steps (lines 2 and 3). In the loop, the neuron executes the procedure *ModelBehavior*, which is model-specific. At each time step, depending on the current state, the neuron may emit spikes to be transmitted to neighboring neurons. The *SimulationManager* invokes the *SimulationUpdate* function, which iterates through the neurons of the local VP and invokes the `Update` function on each of them. The above procedure may be executed by multiple VPs on disjunct sets of neurons concurrently.

To enable execution of the `Update` function on heterogeneous hardware, the computation scheme is modified as shown in Algorithm 2. Now, the *SimulationManager* invokes `MassUpdate`, which processes all neurons of the given type. In `MassUpdate`, if the neuron data has not been initialized on the device, we copy the neuron data from the host to device memory (lines 2 and 3). Hence, the data is transferred to the device only once. Then, at each time step, all neurons are updated in parallel on the device. During the update, neurons may emit spikes. Thus, we invoke *CopyDataDtoH* (line 6), which copies *spike counters* from the device to the host that indicate to the host the number of spikes generated by each neuron. The number of counters is equal to the number of neurons. Currently, we represent the counters as 4-byte variables. However, since in practice the values are small, depending on the scenario it may be sufficient to represent them using even fewer bytes. In our experiments, this data did not incur substantial data transfers overhead.

An important step in the transformation is the generation of a device kernel for *ModelBehaviorGPU*. To this end, we implemented a parsing tool based on LibTooling from the CLang/LLVM toolkit[3] that parses the C++ code of the *ModelBehavior* function of the neuron model class. The parsing tool allows us to obtain the following required pieces of information:

(1) The list of state variables used by the *ModelBehavior* function to identify the variables to be transferred to the device.

[3]https://clang.llvm.org/docs/LibTooling.html.

Listing 1. Original C++ code

```
 1  if ( S_.r_ == 0 ) {
 2    S_.y3_ = V_.P30_ * ( S_.y0_ +
 3             P_.I_e_ ) + ... ;
 4
 5    S_.y3_ = ( S_.y3_ < P_.LowerBound_ ?
 6                       P_.LowerBound_ : S_.y3_ );
 7  } else {
 8    --S_.r_;
 9  }
10  S_.I_ex_ = V_.P21_ex_ * S_.dI_ex_
11             + ...;
12  ...
13  // Ring buffer
14  V_.weighted_spikes_ex_ = B_.ex_spikes_.get_value
15                                           (lag);
16
17  ...
18  // Ring buffer
19  V_.weighted_spikes_in_ = B_.in_spikes_.get_value
20                                           (lag);
21
22  ...
23  if ( S_.y3_ >= P_.Theta_ ) {
24    S_.r_ = V_.RefractoryCounts_;
25    S_.y3_ = P_.V_reset_;
26    // Firing spikes
27    set_spiketime( Time::step( origin.get_steps()
28                             + lag + 1 ) );
29    SpikeEvent se;
30    kernel().event_deliv_mgr.send(*this,se,lag);
31  }
32  // Ring buffer
33  S_.y0_ = B_.currents_.get_value( lag );
34
35
36  ...
```

Listing 2. Generated OpenCL code

```
 1  if ( S__r_[tid] == 0 ) {
 2    S__y3_[tid] = V__P30_[tid] * ( S__y0_[tid] +
 3                  P__I_e_[tid] ) + ...;
 4
 5    S__y3_[tid] = (S__y3_[tid] < P__LowerBound_[tid] ?
 6                   P__LowerBound_[tid] : S__y3_[tid]);
 7  } else {
 8    --S__r_[tid];
 9  }
10  S__I_ex_[tid] = V__P21_ex_[tid] * S__dI_ex_[tid]
11                  + ...;
12  ...
13  // Ring buffer
14  V__weighted_spikes_ex_[tid] = ring_buffer_get_value(
15      ex_spikes_, ring_buffer_size, num_nodes, tid,
16      lag, time_index);
17  ...
18  // Ring buffer
19  V__weighted_spikes_in_[tid] = ring_buffer_get_value(
20      in_spikes_, ring_buffer_size, num_nodes, tid,
21      lag, time_index);
22  ...
23  if ( S__y3_[tid] >= P__Theta_[tid] ) {
24    S__r_[tid] = V__RefractoryCounts_[tid];
25    S__y3_[tid] = P__V_reset_[tid];
26    // Firing spikes
27    spike_count[tid]++;
28
29
30
31  }
32  // Ring buffer
33  S__y0_[tid] = ring_buffer_get_value(currents_,
34                   ring_buffer_size, num_nodes, tid,
35                   lag, time_index);
36  ...
```

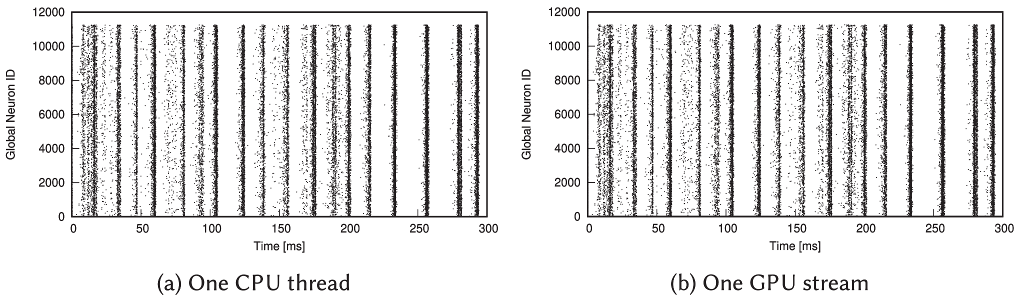
Fig. 6. Update function of the *iaf_psc_alpha* neuron model in the C++ and OpenCL implementation.

(2) The type and dimensionality of each variable. This information is used to allocate device memory.

(3) Function calls performed during the update. The update function may invoke utility functions to (i) retrieve incoming spikes from a neuron's local storage and (ii) emit spikes.

Listing 1 of Figure 6 shows an example of the C++ code of the *ModelBehavior* routine for the *iaf_psc_alpha* model. This code executes a number of instructions on the neuron's state variables such as *S_.r_*, *S_.y3_*, *S_.I_ex_*. These variables are of primitive types (**int** or **double**), often as members of a C++ struct. The parsing tool gathers the type information from the class header file. Further, the accesses to the ring buffer holding incoming spikes (lines 14, 19, and 33) and the transmission of spikes (lines 27–30) are detected as function calls to *get_value* and *event_delivery_manager.send.*

Listing 2 of Figure 6 shows the generated OpenCL kernel code. All accesses to neuron state variables have been transformed from an array-of-struct (AoS) to a GPU-friendly struct-of-array (SoA) pattern. For instance, the variable *S_.r_* is of type **int**, where *S_* is an instance of the *State_* structure defined in the neuron class. In the OpenCL implementation, we define an **int** array *S__r_* of size identical to the number of neurons. Now, the device thread with index *tid* accesses the *S__r_[tid]* entry in the array. The utility function for accessing the ring buffer of incoming spikes is replaced with a kernel call. For sending spikes, we maintain a counter *spike_count* of the number of spikes sent by each neuron. The code portion for triggering spikes in the neuron *tid* is replaced by an incrementation of *spike_count*[*tid*]. This counter is later copied back to the host (Algorithm 2, line 6) so that the sending of spikes can rely on the existing facilities of NEST (Listing 1, lines 27-30).



(a) One CPU thread (b) One GPU stream

Fig. 7. Spike output of CPU and GPU execution on the network size of 11,250 neurons.

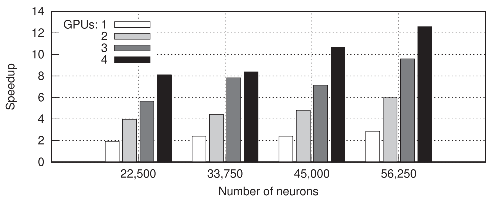


Fig. 8. Speedup with one stream per GPU compared to execution on a single CPU core.

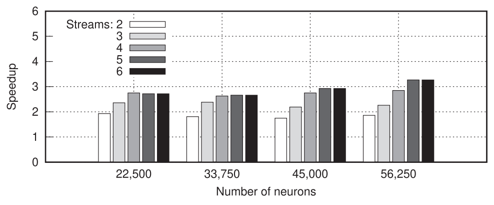


Fig. 9. Speedup when executing multiple streams on a single GPU over a single stream.

5.3 Evaluation

We executed the simulations on a system comprised of two nodes using Intel Xeon Gold 6148 2.4 G CPUs with 20 physical cores and 40 threads each, and four NVIDIA Tesla V100 SXM2 GPUs with 16 GiB of RAM. The toolchain is comprised of GCC 7.2.0, CUDA toolkit 9.0, and OpenMPI 1.10.7. The GPU implementation is developed from version 2.14 of the NEST simulator. However, the CPU results are obtained using the development branch of NEST 2.14, which in our experiment provided better performance than NEST 2.14.

5.3.1 Verification. The correctness of our implementation is verified by comparing the total number of spikes between CPU and GPU runs for a number of scenarios with different network sizes and simulation times. Since a random number generator is associated with each VP, CPU and GPU runs are compared using the same number of CPU threads and GPU streams, respectively. In all tested cases, the results produced by the CPU and GPU variants are identical.

As the number of processed spikes has a strong impact on NEST performance, the comparison of spike frequencies is also important [31]. Figure 7 shows the spikes generated by each neuron for each time step for the CPU execution using 1 CPU thread and GPU execution using 1 GPU stream. Identical spike patterns are generated during CPU and GPU execution, as illustrated in Figures 7(a) and 7(b).

5.3.2 Performance. In the first experiment, we measure the speedup obtained by our GPU version. Figure 8 shows the throughput improvement using single and multiple GPUs compared to single-threaded CPU execution at different SNN scales. In the CPU version, we run the NEST 2.14 with only 1 VP, i.e., in a single thread. Our GPU program runs in a single process with one OpenMP thread controlling each GPU. This setup does not involve MPI communication. We observe that in most cases, the throughput scales almost linearly with the number of GPUs. For example, at a network size of 56,250 neurons, the speedup of 2, 3, and 4 GPUs is 6.0, 9.6, and 12.6 over a single CPU; or 1.7, 2.8, and 3.7 over a single GPU. The results also show that the GPU implementation

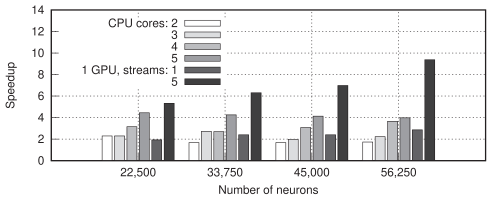


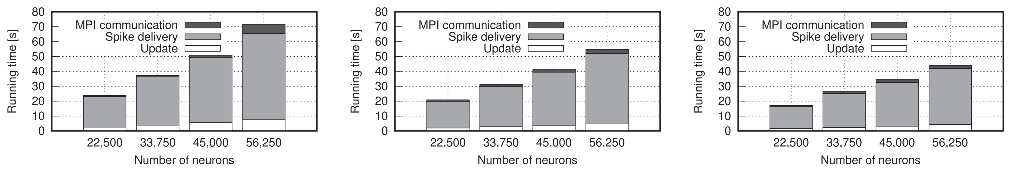
Fig. 10. Performance comparison between a single multi-core CPU and a single GPU. The results are given as speedup factors over one CPU core.

achieves better performance as the network size increases. At 22,500 neurons, the speedup factors over a single CPU core using a single and 4 GPUs are 1.92 and 8.10. Meanwhile, at 56,250 neurons, these ratios are 2.86 and 12.58. The reason for this result is the improved utilization of the GPU at larger network sizes.

In the previous experiment, each GPU was controlled by a single thread. To better exploit the GPU's computing resources, we vary the number of CPU threads (or GPU streams) sharing a single GPU. We compare the performance of multi-stream single-GPU runs with the performance of a single-stream single-GPU execution. The results are shown in Figure 9. Note that, since the remaining CPU portion of the simulator still performs minor amounts of work, e.g., for data serialization, larger numbers of threads also reduce the time spent on the CPU portion. Further, the performance is improved by overlapping CPU-GPU data transfers in one thread with kernel computation in other threads. Generally, by increasing the number of threads (GPU streams) sharing a single GPU, the performance improves. However, at 6 threads, the performance starts decreasing slightly. A likely cause is the overhead of the additional kernel calls in relation to the smaller amount of computation per call.

We also compare the performance between the multi-core CPU and GPU execution. Figure 10 shows the speedup over single-threaded CPU execution at different network sizes. Overall, the performance gain yielded on the multi-core CPU is consistent across network sizes. For example, a run using 4 threads is about 2.7 to 3.7 times faster than a single-threaded run. Meanwhile, as mentioned above, the GPU performance increases with the network size. When using a single stream to control the GPU, the GPU performance is comparable to 2 to 3 CPU cores. When relying on 5 threads to control the GPU, substantially higher performance is achieved: even at 22,500, the GPU outperforms 5 CPU cores. At 56,250 neurons, the speedup over a single CPU core is 9.4, or 2.4 times faster than a run using 5 CPU cores.

We also evaluate the effect of MPI communication on the performance of our NEST GPU implementation. Instead of executing multiple threads per process as in the previous experiments, we execute multiple MPI processes, each of which is single-threaded and uses one GPU. To quantify the proportion of runtime spent on different phases of the simulation, each process controls the GPU using only one stream, although we have previously seen that employing multiple threads per GPU increases the performance dramatically. Inter-node communication relies on the existing MPI communicating scheme of NEST 2.14. We measure the running time of each of the following: *update*, *spike delivery*, and *MPI communication*. The results are illustrated in Figure 11. It is evident that the running time is dominated by the *synapse update*. In our experiments, the proportion of time spent on *spike delivery* is about 81% to 87%. Compared with the original NEST implementation on a single CPU, our GPU method reduces the absolute running time of the *spike delivery* by a factor of about 2 compared to the single-threaded run. The results show the importance of the performance of the *spike delivery* for the overall running time. The *update* step accounts for about 10% of the running time, while *MPI communication* accounts for 2% to 8%. We conclude that in our



(a) 2 MPI processes (b) 3 MPI processes (c) 4 MPI processes

Fig. 11. Breakdown of the running time with different numbers of MPI processes on a single execution node, each using one GPU.

small-scale setup, MPI communication does not hinder performance increases through the use of GPUs.

5.3.3 Memory Consumption. Since accelerators such as GPUs are typically equipped with lower amounts of memory than host CPU memory, efficient use of the available memory is critical to be able to process large segments of the network concurrently. Due to the large degree of connectivity in SNNs, the synapse data typically constitutes the largest part of the memory consumption of an SNN simulation. For example, in a network of 22,500 neurons, the synapse data accounts for 97% of the total memory consumptions. In Section 4.3, we presented our strategy to efficiently represent the network in memory. The memory consumption of the benchmark from our performance measurements is roughly proportional to the number of neurons in the network. A simulation of a network of 11,250 neurons required 3.7 GiB of host memory and 0.8 GiB of graphics memory. At 56,250 neurons, about 18 GiB of host memory and 2.9 GiB of graphics memory are required. Our experiments consumed about a factor of 4.6 to 6.2 more host memory than graphics memory.

The main factor limiting the maximum size of the simulated network is the available memory on the accelerator. The applicability of existing work on GPU-accelerated SNN simulations to the large-scale networks frequently encountered in practice is limited. Often, the considered networks contained only 100 to 1,000 connections per neuron so that the entire network could be stored in GPU memory. By batching the computational work (cf. Section 4.3), the accelerator memory consumption in our implementation is largely decoupled from the network size. Hence, our GPU implementation is still applicable to large-scale SNNs.

6 CPU-GPU CO-EXECUTION

Since recent supercomputers frequently rely on combinations of CPU and GPU devices, a CPU-GPU co-execution enables the full use of such environments. For a CPU-GPU co-execution, the workload consisting of neurons and synapses is divided based on the total number of VPs, each of which is processed either by the original CPU-based NEST or the generated OpenCL implementation running on GPU. By balancing the workload between CPU and GPU, better performance can be achieved compared to executing only on CPU or GPU. The main challenge lies in balancing the workload between CPU and GPU, since typically, a GPU will process each time step substantially faster than a CPU core.

6.1 Evaluation

We evaluated the simulations on a single compute node using two Intel Xeon E5-2670 2.6 GHz CPUs with 8 physical cores and 16 threads each, and 4 NVIDIA Tesla K20 GPUs with 6 GiB of RAM. NEST has been compiled with the same configurations described in the previous section. This setup does not involve any MPI communication. The GPU streams are distributed among the GPUs.

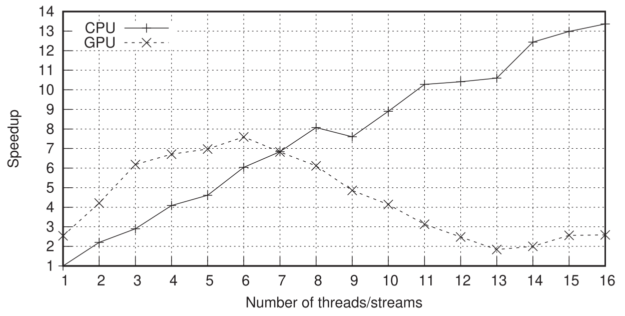
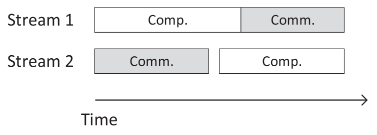
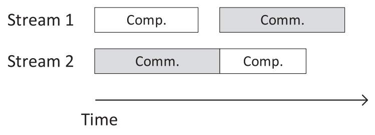


Fig. 12. Strong scaling when simulating 56,250 neurons on a single compute node. Speedup of multiple CPU threads and multiple GPU streams compared to a single CPU core.



(a) Communication time is overlapped with computation time.



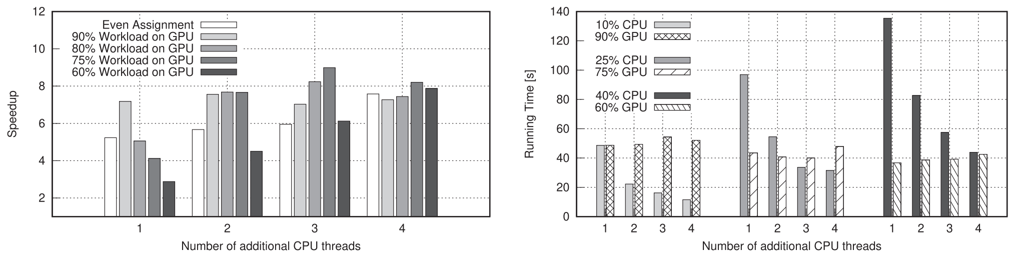
(b) Computation time is too short to hide communication time.

Fig. 13. Overlapping communications and computations.

We conducted experiments to empirically determine the number of CPU threads and GPU streams that achieve the best performance. First, a strong-scaling experiment is conducted for CPU execution and GPU execution, where the total problem size is fixed at a scale of 56,250 neurons while the number of threads (or streams, respectively) is varied. An *even assignment* is used to divide the workload among the VPs, i.e., the same number of neurons and synapses are allocated to each VP. Increasing the number of VPs will reduce the workload per thread. The speedups of CPU threads or GPU streams are compared to a single CPU core, as shown in Figure 12. With the CPU execution, the parallelization efficiency decreases only slightly as the number of CPU threads increases. GPU execution shows a strong scaling up to 6 GPU streams, where it exhibits higher speedup (7.59) compared to 6 CPU threads (6.04). However, GPU execution shows a slowdown when more than 6 streams are used. As GPU-only execution is faster than CPU-only execution in some cases, our implementation can gain performance benefits from a CPU-GPU co-execution.

When the number of GPU streams is low, the computation workload per GPU stream is higher than the communication workload. Hence, it is possible to overlap the communication time with the computation time, as shown in Figure 13(a). As the number of GPU streams increases, the computation workload per GPU stream becomes lower, because the workload processed by each GPU becomes smaller. The computation time is too short to hide the communication time. For example in Figure 13(b), the communication in stream 2 cannot start even if the computation in stream 1 is completed earlier. This is because the bandwidth for communication is fully utilized by stream 1, blocking the communication in stream 2. Therefore, when the number of GPU streams increases, the overall running time increases due to the serialization of communication time. If a faster interconnect between the host CPU and GPU is used, e.g., NVSwitch, then the communication time can be reduced. CPU-only execution does not have this additional overhead, hence increasing the number of CPU cores will give better speedup.

For a system with low numbers of CPU cores and GPUs, it is possible to achieve a performance benefit with CPU-GPU co-execution. For example, executing up to 6 GPU streams can achieve



(a) Speedup compared to a single CPU core execution. (b) Comparing the average running time of CPU threads and GPU streams.

Fig. 14. CPU-GPU co-execution using a different number of CPU threads and workload assignments on a network size of 56,250 neurons. Number of GPU streams is fixed at 4.

a better speedup compared to 6 CPU threads (as shown in Figure 12). Since the GPU execution does not scale efficiently beyond 6 streams, the remaining workload can be processed by the CPU threads to improve overall performance over executing only using CPUs or GPUs. Due to the differences in processing speed, the amount of workload assigned to a CPU core or GPU stream should be chosen separately.

In Figure 14, we simulate a network size of 56,250 neurons and measure the speedup compared to a single CPU core execution. By selecting the number of streams where GPU-only execution is faster than CPU-only execution, we can determine the trade-off between the number of streams and the number of threads. We used a fixed number of 4 GPU streams and vary the number of additional CPU threads up to four threads. Figure 14(a) compares the speedup across a different number of CPU threads and workload assignments. For even assignment, when 4 GPU streams and 4 CPU threads are used, there is a speedup of 7.57× compared to a single-threaded run. This is comparable to just running 6 GPU streams (7.59×), but slower compared to 8 CPU threads (8.07×).

Figure 14(b) shows the average running times of the CPU threads and GPU streams for the whole simulation when using different numbers of CPU threads and workload assignments. The synchronization time between the threads/streams are not included. Comparing with Figure 14(a), we can see that the best performance is achieved when the difference between the average running times for CPU threads and GPU streams is small. The reason is that these threads/streams are fully utilized and not idle-waiting for other threads/streams to complete. For example, when using only one CPU thread, the best configuration is an assignment of 10% of the workload to the CPU, and it can be seen in Figure 14(b) that the CPU and GPU running time is almost equal. Similarly, by distributing 25% of the workload among these 3 CPU threads, well-balanced running time can be achieved. This configuration yields the best speedup by CPU-GPU co-execution at 8.99×.

From this experiment, we conclude that for a compute node with a small number of cores and GPUs, it is possible to achieve better performance using CPU-GPU co-execution by balancing the workload between CPU and GPU through the assignment of a different number of neurons to a CPU core or GPU stream separately. However, benchmarking is required to determine the optimal number of CPU threads and GPU streams for co-execution, and the number of neurons allocated to each CPU and GPU.

6.2 Simulation Performance Model

6.2.1 CPU and GPU Performance Model. In this subsection, we propose to use an analytical model of the simulation performance to determine suitable co-execution parameters to achieve the best possible performance given an SNN model to be simulated. An existing performance model for

CPU-based NEST simulations has been developed based on a semi-empirical approach [45]. The authors collected measurements of the runtime performance of NEST under various parameter settings and subsequently fitted the analytical model to the empirical data. The performance model is specific to an SNN model, as the SNN model determines the connectivity between the neurons and the spiking behaviors, which strongly influences the spike delivery time. It allows predictions regarding the SNN model for any simulation size without the need for new profiling runs. We extend the original CPU performance model for GPU and then combine them to form a CPU-GPU co-execution performance model.

First, we present the original CPU performance model in Reference [45]. The simulation is executed on the benchmark model described in Section 2.3. The original CPU-based NEST simulator is executed in a distributed manner over M MPI processes and within each process over T OpenMP threads. The network model contains N neurons, which is scaled by the scale factor S such that $N = S \times 11{,}250$. It also has a constant fan-in of $K = 11{,}250$ synapse per neuron. The random external input is provided by Poisson generators with a total firing rate of $F = K_x \times V_x$, with K_x being the number of external inputs and set to 9,000, and V_x being the rate of single external input and set to 2.3 spikes per second [28]. The total firing rate is also scaled by S. Due to the homogeneous neurons and synapses as well as even distribution of spikes among all synapses, each VP (i.e., thread) has to deal with about the same amount of workload.

The NEST simulation has been broken down into the following components, showing the time complexity of each component:

(1) Update of neuron and synapse states:
$t_{update} \in O(\frac{N}{MT})$.
(2) Main loop of spike delivery within each virtual process:
$t_{deliver\_main} \in O(MT)$.
(3) Checking every spike event for its relevance to the virtual process (part of spike delivery):
$t_{all\_spikes} \in O(FS)$.
(4) Processing relevant spikes within each virtual process (part of spike delivery):
$t_{relevant\_spikes} \in O((1 - e^{-\frac{K}{MT}})FS)$.
(5) Generating random external input (spikes) to the neural network:
$t_{poisson} \in O(\frac{N}{M})$.
(6) MPI communication (B_{size} is the size of the per-process send buffer):
$t_{COM} \in O(B_{size}M)$.

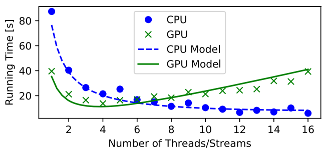
The estimated overall running time $\widehat{t}_{CPU}$ using CPU cores only is

$$\widehat{t}_{CPU} = \widehat{t}^{C}_{update} + \widehat{t}_{deliver\_main} + \widehat{t}_{all\_spikes} + \widehat{t}^{C}_{relevant\_spikes} + \widehat{t}_{poisson} + t_{COM} \tag{1}$$

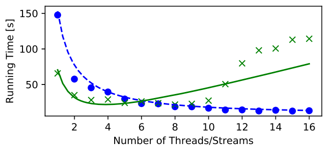
$$= p_0 \frac{N}{MT} + p_1 MT + p_2 FS + p_3 \left(1 - e^{-\frac{K}{MT}}\right) FS + p_4 \frac{N}{M} + p_5 B_{size} M, \tag{2}$$

with $(p_0, \ldots, p_4)$ being the vector of free parameters for model fitting. p_5 can be set to a fixed value based on NEST MPI communication benchmarks to limit the number of free parameters. If executed on a single compute node only, then the component t_{COM} can be ignored.

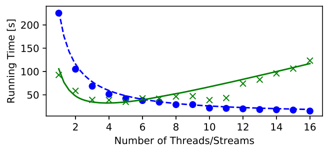
Our GPU version executes the neuron update (1) and spike processing (4) on GPU, while the rest of the components are executed in the same way as the CPU execution. Hence, parameters (p_1, p_2, p_4, p_5) are the same for GPU execution. There are three additional parameters for the GPU performance model (p_6, p_7, p_8). For the spike processing, $\widehat{t}^{G}_{relevant\_spikes}$ is scaled by $p_7 T + p_8$ to



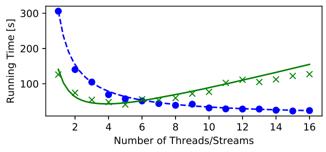
(a) 11,250 Neurons



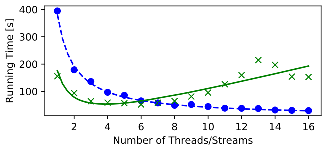
(b) 22,500 Neurons



(c) 33,750 Neurons



(d) 45,000 Neurons



(e) 56,250 Neurons

Fig. 15. Results of performance model fitting on CPU and GPU execution, comparing between estimated and measured running times.

consider the serialization of the GPU communication:

$$\widehat{t}^{G}_{update} = p_6 \frac{N}{MT}, \tag{3}$$

$$\widehat{t}^{G}_{relevant\_spikes} = (p_7 T + p_8) \times \left(1 - e^{-\frac{K}{MT}}\right) FS. \tag{4}$$

Hence, the overall analytical model for modeling the GPU performance is as follows:

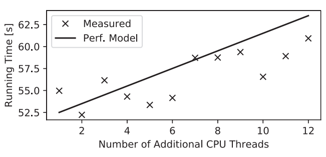
$$\widehat{t}_{GPU} = \widehat{t}^{G}_{update} + \widehat{t}_{deliver\_main} + \widehat{t}_{all\_spikes} + \widehat{t}^{G}_{relevant\_spikes} + \widehat{t}_{poisson} + t_{COM} \tag{5}$$

$$= p_6 \frac{N}{MT} + p_1 MT + p_2 FS + (p_7 T + p_8) \times \left(1 - e^{-\frac{K}{MT}}\right) FS + p_4 \frac{N}{M} + p_5 B_{size} M. \tag{6}$$

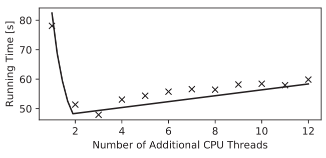
First, the analytical model for $\widehat{t}_{CPU}(M, T, S)$ was fitted to the empirical data of CPU execution by non-linear least-squares optimization. After the model fitting, parameters $(p_0, p_1, p_2, p_3, p_4, p_5)$ are determined. Parameters (p_1, p_2, p_4, p_5) are reused in $\widehat{t}_{GPU}(M, T, S)$. Next, the analytical model for $\widehat{t}_{GPU}(M, T, S)$ was fitted to the empirical data of GPU execution to determine the remaining parameters (p_6, p_7, p_8).

After fitting the analytical performance models to the empirical data, the coefficient of determination for CPU execution and GPU execution amount to $R^2_{CPU} = 0.991$ and $R^2_{GPU} = 0.846$, respectively. A detailed comparison is shown in Figure 15, where it compares between estimated and measured running times. The measured running times are plotted as dots, and fitted models are drawn as lines. Each sub-figure compares the estimated and measured running times for different network sizes with CPU and GPU execution. We can observe that $\widehat{t}_{CPU}(M, T, S)$ is fitted with high accuracy across different network sizes. While higher deviations are observed for $\widehat{t}_{GPU}(M, T, S)$ for some network sizes, a good fit is achieved in most cases. Hence, the fitted analytical models can be used to predict the execution time for any simulation size and determine the performance benefits of executing the simulation on CPU or GPU.

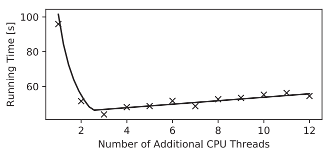
6.2.2 CPU-GPU Co-execution Performance Model. Based on the CPU and GPU performance model, we propose a CPU-GPU co-execution performance model. The parameters from the CPU and GPU performance models are used to construct the performance model for CPU-GPU co-execution. Since the components (2, 3, 5, 6) are the same in the CPU and GPU models, we can split



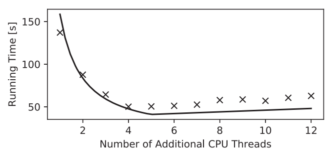
(a) 90% workload on GPU



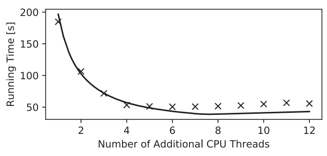
(b) 80% workload on GPU



(c) 75% workload on GPU



(d) 60% workload on GPU



(e) 50% workload on GPU

Fig. 16. Comparison between estimated and measured running times for CPU-GPU co-execution on a network size of 56,250 neurons, using 4 GPU streams.

the performance models into:

$$\widehat{t}_{base} = \widehat{t}_{deliver\_main} + \widehat{t}_{all\_spikes} + \widehat{t}_{poisson} + t_{COM}, \tag{7}$$

$$\widehat{t}'_{CPU} = \widehat{t}^{C}_{update} + \widehat{t}^{C}_{relevant\_spikes}, \tag{8}$$

$$\widehat{t}'_{GPU} = \widehat{t}^{G}_{update} + \widehat{t}^{G}_{relevant\_spikes}. \tag{9}$$

$\widehat{t}_{base}$ is the common components that are executed for both CPU and GPU execution, while the remaining workload depends on either the CPU or GPU execution. For a CPU-GPU co-execution, T_C CPU threads and T_G GPU streams will be used, which are determined through experimental runs of CPU-only and GPU-only executions. T_G GPU streams are selected such that the GPU-only execution is faster than CPU-only execution. This means that it is more efficient to execute using GPU streams compared to the same number of CPU threads. Given the ratio α of workload assigned to GPU, the overall analytical model for co-execution is as follows:

$$\widehat{t}^{C}_{co} = \widehat{t}_{base}(M, T_C + T_G, S) + \widehat{t}'_{CPU}(M, T_C, S) \times (1 - \alpha), \tag{10}$$

$$\widehat{t}^{G}_{co} = \widehat{t}_{base}(M, T_C + T_G, S) + \widehat{t}'_{GPU}(M, T_G, S) \times \alpha, \tag{11}$$

$$\widehat{t}_{co} = \max(\widehat{t}^{C}_{co}, \widehat{t}^{G}_{co}). \tag{12}$$

The optimal α can be determined empirically based on fixed T_C and T_G. $\widehat{t}^{C}_{co}$ is the estimated running time for the CPU threads and $\widehat{t}^{G}_{co}$ is the estimated running time for the GPU streams. The overall estimated running time is the slower time between $\widehat{t}^{C}_{co}$ and $\widehat{t}^{G}_{co}$.

The results of the co-execution analytical model are validated against empirical data of CPU-GPU co-execution from the previous subsection. The detailed comparison between estimated and measured running times for CPU-GPU co-execution on a network size of 56,250 neurons is shown in Figure 16. Each sub-figure compares different workload assigned to GPU. The coefficient of determination based on the validation is $R^2_{co} = 0.919$. When 90% of the workload is assigned to the GPU in Figure 16(a), the running time is dominated by the GPU execution time. When the workload assigned to GPU decreases, if a small number of CPU threads are used, the running time is dominated by the CPU execution time. When more CPU threads are used, the overall running time decreases as the remaining workload is divided among the additional CPU threads. The running time is minimal when the workload is well-balanced between the CPU and GPU.

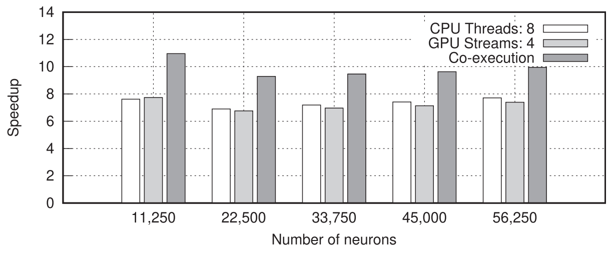


Fig. 17. Estimated speedup for 8 CPU threads, 4 GPU streams, and CPU-GPU co-execution compared to a single CPU core.

These performance patterns are modeling with high accuracy by our co-execution performance model.

To showcase the benefits of the analytical model for co-execution, Figure 17 shows the speedup for 8 CPU threads, 4 GPU streams, and CPU-GPU co-execution compared to a single CPU core across different network size. The optimal parameters for $\widehat{t_{co}}$ are determined through an exhaustive parameter exploration to find the minimal estimated running time. CPU-GPU co-execution has an average speedup of 1.33× and 1.36× over CPU or GPU execution, respectively. In conclusion, by fitting the analytical model of CPU and GPU execution, we can estimate the co-execution performance and determine the suitable parameters to achieve better performance.

7 DISCUSSION AND CONCLUSIONS

We presented an approach to transform CPU-based spiking neural network simulators to enable their execution on heterogeneous hardware. Our approach relies on manual porting of static core simulator functionalities, whereas neuron model code is analyzed and transformed automatically. We demonstrated our approach by transforming the well-known NEST simulator to support OpenCL, enabling its acceleration using heterogeneous hardware platforms such as GPUs, FPGAs, and DSPs. Since the transformed code supports co-execution on multiple device types, better hardware utilization and lower runtimes can be achieved on modern supercomputers with GPUs. Our performance measurements show that at sufficient utilization, a single GPU achieves the performance of about nine CPU cores. A CPU-GPU co-execution with load balancing is also demonstrated, which shows better performance compared to purely CPU-only or GPU-only execution. Based on an existing CPU performance model, the corresponding analytical performance models for GPU execution and CPU-GPU co-execution are also proposed and validated against the empirical data.

Compared to other GPU-accelerated simulators that use NVIDIA's CUDA platform [10, 15, 24, 40, 52], the OpenCL code generated by our code transformation can also exploit other types of accelerators (e.g., FPGAs and DSPs) in addition to GPU, which may further accelerate certain experiments. Future research in hardware-specific optimizations is required to achieve good performance. In addition, investigating the energy cost of execution on accelerators is important to justify the transition to a different architecture where energy utilization is one of their key design considerations.

Our implementation targets NEST version 2.14. NEST 2.16 [36] has been released with improvements in network build time and scalability over the NEST 2.14 [28] and comparable performance in small-scale and medium-scale systems. Extending our work to NEST 2.16 could allow us to support large networks at an even higher performance. Further, our transformation approach could be applied to other established CPU-based simulators such as NEURON [23] and PCSIM [44].

Finally, the OpenCL-accelerated NEST stores neuron state data permanently in accelerator memory. To support the data collection requirements of real-world studies, it may be necessary to periodically transfer parts of this data to host memory. In future work, it would be also interesting to explore methods for *in situ* data analysis in accelerator memory, transferring only the analysis results to the host.

REFERENCES

[1] 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)*. IEEE, 3471–3482.

[2] 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.

[3] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. 2010. Putting automatic polyhedral compilation for GPGPU to work. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC'10)*. Vienna, Austria.

[4] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, 244–263.

[5] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C. Stewart, Daniel Rasmussen, Xuan Choo, Aaron Voelker, and Chris Eliasmith. 2014. Nengo: A Python tool for building large-scale functional brain models. *Front. Neuroinform.* 7 (2014), 48.

[6] Mohammad A. Bhuiyan, Vivek K. Pallipuram, Melissa C. Smith, Tarek Taha, and Rommel Jalasutram. 2010. Acceleration of spiking neural networks in emerging multi-core and GPU architectures. In *Proceedings of the International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW'10)*. IEEE, IEEE, 1–8.

[7] Zhenshan Bing, Claus Meschede, Florian Röhrbein, Kai Huang, and Alois C. Knoll. 2018. A survey of robotics control based on learning-inspired spiking neural networks. *Front. Neurorobot.* 12 (2018), 1–35.

[8] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. 2014. Tiling and optimizing time-iterated computations over periodic domains. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*. IEEE, IEEE, 39–50.

[9] Nicolas Brunel. 2000. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *J. Comput. Neurosci.* 8, 3 (2000), 183–208.

[10] Ting-Shuo Chou, Hirak J. Kashyap, Jinwei Xing, Stanislav Listopad, Emily L. Rounds, Michael Beyeler, Nikil Dutt, and Jeffrey L. Krichmar. 2018. CARLsim 4: An open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'18)*. IEEE, 1–8.

[11] 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*. Springer, 505–518.

[12] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy et al. 2011. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 1–12.

[13] Peng Di, Hui Wu, Jingling Xue, Feng Wang, and Canqun Yang. 2012. Parallelizing SOR for GPGPUs using alternate loop tiling. *Parallel Comput.* 38, 6–7 (2012), 310–328.

[14] Christian Feichtinger, Johannes Habich, Harald Köstler, Ulrich Rüde, and Takayuki Aoki. 2015. Performance modeling and analysis of heterogeneous lattice boltzmann simulations on CPU–GPU clusters. *Parallel Comput.* 46 (2015), 1–13.

[15] Andreas K. Fidjeland, Etienne B. Roesch, Murray P. Shanahan, and Wayne Luk. 2009. NeMo: A platform for neural modelling of spiking neurons using GPUs. In *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors*. IEEE, IEEE, 137–144.

[16] Enrico Franchi. 2012. A domain specific language approach for agent-based social network modeling. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining (ASONAM'12)*. IEEE, 607–612.

[17] Marc-Oliver Gewaltig and Markus Diesmann. 2007. NEST (NEural Simulation Tool). *Scholarpedia* 2, 4 (2007), 1430.

[18] Dan F. M. Goodman and Romain Brette. 2009. The brian simulator. *Front. Neurosci.* 3 (2009), 26.

[19] Qingfeng Guan, Xuan Shi, Miaoqing Huang, and Chenggang Lai. 2016. A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures. *Int. J. Geogr. Info. Sci.* 30, 3 (2016), 494–514.

[20] K. A. Hawick and D. P. Playne. 2013. Simulation software generation using a domain-specific language for partial differential field equations. In *Proceedings of the International Conference on Software Engineering Research and Practice*

(SERP'13). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 69–75.

[21] Moritz Helias, Susanne Kunkel, Gen Masumoto, Jun Igarashi, Jochen Martin Eppler, Shin Ishii, Tomoki Fukai, Abigail Morrison, and Markus Diesmann. 2012. Supercomputers ready for use as discovery machines for neuroscience. *Front. Neuroinform.* 6 (2012), 26.

[22] Suzana Herculano-Houzel. 2012. The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. *Proc. Natl. Acad. Sci. U.S.A.* 109, Supplement 1 (2012), 10661–10668.

[23] Michael L. Hines and Nicholas T. Carnevale. 1997. The NEURON simulation environment. *Neural Comput.* 9, 6 (1997), 1179–1209.

[24] Roger V. Hoang, Devyani Tanna, Laurence C. Jayet Bray, Sergiu M. Dascalu, and Frederick C. Harris Jr. 2013. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Front. Neuroinform.* 7 (2013), 19.

[25] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, ACM, 349–362.

[26] Kaixi Hou, Hao Wang, Wu-chun Feng, Jeffrey S. Vetter, and Seyong Lee. 2018. Highly efficient compensation-based parallelism for wavefront Loops on GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'18)*. IEEE, 276–285.

[27] Tammo Ippen, Jochen M. Eppler, Hans E. Plesser, and Markus Diesmann. 2017. Constructing neuronal network models in massively parallel environments. *Front. Neuroinform.* 11 (2017), 30.

[28] Jakob Jordan, Tammo Ippen, Moritz Helias, Itaru Kitayama, Mitsuhisa Sato, Jun Igarashi, Markus Diesmann, and Susanne Kunkel. 2018. Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Front. Neuroinform.* 12 (2018), 2.

[29] Gordon L. Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John H. Reppy. 2016. Diderot: A domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Trans. Visual. Comput. Graph.* 22, 1 (2016), 867–876.

[30] Alois Knoll and Marc-Oliver Gewaltig. 2016. Neurorobotics: A strategic pillar of the human brain project. *Sci. Robot.* (2016), 25–34.

[31] Susanne Kunkel and Wolfram Schenck. 2017. The nest dry-run mode: Efficient dynamic analysis of neuronal network simulation code. *Front. Neuroinform.* 11 (2017), 40.

[32] Susanne Kunkel, Maximilian Schmidt, Jochen M. Eppler, Hans E. Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann et al. 2014. Spiking network simulation code for petascale computers. *Front. Neuroinform.* 8 (2014), 78.

[33] Christian Lengauer. 1993. Loop parallelization in the polytope model. In *Proceedings of the International Conference on Concurrency Theory*. Springer, Springer, 398–416.

[34] Da Li, Hancheng Wu, and Michela Becchi. 2015. Nested parallelism on GPU: Exploring parallelization templates for irregular loops and recursive computations. In *Proceedings of the International Conference on Parallel Processing (ICPP'15)*. IEEE, IEEE, 979–988.

[35] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. 2015. Heterospark: A heterogeneous CPU/GPU spark platform for machine learning algorithms. In *Proceedings of the International Conference on Networking, Architecture and Storage (NAS'15)*. IEEE, 347–348.

[36] Charl Linssen, Mikkel Elle Lepperød, Jessica Mitchell, Jari Pronold, Jochen Martin Eppler, Chrisitan Keup, Alexander Peyser, Susanne Kunkel, Philipp Weidel, Yannick Nodem, et al. 2018. NEST 2.16.0. (Aug. 2018).

[37] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Netw.* 10, 9 (1997), 1659–1671.

[38] Liam P. Maguire, T. Martin McGinnity, Brendan Glackin, Arfan Ghani, Ammar Belatreche, and Jim Harkin. 2007. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing* 71, 1–3 (2007), 13–29.

[39] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2016. HIPA^{cc}: A domain-specific language and compiler for image processing. *IEEE Trans. Parallel Distrib. Syst.* 27, 1 (2016), 210–224.

[40] Kirill Minkovich, Corey M. Thibeault, Michael John O'Brien, Aleksey Nogin, Youngkwan Cho, and Narayan Srinivasa. 2014. HRLSim: A high performance spiking neural network simulator for GPGPU clusters. *IEEE Trans. Neural Netw. Learn. Syst.* 25, 2 (2014), 316–331.

[41] Abigail Morrison, Ad Aertsen, and Markus Diesmann. 2007. Spike-timing-dependent plasticity in balanced random networks. *Neural Comput.* 19, 6 (2007), 1437–1467.

[42] Quang Anh Pham Nguyen, Philipp Andelfinger, Wentong Cai, and Alois Knoll. 2019. Transitioning spiking neural network simulators to heterogeneous hardware. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* 115–126.

[43] David M. Nicol. 1993. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM* 40, 2 (1993), 304–333.

[44] Dejan Pecevski, Thomas Natschläger, and Klaus Schuch. 2009. PCSIM: A parallel simulation environment for neural circuits fully integrated with Python. *Front. Neuroinform.* 3 (2009), 11.

[45] Wolfram Schenck, A. V. Adinetz, Y. V. Zaytsev, Dirk Pleiter, and A. Morrison. 2014. Performance model for large–scale neural simulations with NEST. In *Proceedings of the Conference for Supercomputing (SC'14).*

[46] Marcel Stimberg, Dan F. M. Goodman, and Thomas Nowotny. 2020. Brian2GeNN: Accelerating spiking neural network simulations with graphics hardware. *Sci. Rep.* 10, 1 (2020), 1–12.

[47] Jan-Phillip Tiesel and Anthony S. Maida. 2009. Using parallel GPU architecture for simulation of planar I/F networks. In *Proceedings of the International Joint Conference on Neural Networks.* IEEE, 3118–3123.

[48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Architect. Code Optimiz.* 9, 4 (2013), 54.

[49] Jiajian Xiao, Philipp Andelfinger, Wentong Cai, Paul Richmond, Alois Knoll, and David Eckhoff. 2020. OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms. *Concurr. Comput.: Pract. Exper.* (2020), 32:e5807.

[50] 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 International Symposium on Distributed Simulation and Real Time Applications (DS-RT'18).* IEEE, 243–252.

[51] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2019. A survey on agent-based simulation using hardware accelerators. *ACM Comput. Surveys* 52, 1 (Jan. 2019), 35.

[52] Esin Yavuz, James Turner, and Thomas Nowotny. 2016. GeNN: A code generation framework for accelerated brain simulations. *Sci. Rep.* 6 (2016), 18854.

Received February 2020; revised June 2020; accepted September 2020