High Performance Simulation of Spiking Neural Network on GPGPUs

Peng Qu, Youhui Zhang, *Member, IEEE*, Xiang Fei, and Weimin Zheng, *Member, IEEE*

Abstract—Spiking neural network (SNN) is the most commonly used computational model for neuroscience and neuromorphic computing communities. It provides more biological reality and possesses the potential to achieve high computational power and energy efficiency. Because existing SNN simulation frameworks on general-purpose graphics processing units (GPGPUs) do not fully consider the biological oriented properties of SNNs, like spike-driven, activity sparsity, etc., they suffer from insufficient parallelism exploration, irregular memory access, and load imbalance. In this article, we propose specific optimization methods to speed up the SNN simulation on GPGPU. First, we propose a fine-grained network representation as a flexible and compact intermediate representation (IR) for SNNs. Second, we propose the cross-population/-projection parallelism exploration to make full use of GPGPU resources. Third, sparsity aware load balance is proposed to deal with the activity sparsity. Finally, we further provide dedicated optimization to support multiple GPGPUs. Accordingly, BSim, a code generation framework for high-performance simulation of SNN on GPGPUs is also proposed. Tests show that, compared to a state-of-the-art GPU-based SNN simulator GeNN, BSim achieves $1.41\times \sim 9.33\times$ speedup for SNNs with different configurations; it outperforms other simulators much more.

Index Terms—Spiking neural network, SNN simulation, GPGPU, load balance, computational neuroscience

1 Introduction

With the rapid development of neuroscience, there is an increasing enthusiasm to create large scale biological neural network models [1], [2], [3]. Currently, numerical simulation has become the third pillar supporting neuroscience research, next to experimental and theoretical approaches [4]. And the most widely used model in the field of neuroscience and neuromorphic computing is Spiking Neural Network (SNN) [5], which introduces higher biological authenticity than traditional neural networks (e.g., deep neural network, DNN). Further, as SNN possesses the potential to achieve high computational power and energy efficiency, it also attracts research interests from the computer system community [6], [7], [8], [9], [10], [11].

An SNN can be described as a directed network, in which neurons connect with each other arbitrarily through synapses (Fig. 1a). The simulation of SNN is a cyclic process that contains thousands of cycles, and in each cycle, we have to update the states of a large number of neurons and synapses according to their mathematic models and deliver millions of spikes. Therefore, it is urgent to improve the performance of SNN simulation on off-the-shelf hardware. It will not only help the neuroscientists to understand the functionality of the brain but also be beneficial to give new architectural hints to the computer and AI communities. However, the biological oriented properties, like spike-driven, sparsity, and long-range connections [12], distinguish SNN from traditional applications such as DNN, graph computing, and physical systems. Thus, existing optimization methods and frameworks cannot be applied to SNN directly.

As a result, quite a few simulators and frameworks for SNN which originate from both the neuroscience communities [13], [14], [15], [16], [17], [18] and the computer communities [19], [20] are proposed. Among them, PyNN [13], a widely supported, simulator-independent language, was proposed in 2008. In PyNN, SNNs are organized as populations (a group of neurons that have similar parameters) and projections (a group of synapses that connect two populations). This high-level, population/projection based description method and its variations have become the mainstream descriptions of SNNs ever since.

Recently, the general-purpose graphic processing unit (GPGPU) has been widely used to speed up a wide range of applications (like DNNs, graph computing, and so on). Thus, many studies [17], [18] also try to involve the computation power of GPGPU to further speed up the simulation of SNN. The prominent one is GeNN [17], which is an optimized code generation framework for efficient and flexible implementation of SNN simulations on CUDA.

In GeNN, each simulation cycle is divided into two separated phases: (1) Neuron computation phase: All the neurons that are not in the refractory period[1] integrate the input currents, update the internal states, and determine whether to issue new spikes. (2) Spike propagation phase: The spikes issued by neurons in the current and the previous cycles are

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology, Beijing 100084, China. E-mail: {qup13, feix16}@mails.tsinghua.edu.cn, zyh02@tsinghua.edu.cn, zwm-dcs@mail.tsinghua.edu.cn.*

Manuscript received 5 Sept. 2019; revised 1 May 2020; accepted 8 May 2020. Date of publication 11 May 2020; date of current version 26 May 2020. (Corresponding author: Youhui Zhang.) Recommended for acceptance by M. Guo. Digital Object Identifier no. 10.1109/TPDS.2020.2994123

1. A neuron in the refractory period will remain inactive regardless of input spikes, namely, perform no calculation. It is a run-time state that is dynamically calculated by the computational model of a neuron.

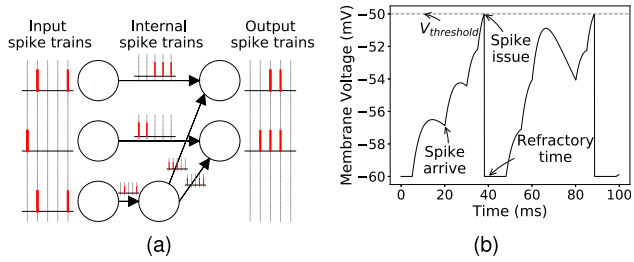

Fig. 1. (a) An example of a simple SNN. The neurons are shown as the nodes while the synapses are expressed as the directed edges. Each neuron will receive a spike train (a spike train is a series of discrete spikes from an individual source) and issue some output spikes. (b) The variation of the membrane voltage of a neuron over time. The membrane voltage increases for a short period after a spike arrives. When the membrane voltage reaches $V_{threshold}$, the neuron will issue a spike and enter the refractory period.

delivered to their target synapses according to their delay properties. Then, the synapses will inject current into their target neurons according to the synapse models. This procedure or its variations are accepted by most simulators and frameworks [15], [17], [18], [21].

Existing SNN simulation systems [14], [15], [17], [19], including GeNN, usually manage and store different populations and projections separately. Although some research [17] may process multiple populations/projections in parallel, they still use separated populations/projections as the basic unit of computing and storage. This management will lead to discontinuous memory access and additional branch operations. Thus, it limits the resource utilization and the overall parallelism. Moreover, during the spike propagation phase, GPGPU-powered SNN simulators usually introduce synapse-based thread mapping methods that ignore the inherent sparsity property of SNN. It will lead to load imbalance and severe limitations on parallelism. Therefore, existing SNN simulators can not make full use of the resources of GPGPU hardware, as illustrated in Table 1.

Another disadvantage of these systems is that they do not support multiple GPUs [15], [17]. Thus, the network size that these systems can simulate is constrained by the memory size of a single GPU.

In this paper, we propose a GPGPU accelerated high performance simulation framework for SNN, and our goal is to make full use of the intrinsic characteristics of SNN. Accordingly, we propose four optimization methods to solve the problems of the existing frameworks. First, we introduce the *fine-grained network representation* to break the population/projection boundaries and optimize the memory access patterns. Then, we propose the *cross-population/-projection parallelism exploration* to improve the degree of computational parallelism across the populations/projections. And this method is also beneficial to memory access patterns. Third, we propose the *sparsity aware load balance* which further exploits parallelism from activity sparsity and reduces the load imbalance. Finally, our work provides dedicated optimization methods to support multiple GPUs.

As a summary, the following contributions are accomplished:

1) We recognize that the parallelism exploration and memory access patterns in existing SNN simulation frameworks significantly limit their performance. Thus, we propose the fine-grained network representation and the cross-population/-projection parallelism exploration to deal with these problems. Besides, the load imbalance among GPU threads caused by activity sparsity also has a considerable impact on the performance. So, we introduce the sparsity aware load balance to further exploit parallelism from activity sparsity and achieve better load balance. We also provide dedicated optimization methods to support multiple GPUs.
2) We provide BSim, a code generation framework that applies all the above techniques. BSim provides a PyNN-compatible description interface as well as a fine-grained network representation of SNN to bridge the gap between the coarse-grained network description and the fine-grained parallelism required by high performance. It also supports multiple GPUs. The source code, tutorials, example projects, and all other related information can be found on the project website https://github.com/CRAFT-THU/BSim.
3) We carry out extensive performance evaluations for SNNs with a set of different configurations. Experimental results on GPGPU show that our implementation can achieve $1.41\times \sim 9.33\times$ speedup over GeNN (a state-of-the-art SNN simulation tool that also supports GPGPU). And compared with other SNN simulators, it outperforms much more. Using more GPUs, the performance is even better.

TABLE 1
Profiling Results of Two Typical GPU-Enabled SNN Simulators: GeNN and Brian2

	GeNN	Brian2
Achieved occupancy (Neuron)	0.123	0.249
Global hit rate (Neuron)	51.82%	9.01%
Achieved occupancy (Synapse)	0.681	0.476
Global hit rate (Synapse)	84.65%	99.61%

The rest of the paper is organized as follows. Section 2 provides background information and related work. Section 3 outlines the framework of BSim. Section 4 describes the concrete techniques of SNN-specific optimization for single and multiple GPUs. Section 5 explains the evaluation methodology and Section 6 presents the detailed experimental results and analyses. Finally, we conclude the paper in Section 7.

2 Background and Related Work

2.1 Spiking Neural Network

SNNs use biologically oriented spikes (i.e. action potentials) to deliver information between neurons connected by synapses. Unlike the signals in traditional computing systems, the presence and timing of an individual spike are both meaningful, thus it can be viewed as a binary value (0/1) with a timestamp. Neuroscientists usually use SNNs to evaluate their theories and reproduce experimental results [22], [23], [24]. Recently, inspired by the success of DNNs, there is an increasing interest in using SNNs for specific tasks [7], [25], [26].

As mentioned in Section 1, the procedure of SNN simulation is usually divided into two phases: neuron computation and spike propagation. And the computation of each phase is defined by the neuron/synapse model separately.

The neuron model describes the dynamic states of the neurons and how to issue spikes. Different mathematical expressions of the states distinguish different neuron models [27], [28], [29], [30]. Among them, the LIF model provides a certain degree of biological authenticity while maintaining limited computational complexity. Therefore, it is widely used by SNN description languages [13], simulators [14], [15], [17], [18], and neuromorphic chips [10], [31], [32].

$$C_m \frac{\mathrm{d}v(t)}{\mathrm{d}t} = -\frac{C_m}{\tau_m}[v(t) - V_{rest}] + i_e(t) + i_i(t) + I_{offset} + I_{injection}(t). \tag{1}$$

The state expression of LIF model is shown in (1). Here, C_m is the neuron capacity, $v(t)$ is the membrane voltage, τ_m is the membrane time constant,[2] V_{rest} is the resistant voltage, $i_e(t)/i_i(t)$ is the excitatory/inhibitory input current, and I_{offset} and $I_{injection}(t)$ are the constant input currents. If the membrane voltage reaches $V_{threshold}$, the neuron will issue a spike, and then its voltage will be held at V_{reset} for a refractory period (τ_{ref}). Once this refractory period ends, the neuron follows this expression again until another spike is issued.

$$\begin{cases} i_e(t) = C_m \sum_{k=1}^{N_E} W_{e,k} S_{e,k}(t) \\ i_i(t) = C_m \sum_{k=1}^{N_I} W_{i,k} S_{i,k}(t) \end{cases}. \tag{2}$$

The synapse model defines how spikes affect the membrane voltage of a post-synapse neuron. In this paper, a widely-used current-based synapse model is adopted, in which synapse weights are directly used as the input currents, as shown in (2) [28]. In (2), $W_{e,k}/W_{i,k}$ is the weight of the kth excitatory/inhibitory synapse, $S_{e,k}/S_{i,k}$ is the corresponding input spike train. Here 'excitatory' means the spikes will increase the target neuron's membrane voltage while 'inhibitory' works just the opposite. Afterward, the synapse will inject $i_e(t)/i_i(t)$ to its target neuron and the states of that neuron will be updated according to (1).

There are various methods to get the parameters of SNN, from biological experiments and handmade models [12], [33] to learning methods [34], [35], [36]. However, unlike DNN, there are no dominant learning procedures for SNN. Recently, inspired by the success of DNN, several researchers have tried to employ deep learning methods to get the parameters of SNNs [36], [37]. And these SNNs are usually referred to as deep spiking neural networks (DSNNs).

To improve portability, simulator-independent languages (e.g., PyNN [13] and SpineML [38]) have been proposed. Among them, PyNN, a domain-specific modeling language in Python, is supported by many SNN simulators [14], [15], [19], [20], [39], as well as neuromorphic chips [31], [32] and FPGA systems [40]. Large-scale SNNs are usually described as collections of populations and projections in software simulators and description languages [13], [15], [38]. Listing 1 gives a detailed SNN sample in PyNN: Two populations are connected by a projection in an all-to-all manner.

2. The membrane time constant measures how quickly a neuron's membrane voltage decays to the resistant voltage.

Listing 1. An SNN sample in PyNN

```
...
# Create two populations and set
  the number and type of the neurons.
pre = Population(1000, sim.IF_curr_exp(params))
post = Population(500, sim.IF_cond_exp(params))
# Create an all-to-all projection between the two
  populations.
  Set the weight and latency.
syn = sim.StaticSynapse(weight=w, delay=d)
pro = Projection(pre, post, sim.AllToAll
Connector(),
    syn, receptor_type='excitatory')
...
```

2.2 SNN Simulation Tools

With the growing interest in building large-scale SNN models [1], [2], [3], many simulation tools have been carried out to help the research communities. There are already a variety of SNN simulators on off-the-shelf hardware, from software simulators [14], [15], [16], [17], [18], [19], [20] to neuromorphic chips [10], [32]. Although neuromorphic chips can provide high performance and energy efficiency, they are rather inconvenient and expensive. Thus, software simulators are currently the main tools [22] for users to develop biologically relevant and accurate models.

Brian2 [15] is an equation based SNN simulator, which uses the SciPy package [41] to optimize mathematical equation expressions. One variety of it, Brian2CUDA, further uses the PyCUDA package to generate CUDA codes for GPGPU. Brain2 supports a variety of neuron/synapse models and connection types. It also provides equation-oriented methods to define new models. GeNN [17] is an optimized code generation framework for efficient SNN simulations on GPGPU. GeNN can update multiple populations and projections in parallel, thus it is much more efficient than other simulators [17]. GeNN mainly integrates the Izhikevich neuron model, but it also provides methods to define customized models. NEST [14] is another widely used SNN simulator that tries to support supercomputers.

2.3 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface (API) for NVIDIA GPGPU. It exposes the underlying hardware abstractions to developers, including a hierarchy of thread groups, shared memories, and barrier synchronization, etc. To fully exploit the high concurrent execution characteristic of the single instruction, multiple thread (SIMT) model of thread groups, a *high degree of parallelism* is conducive. It is also helpful to hide the stalling effects caused by memory accesses. *Memory coalescing* is a key factor in improving global memory access efficiency, that is, when parallel threads that running the same instruction access to consecutive locations in the global memory, the most favorable access pattern is achieved. Besides, *minimizing warp*

divergence is very important to prevent multiple threads in the SIMT mode from following different branches which will lead to sub-optimal performance.

3 System Design

3.1 SNN Simulation Analysis

As SNN has introduced more biological realities, it possesses several biological related properties. The two major and distinguished properties of SNN are *cross-population/-projection parallelism* and *activity sparsity*.

Cross-population/-projection parallelism stems from the fact that we can update the states of different neurons/synapses simultaneously whether they belong to the same population/projection or not. Moreover, one input sample, which is formally some simultaneous spike trains (as illustrated in Fig. 1a), is processed by the neurons of a few successive populations at the same time. That is, even in a pure forward network, when the spike S_i (the ith signal in the spiking sequence) is processed by the first population, the signal caused by S_1 may enter into the ith population. For spike propagation, there are similar phenomena.

Thus, to make full use of the cross-population/-projection parallelism, we should process the neurons/synapses in a fine-grained manner (as individual neurons/synapses) instead of in a coarse-grained manner (as populations/projections). Accordingly, specific optimization is necessary to achieve better memory access patterns and resource occupancy. In Section 4.1 and Section 4.2, we propose the *fine-grained network representation* to break the population/projection boundaries and optimize the memory access patterns, as well as the *cross-population/-projection parallelism exploration* to make full use of the cross-population/-projection parallelism.

Activity sparsity originates from the event-driven behavior of SNN. As we only need to update the states of the neurons/synapses that have received currents/spikes, the major operations in both phases are sparse. Besides, the refractory period of the neuron model further complicates this situation. For example, a fired LIF neuron will enter a refractory period and will remain inactive for a specified period (as shown in Fig. 1b). Thus, only a small fraction of the neurons are active in a simulation cycle.[3]

It means that we must process active neurons/synapses (which are not in the refractory period and have received spikes) and inactive neurons/synapses in different manners to avoid unnecessary computation. But this method may lead to load imbalance and resource underutilization on GPGPU. Thus, we propose another optimization named *sparsity aware load balance* to deal with this issue in Section 4.3.

3.2 Code Generation Workflow

Generally, compared with SNN simulators that based on predefined libraries [18], [19], code generation frameworks [17], [39] can achieve higher performance and support custom models more easily. Thus, in this paper, we propose the BSim, a high performance code generation framework for SNN simulation, which implements the core features of PyNN (such as LIF model, one-to-one connection, all-to-all connection, spike-timing-dependent plasticity and so on). BSim also provides the extension ability for users to define new models and connection methods.

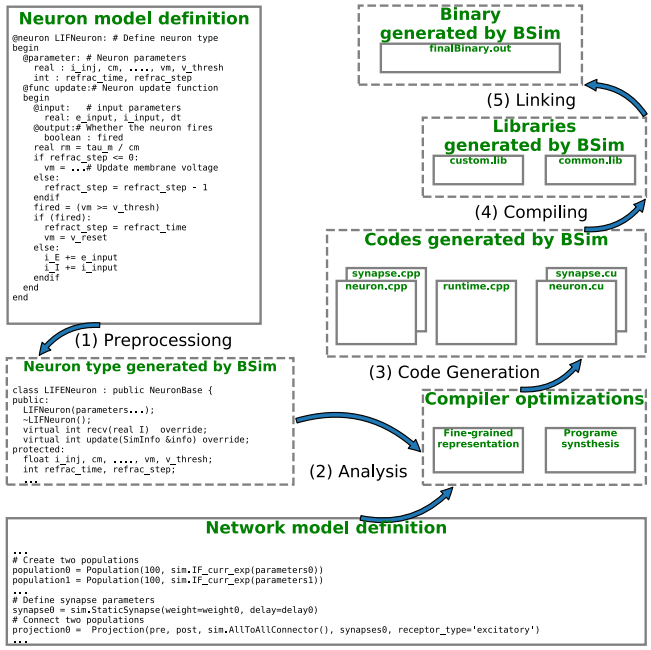

Fig. 2. The workflow of BSim.

BSim consumes the user-provided SNN description (custom models and network description), generates the corresponding fine-grained intermediate representation, C++ and CUDA codes, and finally produces an executable binary file. This procedure is composed of five phases: preprocessing, analysis, code generation, compiling, and linking (Fig. 2).

1) *Preprocessing*. If the network contains user-defined models, these models will be first transformed into corresponding C++ classes.
2) *Analysis*. The network description in PyNN is represented as the fine-grained network representation. Several optimization methods, including parameter sharing, are applied here.
3) *Code Generation*. BSim then converts the numerical expressions, neural/synapse update function(s), and other descriptions to their equivalents in C++ and CUDA. In this phase, the intermediate representation is also converted to C++ structures. Besides the cross-population/-projection parallelism exploration and the sparsity aware load balance, we further refer to program synthesis methods (e.g., parameter folding & propagation) to simplify the computation model and reduce memory consumption.
4) *Compiling*. The C++ and CUDA codes generated in previous phases are compiled by general compilers (e.g., GCC/NVCC) to generate a dedicated simulation library. BSim also provides precompiled common libraries that implement the standard models in PyNN.
5) *Linking*. The dedicated simulation library and common libraries are linked together to produce the final binary file.

3. Other types of neuron model also have the similar phenomenon of inactivity [18], [30], which does coincide with the actual neurobiological observation.

3.3 Simulation Workflow

In BSim, each simulation cycle is divided into two phases: neuron computation and spike propagation.

Neuron Computation. Unlike other simulators, BSim uses neurons/synapses as its basic processing units. During this phase, the cross-population parallelism exploration is introduced to make sure that all the neurons are processed simultaneously regardless of whether they belong to the same population or not. And sparsity aware load balance for neuron computation is applied to deal with the load imbalance and warp-divergence caused by the refractory period.

Spike Propagation. During this phase, we only process the spikes which reach their target synapses according to the delay property. And only the synapses that receive spikes will inject currents to their target neurons. Similar to the neuron computation phase, the cross-projection parallelism exploration and the sparsity aware load balance for spike propagation are exploited to avoid load imbalance and underutilization.

4 Optimization Methods

4.1 Fine-grained Network Representation

A typical SNN contains two types of data: neuron/synapse parameters and network topology. Neuron/synapse parameters are the parameters defined by neuron/synapse models, e.g., the membrane voltage. And these parameters will be accessed during the neuron computation and/or spike propagation phases. The network topology is the connection relationship between neurons and synapses, and it is usually used to guide the spike propagation. Here we focus on the network topology while the parameters are dealt with later (Section 4.2.1). For simplicity, in the rest of the paper, we will refer to the neuron/synapse/population/projection whose identification (ID) is i as $N_i/S_i/P_i/J_i$ respectively.

4.1.1 From Coarse-Grain to Fine-Grain

To bridge the gap between the coarse-grained network description and the fine-grained parallelism, we represent the entire network as a fine-grained, neuron/synapse-based description.

First of all, we break down the population-based description (Fig. 3a) into the neuron-level (Fig. 3b) by re-identifying the neurons. As the neurons of the same type usually have similar parameters and memory access patterns, we re-identify such neurons continuously and store their parameters contiguously. Moreover, the neurons belong to the same population are also re-identified continuously. Because neurons in the same populations usually have similar parameters and activities, putting them together is good for memory coalescing and minimizing warp divergence. For example, N_i and N_{i+1} in P_i are re-identified as N_k and N_{k+1} while N_{NUM_i-1} in P_i and N_0 in P_{i+1} are re-identified as N_m and N_{m+1}, assuming that NUM_i is the number of neurons in P_i, $m = k - i + NUM_i - 1$, and $P_i$and P_{i+1} have the same type.

Then, we reorganize synapses based on their source neurons. The synapses, whose source neuron and delay property are the same, are also re-identified continuously and their parameters are stored contiguously, too. Delay is a special parameter of the synapse models which defines how long it takes to deliver the spike from the source neuron to the target synapse. The synapses that have the same source neuron and same delay property will receive the spikes at the same time and be processed simultaneously. Therefore, their parameters are also accessed simultaneously, which is beneficial to maximize memory coalescence. This strategy is referred to as "sender-first" in the following contents.

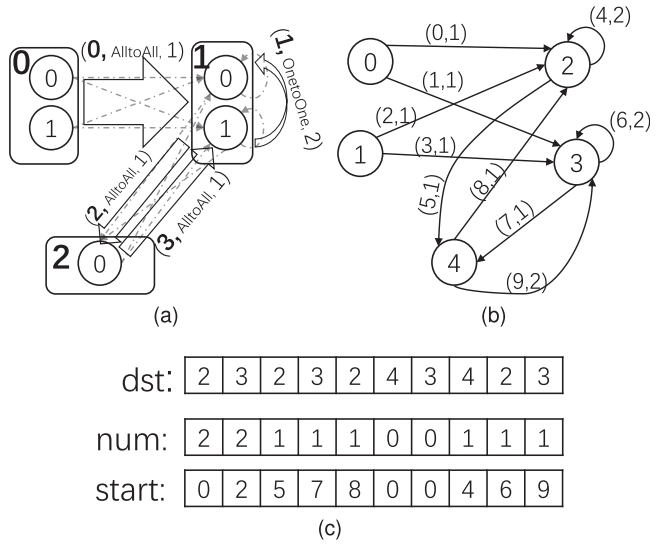

Fig. 3. (a) Population/projection based description of SNN. The populations and the neurons are shown as rectangles and circles respectively. And the numbers inside them are their IDs. The projections are described as arrows and the 3-tuples on them are their IDs, connection types, and delay properties respectively. The thick hollow arrows are the projections described by the users while the dashed arrows are the actual synapse connections. (b) Neuron/synapse based description. It is obtained by re-identifying all the neurons and synapses. ((b) is equivalent to (a)). The neurons are shown as circles and the numbers inside them are their IDs. The synapses are described as arrows and the 2-tuples on them are their IDs and delay properties respectively. (c) Fine-grained network representation (equivalent to (a) and (b)).

Third, we rearrange neuron/synapse parameters according to their new IDs (Section 4.2.1).

Finally, we refer to the *compressed sparse row (CSR)* [42] representation which is widely used for managing sparse structures (like a graph or a sparse matrix) to express the network topology efficiently(Fig. 3c). As the delay property of synapses makes the SNN different from a graph or a sparse matrix, we further integrate the delay information into the CSR representation and propose the compact network representation.

4.1.2 Compact Network Representation

In the fine-grained network representation, the network topology is stored as three compact arrays. The connections from synapses to neurons are organized as an array named "dst", and the reversed ones (i.e. from neurons to synapses) are organized as two arrays named "start" and "num" separately.

The "dst" array stores the IDs of all the target neurons, indexed by the IDs of the synapses. If the value of $dst[i]$ is j, it means that the target neuron of S_i is N_j. The "start" array stores the minimal ID of all the synapses that have the same source neuron and the same delay property, and the "num" array provides the number of these synapses. As the synapses which have the same source neuron and the same delay are re-identified continuously, we can easily access all the

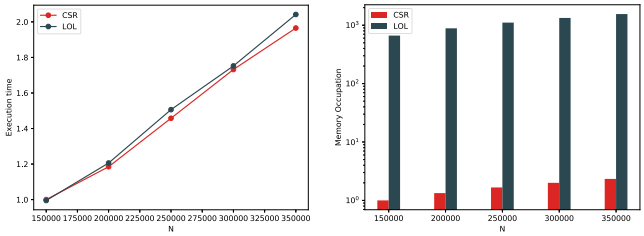

Fig. 4. (a) Performance of different representations. (b) Memory occupation of different representations.

synapses based on these two values. Assuming the number of all the neurons is M_N and the minimum delay of all the synapse is D_{min}. If the value of $start[i * M_N \times (d - D_{min})]$ is j and that of $num[i * M_N \times (d - D_{min})]$ is k, it means that the target synapses of N_i, whose delay properties are d, are $S_j, S_{j+1}, ...,$ and S_{j+k-1}.

Fig. 3 shows a network of three populations (five neurons) and four projections (ten synapses) alongside with its corresponding fine-grained representations. For Fig. 3c, M_N is 5 and D is 1. As $num[0 + M_N * (1 - D_{min})]$ is 2 and $start[0 + M_N * (1 - D_{min})]$ is 0, it means N_0 connects to S_0 and S_1, and the delay properties of S_0 and S_1 are both 1. Similarly, as $dst[0]$ is 2 and $dst[1]$ is 3, we can find out that the targets of S_0 and S_1 are N_2 and N_3 respectively.

4.1.3 Advantage Analysis

Besides the CSR representation, there are also several other representations for sparse matrix [43]. As we have to propagate the spikes to all the destination synapses of the fired neurons, the representation used here should be row-based. Therefore, column-based or block-based representations, such as compressed sparse column (CSC) and compressed sparse blocks (CSB), are not suitable for the SNN. The most widely used row-based representations for the sparse matrix are linked lists, list of lists (LOL), and CSR. Among them, linked lists are not suitable for parallel processing, so here we only compare the performance of LOL and CSR. As shown in Fig. 4, CSR-based representation achieves better performance and far less memory occupation as it only stores the minimal IDs of the synapses that have the same source neuron and the same delay property instead of storing all the IDs of these synapses.

Moreover, as the fine-grained network representation stores the delay information together with the neuron-to-synapse connection, we do not need to store the delay information explicitly. Thus, the space complexity of this format is only $O(D_{max} \times M_N + M_S)$, where D_{max} is the maximum delay of all synapse, M_N is the number of all the neurons and M_S is the number of all the synapses. It is much less than the representation used in some previous studies [17], [18] without this mechanism. As GeNN [17] and Brian2 [15] still use the populations/projections as their basic storage unit, they require additional memory to deal with the "delay" information. What's worse, Brain2 further needs to store the firing information of synapses. Tests show that during the simulation, GeNN consumes 1.08× GPU memory than ours, and Brian2 consumes 2.25×.

Sender-first rule has more advantages. As we propagate the spikes to their target synapses in parallel, this rule is beneficial for better memory coalescence. Moreover, synapses with the same target neuron will usually use the *atomic add* operation to inject currents to the latter. Thus, it is beneficial to distribute them into different CUDA blocks to avoid access conflicts. The sender-first rule just coincides with this case: It organizes all the synapses with the same source neuron together, which means it will arrange synapses with the same target neuron separately (different synapses cannot have both the same source neuron and target neuron).

4.2 Cross-Population/-Projection Parallelism Exploration

Existing SNN simulators usually use the population as their basic computation unit and process different populations sequentially or in parallel, which will lead to limited parallelism, discontinuous memory access, and/or warp divergence.

For example, Listing 2 outlines a typical CUDA sample code from GeNN [17] in which multiple populations are processed in parallel. In line 3, an 'if' statement is used to distinguish different populations; it introduces the additional branch operation and may raise the risk of warp divergence. In line 6, as the parameters in different populations are still stored separately, it is difficult to coalesce the memory access. For instance, assuming the number of neurons in P_0 is M_0, thus the memory accesses of $p[0]- > vm[M_0 - 1]$ and $p[1]- > vm[0]$ cannot coalesce. Especially, if two nearby neurons have different types, the situation will be even worse.

Listing 2. Update Populations in Parallel

```
1: __device__ update_populations
   (Population **p) {
2:     int id = blockIdx.x * blockDim.x + threadIdx.x;
3:     if(isPopulation1(id)) { (*@
4:         int off = getOffsetInPopulation1(id);
5:         ...
6:         // Update the membrane voltage.
7:         p[1]->vm[off] = updateVm(p[1]->vm
           [off], p[1]->Cm[off], p[1]->C_E[off], p
           [1]->C_I[off], p[1]->i_E[off], p[1]-
           >i_I[off]);
8:         ...
9:     } else if (isPopulation2(id)) { ...
10:    } else if (isPopulation3(id)) { ...
11:    } ...
12: }
```

To avoid these disadvantages, we first rearrange the organization of the parameters to get better memory access patterns and then introduce fine-grained parallelism exploration to achieve higher occupancy.

4.2.1 Fine-Grained Parameter Organization

Beyond the network topology, we further reorganize the parameters of neurons and synapses to achieve better memory access patterns. Each type of neurons is organized as several arrays, each of which stores one type of parameter. As the neurons of the same type and in the same population have been identified continuously, all the values of the same parameter are stored contiguously in the array. And it

is convenient to access these parameters through the IDs of neurons. Here, the structure of arrays (SOA) is used instead of the array of structures (AOS), as the former matches GPGPU memory structure better. The synapse parameters are organized roughly the same, while they further obey the "sender-first" rule, in which the parameters of the synapses that have the same source neuron and the same delay properties are stored contiguously.

4.2.2 Fine-Grained Parallelism Exploration

Besides the fine-grained parameter organization, we further introduce the fine-grained parallelism exploration method to achieve higher parallelism and higher resource occupancy.

Listing 3 outlines the fine-grained parallelism exploration method which uses neurons as the basic computation units instead of populations. As we can access the parameters of neurons through their IDs directly (line 4), there is no need to distinguish which population a neuron belongs to (Listing 2, line 3) and the disadvantages introduced by the 'if' statement are avoided. Further, as all the neuron parameters of the same type are stored contiguously, it is easy to coalesce the memory accesses. For instance, the memory access of $n- > vm[M_0 - 1]$ and $n- > vm[M_0]$ (corresponding to $p[0]- > vm[M_0 - 1]$ and $p[1]- > vm[0]$ in Listing 2) are coalesced.

Listing 3. Cross-Population Parallelism Exploration

```
1: __device__ update_neurons(Neurons *n) {
2:     int id = blockIdx.x * blockDim.x + threadIdx.x;
3:     ...
4:     // Update the membrane voltage.
5:     n->vm[id]=updateVm(n->vm[id], n->Cm
       [id], n->C_E[id], n->C_I[id], n->i_E
       [id], n->i_I[id]);
6:     ...
```

Thus, combined with the fine-grained network representation, the fine-grained parallelism exploration can fully exploit the cross-population parallelism and coalesce memory accesses. As the cross-projection parallelism just follows the same computing scheme, these optimization methods can be applied to spike propagation directly.

Although the generation of the fine-grained network representation and parameters might take a certain amount of time, it only takes place once. Therefore, considering that the simulation procedure may occur thousands or millions of times, the overhead of the optimization can be ignored.

4.3 Sparsity Aware Load Balance

4.3.1 Sparsity Aware Load Balance for Neuron Computation

Typically, neurons in the refractory period perform much less computation than others. As the refractory period may last several simulation cycles, the total number of neurons in refractory time cannot be ignored. Thus, traditional processing methods that deal with all the neurons equally may lead to considerable load imbalance.

To address this issue, we propose sparsity aware load balance for neuron computation. During the neuron computation phase, we first process all the neurons in parallel and distinguish between the active neurons (neurons that are not in refractory period) and refractory neurons; Then, we place those active neurons into an active queue (as shown in Listing 4); Finally, we carry out the computation of all these active neurons in parallel.

Using this method, we can not only avoid the computation of inactive neurons but also minimize the load imbalance. Moreover, although the number of active neurons is less than the total number of neurons, the cross-population parallelism exploration method mentioned before can still provide adequate parallelism for neuron computation.

Listing 4. Sparsity Aware Load Balance for Neuron Computation

```
1: __device__ enqueue_active_neurons
   (Neurons *n) {
2:   int id = blockIdx.x * blockDim.x + threadIdx.x;
3:   if (is_active(n[id])) {
4:     thread_safe_enqueue(active_queue, id);
5:   } else {
6:     update_refractory_period(n[id]);
7:   }
8: }
```

4.3.2 Sparsity Aware Load Balance for Spike Propagation

Spike propagation inherits the sparsity of neurons, as only the post-synapses of a fired neuron will receive spikes. A widely-used processing method is mapping each fired/target neuron to a separate thread, which then loops through all the outgoing/incoming spikes [17]. Another method is mapping each projection to several blocks and then looping through the fired source neurons of this projection and processing the spikes in parallel across these blocks.

However, both methods have significant disadvantages. The parallelism of the former is limited to the number of fired/target neurons. If the firing rate decreases, the overall parallelism will be significantly affected. In the latter case, considering that different projections may have different numbers of fired source neurons and the fired neurons may have different numbers of target synapses, this method will lead to serious load imbalance between and inside blocks.

Therefore, we propose the sparsity award load balance method to solve these problems. Owing to the fine-grained network representation, we can propagate the spikes at the neuron/synapse level. First, we deploy the fired neurons among CUDA blocks equally. Then the threads in each block will loop through these neurons and propagate the corresponding spikes to target synapses in parallel (as shown in Listing 5).

In this way, we can exploit two types of parallelism: different fired neurons and different synapses of one fired neuron. Thus, even when the number of fired/target neurons is small, we can still exploit the parallelism of the synapses of these neurons. As the threads in each block will process similar numbers of fired neurons, we also achieve a better load balance between blocks. Furthermore, as the spikes are propagated in parallel inside a block instead of cross several blocks, thus the load imbalance among these threads is also

alleviated. Subsequent experiment results show that the advantages of our implementation are more pronounced while the firing rate decreases.

Listing 5. Sparsity Aware Load Balance for Spike Propagation

```
__device__ spike_propagation(Neurons *n) {
  int neurons_per_block = ceil(fired_neurons/
  gridDim.x);
 for (int i=0; i<neurons_per_block; i++) {
  int nid = fired_neuron_ids[blockIdx.x*neuro-
  ns_per_block+i];

  int start = get_synapse_id(nid);
  int num = get_synapse_num(nid);

  for (int j=threadIdx.x; j<num; j += blockDim.
  x) {
    int sid = start + j;
    process_synapse(sid);
  }
 }
}
```

4.4 Multiple GPU Optimization

A noticeable drawback of GPGPU acceleration is that it will limit the size of the network, as the GPU memory is usually much smaller than the host memory. An intuitive solution is to support multiple GPUs. To achieve this goal, we first partition the entire network into several sub-networks; then each sub-network is transformed into the fine-grained network representation separately and the latter is distributed to each GPU and executed simultaneously afterward.

Moreover, it is necessary to consider several problems related to multiple GPUs, namely, communication and load balance.

4.4.1 Communication Optimization

During the spike propagation phase, spikes issued by fired neurons should be delivered to all the target synapses. Thus if a neuron and some of its target synapses are partitioned into different sub-networks, we have to deliver spikes to all the synapses using cross-GPUs communication. As a neuron usually connects to hundreds of synapses, the cost is considerable. In this case, if we can just deliver the information of the fired neurons to the target GPU and then complete the spike delivery on that GPU locally, it will significantly reduce the amount of communication.

To achieve this, shadow neurons, which are similar to the ghost vertexes in graph computation [44], are introduced to reduce the total amount of communication data. A shadow neuron is a dummy representation of the real neuron whose target synapses have been partitioned into other sub-networks. During the neuron computation phase, when a neuron fires, all of its shadow neurons will also be marked as fired.

4.4.2 Load Balance

During the spike propagation phase, fine-grained synchronizations among synapses that have the same target neuron (e.g., atomic operation on the input current of a dedicated neuron) are needed to avoid conflicts. Thus, partitioning the synapses together with their target neurons will significantly improve the overall performance. Further, as the number of the synapses is usually much larger than that of the neurons, we focus on balancing the number of synapses to achieve load balance.

When partitioning, we gradually assign neuron populations into multiple sub-networks until the number of the synapses of the sub-networks is approximately equal.

5 Evaluation Methodology

The performance of SNN simulation is affected by several factors, mainly including:

- *Network size*, namely the number of the neurons, is the most straightforward and important factor that affects the performance of SNN simulation. That's because the internal states of all neurons have to be updated in every simulation cycle and the network size determines the amount of computation during the neuron computation phase directly. When other factors remain the same, more neurons mean more synapses. And more spikes will be issued which will further affect the performance during the spike propagation phases. Besides, network size will also affect the memory occupation and memory access patterns. In one word, the larger network size, the more simulation time.
- *Firing rate* refers to the average number of times a neuron fires in a second. Assuming that a network with N neurons fires F times over the time interval T, then, its firing rate is defined as $f = F/N/T$. It is the main factor that affects the degree of sparsity of the entire network. A low firing rate means that only a small number of neurons fires within a certain period. The firing rates of typical biological SNNs are lower than 100 Hz which is usually regarded as the upper bound [45], [46], [47], [48]. And the firing rates of DSNNs are usually between 200 Hz and 1000 Hz [7], [49], [50].
- *Network connectivity* can be viewed as the number of synapses that a neuron connects to on average. It determines how many spikes need to be propagated when a neuron fires. Beyond that, it also affects the parallelism during the spike propagation phase, as all the spikes of a fired neuron can be propagated to the target synapses simultaneously.
- *Inhibition ratio* is the ratio of the number of inhibitory neurons to the number of all the neurons. The spikes issued by non-inhibitory neurons have a fundamentally different effect from those issued by inhibitory neurons. The former may encourage the target neurons to issue spikes while the later will prevent them from issuing spikes. Thus, networks with different inhibition ratios may have distinct activities.
- *Neuron model* defines how to update the internal states of the neurons. It determines the complexity of the neuron computation directly. Moreover, as neuron computation can be evaluated in parallel and

TABLE 2
Detailed Configurations of the Platforms

	Machine for Single GPU	Machine for Multiple GPUs
CPU	Intel Xeon E5-2680 v4@2.40 GHz	Intel Xeon E5-2620 v4@2.10 GHz
GPU	NVIDIA Tesla P100 3584 CUDA Cores@1.33 GHz	4x NVIDIA Tesla V100 5120 CUDA Cores@1.53 GHz
Memory	64 GB	256 GB
GPU Memory	12 GB	4 x 16 GB

independently by separate CUDA threads, a more complex neuron model is relatively more beneficial to the GPU. In this paper, we focus on the LIF model while the optimization methods can also be applied to other models. For neural models with higher computational complexity, a higher speedup can be expected, which is confirmed by [17].

Other factors may also affect the performance, such as the network topology, the number of synapses, and so on. But these factors indirectly take effect through the previous five factors (for instance, network topology affects the performance by changing the firing rate and network connectivity). Thus, we mainly focus on the impacts of network size, firing rate, network connectivity, and inhibition ratio. From now on, we will describe a dedicated SNN as a 4-tuple (N, FR, K, R), in which N, FR, K, R mean the network size, firing rate, network connectivity, and inhibition ratio respectively.

To measure the performance of SNN simulation, we refer to the SNN benchmark in [22], which is a CUBA IF Network [51], to build our benchmark. The SNN benchmark used in our experiment consists of many populations and the neurons in one population have similar parameters and the same excitatory/inhibitory type. It is a forward network along with several long-range projections: About 80 percent of all the populations only connect with their front and rear populations, and the rests also connect with remote populations. Other factors, such as the number of populations, the average number of neurons in one population, the firing rate, network connectivity, and the inhibitory ratio, are all configurable. During our experiment, the number of neurons in one population is fixed to 1000. The simulation step is set to 0.1ms and the total duration of the simulation is 1000 cycles. The number of populations, the overall firing rate, network connectivity, and inhibition ratio are modified separately to test the impacts of these factors respectively. The default configuration of the SNNs is (N = 375000, FR = 100 Hz, K = 1000, R = 20%).

We use two different platforms to carry out the experiments. One platform is an X86 server for single GPU evaluation, and the other is an NVIDIA DGX-1 server for multiple GPU tests (DGX-1 provides NVLink connections between every pair of GPUs). Detailed configurations are presented in Table 2. We use GeNN [17] as the baseline, because it is the state-of-the-art SNN simulator and it also employs code generation and GPGPU. As far as we know, now GeNN is the fastest SNN simulator on GPGPU (as shown in the following evaluation). By the way, as GeNN does not integrate the LIF model, we use Brian2GeNN to convert the LIF model in Brain2 [15] into its counterpart in GeNN before testing.[4] GeNN does not support multiple GPUs [17], thus, we only test the scalability of BSim with multiple GPUs.

6 Experimental Results

6.1 Correctness Verification

We use two different methods to verify the correctness of BSim. First, we refer to the CUBA example on Brian2's website[5] and compare the detailed results of BSim and Brian2. Testing results show that the values of the membrane voltage of each neuron in BSim and Brain2 are the same at every timestep. Second, during the following experiments, we compare the firing rate of each neuron in BSim with that in GeNN, and they are also identical.

6.2 Comparison With GeNN and Other Simulators

We have tested quite a few networks with different scales, from 150000 neurons to 375000 neurons. For the network which has the largest scale, we further evaluate the impacts of different network connectivity and inhibition ratio. Furthermore, by changing the synapse weights, such a large network with four different firing rates (2000 Hz, 500 Hz, 100 Hz, 50 Hz) are also tested. The firing rates of 500 Hz, 100 Hz, and 50 Hz can be viewed as the typical firing rates of different types of SNNs (as mentioned in Section 5) and the case of 2000Hz, which rarely happens in reality, can be viewed as a stress test. In each test case, we use the default values for other factors.

Figs. 5a, 5b, 5c, and 5d show the speedup achieved by BSim over GeNN for different network sizes, firing rates, network connectivity, and inhibition ratios respectively. As the exact execution time of SNN simulation is determined by how much cycles the user chooses to simulate the network directly, here we focus on the relative speedup.

Fig. 5a evaluates the impacts of different network sizes. The configuration of the network used here is $(N \in [150000, 375000], FR = 100$ Hz$, K = 1000, R = 20\%)$. From this figure, we can find out that BSim outperforms GeNN regardless of the network size. When the network size increases, BSim achieves almost linear speedup over GeNN. That is because when the network size increases, BSim can extract higher fine-grained parallelism.

Fig. 5b evaluates the impacts of different firing rates. The configuration of the network used here is $(N = 375000, FR \in \{50$ Hz$, 100$ Hz$, 500$ Hz$, 2000$ Hz$\}, K = 1000, R = 20\%)$. From this figure, it shows that BSim achieves higher speedup when the firing rate decreases. It means that the effect of the sparsity aware load balance optimization is remarkable: The sparser the activity, the higher the speedup. Moreover, as mentioned in Section 5, the firing rates used in our tests are higher than the typical firing rates of SNNs, which means that BSim can achieve better performance for usual SNNs.

Fig. 5c evaluates the impacts of different network connectivity. The configuration of the network used here is $(N = 375000, FR = 100$ Hz$, K \in [200, 1000], R = 20\%)$. From this figure, we can find out that BSim achieves a higher speedup

4. Test results show that the slow down introduced by Brian2GeNN is less than 5 percent.

5. https://brian2.readthedocs.io/en/stable/examples/CUBA.html

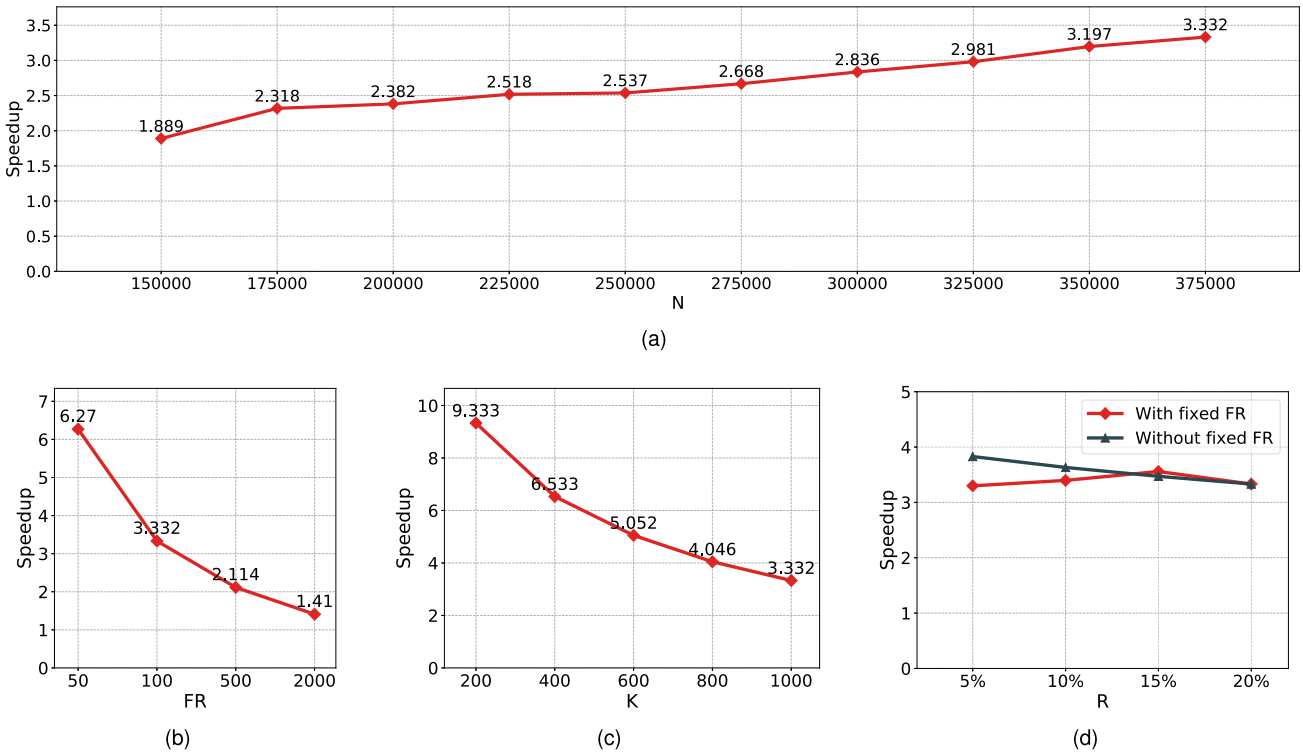

Fig. 5. Speedup of BSim over GeNN with different network configurations. (a) $(N \in [150000, 375000], FR = 100$ Hz$, K = 1000, R = 20\%)$. (b) $(N = 375000, FR \in \{50$ Hz$, 100$ Hz$, 500$ Hz$, 2000$ Hz$\}, K = 1000, R = 20\%)$. (c) $(N = 375000, FR = 100$ Hz$, K \in [200, 1000], R = 20\%)$. (d) $(N = 375000, FR = 100$ Hz$, K = 1000, R \in \{5\%, 10\%, 15\%, 20\%\})$.

when the connectivity decreases. That's because the activities also become sparser in this case, which is similar to that in Fig. 5b.

Fig. 5d evaluates the impacts of different inhibition ratios. The results of both the fixed firing rates and varying firing rates are provided. The configuration of the network used here is $(N = 375000, FR = 100Hz, K = 1000, R \in \{5\%, 10\%, 15\%, 20\%\})$. From this figure, we can find out that the inhibition ratio does not affect the speedup much. That's because when the firing rate is fixed, the number of active neurons is also roughly fixed. Thus, BSim may not exploit further optimization chances, although different inhibition ratios may lead to different activity patterns. When the firing rate varies, the higher inhibition ratio leads to more irregular activity patterns and lower firing rates, which affects the overall speedup to some extent. Our experimental results show that the changing inhibition ratio has a really limited impact on the overall firing rate. But it will lead to dissimilar firing patterns of the neurons, which will further cause discontiguous memory access. On the other hand, as BSim has also explored the activity sparsity of neuron computation, the overall impact is still limited.

From the previous results, we can find out that BSim outperforms GeNN regardless of the detailed configurations of SNNs. When the scale of the network increases, BSim achieves a higher speedup. Besides, the lower the firing rate or the sparser the connectivity, the higher the speedup. Moreover, BSim outperforms GeNN whatever the inhibition ratio is. That is because our implementation exploits the inherent sparsity of neuron computation and spike propagation effectively. Thanks to cross-population/-projection parallelism, even for small networks with high firing rates, the speedup over GeNN is still considerable. Further, in these experiments, we configure the SNN benchmark with relatively high firing rates and dense connectivity, which are not friendly with our optimization methods. Considering that normal SNNs are sparser, the performance of BSim will be even better.

From the perspective of GPU memory consumption, owing to the fine-grained network representation, BSim consumes less GPU memory than GeNN and Brian2 (Table 3). All the results are normalized and BSim is used as the baseline.

We further compare our implementation with some other widely used simulators (Table 4). The configuration of the network used here is $(\mathrm{N} = 375000, \mathrm{FR} = 100Hz, \mathrm{K} = 1000, \mathrm{R} = 20\%)$. For NEST and NEURON, as they don't support GPGPU, we test them on an X86 server with four Intel Xeon E5-2620 CPUs (24 cores).[6]

From this table, we can see that BSim is much faster than others and GeNN is the most efficient one among these existing simulators if BSim is excluded.

TABLE 3
Relative GPU Memory Consumption

	BSim	GeNN	Brian2
GPU memory	1.00	1.08	2.25

6. There is an unofficial GPGPU-based NEURON accelerator, named CoreNEURON (https://github.com/nrnhines/ringtest), which is about 8× faster than NEURON.

TABLE 4
Speedup of BSim Over Other Simulators

	BSim over GENN	BSim over Brian2	BSim over NEST (CPU)	BSim over NEURON (CPU)
Speedup	6.12	32.74	51.82	193.89

6.3 Evaluation of Optimization Methods

6.3.1 Single GPU Optimization Evaluation

To further analyze the performance enhancement, we examine the impacts of different optimization techniques (including the cross-population/-projection parallelism exploration, the sparsity aware load balance, and the optimization for multiple GPUs) on the neuron computation and spike propagation phases separately. We also use nvprof to collect the profiling data of BSim and GeNN, including the average number of instructions executed per cycle (*IPC*), the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor (*Achieved Occupancy (AO)*), the hit rate for global loads in unified L1/TEX cache (*Global Hit Rate (GHR)*) and the ratio of the average active threads per warp to the maximum number of threads per warp (*Warp Execution Efficiency (WEE)*). Higher *IPC* indicates more efficient use of all the available resources, the *AO* is used to measure whether there is sufficient and efficient parallelism to hide latency, the *GHR* is used to evaluate the efficiency of memory access pattern and the *WEE* can tell the effect of warp-divergence. All of the profiling results are higher the better.

Figs. 6a, 6b, and 6c show the speedup of each optimization method for the network whose configuration is $(\mathrm{N} = 375000,\ \mathrm{FR} = 100\ \mathrm{Hz},\ \mathrm{K} = 1000,\ \mathrm{R} = 20\%)$. Figs. 6d and 6e show the corresponding profiling results.

Accordingly, we can see that the cross-population/-projection parallelism exploration speeds up the neuron computation significantly, but has no obvious effect on spike propagation. On the contrary, the sparsity aware load balance optimization accelerates spike propagation very much but it is not important to neuron computation.

The reason is that the former can fully explore the parallelism and coalesce memory access during the neuron computation phase, as shown in Fig. 6d, it achieves considerably better *IPC*, *AO* and *GHR*. Thus, it can significantly speed up the neuron computation phase. But for the spike propagation phase, if we only use the cross-projection parallel optimization, the performance may be worse than that of GeNN. As shown in Fig. 6e, although other profiling results are slightly better, the *AO* is much lower. That is because the activity sparsity property can lead to limited parallelism and considerable load imbalance which outweigh the gain from better memory access pattern (which manifested as the low *AO*). As mentioned in Section 4.3.2, the sparsity aware load balance for spike propagation will solve this problem, thus the combined result is much better. All the profiling results with sparsity aware load balance is better than GeNN in this case (Fig. 6e). Moreover, although the *GHR* and *WEE* are only slightly better, it can still achieve much better *IPC* due to better load balance.

Comparing with GeNN, we can figure out that the efficiency of BSim stems from the more sufficient exploration of parallelism and the better memory access patterns (Figs. 6d and 6e). Less warp divergence is also beneficial. Accordingly, during the neuron computation phase, BSim gets higher *IPC*, *achieved occupancy*, *global hit rate* and *warp execution efficiency*, which means that it has advantages in every way. For spike propagation phases, the situation is similar.

From Figs. 6a and 6c, we may find out that sparsity aware load balance for neurons affects the performance a little bit. That is because the cross-population parallelism optimization method has sped up the neuron computation significantly, thus the gain introduced by sparsity exploration for neurons is rather limited (as shown in Fig. 6d). It is even less than the cost, i.e. the overhead of constructing the active neuron queue (Section 4.3.1). However, this does not mean that sparsity aware load balance for neuron computation is useless. If the time spent for neuron computation increases (e.g., larger network size, the more complex neuron models or the less powerful hardware are used), this optimization method will achieve considerable speedup.

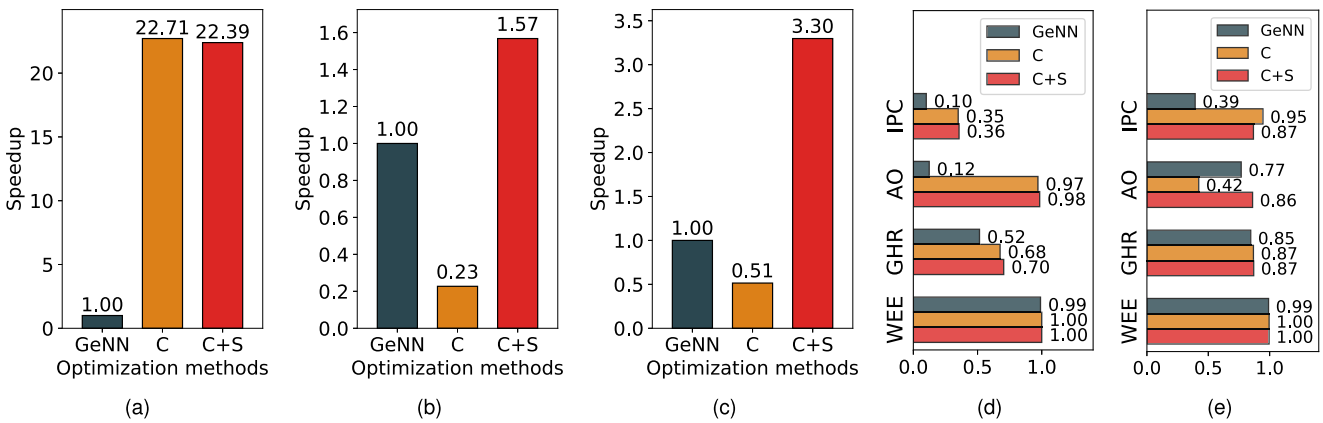

Fig. 6. (a)-(c) Impact of different optimization techniques for different computation phases: (a) Neuron computation, (b) Spike propagation, (c) Overall. *C* stands for *cross-population/-projection parallelism exploration*, *S* stands for *sparsity aware load balance*. (d)–(e) The profiling results for different computation phases: (d) Neuron computation, (e) Spike propagation. *IPC* is the abbreviation of instructions executed per cycle, *Achieved Occupancy (AO)* means the ratio of the average active warps per active cycle to the maximum number of warps, *Global Hit Rate (GHR)* is the hit rate for global loads in unified L1/TEX cache and *Warp Execution Efficiency (WEE)* stands for the ratio of the average active threads per warp to the maximum number of threads per warp.

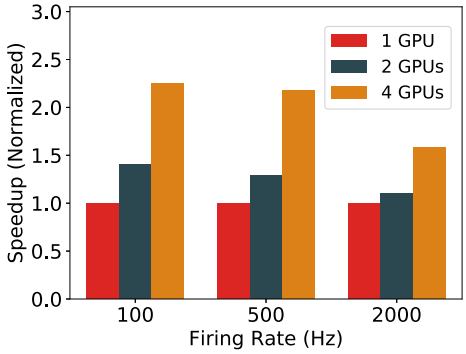

Fig. 7. Speedup of BSim with multiple GPUs.

For example, for the same SNN, this method itself can achieve about $1.08\times$ speedup on an NVIDIA GeForce GTX 1080Ti GPU.

6.3.2 Multiple GPU Optimization Evaluation

We also evaluate the performance of BSim on multiple GPUs. The network used here is $(N = 1000000,\ FR \in \{100\ \text{Hz}, 500\ \text{Hz}, 2000\ \text{Hz}\}, K = 1000, R = 20\%)$. In addition to the computation time, we also analyze the overhead of communication and synchronization.

Fig. 7 shows the overall speedup of BSim on multiple GPUs with different firing rates. And Fig. 8 illustrates the size of data transferred between GPUs with different firing rates. Fig. 9 further breaks down the overall simulation time into separated computation, communication, and synchronization time.

From Fig. 7, we can find out that when more GPUs are employed, the overall performance is much better. But the speedup-per-GPU decreases a little. That's because when more GPUs are employed, the size of the transferred data increases significantly (as shown in Fig. 8). And it will further lead to worse computation to communication/synchronization ratio (as shown in Fig. 9). When the firing rate increases, there is a similar phenomenon. Further, Fig. 9 shows that in both cases, the time spent on synchronization increases more than that for communication. That's because when more GPUs are employed, the overall parallelism of data transfer is also increased. On the other hand, there are more threads to be synchronized.

We also test the impact of shadow neuron optimization. Without shadow neurons, the size of transferred data

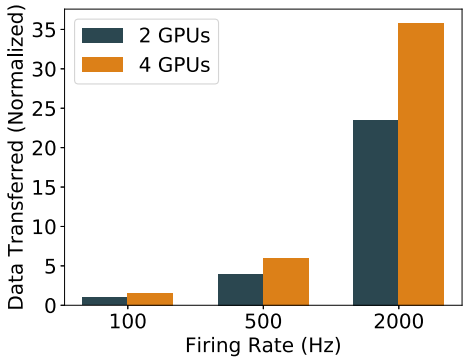

Fig. 8. The size of transferred data for different networks. The data is normalized and that for the network with (2 GPUs, FR=100 Hz) is used as the baseline.

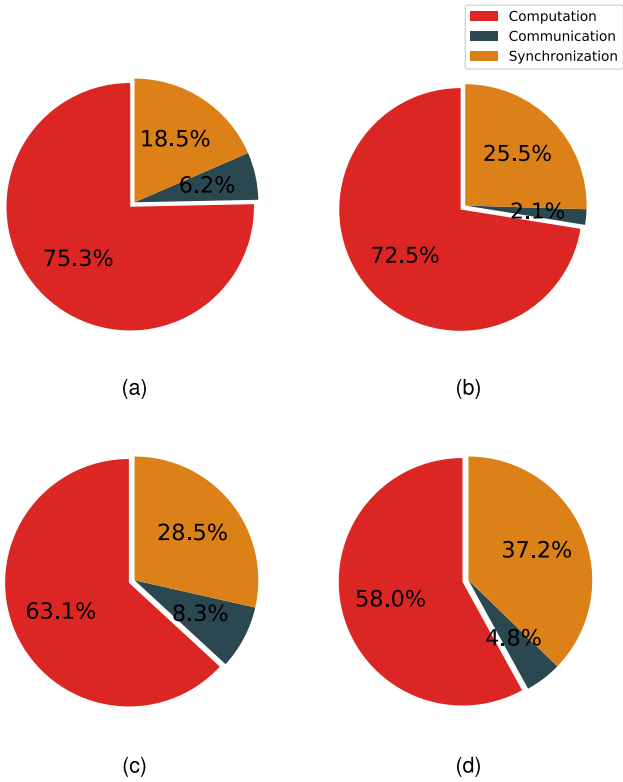

Fig. 9. Breakdown of the simulation time for different networks concerning computation, communication, and synchronization. The network configurations are (a) 2 GPUs, FR=100 Hz, (b) 2 GPUs, FR=2000 Hz, (c) 4 GPUs, FR=100 Hz, and (d) 4 GPUs, FR=2000 Hz.

increases about $1000\times$ when four GPUs are used and the firing rate is 100 Hz. That is because the simulator has to deliver the information of each synapse that received spikes instead of the information of shadow neurons, and the former is about $K\times$ larger than the later. Further, without shadow neurons, the overall simulation time increases by $2.24\times$, the computation to communication/synchronization ratio drops to about 33.9 percent, and the memory consumption increases 22.9 percent.

Although more GPUs result in a higher speedup, the overall speedup of four GPGPUs is about 2.3. That is because the current simulation procedure of SNNs requires the synchronization of global simulation steps among different GPUs in every simulation cycle and this synchronization takes a lot of time. As shown in Fig. 9, when more GPUs are employed, the time spent on synchronization increases considerably. Improving performance in this aspect is part of our future work.

On the other hand, when the network size exceeds the memory capability of a single GPU, the multiple-GPUs-solution is more than one order of magnitude faster than the single-GPU-solution, as the latter has to transfer data between the GPU memory and the host memory every cycle. Therefore, the support for multiple GPUs is more meaningful to increase the maximum size of supported networks, while also improving the execution efficiency.

7 Conclusion

In this paper, we make full use of the intrinsic properties of SNN to speed up its simulation on GPGPUs. Specifically, we propose the fine-grained network representation and the

cross-population/-projection parallelism exploration to fully explore the parallelism and optimize the memory access patterns. We also propose the sparsity aware load balance to deal with the resource underutilization and load imbalance caused by activity sparsity. Besides, we further provide dedicated optimization methods to support multiple GPUs. Based on these technologies, we propose BSim, an efficient and easy-to-use simulation framework for SNN. As demonstrated by experimental evaluation, it can achieve up to $9.33\times$ speedup over a state-of-the-art simulator, GeNN. And it outperforms other simulators much more. Test results also show that by employing multiple GPUs, BSim can support much larger networks while achieving higher speedup.

Acknowledgments

The authors would like to thank for the support from Beijing Academy of Artificial Intelligence, under Grant BAAI2019ZD0503, the support from Beijing National Research Center for Information Science and Technology, and the support from Beijing Innovation Center for Future Chips, Tsinghua University.

References

[1] C. Eliasmith *et al.*, "A large-scale model of the functioning brain," *Science*, vol. 338, no. 6111, pp. 1202–1205, 2012.
[2] M. W. Reimann, C. A. Anastassiou, R. Perin, S. L. Hill, H. Markram, and C. Koch, "A biophysically detailed model of neocortical local field potentials predicts the critical role of active membrane currents," *Neuron*, vol. 79, no. 2, pp. 375–390, 2013.
[3] A. Katrin *et al.*, "The human brain project: Creating a european research infrastructure to decode the human brain," *Neuron*, vol. 92, no. 3, pp. 574–581, 2016.
[4] J. Jordan *et al.*, "Extremely scalable spiking neuronal network simulation code: from laptops to exascale computers," *Frontiers Neuroinformatics*, vol. 12, 2018, Art. no. 2.
[5] W. Maass and H. Markram, "On the computational power of circuits of spiking neurons," *J. Comput. Syst. Sci.*, vol. 69, no. 4, pp. 593–616, 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000004000406
[6] F. Akopyan *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1537–1557, Oct. 2015.
[7] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *Int. J. Comput. Vis.*, vol. 113, no. 1, pp. 54–66, 2015.
[8] W. Gerstner, H. Sprekeler, and G. Deco, "Theory and simulation in neuroscience," *Science*, vol. 338, no. 6103, pp. 60–65, 2012.
[9] X. Jin and S. Furber, *Parallel Simulation of Neural Networks on Spinnaker Universal Neuromorphic Hardware*. Univ. Manchester, 2010. [Online]. Available: https://www.escholar.manchester.ac.uk/uk-ac-man-scw:85336
[10] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.
[11] M. Davies *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, Jan./Feb. 2018.
[12] V. Braitenberg and A. Schüz, *Anatomy of the Cortex: Statistics and Geometry*, vol. 18, Berlin, Germany: Springer, 2013.
[13] A. Davison *et al.*, "PyNN: A common interface for neuronal network simulators," *Frontiers Neuroinformatics*, vol. 2, 2009, Art. no. 11. [Online]. Available: https://www.frontiersin.org/article/10.3389/neuro.11.011.2008
[14] M.-O. Gewaltig and M. Diesmann, "NEST (neural simulation tool)," *Scholarpedia*, vol. 2, no. 4, 2007, Art. no. 1430.
[15] M. Stimberg, D. Goodman, V. Benichoux, and R. Brette, "Equation-oriented specification of neural models for simulations," *Frontiers Neuroinformatics*, vol. 8, 2014, Art. no. 6. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2014.00006
[16] M. L. Hines and N. T. Carnevale, "The neuron simulation environment," *Neuron Comp.*, vol. 9, no. 6, pp. 1179–209, 2006.
[17] E. Yavuz, J. Turner, and T. Nowotny, "GeNN: A code generation framework for accelerated brain simulations," *Scientific Rep.*, vol. 6, 2016, Art. no. 18854.
[18] M. Beyeler, K. D. Carlson, T.-S. Chou, N. Dutt, and J. L. Krichmar, "CARLsim 3: A user-friendly and highly optimized library for the creation of neurobiologically detailed spiking neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, 2015, pp. 1–8.
[19] D. Pecevski, T. Natschläger, and K. Schuch, "PCSIM: A parallel simulation environment for neural circuits fully integrated with python," *Frontiers Neuroinformatics*, vol. 3, 2009, Art. no. 11. [Online]. Available: https://www.frontiersin.org/article/10.3389/neuro.11.011.2009
[20] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk, "NeMo: A platform for neural modelling of spiking neurons using gpus," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst. Architectures Process.*, 2009, pp. 137–144.
[21] A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," in *Proc. Int. Joint Conf. Neural Netw.*, 2009, pp. 2145–2152.
[22] R. Brette *et al.*, "Simulation of networks of spiking neurons: a review of tools and strategies," *J. Comput. Neurosci.*, vol. 23, no. 3, pp. 349–398, 2007.
[23] R. Kass *et al.*, "Computational neuroscience: Mathematical and statistical perspectives," *Annu. Rev. Statist. Appl.*, vol. 5, pp. 183–214, 2018.
[24] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Netw.*, vol. 10, pp. 1659–1671, 1997.
[25] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Frontiers Comput. Neurosci.*, vol. 9, pp. 99–99, 2015.
[26] P. O'Connor, D. Neil, S.-C. Liu, T. Delbruck, and M. Pfeiffer, "Real-time classification and sensor fusion with a spiking deep belief network," *Front. Neurosci.*, vol. 7, 2013, Art. no. 178. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2013.00178
[27] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *IEEE Trans. Neural Netw.*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.
[28] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biol. Cybern.*, vol. 95, no. 1, pp. 1–19, Jul. 2006. [Online]. Available: https://doi.org/10.1007/s00422–006-0068-6
[29] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Bull. Math. Biol.*, vol. 52, no. 1, pp. 25–71, Jan. 1990. [Online]. Available: https://doi.org/10.1007/BF02459568
[30] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.
[31] J. Schemmel *et al.*, "Live demonstration: A scaled-down version of the brainscales wafer-scale neuromorphic system," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2012, p. 702.
[32] S. B. Furber *et al.*, "Overview of the spinnaker system architecture," *IEEE Trans. Comput.*, vol. 62, no. 12, pp. 2454–2467, Dec. 2013.
[33] K. Gurney, T. J. Prescott, and P. Redgrave, "A computational model of action selection in the basal ganglia. i. a new functional anatomy," *Biol. Cybern.*, vol. 84, no. 6, pp. 401–410, 2001.
[34] S. Song, K. Miller, and L. Abbott, "Competitive hebbian learning through spike timing-dependent plasticity," *Nature Neurosci.*, vol. 3, pp. 919–26, 2000.
[35] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Proc. Int. Joint Conf. Neural Netw.*, 2015, pp. 1–8.
[36] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training deep spiking neural networks using backpropagation," *Frontiers Neurosci.*, vol. 10, 2016, Art. no. 508. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2016.00508
[37] E. Hunsberger and C. Eliasmith, "Training spiking deep networks for neuromorphic hardware," *CoRR*, vol. abs/1611.05141, 2016. [Online]. Available: http://arxiv.org/abs/1611.05141
[38] P. Richmond, A. Cope, K. Gurney, and D. J. Allerton, "From model specification to simulation of biologically constrained networks of spiking neurons," *Neuroinformatics*, vol. 12, no. 2, pp. 307–323, 2014.
[39] R. Brette, S. Marcel, and D. Goodman, "The brian spiking neural network simulator," Accessed: Jul. 19, 2017, 2017. [Online]. Available: http://briansimulator.org/

[40] K. Cheung, S. R. Schultz, and W. Luk, "NeuroFlow: A general purpose spiking neural network simulation platform using customizable processors," *Frontiers Neurosci.*, vol. 9, 2016, Art. no. 516. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnins.2015.00516
[41] T. E. Oliphant, "Scipy: Open source scientific tools for python," *Comput. Sci. Eng.*, vol. 9, pp. 10–20, 2007.
[42] D. Merrill, M. Garland, and A. S. Grimshaw, "Scalable GPU graph traversal," in *Proc. ACM Sigplan Symp. Principles Practice Parallel Program.*, 2012, pp. 117–128.
[43] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proc. 21st Annu. Symp. Parallelism Algorithms Architectures*, 2009, pp. 233–244.
[44] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, Art. no. 2.
[45] A. K. Fidjeland and M. P. Shanahan, "Accelerated simulation of spiking neural networks using GPUs," in *Proc. Int. Joint Conf. Neural Netw.*, 2010, pp. 1–8.
[46] A. Russell *et al.*, "Optimization methods for spiking neurons and networks," *IEEE Trans. Neural Netw.*, vol. 21, no. 12, pp. 1950–1962, Dec. 2010.
[47] R. Guyonneau, R. Vanrullen, and S. J. Thorpe, "Temporal codes and sparse representations: A key to understanding rapid processing in the visual system," *J. Physiol.-Paris*, vol. 98, no. 4, pp. 487–497, 2004.
[48] A. Roxin, N. Brunel, D. Hansel, G. Mongillo, and V. V. C, "On the distribution of firing rates in networks of cortical neurons," *J. Neurosci.*, vol. 31, no. 45, pp. 16 217–16 226, 2011.
[49] E. Hunsberger and C. Eliasmith, "Spiking deep networks with lif neurons," 2015, *arXiv:1510.08829*.
[50] D. Zambrano and S. M. Bohte, "Fast and efficient asynchronous neural computation with adapting spiking neural networks," 2016, *arXiv: 1609.02053*.
[51] T. P. Vogels and L. F. Abbott, "Signal propagation and logic gating in networks of integrate-and-fire neurons," *J. Neurosci.*, vol. 25, no. 46, pp. 10 786–10 795, 2005.

Peng Qu received the BS and PhD degrees in computer science and technology from Tsinghua University, Beijing, in 2013 and 2018, respectively. He is currently a postdoc with the Department of Computer Science and Technology, Tsinghua University, Beijing. His research interests include computer architecture and neuromorphic computing.

Youhui Zhang (Member, IEEE) received the BS and PhD degrees in computer science from Tsinghua University, Beijing, in 1998 and 2002, respectively. He is currently a professor with the Department of Computer Science and Technology, Tsinghua University, Beijing. His research interests include computer architecture and neuromorphic computing. He is a member of ACM.

Xiang Fei received the BS degree in computer science from Zhejiang University, Hangzhou, in 2016. He is currently working toward the PhD degree with the Department of Computer Science and Technology, Tsinghua University, Beijing. His research interests include computer architecture and neuromorphic computing.

Weimin Zheng (Member, IEEE) received the BSc and MSc degrees in computer science from Tsinghua University, China, in 1970 and 1982, respectively, where he is a professor with the Department of Computer Science. His research interests include high-performance computing, network storage, and parallel compilers. He is a member of the IEEE Computer Society.