Accelerated Optimization for Simulation of Brain Spiking Neural Network on GPGPUs

Fangzhou Zhang[1], Mingyue Cui[1], Jiakang Zhang[1], Yehua Ling[2], Han Liu[1], and Kai Huang[1(✉)]

[1] Sun Yat-Sen University, Guangzhou 510006, China
huangk36@mail.sysu.edu.cn

[2] Guangxi Transportation Science and Technology Group Co., Ltd., Nanning 530007, China

Abstract. As the application scenarios for large-scale spiking neural networks (SNN) increase, efficient SNN simulation becomes more essential. However, simulating such a large-scale network faces expensive overhead in terms of computation and communication, especially for high firing rates. To address this problem, we propose an effective accelerated optimization method for simulating SNN on GPGPUs, which simultaneously takes into account workload balancing and communication overhead. We design a workload-oriented network partition algorithm to minimize the number of external synapses and ensure workload balance. Additionally, we propose spike synchronization optimization by incorporating fine-grained scale, data compression, and full-duplex communication. This optimization aims to achieve lower communication overhead and better performance improvement. Furthermore, to avoid thread warp divergence, we assign an entire thread block for each neuron without collecting information on fired neurons in the spike propagation phase, which simplifies the execution flow and enhances performance. Experimental results demonstrate that our simulator can achieve up to 1.31×∽6.74× speedup for SNN with different configurations, and the efficiency is improved by 40.21%∽51.11% compared with the state-of-the-art methods.

Keywords: SNN simulation · Accelerated optimization · Load balance · Spiking neural network · High performance · GPGPUs

1 Introduction

Compared with traditional deep neural networks (DNN), spiking neural networks (SNN) are considered to be a more accurate representation of the human brain in biology, offering stronger interpretability and adaptability [23,24]. Simulating SNN serves as a pivotal bridge between neuroscience and artificial intelligence, offering a unique opportunity to unravel the intricacies of neural information

Z. Tari et al. (Eds.): ICA3PP 2023, LNCS 14492, pp. 172–189, 2024.
https://doi.org/10.1007/978-981-97-0811-6_10

processing. SNN incorporates time as an additional input dimension, and their unique spike-driven mechanism provides natural advantages in the field of bionic robots. IBM specifically designs an ultra-low-power programmable neurosynaptic chip called TrueNorth [2]. This chip integrates 5.4 billion transistors, including 1 million neurons and 256 million synapses. Intel also develops Loihi chip [19] for realizing brain-inspired simulation functions, which has a total area of 3840 square millimeters, 8 million neurons, and 8 billion synapses. However, due to selling price or policy restrictions, this has greatly limited the widespread use of SNN.

SNN simulation deployment on traditional commercial semiconductor platforms is one of the solutions. Balaji *et al.* [4] propose a SNN-accelerated GeNN framework, using technologies such as sparse representation of synaptic connections and block size determination based on occupancy to deploy SNN. Ahmad *et al.* [1] further propose an activation synaptic grouping strategy to solve the unbalanced workload of CUDA thread warps. Hazan *et al.* [14] present Bindsnet SNN simulation framework which uses PyTorch library for scheduling heterogeneous resources to achieve better scalability and computation acceleration. These methods realize the simulation deployment of SNN by adjusting the representation of neurons and synapses, but the acceleration effect is often very limited.

The sparsity of data in both time and space adds complexity to achieve a high degree of parallelism in SNN simulations. GPGPU-powered SNN simulators usually introduce synapse-based thread mapping methods that ignore the inherent sparsity property of SNN, which leads to load imbalance and severe limitations on parallelism. Compared with DNN, the topology of SNN tends to be more randomized. This characteristic poses a considerable challenge in achieving an even distribution of resources. Existing simulators focus on the number of synapses to design load balancing, which ignores the impact of the topology of SNN. Maintaining the connectivity and coherence of distributed networks means frequent data exchange during the spike synchronization phase, which brings significant communication overhead. Existing SNN simulators are difficult to make full use of the resources of GPGPUs.

This paper proposes an effective multi-GPU SNN simulator with accelerated optimization. Firstly, we introduce a novel workload-oriented network partition model, which modifies the partition balance condition of Metis [15] algorithm based on the workload model. The partition model reduces the number of external synapses which achieves lower communication overhead in the simulation stage while maintaining workload balance. Subsequently, to boost the computing of the simulation, we simplify execution flow without collecting fired neuron information and propose a branch-matched thread organization strategy to improve the spike propagation performance. This strategy assigns an entire thread block for each neuron to avoid thread warp divergence efficiently for lower computation overhead. Finally, considering the impact of communication overhead on multi-GPU simulation performance, we design a spike synchronization optimization with fine-grained scales, data compression, and full-duplex communication to achieve fewer data transfer sizes and better transfer parallelism during spike synchronization. We conduct extensive performance evaluations for

SNN with a set of different configurations. Experimental results on GPGPUs show that our implementation can achieve better performance, compared with the state-of-the-art methods. The contributions of the work mentioned above can be summarized as follows:

- We propose an accelerated SNN simulation framework on GPGPUs, which can efficiently simulate large-scale SNN networks.
- By introducing the workload-oriented network partition algorithm, our method reduces the number of external synapses, which achieves lower communication overhead during the simulation stage.
- To avoid thread warp divergence, we present a thread organization strategy that allocates thread blocks in units of neurons.
- We design a spike synchronization algorithm with optimization in fine-grained scale, data compression, and full-duplex communication, which further reduces the overhead of communication.

The rest of the paper is organized as follows. Section 2 provides background information and related work. Section 3 describes our method of SNN-specific optimization on GPGPUs. Section 4 presents the experimental details and analyses the simulation results. Finally, we conclude the paper in Sect. 5.

2 Related Work

2.1 Spiking Neural Networks

As the most widely used model in the field of neuroscience and neuromorphic computing, SNN can be described as a directed network, in which neurons connect with each other arbitrarily through synapses. A standard SNN mainly consists of neurons, synapses, and spikes, as shown in Fig. 1.

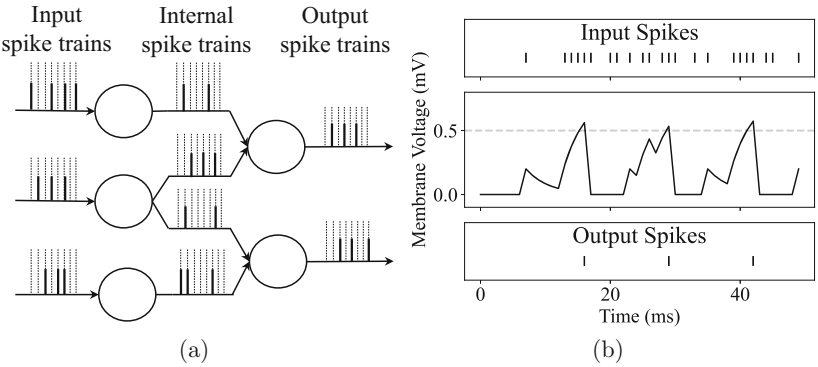


Fig. 1. An example of SNN. (a) The composition of a simple SNN. (b) Membrane voltage, input spike arrival time, and spike generation time for a neuron.

Neurons correspond to signal processing units. Different mathematical expressions of the states distinguish different neuron models. Taking LIF model [13] as an example, its differential equation indicates the iterative formula for neural membrane potential v over time as follows:

$$C_m \frac{dv}{dt} = \frac{C_m(v_{rest} - v)}{\tau_m} + (I_{synE} + I_{synI}) \tag{1}$$

$$\begin{cases} I_{synE} = g_e \times (v_{exc} - v) \\ I_{synI} = g_i \times (v_{inh} - v) \end{cases}, \begin{cases} dg_e/dt = -g_e/\tau_{exc} \\ dg_i/dt = -g_i/\tau_{inh} \end{cases} \tag{2}$$

The iterative update of the membrane potential v depends on the membrane capacitance C_m, membrane time constant τ_m, excitatory and inhibitory synaptic input currents I_{synE} (I_{synI}). The synaptic currents are calculated from the corresponding potentials v_{exc} (v_{inh}) and synaptic conductance g_i (g_e), in which g_i and g_e are also updated iteratively over time based on their respective time constants τ_{exc} (τ_{inh}).

Spikes correspond to the time sequence signal. In SNN, the signals generated by the neurons can be modeled as a binary value (0/1) with a timestamp. Each value in the sequence corresponds to a spike and represents the arrival time of the spike. In general, spikes are used for information transfer between neurons. Spikes can be generated by external input or by neurons during simulation.

Synapses connect neurons and transmit spikes. According to the direction of synapses, neurons are divided into source neurons and target neurons. Spikes sent by the source neurons travel along the synapses to the target neurons. When the source neuron is activated and sends out the spike, it needs to delay d before reaching the target neuron. Besides, if the weight of a synapse is negative, it means that the spikes inhibit the target neuron.

2.2 SNN Simulation Tools

With the growing interest in building large-scale SNN models, several SNN simulators [3,7,10,16,18,22,27] have been proposed to help the research communities. NEST [11] is the first neural network simulator on the traditional computer, which uses multi-core computers and computing cluster resources based on OpenMP. Brain [12] and Brain2 [28] provide equation-oriented methods to define new models, which support a variety of neuron/synapse models. Furthermore, GenEHH [21] lumps some anatomical features into a single compartment to simplify analysis and reduce computational load.

Compare with the CPU-based simulator, using GPU [6,9,17,20,25] has more advantages in simulation speed. SPIKE [1] uses time slice grouping and insensitive delay to improve the simulation speed. CARLsim4 [9] proposes a SNN simulation framework based on heterogeneous resources, which allows multiple GPUs and CPU cores to be used simultaneously. Different from the simulators mentioned above, SNAVA [26] is implemented in modern Field-Programmable Gate Arrays (FPGAs) devices to improve performance execution and flexibility to support large-scale SNN models.

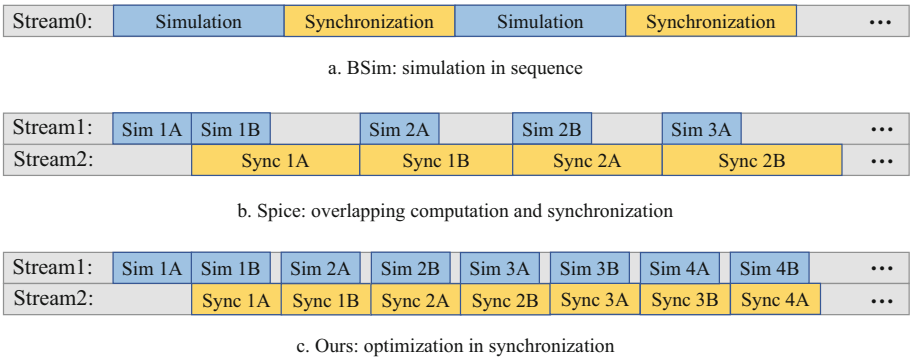


Fig. 2. Comparison of spike synchronization for different methods.

Recently, BSim [22] is a code generation SNN simulation framework, which implements sparsity aware load balance on CUDA threads by distinguishing neuron state in advance. It optimizes transmission by synchronizing neuron states instead of spikes, but the absence of time information leads to no space for further performance improvements. Spice simulator [6] implements a load balancing strategy for synapses and a cache-aware spike transmission algorithm which overlaps the computation and transmission by bisecting the time slice group. Figure 2 shows the process of simulation for different simulators. Different from the serial simulation execution of BSim, the Spice simulator further bisects the time slice group and uses two CUDA streams to realize the parallelism between the synchronization phase and the simulation phase. However, Spice transfers spikes of all devices directly, which ignores the size of data transferred and introduces large communication overhead. Besides, neither BSim nor Spice consider optimizing the number of external synapses in the pre-simulation stage. By contrast, we propose a workload model that guides the Metis algorithm to more evenly distribute neurons and synapses to different devices and reduce the ratio of edge-cut of partition results, which means less communication overhead. We also design a spike synchronization optimization with fine-grained scales, data compression, and full-duplex communication which achieves fewer data transfer sizes and better transfer parallelism. In this way, our method greatly reduces communication overhead and improves simulation performance, compared with others.

2.3 CUDA Programming on Multiple GPUs

CUDA is a parallel computing platform and programming model developed by NVIDIA. It allows developers to harness the power of NVIDIA GPUs (Graphics Processing Units) for general-purpose computing tasks. CUDA programming enables efficient parallel execution of code on multiple GPUs for high-performance computing. To improve performance when using multiple GPUs, it is essential to employ load balancing techniques to evenly distribute the workload across GPUs, preventing one GPU from becoming a bottleneck while others

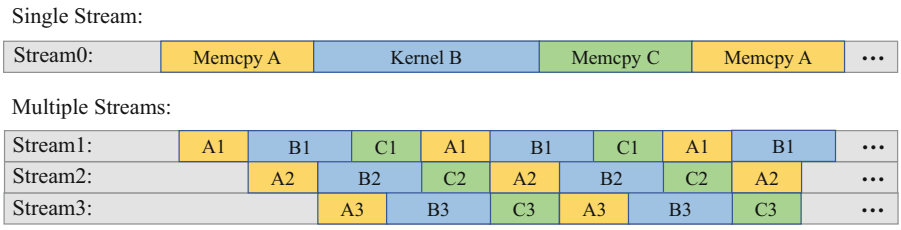


Fig. 3. An example of the overlapping using multiple streams.

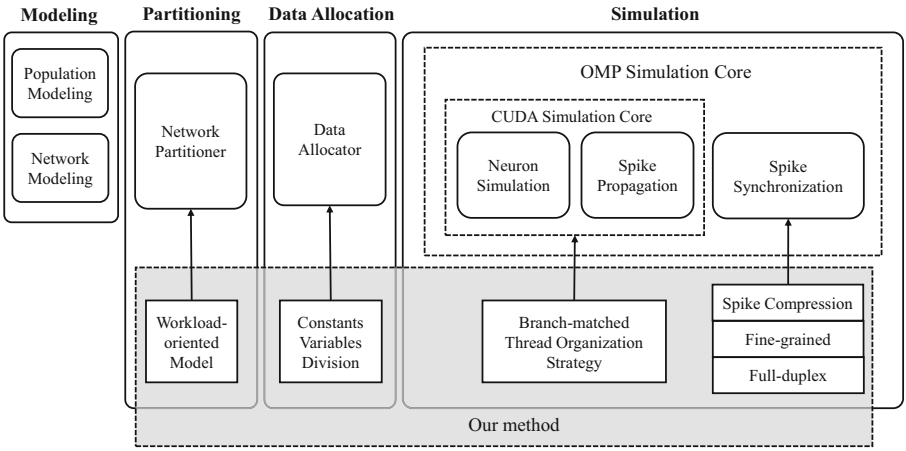


Fig. 4. The framework of accelerated optimization for simulation.

remain idle. Moreover, multi-stream parallelism enables the simultaneous execution of multiple kernel launches and memory operations. This approach effectively reduces the impact of data transfer latency by overlapping transmission and computation through asynchronous memory operation and CUDA streams. By initiating data transfers while the GPUs are still processing previous computations, the overall performance can be greatly enhanced, resulting in improved parallelism and better GPU utilization. Figure 3 shows an example of overlapping transmission and computation. By decomposing tasks and putting them into multiple streams, the computing resources of the GPU are more fully utilized. Incorporating these techniques can optimize the performance of multi-GPU CUDA applications.

3 Method

As is shown in Fig. 4, we propose an effective multi-GPU SNN simulation framework, where the grey area describes the focus of our optimization. The entrance of the framework is a modeling module that defines the model and specifies the network architecture. We provide a series of easy-to-use modeling tools for users

in this module. Our method mainly works on the following three fields: partitioning, data allocation, and simulation. Firstly, in the partitioning module, we propose a workload-oriented network partition algorithm combining the workload model and Metis algorithm. Our workload-oriented partition algorithm can achieve a better partition effect by reducing the number of external synapses and taking into account workload balancing. Secondly, at the data allocation module, we take the memory access optimization into the module by extracting invariants in the simulation flow and allocating them to the constant memory of GPGPUs. Besides, we propose a branch-matched thread organization strategy to improve the spike propagation performance, which simplifies execution flow without collecting fired neuron information. This strategy assigns an entire thread block for each neuron to avoid thread warp divergence efficiently which is of benefit for lower computation overhead. Finally, at the simulation module, we present a spike synchronization algorithm with optimization in fine-grained scale, data compression, and full-duplex communication to achieve lower communication overhead and better performance improvement during spike synchronization.

3.1 Workload-Oriented Partitioning

Network partitioning is a critical stage before simulation, which is used to implement static load balancing and may affect the communication overhead during the synchronization phase. The partition algorithm divides a large-scale network into multiple devices, where neurons and their post-synapses are assigned to different devices. For those synapses whose source neuron and target neuron are located in different devices are defined as external synapses. And the ratio of edge-cut indicates the proportion of external synapses to the number of all synapses. The number of neurons and synapses directly affects the workload of the device, while the number of external synapses affects the communication overhead during simulation. In this section, we focus on reducing the number of external synapses while maintaining the load balance. Specifically, we propose a workload model that guides the Metis algorithm to more evenly distribute neurons and synapses to different devices. In this way, our method reduces the ratio of edge-cut, which means less communication overhead of different devices in the simulation stage.

In the workload model, we use piecewise functions wgt_n and wgt_s to specify the simulation overhead of different kinds of neurons and synapses. The equations are as follows:

$$wgt_n(n) = \begin{cases} C_{lif}, & type(n) = lif \\ C_{poisson}, & type(n) = poisson \end{cases} \tag{3}$$

$$wgt_s(s) = \begin{cases} C_{static}, & type(s) = static \\ C_{stdp}, & type(s) = stdp \end{cases} \tag{4}$$

where C_{lif}, $C_{poisson}$, C_{static}, and C_{stdp} define the computational overhead of different types of neurons and synapses. We further define the number of devices

as N_d and the number of external synapses is defined as N_{outsyn}. For a device d_i, the number of neurons and synapses are N^i_{neu} and N^i_{syn}. neu^i_m is the m'th neuron in the device d_i. Similarly, syn^i_n is the n'th synapse in the device d_i. Then we can describe the workload V_i of device d_i as follows:

$$V_i = \sum_{m}^{N^i_{neu}} (wgt_n(neu^i_m)) + \sum_{n}^{N^i_{syn}} (wgt_s(syn^i_n)) \tag{5}$$

For static load balancing, the goal of our method is to seek the minimum value of the standard deviation of different device workloads:

$$min \quad SD_{load} = \sqrt{\frac{\sum^i_{N_d}(V_i - \bar{V})^2}{N_d}} \tag{6}$$

where $\bar{V}$ is the average workload of all the devices. Finally, we combine the workload model with the Metis algorithm to ensure the workload balance among devices while obtaining partition results with fewer external synapses.

3.2 Memory Access Optimization

Constant memory is a cache that shares data across multiple thread blocks and is read-only for each thread block. It offers lower latency and higher bandwidth compared with global memory, allowing for faster memory access. Based on GPU's constant memory, we divide simulation data into variables and constants and store constant data in constant memory to reduce memory access overheads. As shown in Eq. 1 and Eq. 2, some of elements do not change during the simulation, such as C_m, v_{rest}, τ_m, v_{reset}, τ_{exc}, τ_{inh}, v_{exc}, and v_{inh}. So we divide them from others and copy them into GPU constant memory. Compared with other simulators that store all these quantities in the global memory, it can be shared efficiently among threads without requiring expensive global memory accesses.

3.3 Branch-Matched Thread Organization

The spike propagation phase only performs the spike propagation process on the fired neurons. This indicates using GPU to perform spike propagation in a normal way leads to warp divergence and decreases the performance. When thread wrap diverges, the GPU needs to serialize the execution of different instructions, which leads to high latency and low parallelism. Different from the BSim-based spike propagation algorithm, our method does not need to collect the fired neuron, which simplifies the execution flow. Besides, we introduce a branch-matched thread organization strategy by allocating thread blocks in units of neurons to avoid thread warp divergence for higher performance.

The pseudocode of the spike propagation kernel function is shown in Algorithm 1. Firstly, we launch the spike propagation kernel function by allocating thread blocks consistent with the number of neurons. The dimension of the

thread block is set to 32, 64, or 128, according to the scale of the network. Then, we use the index of the thread block as the index of the neuron of the spike to be propagated (line 1). If the neuron is not fired, we terminate the entire thread block to avoid warp divergence (lines 2–3). For a fired neuron, we find all the post-synapses of the neuron (lines 4–5). Then the algorithm assigns all spike propagation tasks of this neuron to threads of the block evenly and efficiently (lines 6–7). Before propagating, the method further judges whether the synapse is an external synapse (8–9). For an internal synapse, the spike is pushed to the corresponding neuron (line 10). For an external synapse, the spike is pushed into the buffer pool of the corresponding device (line 12). By using this thread organization strategy, our method obtains lower overhead in both the neuron simulation phase and the spike propagation phase.

Algorithm 1: Spike propagation algorithm

```
   Input: netId, inBuffers, outBuffers
1  nid = blockIdx.x;
2  if fird (nid) then
3      return;
4  startloc = get_axon_loc (nid);
5  synsize = get_axon_size (nid);
6  for i = threadIdx.x; i < synsize ; i += blockDim.x do
7      synidx = startloc + i;
8      zone = get_zone (synidx);
9      if zone == netId then
10         push_inner_spike (synidx, inBuffers);
11     else
12         push_outter_spike (synidx, zone, outBuffers);
```

3.4 Spike Synchronization Optimization

The spike synchronization which mainly propagates the spikes generated by neurons to neurons in other devices is critical for multi-GPU simulation. This is because the communication overhead in this process has a huge impact on multi-GPU simulation performance. To further reduce simulation overhead, our method overlaps transmission and computation by bisecting time slice group [5]. As shown in Fig. 5, we propose a spike synchronization optimization with fine-grained scales, data compression, and full-duplex communication. We design a compressed spike buffer data structure for spike synchronization, which merges the spikes that have the same target neuron and same arrival time (see Fig. 5(b)). This organization structure can reduce the communication overhead between devices during spike synchronization, especially for the high firing rate.

Different from synchronizing all spike information of the device, our method performs a finer-grained division of spike data to be synchronized for reducing

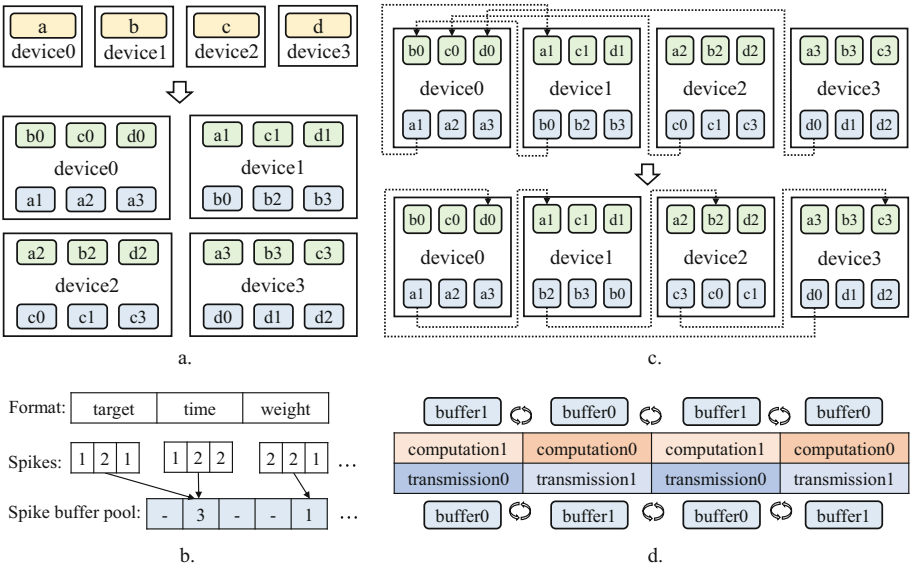


Fig. 5. Schematic diagram of spike synchronization optimization. (a) Finer-grained spike buffer design. (b) Spike data compression. (c) The order of synchronizing spike buffer pools. (d) Double buffering for avoiding conflicts.

redundant information. As is shown in Fig. 5(a), the finer-grained optimization divides the spike buffer of a device into multiple pools according to the device to which the spike propagates, in which green areas are the corresponding receive buffer pool for the spike buffer pool. Considering that computation and transmission to the same data area may lead to conflicts, our method uses double buffering for each buffer pool, as shown in Fig. 5(d). Furthermore, we also optimize the process of spike synchronization of full-duplex communication between GPUs by rearranging the synchronization order of spike buffer pools. From Fig. 5(c), we can see that in the case of no rearrangement, neither device 2 nor device 3 has any download tasks, while device 1 has three download tasks. They do not take full advantage of full-duplex communication at the same time period. However, when using rearrangement, all devices have upload and download tasks at the same period, which means that the data transmission using rearrangement has higher parallelism than that without rearrangement.

4 Experiments

4.1 Setup

In order to verify the effectiveness of our simulator, we consider the following various factors including network scales, firing rate, and network connectivity. Network scales mainly rely on the number of neurons and synapses, which can

Table 1. Hardware specification of the platform

Hardware Type	Hardware Environments
CPU	2× Intel(R) Xeon(R) Gold 6134 CPU
Basic Freq. (GHz)	3.20
Memory (GB)	256
GPU	8× Nvidia Tesla V100
Host connection	PCIe P2P
GPU Memory (GB)	8× 32
CUDA version	10.0

directly affect the memory occupation and simulation time. The firing rate is the average number of times a neuron fires in a second, which has a great effect on the simulation time. A high firing rate means that many spikes need to be propagated in simulation. Network connectivity is the average number of synapses connected by a neuron. It can determine the number of synapses in the network when the number of neurons in the network is certain. By default, we set the network scale and firing rate to 0.75 B and about 150 Hz, respectively. We use the Brunel [8] model as the default network structure, which has Poisson neurons that can manually set the firing rate. For network connectivity, we set the connection probability between populations as 0.1.

As shown in Table 1, we list the hardware specifications of the platform for comparison. All the experiments are deployed on commercial off-the-shelf platforms with Nvidia Tesla V100 GPU and Intel(R) Xeon(R) Gold 6134 CPU. For SNN benchmarks, we choose state-of-the-art methods for comparison, BSim [22] and Spice [5]. For the Spice simulator, in order to provide a corresponding compatible environment, we use docker as a container to deploy in our experimental environment. We use the LIF model to update the internal states of the neurons, which provides a certain degree of biological authenticity while maintaining limited computational complexity. Note that in order to present the results more intuitively, the values of the ordinates of the figures are not consistent. We use the ratio of edge-cut and the standard deviation of workload to evaluate the partition effect of different methods. The ratio of edge-cut reflects the number of external synapses, which affects the communication overhead of simulation. The standard deviation of workload between regions of partition results describes the load balance of the simulation workload on GPGPUs.

4.2 Simulation Results

As shown in Fig. 6, we represent the simulation overhead for different simulators on 8 GPGPUs. We can see that our method is almost the best compared with other methods, especially under high firing rates. When the firing rate is greater than 700 Hz, the efficiency of our simulator is improved by 40.21%∽51.11%

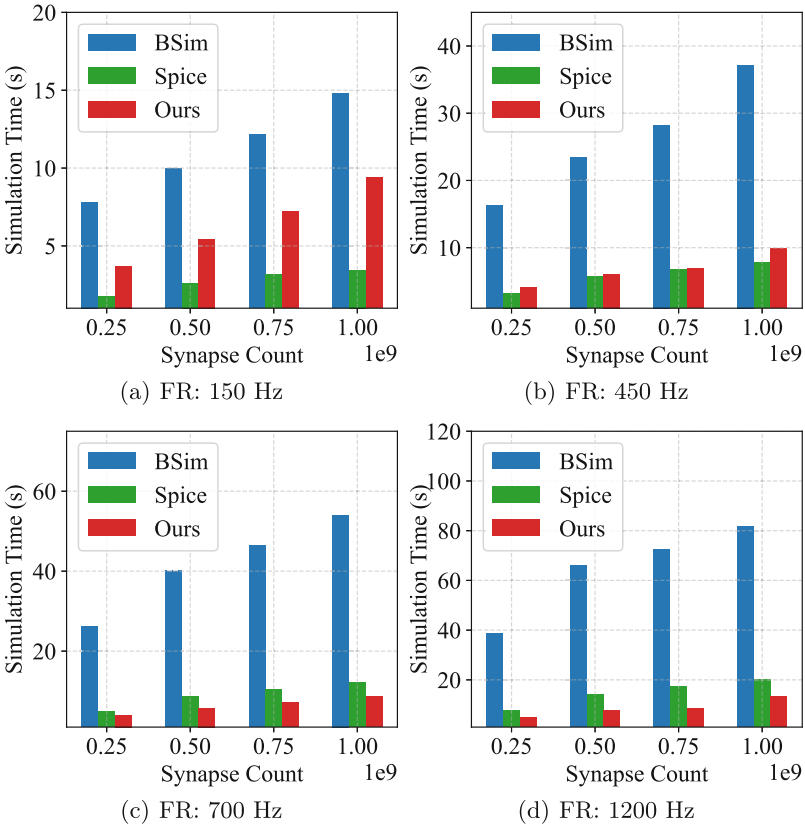


Fig. 6. Simulation time of different simulators with different network scales on 8 GPGPUs.

compared with the Spice simulator, respectively. Benefiting from the compression of spike information, fine-grained spike buffer division, and optimization of spike synchronization order, our method achieves lower communication overhead, especially for high firing rates. Another observation is that the Spice simulator slightly outperforms our method on low firing rate. The reason is that the Spice simulator calculates the synchronized data transfer size according to the number of spikes, which makes a lower communication overhead at lower firing rates. Our method uses a compressed spike data structure that introduces an inherent overhead, but the data transfer size is kept low at high firing rates. The experimental results also confirmed this point in Fig. 6. In addition, our simulator outperforms the BSim simulator for all conditions. Experimental results demonstrate that our method achieves better accelerated optimization than other methods.

We further verify the speedup performance under different GPGPUs when the number of synapses is 0.75B, as shown in Fig. 7. We can observe that our method achieves significant speedup improvements compared with other meth-

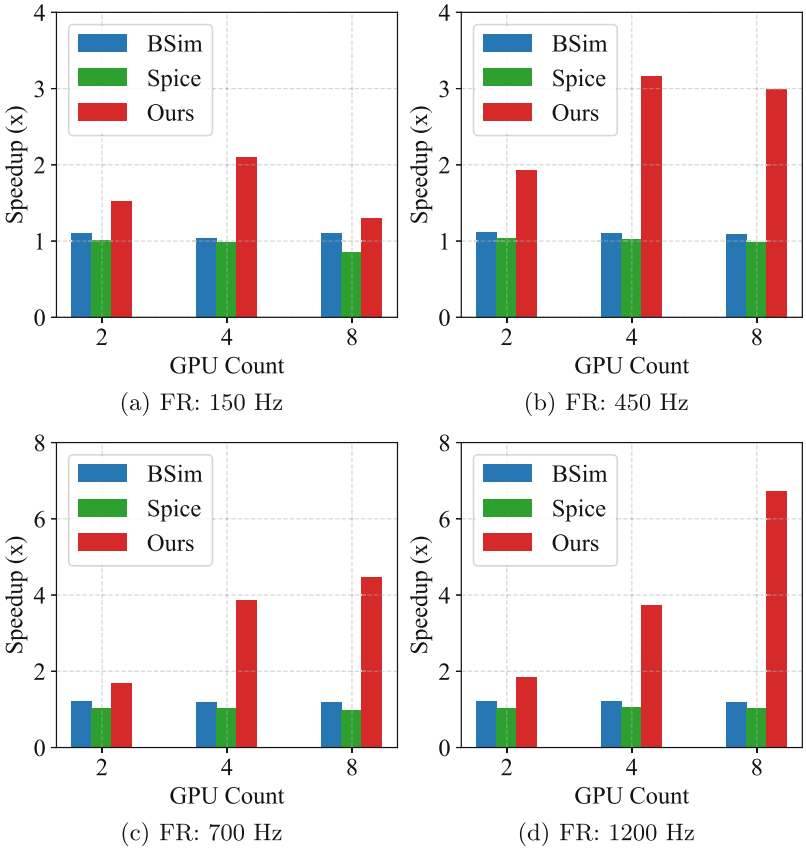
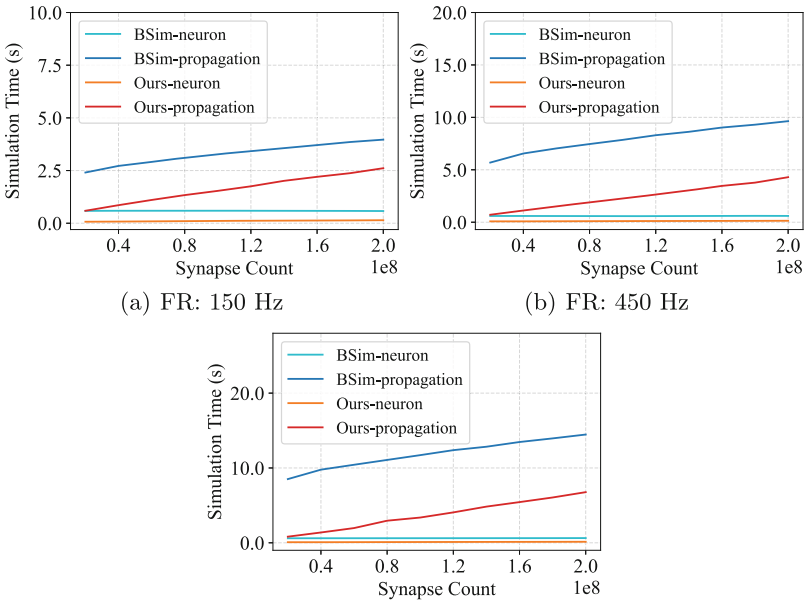


Fig. 7. Speedup of different simulators under different GPGPUs.

ods. The maximum speedup of our simulator can reach 6.74 for a high firing rate on 8 GPGPUs. By contrast, the multi-GPU speedup effect of the BSim and Spice simulator is not obvious. Besides, we notice that the speedup of 4 GPGPUs is better than 8 GPGPUs at low firing rates for each method. Although using 8 GPGPUs provides computation acceleration, it introduces more communication overhead compared with using 4 GPGPUs.

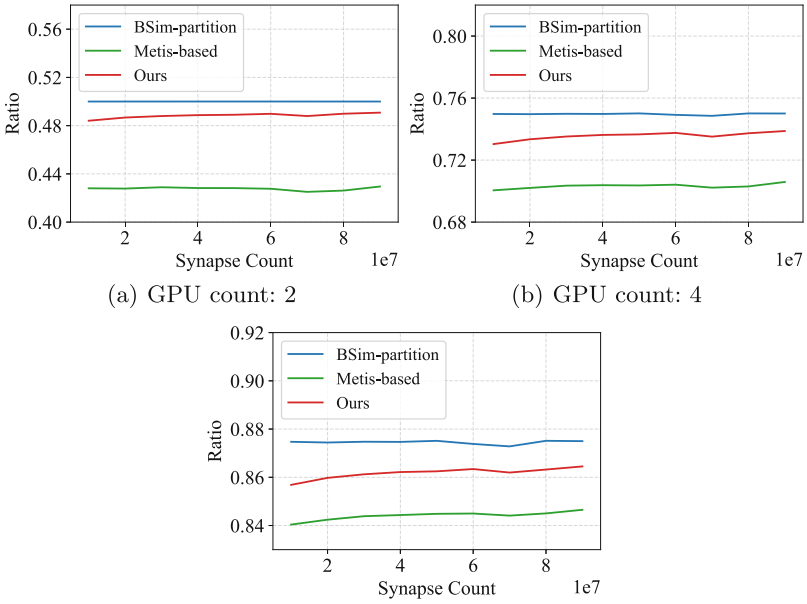
In Fig. 8, we evaluate the computation performance improvement of our method at different stages of the simulation. We extract the time overhead of the CUDA kernel function for neuron simulation and spike propagation and make comparisons on a single GPU to exclude the impact of communication overhead. From Fig. 8, we can see that the time overhead of our simulator for simulating neurons and spike propagation phases on a single GPU is much lower than that of the BSim simulator. Our method simplifies the simulation process through different thread organization methods, which reduces unnecessary computational work while avoiding warp divergence. Experimental results demonstrate that



(a) FR: 150 Hz (b) FR: 450 Hz

(c) FR: 700 Hz

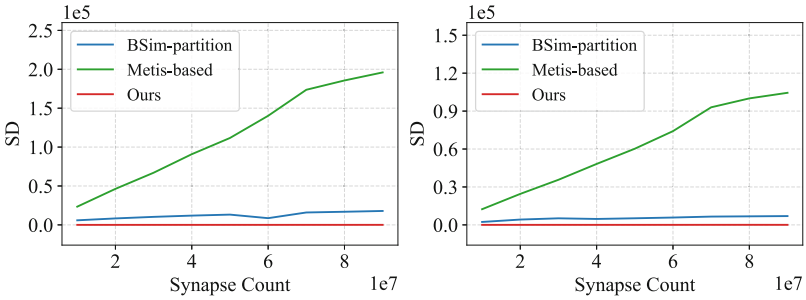
Fig. 8. Time overhead of different simulation stages.



(a) GPU count: 2 (b) GPU count: 4

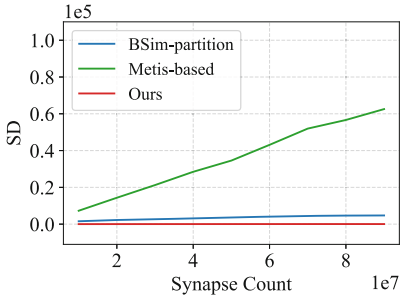
(c) GPU count: 8

Fig. 9. The edge-cut ratio with different network scales.



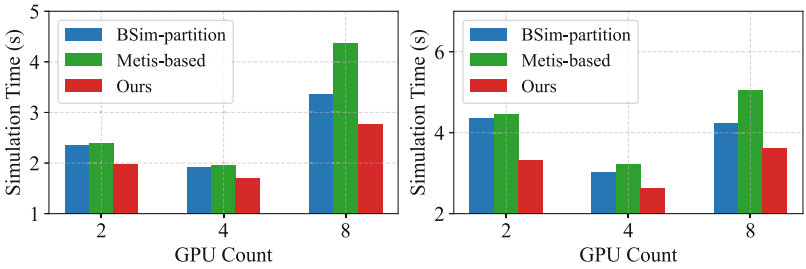
(a) GPU count: 2

(b) GPU count: 4



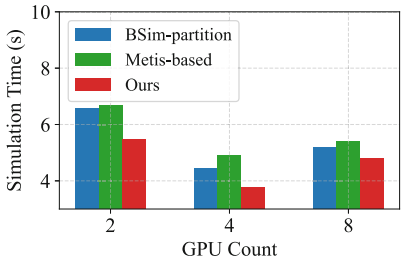
(c) GPU count: 8

Fig. 10. The standard deviation of workload with different network scales.



(a) Synapse Count: 200 M

(b) Synapse Count: 400 M



(c) Synapse Count: 600 M

Fig. 11. Simulation overhead with different partition ways on GPGPUs.

our method achieves performance improvements at neuron simulation and spike propagation stage. Note that since the Spice simulator does not perform some necessary computational processes, such as counting the number of neuron fires, for the sake of fairness, we do not show the results of Spice.

The edge-cut ratio and the standard deviation of workload using different partition methods with different network scales are shown in Fig. 9 and Fig. 10. We can observe that the partition results of our method have a lower edge-cut ratio than BSim, which means fewer external synapses in Fig. 9. Our partition results have a better balanced workload effect than others by introducing a workload model, as shown in Fig. 10. Figure 11 further shows the simulation results when using different partition algorithms. Obviously, our partition algorithm is better than directly using the Metis algorithm, and also better than the BSim. Although the edge-cut ratio of the Metis is lower than our method, it ignores the workload balancing that greatly affects the simulation performance.

5 Conclusion

In this paper, we propose an effective accelerated optimization for the simulation of SNN on GPGPUs, simultaneously taking into account the workload balancing and communication overhead. Our method consists of a workload-oriented network partition, spike synchronization optimization with fine-grained scale, data compression, and full-duplex communication, and the spike propagation process with branch-matched thread organization. Experimental results show that our method achieves effective simulation performance and is insensitive to the firing rate variety. We believe that our method provides a novel perspective for this study. In the future, we will further analyze the impact of the optimization algorithm on accuracy performance. More schemes also need to be considered in terms of dynamic load balancing.

Acknowledgements. This work is supported in part by the Open Project Program for the Engineering Research Center of Software/Hardware Co-design Technology and Application, Ministry of Education (East China Normal University), Grant No. 67000-42990016, and in part by Fundamental Research Funds for the Central Universities, Sun Yat-sen University, Grant No. 23qnpy30/67000-31610023.

References

1. Ahmad, N., Isbister, J.B., Smithe, T., Stringer, S.M.: Spike: a GPU optimised spiking neural network simulator. Cold Spring Harbor Laboratory (2018). https://doi.org/10.1101/461160
2. Akopyan, F., Sawada, J., Cassidy, A., Alvarez-Icaza, R., Modha, D.S.: TrueNorth: design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(10), 1537–1557 (2015). https://doi.org/10.1109/TCAD.2015.2474396
3. Balaji, A., et al.: PyCARL: a PyNN interface for hardware-software co-simulation of spiking neural network (2020). https://doi.org/10.48550/arXiv.2003.09696

4. Balaji, N., Yavuz, E., Nowotny, T.: Scalability and optimization strategies for GPU enhanced neural networks (GENN). Comput. Sci. (2014). https://doi.org/10.48550/arXiv.1412.0595
5. Bautembach, D., Oikonomidis, I., Kyriazis, N., Argyros, A.: Faster and simpler SNN simulation with work queues. In: 2020 International Joint Conference on Neural Networks (IJCNN) (2020). https://doi.org/10.1109/IJCNN48605.2020.9206752
6. Bautembach, D., Oikonomidis, I., Argyros, A.: Multi-GPU SNN simulation with static load balancing. In: 2021 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2021). https://doi.org/10.1109/IJCNN52387.2021.9533921
7. Bekolay, T., et al.: Nengo: a Python tool for building large-scale functional brain models. Front. Neuroinform. **7**, 48 (2014). https://doi.org/10.3389/fninf.2013.00048
8. Brunel, N.: Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. J. Comput. Neurosci. **8**, 183–208 (2000). https://doi.org/10.1023/A:1008925309027
9. Chou, T., Kashyap, H., Xing, J., Listopad, S., Rounds, E.L.: CARLsim 4: an open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In: IEEE International Joint Conference on Neural Networks (2018). https://doi.org/10.1109/IJCNN.2018.8489326
10. Eppler, J., Helias, M., Muller, E., Diesmann, M., Gewaltig, M.O.: PyNEST: a convenient interface to the nest simulator. Front. Neuroinf. **2** (2009). https://doi.org/10.3389/neuro.11.012.2008
11. Gewaltig, M.O., Diesmann, M.: NEST (neural simulation tool). Scholarpedia **2**(4), 1430 (2007). https://doi.org/10.4249/scholarpedia.1430
12. Goodman, D., Brette, R.: Brian: a simulator for spiking neural networks in Python. Front. Neuroinf. **2** (2008). https://doi.org/10.3389/neuro.11.005.2008
13. Goodman, D.F.M., Brette, R.: The brian simulator. Front. Neurosci. **3**(2) (2009). https://doi.org/10.3389/neuro.01.026.2009
14. Hazan, H., et al.: BindsNET: a machine learning-oriented spiking neural networks library in Python. Front. Neuroinf. **12** (2018). https://doi.org/10.3389/fninf.2018.00089
15. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998). https://doi.org/10.1137/S1064827595287997
16. Kasap, B., Opstal, A.V.: Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations. Neurocomputing, S0925231218304168 (2018). https://doi.org/10.1016/j.neucom.2018.04.007
17. Knight, J.C., Nowotny, T.: GPUs outperform current HPC and neuromorphic solutions in terms of speed and energy when simulating a highly-connected cortical model. Front. Neurosci. **12** (2018). https://doi.org/10.3389/fnins.2018.00941
18. Lee, H., Kim, C., Kim, M., Chung, Y., Kim, J.: NeuroSync: a scalable and accurate brain simulator using safe and efficient speculation. In: 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 633–647. IEEE (2022). https://doi.org/10.1109/HPCA53966.2022.00053
19. Lin, C.K., et al.: Programming spiking neural networks on Intel's Loihi. Computer **51**, 52–61 (2018). https://doi.org/10.1109/MC.2018.157113521
20. Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Masquelier, T.: SpykeTorch: efficient simulation of convolutional spiking neural networks with at most one spike per neuron. Front. Neurosci. **13**, 625 (2019). https://doi.org/10.3389/fnins.2019.00625

21. Panagiotou, S., Miedema, R., Sidiropoulos, H., Smaragdos, G., Soudris, D.: A novel simulator for extended Hodgkin-Huxley neural networks. In: 2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (2020). https://doi.org/10.1109/BIBE50027.2020.00071
22. Qu, P., Zhang, Y., Fei, X., Zheng, W.: High performance simulation of spiking neural network on GPGPUs. IEEE Trans. Parallel Distrib. Syst. **31**, 2510–2523(2020). https://doi.org/10.1109/TPDS.2020.2994123
23. Sakemi, Y., Morino, K., Morie, T., Aihara, K.: A supervised learning algorithm for multilayer spiking neural networks based on temporal coding toward energy-efficient VLSI processor design. IEEE Trans. Neural Netw. Learn. Syst. (2021). https://doi.org/10.1109/TNNLS.2021.3095068
24. Shang, Y., Li, Y., You, F., Zhao, R.L.: Conversion-based approach to obtain an SNN construction. Int. J. Software Eng. Knowl. Eng. (2021). https://doi.org/10.1142/S0218194020400318
25. Smaragdos, G., et al.: BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations. J. Neural Eng. **14**(6), 066008.1–066008.15 (2017). https://doi.org/10.1088/1741-2552/aa7fc5
26. Sripad, A., Sanchez, G., Zapata, M., Pirrone, V., Madrenas, J.: SNAVA-a real-time multi-FPGA multi-model spiking neural network simulation architecture. Neural Netw. **97**, 28–45 (2018). https://doi.org/10.1016/j.neunet.2017.09.011
27. Stewart, T.C., Tripp, B., Eliasmith, C.: Python scripting in the Nengo simulator. Front. Neuroinf., 7 (2009). https://doi.org/10.3389/neuro.11.007.2009
28. Stimberg, M., Brette, R., Dan, F.: Brian 2: an intuitive and efficient neural simulator. Cold Spring Harbor Laboratory (2019). https://doi.org/10.7554/eLife.47314