Temporal Backpropagation for Spiking Neural Networks (SNNs) in CUDA

Vineet Kotariya
Department of Electrical Engineering
IIT-Bombay
Mumbai, India
vineetkotariya@gmail.com

Tarang Jain
Department of Aerospace Engineering
IIT-Bombay
Mumbai, India
170010005@iitb.ac.in

***Abstract*—Spiking Neural Networks (SNNs) are brain inspired networks that try to mimic biological information processing to perform tasks like classification in a highly sparse and power efficient manner. However, training and simulating these networks is extremely time consuming. This is because of an additional temporal domain as compared to traditional neural networks. However, there is a lot of scope for parallelizing the computations involved in training SNNs, both in the spatial and temporal domain. However, there is a lack of a generalized GPU-based training framework for SNNs which can exploit this. We propose a CUDA C based implementation of an SNN which uses a form of rank-order temporal coding called time-to-first-spike coding to enable GPU-based training for such SNNs. This approach reduces the simulation time by a factor of 358 as compared to a serial implementation and by a factor of 32 as compared to an OpenMP implementation using 16 threads.**

***Index Terms*—Spiking Neural Networks, CUDA, OpenMP**

I. Introduction

Spiking Neural Networks (SNNs) are widely touted as the next generation of neural networks. These are biologically-inspired networks that process information in terms of spikes (inspired by action potentials in brain) instead of analog values like traditional neural networks. SNNs, when implemented on on neuromorphic hardware, exhibit extremely low power consumption, fast inference, and sparse and event-driven information processing [1]. However, these spiking activation functions are non-differentiable which makes training them extremely tricky. Before mapping these networks to neuromorphic hardware, the algorithms have to be tested and simulated. However, there is a lack of a generalized GPU based framework for such simulations which leads to extremely large training time. We have come up with a CUDA based framework for training fully-connected temporal SNNs which significantly reduces the simulation time. We compare the time required with a serial and OpenMP based implementation of the same network and demonstrate the excellent speedup achieved.

II. Theoretical Background

We have used the SNN model (sec. II) described in [2], [3].

A. Neuronal Dynamics and Coding

Our network consists of two layers of non-leaky Integrate and Fire (IF) neurons. The membrane potential of the j^{th} neuron in the l^{th} layer at time t is given by:

$$V_j^l(t) = \sum_i w_{ji}^l \sum_{\tau=1}^{t} S_i^{l-1}(\tau) \tag{1}$$

where S_i^{l-1} and $w^l ji$ are the input spike train and the input synaptic weight from the i^{th} neuron in the previous layer to neuron j respectively. The IF neuron emits a spike the first time its membrane potential reaches the threshold, θ_j^l,

$$S_j^l(t) = \begin{cases} 1 & \text{if } V_j^l(t) > \theta_j^l \ \ \& \ \ S_j^l(< t) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For information processing, we use a special kind of temporal coding called time-to-first-spike (TTFS) coding. All the information is encoded in the time of the first spike of a neuron. To ensure this the input has to be similarly encoded. For a gray image with pixel intensity in the range [0,I_{max}] for a network with simulation time t_{max}, the firing time of the i^{th} input, t_i is given by:

$$t_i = \left(\frac{I_{max} - I_i}{I_{max}} \right) t_{max} \tag{3}$$

where I_i is the intensity of the i^{th} pixel. So, the spike train

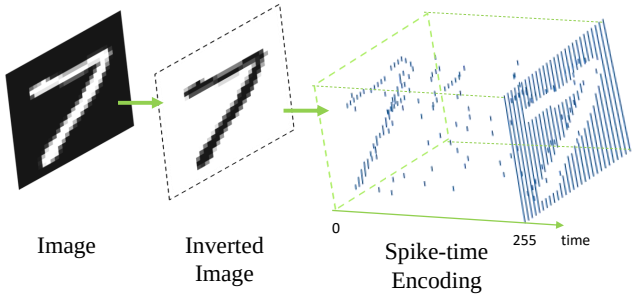

Fig. 1. Input Encoding (using TTFS coding). [3]

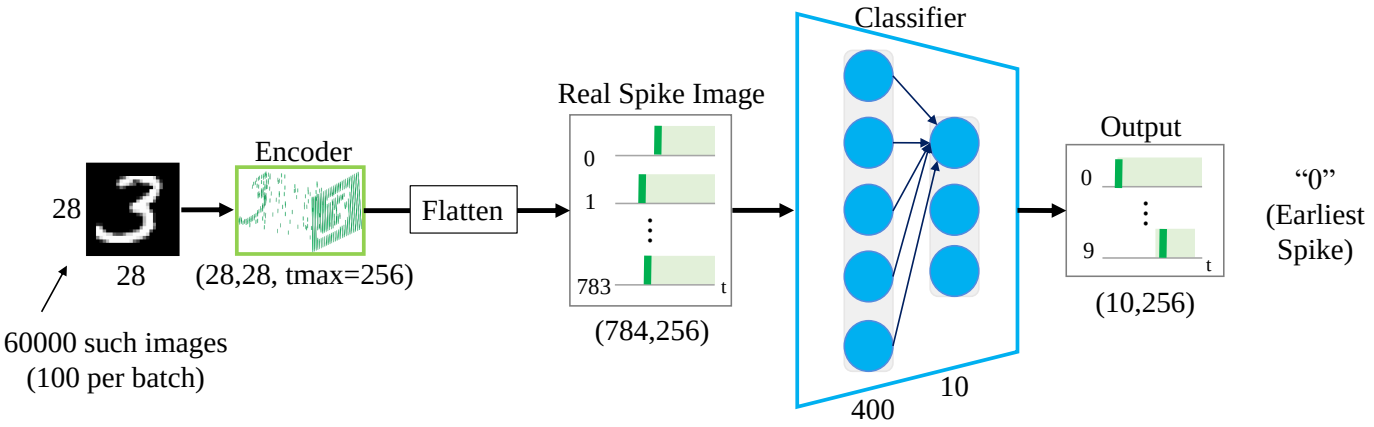

Fig. 2. Our Network Architecture.

of the i^{th} input is given by:

$$S_i^0(t) = \begin{cases} 1 & \text{if } t = t_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The input encoding is demonstrated in fig.1. Presynaptic neurons that spike earlier have a bigger impact on a neuron that input neurons that spike later. So, a neuron which has input spike trains with early spikes, in turn spikes early. This ensures that the encoding is preserved for the deeper layers. This in turn means that the network decision depends on these earliest spikes. For our classification task, in the output layer, the neuron that spikes the earliest, is taken to be the class prediction.

B. Network Architecture

We use the MNIST handwritten digit dataset to evaluate our model. We have trained a 2-layer fully-connected (dense) network with 400 hidden neurons and 10 output neurons. The network is depicted in 2. We flatten the 28x28 MNIST images after TTFS encoding before feeding it to the network.

C. Temporal Backpropagation

We define $e = [e_1, ..., e_c]$ as the temporal error function.

$$e_j = \frac{T_j^o - t_j^o}{t_{max}} \tag{5}$$

where T_j^o ans t_j^o are the target and actual firing times of the j^{th} neuron respectively. The loss fuction of the network is defined as:

$$L = \frac{1}{2}||e||^2 \ = \frac{1}{2}\sum_{j=1}^{c} e_j^2 \tag{6}$$

For training the network we use stochastic gradient descent. We update w_{ji}^l, the weight for the connection from the i^{th} neuron of the $(l-1)^{th}$ layer to the j^{th} neuron of the l^{th} layer as follows: (η is the learning rate)

$$w_{ji}^l = w_{ji}^l - \eta \frac{\delta L}{\delta w_{ji}^l} \tag{7}$$

Now, the IF neuron activation function is not differentiable. However, it approximates ReLU. The ReLU activation function and it's derivative are given by eq.8,10.

$$y_j^l = \max\left(0, z_j^l = \sum_i w_{ji}^l x_i^{l-1}\right) \tag{8}$$

So, a large output value y_j^l corresponds to a large input z_j^l (large $x_i^{(l-1)}$ coupled with a large corresponding synaptic weight w_{ji}^l). Equivalently in TTFS coding, larger input values correspond to earlier spikes and so early input spikes with strong synaptic weights would mean that the neuron fires earlier too. So, we can assume an equivalence relation between the ReLU output , y_j^l, and the firing time of t_j^l, a TTFS IF neuron:

$$y_j^l \sim t_{max} - t_j^l \tag{9}$$

The ReLU derivative is given by:

$$\frac{\delta y_j^l}{\delta w_{ji}^l} = \frac{\delta y_j^l}{\delta z_j^l}\frac{\delta z_j^l}{\delta w_{ji}^l} = \begin{cases} x_i^{l-1} & \text{if } y_j^l > 0 \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

From eq.9, we can approximate:

$$\frac{\delta t_j^l}{\delta V_j^l} = \begin{cases} -1 & \text{if } t_j^l < t_{max} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

From eq.1 and eq.11, we have:

$$\frac{\delta t_j^l}{\delta w_{ji}^l} = \frac{\delta t_j^l}{\delta V_j^l}\frac{\delta V_j^l}{\delta w_{ji}^l} = \begin{cases} -\sum_{\tau=1}^{t_j^l} S_i^{l-1}(\tau) & \text{if } t_j^l < t_{max} \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

So, from eq. 5,6,12:

$$\frac{\delta L}{\delta t_j^l} = \frac{\delta L}{\delta t_j^l}\frac{\delta t_j^l}{\delta w_{ji}^l} = \begin{cases} -\sum_{\tau=1}^{t_j^l} S_i^{l-1}(\tau) & \text{if } t_j^l < t_{max} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

where $\delta_j^l = \delta L / \delta t_j^l$. For the output layer:

$$\delta_j^{out} = \frac{\delta L}{\delta e_j^l}\frac{\delta e_j^l}{\delta t_j^{out}} = -e_j \tag{14}$$

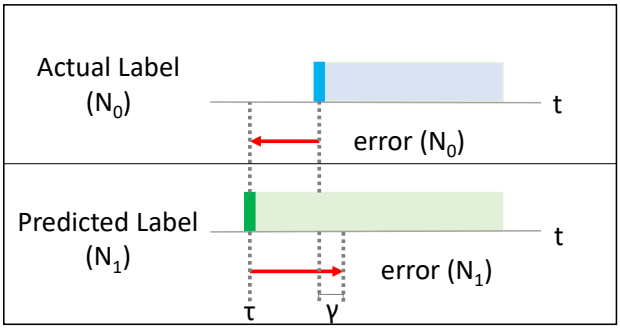

Fig. 3. Margin-based Temporal Loss function

We backpropagate the error for the hidden layers:

$$\delta_j^l = \sum_k \left(\frac{\delta L}{\delta t_j^{l+1}} \frac{\delta t_j^{l+1}}{\delta V_k^{l+1}} \frac{\delta V_k^{l+1}}{\delta t_j^l} \right) = \begin{cases} \sum_k \delta_k^{l+1} w_{kj}^{l+1} & \text{if } t_j^l < t_k^{l+1} \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

To prevent over-fitting, we have also used an L2 regularization term $\lambda \sum_l \sum_{i,j} (w_{ji}^l)^2$. We have used a margin-based temporal loss with relative target firing times. Fig.6 depicts it is calculated. The target firing times are set as follows:

$$T_j^o = \begin{cases} \tau & \text{if } j = i \\ \tau + \gamma & \text{if } j \neq i \ \& \ t_j^0 < \tau + \gamma \\ t_j^0 & \text{if } j \neq i \ \& \ t_j^0 \geq \tau + \gamma \end{cases} \quad (16)$$

where $\tau = \min\{t_j^0 | 1 < j < c\}$. In the special case when none of the neurons fire:

$$T_j^o = \begin{cases} t_{max} - \gamma & \text{if } j = i \\ t_{max} & \text{if } j \neq i \end{cases} \quad (17)$$

III. Scope for Parallelization

Unlike traditional neural networks, SNNs have an additional 'time' dimension. This implies that the serial run would be slower by more than a factor of the simulation time (here: $t_{max} = 256$). At the same time, this also allows for parallelization across three dimensions: parallel processing of multiple training examples in a batch, parallelization across the neurons in each layer of the dense network and parallelization along the time dimension for each input neuron in a training example. We have developed fully parallel CUDA and OpenMP implementations in addition to a serial implementation for the SNN described in section 2. In the algorithms 1-3, which describe forward pass, loss calculation and backward pass algorithms respectively, we clearly show the loops that have been parallelized in our implementation.

The forward pass algorithm is run for each layer serially, as in the forward direction, the output of $(l-1)^{th}$ layer is needed as an input to the l^{th} layer. Forward Pass consists of a accumulation operation (dot product with prefix sum - as the input is integrated) followed by a thresholding operation. The

Algorithm 1 Forward Pass

```
for b ∈ 0 → BatchSize do                                   (parallel)
  for j ∈ 0 → N_in do                                      (parallel)
    for i ∈ 0 → N_out do                                   (parallel)
      for t ∈ 0 → t_max do                                 (parallel)
        V^l(b,i,t) += x^{l-1}(b,j,t) * w^l(j,i)

for b ∈ 0 → BatchSize do                                   (parallel)
  for i ∈ 0 → N_out do                                     (parallel)
    for t ∈ 0 → t_max do
      V^l(b,i,t) += V^l(b,i,t-1)
      if V^l(b,i,t) ≥ θ^l then
        x^l(b,i,t) = 1
        firing_time^l(b,i) = t
        break
```

dot product operation is parallelized in all three dimensions (image, space and time). However, for the prefix sum, we go serially in the time domain. As soon, as we hit the threshold, the spike is issued and computation for the given neuron is suspended as it is no longer necessary to compute the voltages for the further time steps (because of our TTFS coding where a neuron spikes only once).

Algorithm 2 Loss Calculation

```
for b ∈ 0 → BatchSize do                                   (parallel)
  for i ∈ 0 → N_out do                                     (parallel)
    min_time(b) = min_i(firing_time^out(b,i))
    winner(b) = arg min_i(firing_time^out(b,i))

for b ∈ 0 → BatchSize do                                   (parallel)
  for i ∈ 0 → N_out do                                     (parallel)
    Calculate T_i^o : target_time(b,i)
    Calculate δ_i^o : delta^out(b,i)
```

Loss calculation evaluates the model. It is used in the output layer to find the winner (prediction), the minimum firing time in the output layer to calculate the desired target time for each output neuron and the error vector, e, in the loss calculation (which also gives us the gradient δ_j^{out} of the output layer). We parallelize the computation in both the image and spatial domain (time domain is not involved in this step).

Algorithm 3 Backpropogation

```
for b ∈ 0 → BatchSize do                                   (parallel)
  for i ∈ 0 → N_in do                                      (parallel)
    for j ∈ 0 → N_out do                                   (parallel)
      if firing_time^{l-1}(b,i) < firing_time^l(b,j) then
        dw^l(j,i) += delta^l(b,j)
        delta^{l-1}(b,i) += delta^l(b,j) * w^l(j,i)

for j ∈ 0 → N_out do                                       (parallel)
  for i ∈ 0 → N_in do                                      (parallel)
    w^l(j,i) += η * (dw^l(j,i) + γ * w^l(j,i))
```

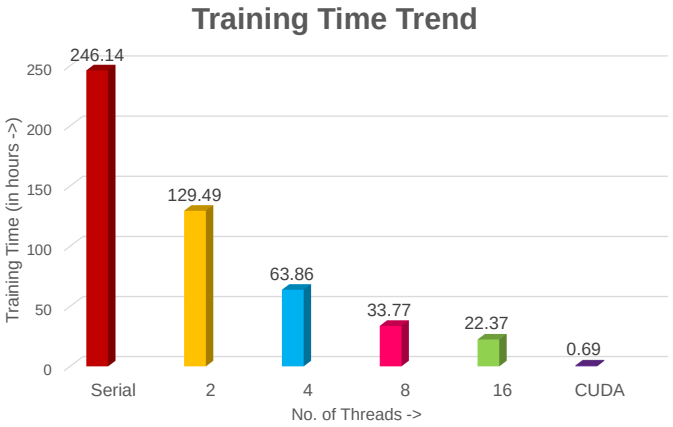

Fig. 4. Simulation time comparison.

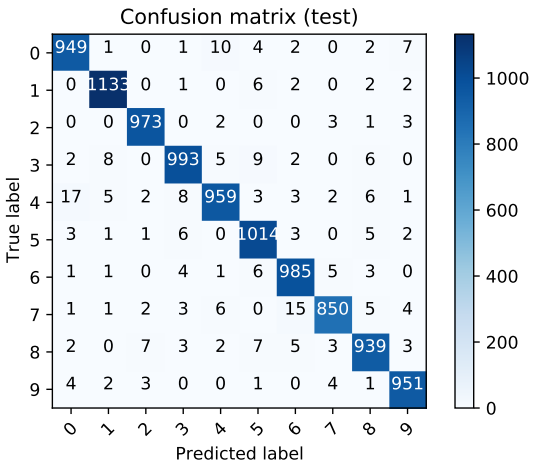

Fig. 5. Confusion Matrix for our trained network on MNIST test data.

In backpropagation, we have to run each layer serially, but in the reverse direction. The gradients of l^{th} layer are needed as input to the $(l-1)^{th}$ layer. For the gradient and delta weight calculations for the hidden layer neurons, updates have to be atomic (see Algorithm 3). This is because the same values are being updated by multiple threads (we are accumulating values in dw and delta). This is done for each layer (one after the other). The weight update is then just a straightforward parallel matrix addition.

IV. Experimental Setup

For our serial and OpenMP implementations, we have used the IIT-K ParamSanganak (Standard) server. For the timing study of our CUDA implementation, we used the Colab GPU for CUDA (Nvidia Tesla K80).

TABLE I
Relative Performance of different implementations

Implementation		**Speedup**
Method	Threads	**Achieved**
Serial	1	1
OpenMP	2	1.9
	4	3.85
	8	7.29
	16	11.01
CUDA		**358.22**

V. Results

Fig.4 shows the per epoch training time for the serial run, OpenMP runs with 2,4,8 and 16 threads and the CUDA run with full parallelization. As expected, the run time keeps decreasing with increasing number of parallel threads but the magnitude of the gradient of the run time curve keeps decreasing due to the additional overhead associated with maintaining a larger number of threads. Nonetheless, the OpenMP run with 16 threads shows a substantial reduction in training time over the serial run, with a run time ratio of 0.09 i.e. a speedup of 11. This is very impressive as for OpenMP only minimal changes are needed to the code (adding pragma), so, one might just opt for OpenMP to reduce the coding effort. The CUDA implementation on the other hand shows massive gains in the run time with a speed up of over 358 (see table.I). It brings down the training time from over 10 days (in the serial case) to less than 45min. We use 2D blocks and 3D blocks for loops parallelized across two dimensions and three dimensions respectively. To minimize thread wastage, each CUDA kernel is invoked with 1024 threads, with the x, y, z block sizes chosen appropriately to minimize thread wastage. The number of blocks are thereby determined by the grid dimensions. Coming to the accuracy numbers, we achieved a 97.2% accuracy on MNIST test dataset (10000 samples) after 40 epochs of training for our 784-400-10 fully connected dense architecture (described in sec. II-B). Figure.5 shows the confusion matrix for the same.

VI. Conclusion

We have successfully demonstrated that a GPU-based implementation for SNNs can give substantial gains in the throughput and reduce simulation time significantly over serial implementations if the spatial and temporal parallelizability is fully utilized. Such implementations, if generalized for different types of neurons, encodings and learning rules, have a massive potential in improving the research timelines in the SNN domain.

References

[1] Pfeiffer, Michael and Pfeil, Thomas,"Deep Learning With Spiking Neurons: Opportunities and Challenges". Frontiers in Neuroscience, Vol. 12, pg 774 (2018), DOI=10.3389/fnins.2018.00774

[2] Saeed Reza Kheradpisheh and Timothee Masquelier, "Temporal backpropagation for spiking neural networks", International Journal of Neural Systems, Vol. 30, No. 6 (2020) 2050027, DOI=10.1142/S0129065720500276

[3] V. Kotariya and U. Ganguly, "Spiking-GAN: A Spiking Generative Adversarial Network Using Time-To-First-Spike Coding", unpublished