Accelerating Spatiotemporal Supervised Training of Large-Scale Spiking Neural Networks on GPU

Ling Liang[1] **Zhaodong Chen**[1] **Lei Deng**[2] **Fengbin Tu**[1] **Guoqi Li**[2] **Yuan Xie**[1]

[1] *Department of Electrical Computer Engineering, University of California, Santa Barbara, CA, USA*

[2] *Department of Precision Instrument, Tsinghua University, Beijing, China*

{lingliang,chenzd15thu,fengbintu,yuanxie}@ucsb.edu

{leideng, liguoqi}@mail.tsinghua.edu.cn

***Abstract*—Spiking neural networks (SNNs) have great potential to achieve brain-like intelligence, however, it suffers low accuracy of conventional synaptic plasticity rules and low training efficiency on GPUs. Recently, the emerging backpropagation through time (BPTT) inspired learning algorithms bring new opportunities to boost the accuracy of SNNs, while training on GPUs still remains inefficient due to the complex spatiotemporal dynamics and huge memory consumption, which restricts the model exploration for SNNs and prevents the advance of neuromorphic computing.**

In this work, we build a framework to solve the inefficiency of BPTT-based SNN training on modern GPUs. To reduce the memory consumption, we optimize the dataflow by saving CONV/FC results only in the forward pass and recomputing other intermediate results in the backward pass. Then, we customize kernel functions to accelerate the neural dynamics for all training stages. Finally, we provide a Pytorch interface to make our framework easy-to-deploy in real systems. Compared to vanilla Pytorch implementation, our framework can achieve up to 2.13× end-to-end speedup and consume only 0.41× peak memory on the CIFAR10 dataset. Moreover, for the distributed training on the large ImageNet dataset, we can achieve up to 1.81× end-to-end speedup and consume only 0.38× peak memory.

***Keywords*—Neuromorphic Computing, Spiking Neural Network, GPU Optimization, Training Acceleration**

I. Introduction

Spiking neural networks (SNNs) are known as a family of brain-inspired models that learn the behaviors of neural circuits. Taking the advantages of processing dynamic and noisy information, SNNs have been applied in a wide range of tasks such as pattern recognition [1], solving optimization problems [2], robotics [3]. Also, some researchers use SNNs to process sparse data [4] and security-aware tasks [5].

How to train an SNN model with both high accuracy and high efficiency is a critical problem. In the early stage, the local learning algorithms, such as spike timing dependent plasticity (STDP) [6], are widely adopted in the design of neuromorphic chips [2], [7] and programming libraries [8], [9]. However, these rules suffer low accuracy when handling mainstream intelligence tasks and are difficult to scale up to large models, which prevents their applications in practice. To this end, the global learning algorithms inspired by the backpropagation through time (BPTT) emerge in recent years to train SNNs with much higher accuracy [1], [10], [11].

*Corresponding author: Lei Deng.

Although BPTT-based training algorithms boost the accuracy of SNNs, it is inefficient to train an SNN model on GPUs. Compared to the well-optimized training of an artificial neural network (ANN) model on GPU, training an SNN model with BPTT-based algorithms would encounter two challenges: (1) there are more intermediate data; (2) the spatiotemporal dynamics and computational operations are more complex. The first challenge increases the memory consumption, which limits the exploration space of SNN models given the same hardware resources. The second challenge causes frequent GPU kernel launching that dramatically degrades the execution performance. Some researches explore the optimization of *CUDA* codes on GPU for other applications such as graph neural network [12], and Transformer [13]. These studies identify distinct operations for each application, their optimization methods cannot be directly exploited in BPTT-based SNN learning.

In this work, we aim at accelerating the BPTT-based training for SNNs on GPU. We first characterize the training information flow. Then we propose to optimize the dataflow by abandoning the storage of most intermediate data in the forward pass and recomputing them in the backward pass and the parameter update stage. Furthermore, we integrate the spatiotemporal dynamics and design efficient kernel functions with kernel fusion for each training stage to mitigate the heavy kernel launching overhead. Our framework does not include any approximation and can produce the same result as the vanilla Pytorch implementation. Overall, our optimization framework can achieve up to 2.13× speedup & 0.41× peak memory on the CIFAR10 dataset, and 1.81× speedup & 0.38× peak memory on the ImageNet dataset.

II. Preliminaries

A. Spiking Neural Networks

SNNs usually attempt at emulating the behaviors of neural circuits. In this work, we use leaky integrate-and-fire (LIF) [14] to model each neuron. Fig. 1(a) depicts the neuronal behaviours in the forward pass. At each time step, the neuron may produce a spike s and send to the post-synaptic neurons according to its membrane potential u and the $fire$ function. The membrane potential of each neuron is determined by its weighted inputs and its soma status at the previous time step. The detailed neuron model will be provided in the next subsection.

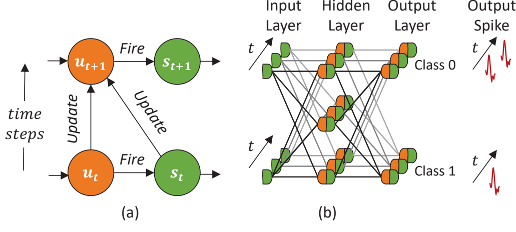

Fig. 1. Illustration of (a) a spiking neuron modeled by LIF and (b) an SNN with spatiotemporal information flow.

An SNN model is displayed in Fig. 1(b). In the forward pass, the information propagate in both temporal and spatial directions. The hidden layers are usually comprised of convolutional layers (Conv), pooling layers (Pool), and fully connected layers (FC), which is similar to ANNs. The final recognition result depends on the spike coding scheme of the last layer. In this work, we adopt the common rate coding that the output neuron fires the most spikes indicates the recognized class.

B. SNN Training via BPTT

In this subsection we give a brief introduction of how to train an SNN through the BPTT-based learning algorithm. There are three stages during training: forward pass (FP), backward pass (BP), and parameter update (PU). In the FP stage, each SNN layer contains three types of operations: Conv/Pool/FC, batch normalization (BN), and LIF dynamics as Fig. 2 shows. The LIF dynamics is governed by

$$\boldsymbol{s}_t^l = fire(\boldsymbol{u}_t^l - th_f), \tag{1}$$

$$\boldsymbol{u}_t^l = \underbrace{\alpha \boldsymbol{u}_{t-1}^l \cdot (1 - \boldsymbol{s}_{t-1}^l)}_{temporal} + \underbrace{\boldsymbol{y}_t^l}_{spatial}, \tag{2}$$

where $\boldsymbol{u}_t^l$ and $\boldsymbol{s}_t^l$ represent the membrane potentials and the spike events of the neurons in the l-th layer at the t-th time step. The neuron would fire a spike and send it to the post-synaptic neurons once the membrane potential larger than the firing threshold th_f. $fire(\cdot)$ is the Heaviside step function, i.e., $fire(x) = 1$ if $x \geq 0$; $fire(x) = 0$ otherwise. The membrane potential is comprised by the temporal part and the spatial part. In the temporal part, if the neuron did not fire a spike in the previous time step, the membrane potential decays by a factor α, otherwise, the it will be reset to 0. The spatial part $\boldsymbol{y}$ is the result after the BN operation, as in Fig. 2. The BN follows

$$\boldsymbol{y}_t^l[j] = \frac{\boldsymbol{x}_t^l[j] - \mu^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}} \gamma^l[j] + \beta^l[j], \tag{3}$$

where $\boldsymbol{x}_t^l[j]$ and $\boldsymbol{y}_t^l[j]$ stand for Conv/Pool/FC and BN results of the j-th channel in the l-th layer at the t-th time step, respectively. The BN [10] is achieved by subtracting the mean $\mu^l[j]$ and dividing the standard deviation $\sqrt{(\sigma^l[j])^2 + \epsilon}$, where $\mu^l[j]$ and $\sigma^l[j]$ are the results from all elements in the j-th channel of $\boldsymbol{x}^l$ across all samples in a batch and time steps. ϵ is a small value to avoid the division error. γ and β are two trainable parameters shared by all neurons in the same channel.

In Eq. (3), $\boldsymbol{x}$ can be the spatial result from a Conv/Pool/FC layer. Taking the Conv result as an example, it is calculated through

$$\boldsymbol{x}_t^l[j] = \sum\nolimits_k \boldsymbol{s}_t^{l-1}[k] * \boldsymbol{w}^l[k, j] + b^l[j]. \tag{4}$$

Conv takes the spike events of the previous layer as inputs, wherein the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ are trainable parameters.

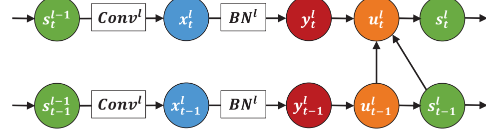

Fig. 2. Illustration of the “Conv→BN→LIF” information flow in the FP stage.

In the BP stage, the gradients propagate along the opposite direction of the arrows in Fig. 2. The gradients of the spikes and the membrane potentials are calculated as

$$\nabla \boldsymbol{s}_t^l[k] = \underbrace{\nabla \boldsymbol{u}_{t+1}^l[k] \cdot (-\alpha \boldsymbol{u}_t^l[k])}_{temporal} + \underbrace{\sum\nolimits_j \nabla \boldsymbol{x}_t^{l+1}[j] * \boldsymbol{w}^{l+1}[k, j]}_{spatial}, \tag{5}$$

$$\nabla \boldsymbol{u}_t^l = \nabla \boldsymbol{u}_{t+1}^l \cdot \alpha (1 - \boldsymbol{s}_t^l) + \nabla \boldsymbol{s}_t^l \cdot fire'(\boldsymbol{u}_t^l). \tag{6}$$

The spike gradients $\nabla \boldsymbol{s}_t^l$ are obtained from both temporal and spatial directions. The temporal part is derived from the temporal part of Eq. (2), while the spatial part is the gradient format of Conv. The membrane potential gradients $\nabla \boldsymbol{u}_t^l$ are acquired by calculating the partial derivative of Eq. (1)-(2). The derivative of the $fire$ function does not exist in principle. We adopt an approximation [1] to simulate the derivative:

$$fire'(u_t^l[i]) \approx \begin{cases} \eta, & th_l < u_t^l[i] < th_r, \\ 0, & otherwise, \end{cases} \tag{7}$$

where η is a decay factor.

From Eq. (2), we can find $\nabla \boldsymbol{u}_t^l = \nabla \boldsymbol{y}_t^l$. Based on $\nabla \boldsymbol{y}_t^l$, we can get the gradient of the Conv result $\nabla \boldsymbol{x}_t^l$ through

$$\nabla \boldsymbol{x}_t^l[j] = \frac{\gamma^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}} \left(\nabla \boldsymbol{y}_t^l[j] - \frac{\boldsymbol{I}}{n} \sum_{t, n_j} \nabla y_t^l[n_j] - \frac{\hat{\boldsymbol{x}}_t^l[j]}{n} \sum_{t, n_j} \nabla y_t^l[n_j] \hat{x}_t^l[n_j] \right), \tag{8}$$

$$\hat{\boldsymbol{x}}_t^l[j] = \frac{\boldsymbol{x}_t^l[j] - \mu^l[j]}{\sqrt{(\sigma^l[j])^2 + \epsilon}}, \tag{9}$$

where $\nabla \boldsymbol{x}_t^l$ is comprised of the partial derivative of $\boldsymbol{y}_t^l$ with respect to $\boldsymbol{x}_t^l$, $\boldsymbol{\mu}^l$, and $\boldsymbol{\sigma}^l$ in Eq. (4), which corresponds to the three terms in the brackets of Eq. (8). Since the neurons in the same channel share identical mean and standard deviation values, n_j indicates the neurons in the jth channel and n denotes the total number of neurons in the jth channel.

In the PU stage, we first update the trainable parameters of the BN operation following

$$\nabla \gamma^l[j] = \sum\nolimits_{t, n_j} \nabla y_t^l[n_j] \hat{x}_t^l[n_j], \tag{10}$$

$$\nabla \beta^l[j] = \sum\nolimits_{t, n_j} \nabla y_t^l[n_j]. \tag{11}$$

The above parameter update is derived from Eq. (3). The parameter update of Conv is governed by

$$\nabla \boldsymbol{w}^l[k,j] = \sum\nolimits_t \nabla \boldsymbol{x}_t^l[j] * \boldsymbol{s}_t^{l-1}[k], \tag{12}$$

$$\nabla b^l[j] = \sum\nolimits_{t,n_j} \nabla x_t^l[n_j]. \tag{13}$$

C. Motivation

Currently, most of the BPTT-based SNN studies are simulated on GPU with the Pytorch programming environment [1], [10]. We use the vanilla Pytorch implementation [10] to characterize the training performance. Fig. 3 presents the latency and peak memory consumption for training an SNN model with one epoch on the CIFAR10 dataset under different model settings. The network size is medium and the structure is provided in Tab. I. The memory consumption for feature maps (i.e., $\boldsymbol{s}$, $\boldsymbol{u}$, $\boldsymbol{x}$, $\boldsymbol{y}$) at each time step is 71.13 MB. $Net.\ with\ LIF + BN$ and $Net.\ with\ LIF$ denote the networks with and without BN. In order to estimate the impact of the $fire$ function and the membrane potential calculation (i.e., LIF dynamics), we additionally design a network with only Conv/FC layers, which is denoted as $Baseline$. The $Baseline$ network is functionally incorrect and we just use it for comparison.

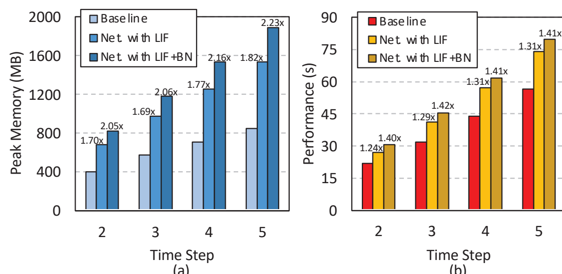

Fig. 3. Characterization of the vanilla Pytorch implementation of SNN training with one epoch on CIFAR10: (a) peak memory consumption; (b) latency.

Fig. 3(a) compares the peak memory consumption during training. Since the peak memory consumption determines how many GPUs are needed, a lower memory consumption can help reduce the resource overhead. It can be seen that incorporating the LIF operation would increase the peak memory consumption by $1.69 \sim 1.82\times$. The increased memory consumption comes from $\boldsymbol{u}$ and $\boldsymbol{s}$. The further incorporation of BN consumes $2.05 \sim 2.23\times$ peak memory compared to the baseline, due to the extra storage of $\boldsymbol{y}$. **We find that the intermediate data consume lots of memory, which finally impacts design space exploration for SNN models**. Fig. 3(b) further compares the training latency. Compared to the baseline, the LIF dynamics increases the training latency by $1.24 \sim 1.31\times$; the further incorporation of BN consumes $1.40 \sim 1.42\times$ latency compared to the baseline. **From the result, the computation latency for the operations besides Conv/FC cannot be ignored.**

Given the above observations, we find that the optimization of LIF and BN operations during SNN training is valuable.

III. Approach

In this section we first present how to optimize the BPTT-based SNN training dataflow for saving the storage consumption. Then we detail our GPU kernel design to fuse arithmetic operations for eliminating frequent kernel launching.

A. Dataflow Optimization

In SNN training, many intermediate variables are generated and stored in the FP stage, which are accessed in BP and PU stages. Whereas, this causes huge memory consumption. In order to reduce the memory overhead, we propose to only store parts of intermediate data in the FP stage, and recompute the missing ones in BP and PU stages.

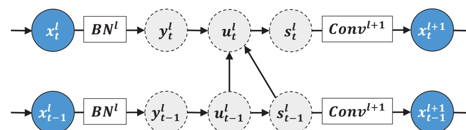

Fig. 4. The optimized information flow that only stores $\boldsymbol{x}$ in the FP stage.

With the above idea, we choose to save Conv (or FC) results (i.e., $\boldsymbol{x}$) as in Fig. 4. In this example, BN results, membrane potentials, and spike events are all abandoned in the FP stage. The reason that we select Conv results to store is that Conv takes most of computations, and thus recomputing Conv results may cause long training latency. Although recomputing other intermediate data needs complex arithmetic computations, all of them are element-wise that occupy fewer workloads.

The goal of BP and PU stages are to calculate the gradients of Conv results (i.e., $\nabla \boldsymbol{x}^l$), the gradients of Conv parameters (i.e., $\nabla \boldsymbol{w}^{l+1}$ & $\nabla \boldsymbol{b}^{l+1}$) and BN parameters (i.e., $\nabla \boldsymbol{\gamma}^l$ & $\nabla \boldsymbol{\beta}^l$). Three steps can be summarized in BP and PU stages as Fig. 5. In the first step, we recompute $\boldsymbol{u}^l$ & $\boldsymbol{s}^l$, which are abandoned in the FP stage, as in Fig. 5(a). Then, according to the recomputed $\boldsymbol{s}^l$ and the backpropagated $\nabla \boldsymbol{x}^{l+1}$, we get the gradients of Conv parameters in layer $l+1$. Also, we get the spatial part of $\nabla \boldsymbol{s}^l$ in Eq. (5), as in Fig. 5(b). In the last step, we compute $\nabla \boldsymbol{x}^l$ and the gradients of BN parameters, as in Fig. 5(c). The detailed algorithms for each step will be provided in the next subsection.

B. Kernel Optimization for BP and PU Stages

Except for the huge memory consumption in SNN training, the complex LIF dynamics and BN in the vanilla Pytorch implementation degrade the performance dramatically due to the frequently kernel launching. In this subsection, we explain how to optimize the GPU kernel functions for those operations to accelerate the gradient calculation in BP and PU stages.

Before providing the detailed design of kernel functions, We first describe the basic thread hierarchy of the thread model in GPU kernels. On GPU, threads are organized into blocks. We use 'block Dim' to identify the number of threads in a block and use 'thread ID' to specify a thread in a given block. Then, we define 'block ID' as the location of a block in a program.

Since we abandoned the storage of membrane potentials and spike events in the FP stage, we need to recompute $\boldsymbol{u}^l$ & $\boldsymbol{s}^l$ based on $\boldsymbol{x}^l$ in BP and PU stages as shown in Fig. 5(a). The detailed kernel function is provided in Alg. 1. Here, we fuse all arithmetic operations into one kernel function to avoid the frequent kernel launching. The update of membrane potentials

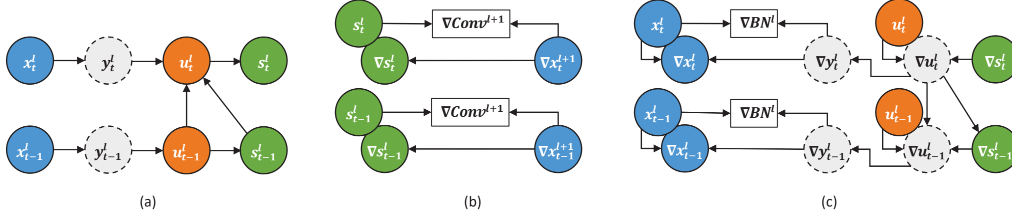

Fig. 5. The three steps in BP and PU stages: (a) recompute $\boldsymbol{u}^l$ and $\boldsymbol{s}^l$ (Alg. 1); (b) compute the gradients of Conv parameters (i.e., $\nabla \boldsymbol{w}^l$ & $\nabla \boldsymbol{b}^l$) and the spatial part of $\nabla \boldsymbol{s}^l$ (Alg. 4); (c) compute the gradients of BN parameters (i.e., $\nabla \boldsymbol{\gamma}^l$ & $\nabla \boldsymbol{\beta}^l$) and the gradients of Conv results (i.e., $\nabla \boldsymbol{x}^l$) (Alg. 2 & 3).

Algorithm 1: Fused FP Kernel with BN

Data: $\boldsymbol{x} \in \mathbb{R}^{T\times B\times C\times H\times W}$; $\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^C$;
Result: $\boldsymbol{u}, \boldsymbol{s} \in \mathbb{R}^{T\times B\times C\times H\times W}$;
1 **begin**
2 $\quad \mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$; $\gamma_{tmp} = \gamma[j]$; $\beta_{tmp} = \beta[j]$;
3 $\quad$ **for** $b \leftarrow 1$ *to* B **do**
4 $\quad\quad u_{pre} = 0$; $s_{pre} = 0$;
5 $\quad\quad$ **for** $t \leftarrow 1$ *to* T **do**
6 $\quad\quad\quad$ // Eq. (2) & (3)
7 $\quad\quad\quad u_{pre} = \alpha u_{pre}(1 - s_{pre}) + \frac{x[idx_n] - \mu_{tmp}}{\sqrt{\sigma^2_{tmp} + \epsilon}} \gamma_{tmp} + \beta_{tmp}$;
8 $\quad\quad\quad$ // Eq. (1)
9 $\quad\quad\quad s_{pre} = u_{pre} \geq th_f$;
10 $\quad\quad\quad u[idx_n] = u_{pre}$; $s[idx_n] = s_{pre}$;
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 **end**

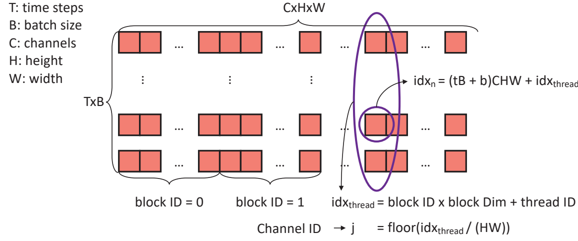

Fig. 6. Correspondence between the feature map and threads for the kernel programming. The feature map is stored with the row major order.

and spike events are dominated by Eq. (1)-(3). The thread model for GPU kernels is presented in Fig. 6. We first reshape the tensor from the size of $T \times B \times C \times H \times W$ to a 2D format with the size of $TB \times CHW$. With this reshaping, each thread processes $T \times B$ elements and the total number of threads is CHW. During the processing of the kernel function, we need to know the channel ID (i.e., j) for each thread to get the corresponding parameters for BN. We also need to know the neuron's location (i.e., idx_n) in the original tensor to write back the calculated results. In Alg. 1, each thread calculates $\boldsymbol{u}^l$ and $\boldsymbol{s}^l$ for different time steps and input samples in a batch, in which $\mu^l[j]$, $\sigma^l[j]$, $\gamma^l[j]$, and $\beta^l[j]$ keep unchanged under the same j. During the kernel processing, $\boldsymbol{u}^l$ and $\boldsymbol{s}^l$ are calculated along the time steps from 1 *to* T, since the calculation at the current time step has data dependency on the previous time step according to Eq. (2). Notice that this fused kernel function can be directly applied in the FP stage to compute the spike events.

After we obtain $\boldsymbol{s}^l$, we can compute the spatial part of $\nabla \boldsymbol{s}^l$ and $\nabla \boldsymbol{w}^l$ & $\nabla \boldsymbol{b}^l$ as Fig. 5(b) by calling the functions in *cuDNN*. Then, we need to compute $\nabla \boldsymbol{x}^l$ and $\nabla \boldsymbol{\gamma}^l$ & $\nabla \boldsymbol{\beta}^l$ as shown in

Algorithm 2: Fused PU Kernel with BN

Data: $\boldsymbol{x}, \boldsymbol{u}, \nabla \boldsymbol{s}_{spatial} \in \mathbb{R}^{T\times B\times C\times H\times W}$; $\boldsymbol{\mu}, \boldsymbol{\sigma} \in \mathbb{R}^C$;
Result: $\nabla \boldsymbol{y}_{sum}, \nabla \hat{\boldsymbol{y}}_{sum} \in \mathbb{R}^{C\times H\times W}$;
1 **begin**
2 $\quad \mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$;
3 $\quad$ **for** $b \leftarrow 1$ *to* B **do**
4 $\quad\quad \nabla u_{next} = 0$;
5 $\quad\quad$ **for** $t \leftarrow T$ *to* 1 **do**
6 $\quad\quad\quad$ // Eq. (5) & (7)
7 $\quad\quad\quad$ **if** $th_l < u[idx_n] < th_r$ **then**
8 $\quad\quad\quad\quad \nabla s_{tmp} = \nabla s_{spatial}[idx_n] - \nabla u_{next}(\alpha u[idx_n])$;
9 $\quad\quad\quad$ **end**
10 $\quad\quad\quad$ **else**
11 $\quad\quad\quad\quad \nabla s_{tmp} = 0$;
12 $\quad\quad\quad$ **end**
13 $\quad\quad\quad$ // Eq. (6)
14 $\quad\quad\quad \nabla u_{next} = \nabla u_{next} \alpha (u[idx_n] < th_f) + \eta \nabla s_{tmp}$;
15 $\quad\quad\quad$ // Eq. (9)
16 $\quad\quad\quad \hat{x}_{tmp} = \frac{x[idx_n] - \mu_{tmp}}{\sqrt{\sigma^2_{tmp} + \epsilon}}$;
17 $\quad\quad\quad$ // Eq. (8), (10) & (11)
18 $\quad\quad\quad \nabla y_{sum}[idx_{thread}] += \nabla u_{next}$;
19 $\quad\quad\quad \nabla \hat{y}_{sum}[idx_{thread}] += \nabla u_{next} \hat{x}_{tmp}$;
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 **end**

Algorithm 3: Fused BP Kernel with BN

Data: $\boldsymbol{x}, \boldsymbol{u}, \nabla \boldsymbol{s}_{spatial} \in \mathbb{R}^{T\times B\times C\times H\times W}$;
$\boldsymbol{\mu}, \boldsymbol{\sigma}, \boldsymbol{\gamma}, \nabla \boldsymbol{y}_{mean}, \nabla \hat{\boldsymbol{y}}_{mean} \in \mathbb{R}^C$;
Result: $\nabla \boldsymbol{x} \in \mathbb{R}^{T\times B\times C\times H\times W}$;
1 **begin**
2 $\quad \mu_{tmp} = \mu[j]$; $\sigma_{tmp} = \sigma[j]$; $\gamma_{tmp} = \gamma[j]$;
3 $\quad \nabla y_{mean_tmp} = \nabla y_{mean}[j]$; $\nabla \hat{y}_{mean_tmp} = \nabla \hat{y}_{mean}[j]$
4 $\quad$ **for** $b \leftarrow 1$ *to* B **do**
5 $\quad\quad$ **for** $t \leftarrow 1$ *to* T **do**
6 $\quad\quad\quad$ Repeat line 7∼19 in Alg. 2 to get ∇u_{next} and $\hat{x}_{tmp}$;
7 $\quad\quad\quad$ // Eq. (8)
8 $\quad\quad\quad \nabla x[idx_n] = \frac{\gamma_{tmp}}{\sqrt{\sigma^2_{tmp} + \epsilon}} (\nabla u_{next} - \nabla y_{mean_tmp} - \hat{x}_{tmp} \nabla \hat{y}_{mean_tmp})$;
9 $\quad\quad$ **end**
10 $\quad$ **end**
11 **end**

Fig. 5(c). Based on Eq. (8), in order to get $\nabla \boldsymbol{x}^l$, we need to get the sum of $\nabla \boldsymbol{y}^l \hat{\boldsymbol{x}}^l$ and $\nabla \boldsymbol{y}^l$ for each channel first, which are $\nabla \boldsymbol{\gamma}^l$ and $\nabla \boldsymbol{\beta}^l$ as Eq. (10)-(11). Thus, we first compute the gradients of BN parameters as the intermediate data in BP and PU stages, as given in Alg. 2. Based on Eq. (6), the gradients of membrane potentials at the current time step has data dependency on the next time step, therefore $\nabla \boldsymbol{u}^l$ is calculated along time steps from T *to* 1, which is opposed to that in Alg. 1. The thread model in Alg. 2 is also based on Fig. 6.

At last, we can obtain $\nabla \boldsymbol{x}^l$ according to Alg. 3. In order

to avoid extra memory consumption, we do not store the gradients of BN results (i.e., $\nabla \boldsymbol{y}^l$). Therefore, in Alg. 3, we recompute $\nabla \boldsymbol{u}^l$ again, and go through the time steps from T *to* 1 considering the data dependency in the calculation of $\nabla \boldsymbol{u}$.

After going through all the required kernels in BP and PU stages, we give the overall function in Alg. 4. The inputs include the Conv results of the current layer (i.e., $\boldsymbol{x}^l$), gradients of Conv results of the next layer (i.e., $\nabla \boldsymbol{x}^{l+1}$), Conv and BN parameters. The outputs contain the gradients of Conv results (i.e., $\nabla \boldsymbol{x}^l$), gradients of Conv and BN parameters.

Algorithm 4: PU & BP *CUDA*

Data: $\boldsymbol{x}^l, \boldsymbol{\mu}^l, \boldsymbol{\sigma}^l, \boldsymbol{\gamma}^l, \boldsymbol{\beta}^l, \boldsymbol{w}^{l+1}, \nabla \boldsymbol{x}^{l+1}$;
Result: $\nabla \boldsymbol{x}^l, \nabla \boldsymbol{\gamma}^l, \nabla \boldsymbol{\beta}^l, \nabla \boldsymbol{w}^{l+1}, \nabla \boldsymbol{b}^{l+1}$;

1 **begin**
2 // Apply FP to recompute $\boldsymbol{u}^l$ & $\boldsymbol{s}^l$
3 $\boldsymbol{u}^l, \boldsymbol{s}^l \leftarrow$ Alg. 1 ($\boldsymbol{x}^l, \boldsymbol{\mu}^l, \boldsymbol{\sigma}^l, \boldsymbol{\gamma}^l, \boldsymbol{\beta}^l$);
4 // Apply PU to get gradients of parameters and intermediate data
5 // Eq. (12), (13), and (5)
6 $\nabla \boldsymbol{w}^{l+1} = \nabla \boldsymbol{x}^{l+1} * \boldsymbol{s}^l$; $\nabla \boldsymbol{b}^{l+1} = \sum \nabla \boldsymbol{x}^{l+1}$;
7 $\nabla \boldsymbol{s}^l_{spatial} = \nabla \boldsymbol{x}^{l+1} * \boldsymbol{w}^{l+1}$;
8 $\nabla \boldsymbol{y}^l_{sum}, \nabla \hat{\boldsymbol{y}}^l_{sum} \leftarrow$ Alg. 2 ($\boldsymbol{x}^l, \boldsymbol{u}^l, \nabla \boldsymbol{s}^l_{spatial}, \boldsymbol{\mu}^l, \boldsymbol{\sigma}^l$);
9 // Eq. (10) & (11)
10 $\nabla \boldsymbol{\gamma}^l, \nabla \boldsymbol{\beta}^l \leftarrow$ sum of $\nabla \hat{\boldsymbol{y}}^l_{sum}, \nabla \boldsymbol{y}^l_{sum}$;
11 $\nabla \hat{\boldsymbol{y}}^l_{mean}, \nabla \boldsymbol{y}^l_{mean} \leftarrow \nabla \boldsymbol{\gamma}^l/(TBHW), \nabla \boldsymbol{\beta}^l/(TBHW)$;
12 // Apply BP to get $\nabla \boldsymbol{x}^l$
13 $\nabla \boldsymbol{x}^l \leftarrow$ Alg. 3 ($\boldsymbol{x}^l, \boldsymbol{u}^l, \nabla \boldsymbol{s}^l_{spatial}, \boldsymbol{\mu}^l, \boldsymbol{\sigma}^l, \boldsymbol{\gamma}^l, \nabla \hat{\boldsymbol{y}}^l_{mean}, \nabla \boldsymbol{y}^l_{mean}$);
14 **end**

In order to make our optimization easy-to-deploy, we design Pytorch interfaces[1] which can also support distributed learning.

IV. Evaluation

A. Experimental Setup

In this work, we focus on the widely adopted image recognition tasks for evaluation. Our experiments are implemented on two popular datasets: CIFAR10 and ImageNet. The network configurations are detailed in Tab. I. For the CIFAR10 dataset, we design three models with different number of Conv layers; for the imageNet dataset, we adopt an AlexNet-like network and ResNet34 to evaluate our optimization framework. We use a vanilla Pytorch implementation [10] for the BPTT-based SNN training as the baseline which contains BN operations.

B. Performance Analysis on CIFAR10

We first analyze the improvement of our optimization framework on the CIFAR10 dataset. Fig. 7(a) shows the comparison without BN. From the results, our optimization framework consumes much less memory for larger models, since the non-optimized functions (e.g., data processing) for small networks consume a large portion of memory. The peak memory consumption under different number of time steps presents little variance, since the portion of saved memory for each layer does not change. For the training latency, our optimization achieves higher speedup as the number of time steps increases. The reason is that our kernel functions fuse the neural dynamics at all time steps. On the contrary, the vanilla Pytorch updates the neural activities step by step which causes more kernel launching time. For the models with different size, the speedup values are similar, indicating that our optimization can provide a considerable speedup no matter what the network structure is. Overall, for the models without BN, we can achieve maximum $2.13\times$ end-to-end speedup and consume only $0.41\times$ peak memory when compared to the vanilla Pytorch implementation.

After involving the BN layer, our framework achieves less speedup but more memory saving as Fig. 7(b). The reason is that the BN layer introduces more intermediate data recompute. Overall, for the models with BN, we can achieve maximum $1.94\times$ end-to-end speedup and consume only $0.33\times$ peak memory when compared to the vanilla Pytorch implementation.

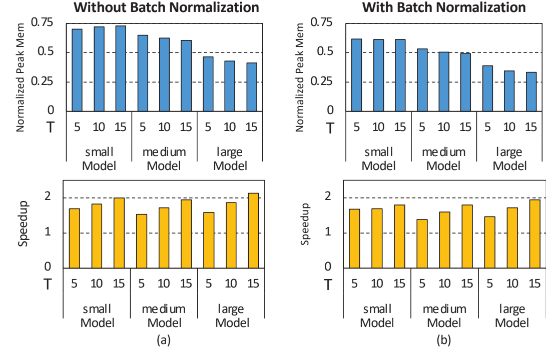

Fig. 7. Peak memory consumption and performance comparison between our optimization framework and the vanilla Pytorch implementation: (a) without BN; (b) with BN.

We further analyze the breakdown of memory consumption and computation time in Fig 8. The vanilla Pytorch spends a higher portion of memory on feature map storage and takes a lower portion of time for the kernel execution. This phenomena demonstrates the efficiency of our framework in saving memory consumption and preventing frequent kernel launching.

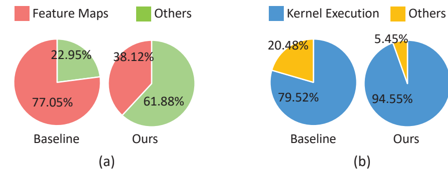

Fig. 8. Comparison of the breakdown of (a) memory consumption and (b) computation time between the vanilla Pytorch implementation and our optimization framework for the medium-size model with BN and $T = 10$.

C. Comparison with TorchScript

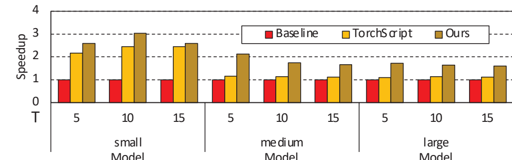

Fig. 9. Performance comparison in the FP stage between our optimization framework and the Pytorch optimization with TorchScript on Cifar10.

The vanilla Pytorch implementation for the FP stage can be easily optimized through TorchScript [15]. We compare

[1] https://github.com/liangling76/snn_gpu_training_bptt

TABLE I

MODEL CONFIGURATIONS. AP-AVERAGE POOLING; mC-A CONV LAYER THAT HAS m OUTPUT CHANNELS; I^0-INPUT; I^l-FEATURE MAP IN LAYER l.

Dataset	Model Size	Accuracy	Network Structure	T	B	α	η	th_f	(th_l, th_r)
Cifar10	Small	83.17%	I^0 - 128C - 256C - 512C - 1024FC - 512FC - 10FC	10	64	0.25	1.0	0.25	(-0.25, 0.75)
	Medium	88.21%	I^0 - 128C - 128C - 256C - 256C - 512C - 512C - 1024FC - 512FC - 10FC						
	Large	90.17%	I^0 - 128C - 128C - 128C - 256C - 256C - 256C - 512C - 512C - 512C - 1024FC - 512FC - 10FC						
ImageNet	AlexNet	54.94%	I^0 - 64C - 128C - 256C - 256C - 384C - 256C - 256C - 4096FC - 4096FC - 1000FC	6	16				
	ResNet34	60.52%	I^0 - 64C - ⟨ 64B0 - 2×64B1 ⟩ - ⟨ 128B0 - 3×128B1 ⟩ - ⟨ 256B0 - 5×256B1 ⟩ - ⟨ 512B0 - 2×512B1 ⟩ - AP - 1000FC mB0: ⟨ I^l - mC - mC ⟩ + ⟨ I^l - mC ⟩; mB1: ⟨ I^l - mC - mC) + ⟨ I^l ⟩						

our framework with TorchScript in Fig. 9. From the results, both frameworks achieve high speedup for small models, since a lower portion of time is spent on Conv/FC layers. Our framework and Torchscript fuse operations in one layer along the temporal and spatial directions, respectively. For larger models, our framework achieves a higher speedup that indicates fusing operations along the temporal direction is more efficient.

D. Performance Analysis on ImageNet

Finally, we analyze our optimization framework on the ImageNet dataset with distributed learning. The comparison to the vanilla Pyotrch implementation is shown in Fig. 10. For the peak memory consumption per GPU, we find that the vanilla Pytorch implementation needs much more storage in distributed learning compared to the single-GPU training. The reason is that the vanilla Pytorch implementation induces extra memory overhead for some intermediate data such as $\hat{\boldsymbol{x}}_t^l$. In our framework, the additional storage consumption only comes from the calculation of the average parameter gradients between GPUs. Compared to the vanilla Pytorch impelementation, our optimization framework only consumes 0.68× and 0.38× peak memory for ResNet34 and AlexNet, respectively.

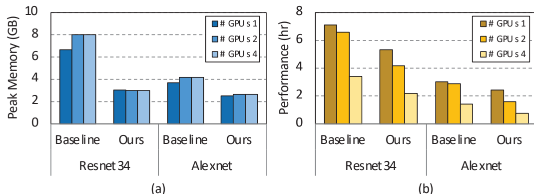

Fig. 10. Comparison between our optimization framework and the vanilla Pytorch implementation for distributed learning on the ImageNet dataset: (a) peak memory consumption per GPU; (b) training latency for one epoch.

Compared to the single-GPU training, both frameworks achieve insignificant speedup when using two GPUs. The reason is that, the distributed training needs additional communication between GPUs, such as parameter synchronization. When further scaling up to more GPUs, the latency is dramatically reduced due to the enhanced computing power but the similar communication burden for each GPU compared to the case with two GPUs. From the simulation results, our optimization framework can achieve maximum 1.58× and 1.81× end-to-end speedup for ResNet34 and Alexnet, respectively, when compared to the vanilla Pytorch implementation.

V. CONCLUSION

In this work, we design an framework that accelerates BPTT-based SNN training on GPU. We first optimize the dataflow by only storing the Conv results in the FP stage to reduce the memory consumption. Then we customize kernel functions to alleviate frequent kernel launching. We also provide Pytorch interfaces to make our framework easy-to-deploy. In the future, we would like to incorporate more SNN models and learning algorithms into our optimization framework.

REFERENCES

[1] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, 2018.

[2] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[3] A. R. Vidal, H. Rebecq, T. Horstschaefer, and D. Scaramuzza, "Ultimate slam? combining events, images, and imu for robust visual slam in hdr and high-speed scenarios," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 994–1001, 2018.

[4] W. He, Y. Wu, L. Deng, G. Li, H. Wang, Y. Tian, W. Ding, W. Wang, and Y. Xie, "Comparing snns and rnns on neuromorphic vision datasets: similarities and differences," *Neural Networks*, vol. 132, pp. 108–120, 2020.

[5] L. Liang, X. Hu, L. Deng, Y. Wu, G. Li, Y. Ding, P. Li, and Y. Xie, "Exploring adversarial attack in spiking neural networks with spike-compatible gradient," *arXiv preprint arXiv:2001.01587*, 2020.

[6] S. Song, K. D. Miller, and L. F. Abbott, "Competitive hebbian learning through spike-timing-dependent synaptic plasticity," *Nature neuroscience*, vol. 3, no. 9, pp. 919–926, 2000.

[7] E. Baek, H. Lee, Y. Kim, and J. Kim, "Flexlearn: fast and highly efficient brain simulations using flexible on-chip learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 304–318, 2019.

[8] P. Qu, Y. Zhang, X. Fei, and W. Zheng, "High performance simulation of spiking neural network on gpgpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2510–2523, 2020.

[9] M. Stimberg, D. F. Goodman, V. Benichoux, and R. Brette, "Equation-oriented specification of neural models for simulations," *Frontiers in neuroinformatics*, vol. 8, p. 6, 2014.

[10] H. Zheng, Y. Wu, L. Deng, Y. Hu, and G. Li, "Going deeper with directly-trained larger spiking neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 11062–11070, 2021.

[11] L. Liang, Z. Qu, Z. Chen, F. Tu, Y. Wu, L. Deng, G. Li, P. Li, and Y. Xie, "H2learn: High-efficiency learning accelerator for high-accuracy spiking neural networks," *arXiv preprint arXiv:2107.11746*, 2021.

[12] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie, "fusegnn: accelerating graph convolutional neural network training on gpgpu," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2020.

[13] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "Turbotransformers: An efficient gpu serving system for transformer models," *arXiv preprint arXiv:2010.05680*, 2020.

[14] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[15] PyTorchContributors., "Torch script. https://pytorch.org/docs/ master/jit.html.," *Accessed: 2018-09-24.*, 2018.