SpikePipe: Accelerated Training of Spiking Neural Networks via Inter-Layer Pipelining and Multiprocessor Scheduling

Sai Sanjeet†, Bibhu Datta Sahoo†, and Keshab K. Parhi, *Fellow, IEEE*‡
†Dept. of Electrical Eng., University at Buffalo, Buffalo, NY, USA. e-mail: bibhu@buffalo.edu
‡Dept. of Electrical and Computer Eng., University of Minnesota, Minneapolis, USA. e-mail: parhi@umn.edu

***Abstract*—Spiking Neural Networks (SNNs) have gained popularity due to their high energy efficiency. Prior works have proposed various methods for training SNNs, including backpropagation-based methods. Training SNNs is computationally expensive compared to their conventional counterparts and would benefit from multiprocessor hardware acceleration. This is the first paper to propose inter-layer pipelining to accelerate training in SNNs using systolic array-based processors and multiprocessor scheduling. The impact of training using delayed gradients is observed using four networks training on different datasets, showing no degradation for small networks and $< 10\%$ degradation for large networks. The mapping of various training tasks of the SNN onto systolic arrays is formulated, and the proposed scheduling method is evaluated on the four networks. The results are compared against standard pipelining algorithms. The results show that the proposed method achieves an average speedup of 1.7$\times$ compared to standard pipelining algorithms, with an upwards of 2$\times$ improvement in some cases. The incurred communication overhead due to the proposed method is less than 0.5% of the total required communication of training in networks with convolutional layers.**

***Index Terms*—Spiking neural networks, inter-layer pipelining, multiprocessor scheduling, hardware accelerators.**

I. Introduction

Spiking neural networks (SNNs) are a type of neural network that mimic the functionality of biological neural networks [1]–[4]. Due to their high energy efficiency, they have been employed in a wide range of applications, including computer vision, robotics, and speech recognition, and can achieve performance similar to conventional neural networks [5]. SNNs can be divided into two main types: synchronous [6]–[8] and asynchronous [9]–[11]. In synchronous SNNs, neuronal outputs are computed at linearly-spaced time intervals, while asynchronous or event-driven SNNs compute neuronal outputs based on the arrival of spikes.

Both synchronous and asynchronous SNNs use models of neurons to compute membrane potentials and spike timings. There are various models of neurons, such as the integrate-and-fire (I&F) [12], the leaky integrate-and-fire (LIF) [12], the Izhevsky model [13], and the Hodgkin-Huxley model [14]. The simplest model, the I&F model, is a linear model that does not account for the time-dependent nature of the membrane potential. The LIF model is a nonlinear model that incorporates decay and accounts for the time-dependent nature of the membrane potential. The Hodgkin-Huxley model, the most accurate model of the neuron, accounts for the time-dependent nature of the membrane potential and the ion channels in the neuron. However, simulating the Hodgkin-Huxley model is computationally expensive.

In the most straightforward implementation, the LIF neuron is modeled as a first-order infinite impulse response (IIR) filter [7], [15], [16], where the IIR filter's output represents the neuron's membrane potential. Once the membrane potential reaches a threshold value, the neuron spikes, and the membrane potential is reset to resting potential. In a digital implementation, the IIR filter is implemented using a digital first-order section.

There are various methods of training SNNs, with the two most common being backpropagation [6]–[10], [17] and Spike Timing Dependent Plasticity (STDP) [11], [18]–[20]. Backpropagation, which inherently introduces feedback loops, is challenging to map onto multiple processors to achieve high throughput. This work focuses on training synchronous SNNs using backpropagation. All the results shown in this work are with the LIF neuron model. However, it is to be noted that the proposed methods can easily be adapted to other neuron models.

Prior works have proposed various methods for training conventional neural networks, such as Convolutional Neural Networks (CNNs), on multiple processors [21]–[26]. In our prior LayerPipe approach [26], we proposed the use of variable delayed gradients to achieve inter-layer pipelining of CNNs. Delayed gradients have been used to pipeline adaptive digital filters [27]. The basic assumption is that the gradients can be replaced by delayed gradients if the gradients are slowly varying. However, these methods cannot be directly applied to SNNs because they do not account for the fact that activations in SNNs are binary. By taking this into account, it is possible to further pipeline the training process of training the SNNs at a fine-grain level without significant overhead in communication. It may be noted that there exist prior works that focused on developing hardware for accelerating the training of SNNs [28]–[30]. However, this work's primary objective is accelerating training by efficient mapping onto multiple processors, and therefore, can be used to further accelerate the prior proposed hardware with small modifications.

The contributions of this paper are three-fold. We consider training a digital SNN model that is based on a first-order IIR digital filter with a forward gain that is not unity, introduced in our prior work [15], [16]. Note that in prior SNNs, the forward gain of the first-order IIR filter had been considered to be

This work was supported in part by the National Science Foundation under grant number CCF-1954749. S. Sanjeet was a Predoctoral visitor at the University of Minnesota during the course of this work.

unity. Simulation results show that the proposed architecture achieves 98.6% accuracy on the MNIST dataset using the proposed structure, compared to 98.3% with a structure with unity gain in the forward path. The hardware overhead for realizing non-unity gain is also minimal. Our first contribution lies in deriving the backpropagation equations for training the proposed SNN architecture. It is shown that the IIR filter structure used in the forward pass of the neuron model can also be used to compute the backward pass, implying no additional specialized blocks are necessary for training the SNN. This is non-intuitive, and is important because the same datapath can be used for both inference and training, especially in edge devices. Second, we exploit a variable delayed-gradient approach to achieve inter-layer pipelining. This is inspired by our prior work on accelerating training in CNNs using Layerpipe [26]. This paper is the first to accelerate training in SNNs using inter-layer pipelining. This creates concurrency that can be exploited to map the computations of different layers to multiple processors. It is shown that the use of delayed gradients does not degrade the accuracy by a significant amount. Third, a fine-grained layer-wise scheduling algorithm is proposed to map the tasks to multiprocessors to accelerate the training of SNNs. The proposed fine-grained pipelining algorithm is evaluated using a few sample networks, and the results are compared with existing scheduling algorithms.

This paper is organized as follows. Section II reviews the SNN model considered in this paper. Section III derives the backpropagation equations for training the SNN, and introduces the delayed gradient approach. Section IV formulates the mapping of the training tasks to systolic arrays. Section V presents a fine-grained scheduling algorithm to map the training tasks to multiple processors such that the underutilization of the processors is minimized. Simulation results are presented in Section VI.

II. Modeling Neurons for SNNs

The first step in designing an SNN is modeling a single neuron. Of the various neuron models, the Leaky Integrate-and-Fire (LIF) [12] model is used in this work. The LIF model is a nonlinear model that incorporates decay and accounts for the time-dependent nature of the membrane potential. The membrane potential, $v_m(t)$, of the LIF neuron follows first-order dynamics and is described by (1). The neuron produces a spike when the membrane potential crosses a threshold and is reset to a resting potential.

$$C_m \frac{dv_m(t)}{dt} = i_{in}(t) - \frac{v_m(t) - V_{rest}}{R_m} \tag{1}$$

where $i_{in}(t)$ is the net input current to the neuron, C_m is the membrane capacitance, and R_m is the membrane resistance that causes decay in the membrane potential. For simplicity, the resting potential V_{rest} is taken to be zero. The Laplace transform of (1) is given by (2).

$$V_m(s) = \frac{I_{in}(s)}{s \cdot C_m + \frac{1}{R_m}} \tag{2}$$

where, $V_m(s)$ and $I_{in}(s)$ are the Laplace transforms of $v_m(t)$ and $i_{in}(t)$, respectively. From (2), it is evident that the transfer function from input currents to membrane potential is a first-order infinite impulse response (IIR) filter. The digital equivalent of this filter can be obtained using the bilinear transform as shown in (3).

$$H(z) = \frac{V_m(z)}{I_{in}(z)} = \frac{1 + z^{-1}}{(c + \lambda) - (c - \lambda)z^{-1}} \tag{3}$$

where, $c = 2C_m/T_s$, T_s is the sampling period, and $\lambda = 1/R_m$. The digital filter in (3) is implemented using a digital first-order section as shown in Fig. 1.

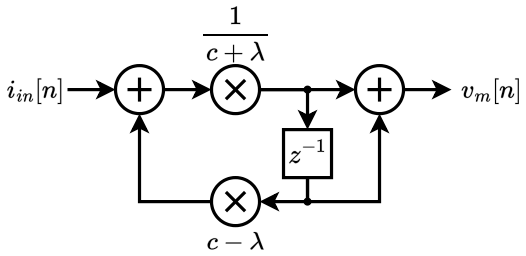


Fig. 1. Digital first-order section for implementing the LIF neuron model.

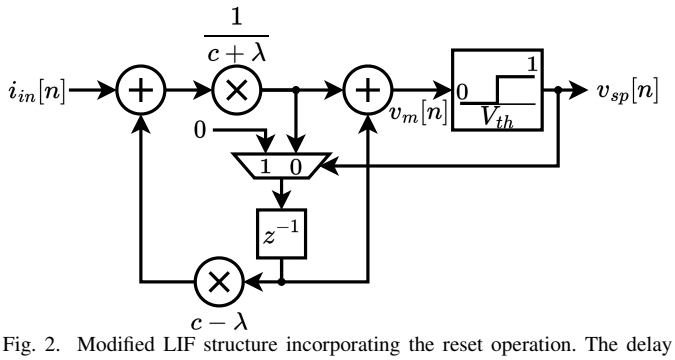


Fig. 2. Modified LIF structure incorporating the reset operation. The delay element is reset when the membrane potential crosses the threshold voltage.

The output of the first-order section is the membrane potential of the neuron. Once the membrane potential reaches a threshold value, the neuron spikes, and the membrane potential is reset to resting potential. The spiking operation is implemented using a simple comparator as shown in Fig. 2[1]. $v_{sp}[n]$ is a binary time series whose value is 1 if there is a spike at timestep n and 0 otherwise. The relation between the input current, the membrane potential, and the output spike train is given by (4) and (5).

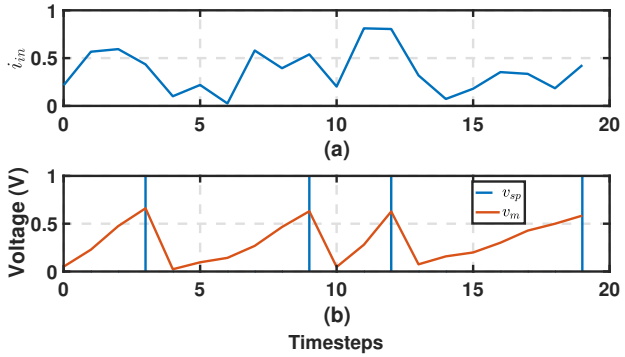


Fig. 3. (a) Sample input current for 20 timesteps and (b) the corresponding membrane potential and spike train. The values of c and λ are 4 and 0.25, respectively, and the threshold voltage is 0.5.

[1]It is to be noted that resetting the delay element also resets the feedforward path, making it harder to produce consecutive spikes. This introduces a soft refractory behavior in the neuron.

$$v_m[n] = \frac{i_{in}[n] + (c-\lambda)\cdot d[n-1]}{c+\lambda} + d[n-1] \quad (4)$$

$$d[n] = \bar{v}_{sp}[n] \cdot \frac{i_{in}[n] + (c-\lambda)\cdot d[n-1]}{c+\lambda}$$

$$v_{sp}[n] = \begin{cases} 1, & \text{if } v_m[n] \geq V_{th} \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

where, $d[n]$ is the intermediate value stored in the delay element at timestep n, and $\bar{v}_{sp}$ is the binary complement of v_{sp}. Figure 3 shows the membrane potential and the spike train for a sample input current.

III. Training SNNs

A neural network is a network of neurons interconnected by synapses. The topology of the network is generally predetermined and does not change over the course of training. In the proposed Spiking Neural Network (SNN), each neuron is modeled using the LIF neuron model as described in Section II. The neurons are arranged in layers, and synapses connect each neuron to a subset of neurons in the previous layer. The weights associated with the synapses are learned during the training process.

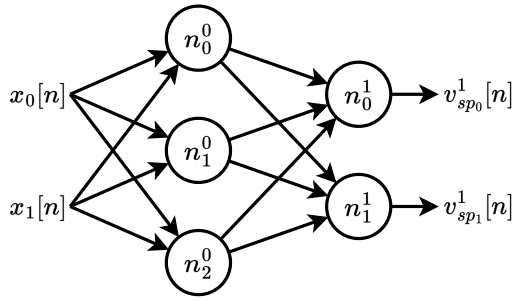


Fig. 4. A sample spiking neural network with two layers.

Figure 4 shows an example network with two inputs, $x_0[n]$ and $x_1[n]$, one hidden layer with three neurons, and an output layer with two neurons. The neuron i in layer l is represented by n_i^l. The same naming convention is used for all intermediate variables i_{in}, v_m, and v_{sp}. The weight connecting input i to neuron j of the hidden layer is given by w_{ij}^0. Therefore, the input current to neuron j of the hidden layer is given by (6).

$$i_{in_j}^0[n] = \sum_{i=0}^{1} w_{ij}^0 \cdot x_i[n] \quad (6)$$

Similarly, the weight connected to neuron i of the hidden layer and neuron j of the output layer is given by w_{ij}^1. The corresponding input currents are used to compute the membrane potentials and spike trains of every neuron using (4) and (5). The output of the network is the spike train of the output neurons. Figure 5 shows the unrolled dataflow graph (DFG) of a neuron in the hidden layer for T timesteps, where nodes i_{in}, v_m, and v_{sp} are computed using (6), (4), and (5), respectively.

A. Loss Function

For supervised learning, the network is trained using a loss function that measures the deviation between the output of the network and the desired output. The loss function is a function of the weights of the network, and the goal of the training process is to obtain the optimal set of weights that minimize the loss function. The use of the appropriate loss function is essential for training the network. Before discussing the loss function, it is crucial to understand the network's output. Spike trains are binary sequences, and encoding the desired output as a binary sequence is not trivial. Therefore, the output of the network is taken to be the membrane potential of the output neurons in the last timestep. To ensure that the membrane potential accurately reflects the network's output, the output neurons are transformed into accumulators with a feedforward path by removing their leaky nature and preventing the membrane potential from resetting, *i.e.,* by setting $c = 1$ and $\lambda = 0$.

Using the membrane potential in the final timestep as the network output, the categorical cross-entropy loss function, given by (7), is used.

$$\mathcal{L} = -\sum_{i=0}^{N-1} y_i \cdot \log\left(\frac{\exp(v_{m_i}^1[T-1])}{\sum_{j=0}^{N-1}\exp(v_{m_j}^1[T-1])}\right) \quad (7)$$

where, $y = [y_0, y_1, \ldots, y_{N-1}]^T$ is the one-hot encoded desired output, N is the number of outputs of the network, $\exp(.)$ is the exponential function, and $\mathcal{L}$ is the loss between the network output and desired output.

B. Backpropagation

To train the network, the gradient of the loss function with respect to the weights of the network is needed. This gradient is used to update the weights of the network using gradient descent. The gradient of the loss function with respect to the weights is computed using backpropagation. The backpropagation algorithm is based on the chain rule of differentiation. For each node in the DFG in Fig. 5, the forward pass and backward pass equations are detailed. The forward pass equations are used to compute the output of the network, and the backward pass equations are used to calculate the gradient of the loss function with respect to the weights and intermediate outputs of the network.

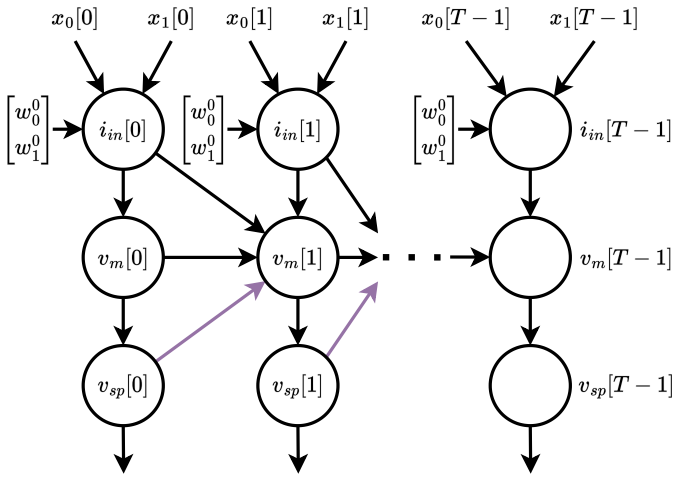


Fig. 5. Dataflow graph of a neuron in the hidden layer for T timesteps. Spiking of the neuron in the previous timestep gates the input current and membrane potential of the current timestep, and is shown by the colored line.

1) $i_{in}[n]$: The forward pass equation for computing $i_{in}[n]$ of the first layer is given by (8).

$$i^0_{in_j}[n] = \sum_i w^0_{ij} \cdot x_i[n] \tag{8}$$

For each subsequent layer, the forward pass equation for computing $i_{in}[n]$ is given by (9).

$$i^l_{in_j}[n] = \sum_i w^l_{ij} \cdot v^{l-1}_{sp_i}[n] \tag{9}$$

Generally, the forward pass equation for computing $i_{in}[n]$ is the same as for other neural networks such as CNNs. The only difference is the operation is repeated for every timestep. Therefore, in a Spiking CNN for each timestep, the input spike train is convolved with the weights to generate the input current to the neurons in the next layer.

The local gradients for the backward pass are given by (10) and (11).

$$\frac{\partial i^l_{in_j}[n]}{\partial v^{l-1}_{sp_i}[n]} = w^l_{ij} \tag{10}$$

$$\frac{\partial i^l_{in_j}[n]}{\partial w^l_{ij}} = v^{l-1}_{sp_i}[n] \tag{11}$$

The gradients with respect to the loss function are given by (12) and (13) using the chain rule of differentiation.

$$\frac{\partial \mathcal{L}}{\partial v^{l-1}_{sp_i}[n]} = \sum_j \frac{\partial \mathcal{L}}{\partial i^l_{in_j}[n]} \cdot \frac{\partial i^l_{in_j}[n]}{\partial v^{l-1}_{sp_i}[n]} = \sum_j \frac{\partial \mathcal{L}}{\partial i^l_{in_j}[n]} \cdot w^l_{ij} \tag{12}$$

$$\frac{\partial \mathcal{L}}{\partial w^l_{ij}} = \sum_n \frac{\partial \mathcal{L}}{\partial i^l_{in_j}[n]} \cdot \frac{\partial i^l_{in_j}[n]}{\partial w^l_{ij}} = \sum_n \frac{\partial \mathcal{L}}{\partial i^l_{in_j}[n]} \cdot v^{l-1}_{sp_i}[n] \tag{13}$$

2) $v_m[n]$: The forward pass equation for computing $v_m[n]$ is given by (4). All the corresponding v_m, v_{sp}, and i_{in} are $v^l_{m_i}$, $v^l_{sp_i}$, and $i^l_{in_i}$, respectively. Here, $v^l_{sp_i}$ acts as the gating function to determine whether the input current and membrane potential of the previous timestep should be used to compute the membrane potential of the current timestep. The backward pass equations for computing the gradients with respect to $i^l_{in_i}[n]$, $v^l_{m_i}[n-1]$, and $i^l_{in_i}[n-1]$ are given by (14), (15), and (16), respectively.

$$\frac{\partial v^l_{m_i}[n]}{\partial i^l_{in_i}[n]} = \frac{1}{c+\lambda} \tag{14}$$

$$\frac{\partial v^l_{m_i}[n]}{\partial v^l_{m_i}[n-1]} = \bar{v}^l_{sp_i}[n-1] \cdot \frac{c-\lambda}{c+\lambda} \tag{15}$$

$$\frac{\partial v^l_{m_i}[n]}{\partial i^l_{in_i}[n-1]} = \bar{v}^l_{sp_i}[n-1] \cdot \frac{1}{c+\lambda} \tag{16}$$

The gradient with respect to the loss function is given by (17) using the chain rule of differentiation.

$$\frac{\partial \mathcal{L}}{\partial i^l_{in_i}[n]} = \frac{1}{c+\lambda} \cdot \left(\frac{\partial \mathcal{L}}{\partial v^l_{m_i}[n]} + \bar{v}^l_{sp_i}[n] \cdot \frac{\partial \mathcal{L}}{\partial v^l_{m_i}[n+1]} \right) \tag{17}$$

For the output layer, the forward pass is given by (18).

$$\begin{aligned} v^L_{m_i}[n] &= i^L_{in_i}[n] + i^L_{in_i}[n-1] + v^L_{m_i}[n-1] \\ v^L_{m_i}[T-1] &= \left(2 \cdot \sum_n i^L_{in_i}[n] \right) - i^L_{in_i}[T-1] \end{aligned} \tag{18}$$

The gradient is given by (19).

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial i^L_{in_i}[n]} &= k \cdot \frac{\partial \mathcal{L}}{\partial v^L_{m_i}[T-1]} \\ k &= \begin{cases} 1, & \text{if } n = T-1 \\ 2, & \text{otherwise} \end{cases} \end{aligned} \tag{19}$$

The relation between $v^L_{m_i}[T-1]$ and $\mathcal{L}$ is given by (7) and the derivative is given by (20).

$$\frac{\partial \mathcal{L}}{\partial v^L_{m_i}[T-1]} = \frac{e^{v^L_{m_i}[T-1]}}{\sum_j e^{v^L_{m_j}[T-1]}} - y_i \tag{20}$$

3) $v_{sp}[n]$: When the membrane potential crosses a threshold value, a spike is produced. The forward pass equation for computing $v_{sp}[n]$ is given by (21).

$$v^l_{sp_i}[n] = \phi(v^l_{m_i}[n]) = \begin{cases} 1, & \text{if } v^l_{m_i}[n] \geq V_{th} \\ 0, & \text{otherwise} \end{cases} \tag{21}$$

The thresholding activation function, $\phi(x)$, that produces the spike is non-differentiable. To ensure that the gradient is propagated through the thresholding function, the function is approximated to be a linear function in the region around the threshold value, as shown in Fig. 6 and described in [6]. The gradient of the approximated function is given by (22). The variable α in (22) and Fig. 6 is a hyperparameter that can be tuned. For simplicity, α is set to 0.5 in this work.

$$\frac{\partial v^l_{sp_i}[n]}{\partial v^l_{m_i}[n]} = \phi'(v^l_{m_i}[n]) = \begin{cases} \frac{1}{2\alpha}, & \text{if } |v^l_{m_i}[n] - V_{th}| \leq \alpha \\ 0, & \text{otherwise} \end{cases} \tag{22}$$

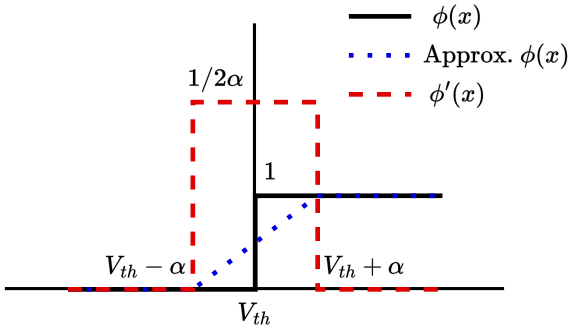


Fig. 6. Approximation of the thresholding function as linear in the region around the threshold value.

For all layers except the output layer, the gradient of the loss with respect to membrane potential is given by (23).

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial v^l_{m_i}[n]} &= \frac{\partial \mathcal{L}}{\partial v^l_{sp_i}[n]} \cdot \frac{\partial v^l_{sp_i}[n]}{\partial v^l_{m_i}[n]} + \frac{\partial \mathcal{L}}{\partial v^l_{m_i}[n+1]} \cdot \frac{\partial v^l_{m_i}[n+1]}{\partial v^l_{m_i}[n]} \\ &= \frac{\partial \mathcal{L}}{\partial v^l_{sp_i}[n]} \cdot \phi'(v^l_{m_i}[n]) \\ &\quad + \frac{\partial \mathcal{L}}{\partial v^l_{m_i}[n+1]} \cdot \bar{v}^l_{sp_i}[n] \cdot \frac{c-\lambda}{c+\lambda} \end{aligned} \tag{23}$$

From (17) and (23), it is apparent that the gradient of the loss with respect to the input current can be computed using

a similar IIR filter structure as the LIF neuron model. The structure of the IIR filter for the gradient computation is shown in Fig. 7. All the variables in the structure are in time-reversed order, *i.e.*, $\frac{\partial \mathcal{L}}{\partial v_{sp_i}[T-1]}$ is the input in the first clock cycle, $\frac{\partial \mathcal{L}}{\partial v_{sp_i}[T-2]}$ in the second clock cycle, and so on. Similarly, the output is also produced in time-reversed order.

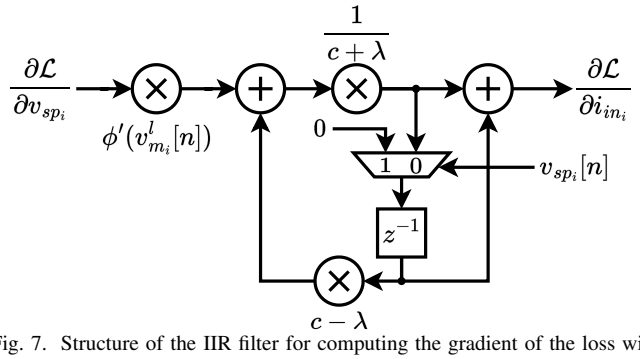


Fig. 7. Structure of the IIR filter for computing the gradient of the loss with respect to the input current.

C. Dataflow Graph of Training

Figure 8 shows the DFG for training a sample two-layer network. Each node in the DFG computes the corresponding variables for a single mini-batch. Therefore, one weight update based on a single input mini-batch is computed per cycle of the DFG. The DFG is similar to the DFG of a conventional neural network, the difference being each computation in the DFG of an SNN has a timestep dimension. The training process inherently has feedback loops. The colored line in Fig. 8 highlights the critical loop in the example DFG, which has the forward and backward computations of all the layers. Since there is only one delay element in the critical loop, it is not straightforward to retime the DFG for mapping onto multiple processors.

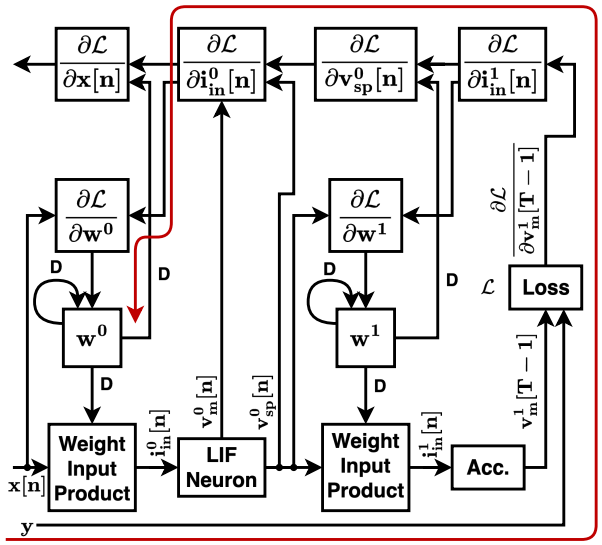


Fig. 8. Dataflow graph of training a sample two-layer network. The colored line indicates the critical loop.

D. Delayed Gradients

To split the DFG into multiple subgraphs and map onto multiple processors, additional delay elements must be added into the critical loop. One way of achieving this is using delayed gradients [27]. The gradient update is delayed by a few cycles, introducing additional delay elements into the critical loop. The modified DFG can now be retimed and split into multiple subgraphs. The use of delayed gradients and retiming was used to achieve inter-layer pipelining in the training of the CNNs in the LayerPipe approach described in [26]. Figure 9 shows the DFG after retiming with delayed gradients.

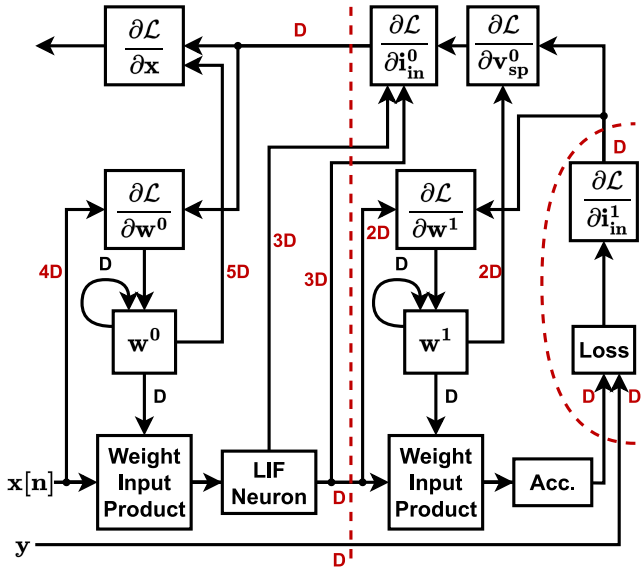


Fig. 9. Retimed DFG with delayed gradients. The DFG can be split into subgraphs at the colored lines. The colored delay elements emerge from retiming the DFG using delayed gradients.

The introduction of delayed gradients calls for an analysis of the trainability and convergence of the network. Delayed gradients will not severely affect the trainability of the network if the gradients change slowly over iterations. In the context of neural networks, this is determined by the gradient update algorithm used to train the network. The most common gradient update algorithm is the stochastic gradient descent (SGD) algorithm. The SGD algorithm is given by (24), where w is the weight, η is the learning rate, and $\frac{\partial \mathcal{L}}{\partial w}$ is the gradient of the loss with respect to the weight.

$$w_{t+1} = w_t - \eta \cdot \frac{\partial \mathcal{L}}{\partial w} \tag{24}$$

From the SGD update equation, the gradients may change rapidly in every iteration during the initial stages of training. This would result in delayed gradients significantly affecting the trainability of the network. The effect of delayed gradients on the convergence of LMS filters is explored in [31] by using the moving average of the gradients. In the context of neural networks, the Adam optimizer [32] ensures a similar behavior by introducing first and second-order momentum. The Adam

optimizer is given by (25).

$$
\begin{aligned}
m_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1) \cdot \frac{\partial \mathcal{L}}{\partial w} \\
v_{t+1} &= \beta_2 \cdot v_t + (1-\beta_2) \cdot \left(\frac{\partial \mathcal{L}}{\partial w}\right)^2 \\
\eta_{t+1} &= \eta \cdot \frac{\sqrt{1-\beta_2^{t+1}}}{1-\beta_1^{t+1}} \\
w_{t+1} &= w_t - \eta_{t+1} \cdot \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon}
\end{aligned} \tag{25}
$$

E. Example Networks

To analyze the effect of delayed gradients, the MNIST [33], Neuromorphic-MNIST [34], DVS128 Gestures [35], and the Spiking Heidelberg Digits (SHD) [36] datasets are considered. Table I shows the networks used for the four datasets which are trained for 8, 30, 40, and 40 timesteps, respectively. The networks are trained using the SGD and Adam optimizers, with and without delayed gradients. The gradients in the layers are delayed by a predetermined number of batches, as tabulated in Table I. This distribution of delays splits the training DFG into subgraphs such that there is only one layer per subgraph, as described in [26]. The learning rate is set to 0.001 for both optimizers. The mini-batch size is set to 32 and the MNIST and N-MNIST networks are trained for 100 epochs, the DVS128 Gestures network is trained for 300 epochs, and the SHD dataset is trained for 30 epochs. The results are shown in Table II. The mean and standard deviation of the accuracy for 10 different weight initializations are reported.

It is observed that the smaller networks trained on the MNIST, N-MNIST, and SHD datasets achieve similar accuracy with delayed gradients as without delayed gradients for both SGD and Adam optimizers. As expected, the accuracy with the Adam optimizer is higher than that of the SGD optimizer. Overfitting is a major problem in the SHD dataset for feed-forward SNNs, as mentioned in [36]. However, the use of delayed gradients consistently achieves a higher accuracy on this dataset, suggesting a form of regularization.

However, for the larger network trained on the DVS128 Gestures dataset, delayed gradients impact the performance. It is also observed that the difference in accuracy in the networks trained with and without delayed gradients increases as the absolute accuracy obtained increases. Therefore, delayed gradients have a more significant effect on the trainability of the network when the network is trained to achieve higher accuracy, as is the case with the Adam optimizer, as opposed to the SGD optimizer. However, the maximum accuracy achieved across the 10 runs in Adam is 81.25%, which is higher than the 77.43% achieved with SGD. Therefore, the Adam optimizer is used for training the networks in the rest of the work. It is to be noted that the networks trained with delayed gradients offer a significant speedup in training time, as discussed in Section VI, at the cost of a small reduction in accuracy. The trained networks can be further fine-tuned without delayed gradients to achieve higher accuracy if required. However, such fine-tuning is not explored in this work.

TABLE I
DETAILS OF THE NETWORKS USED FOR THE FOUR DATASETS. THE DELAY COLUMN INDICATES THE NUMBER OF BATCHES BY WHICH THE GRADIENTS OF THE CORRESPONDING LAYERS ARE DELAYED.

Dataset	Layer	Output Shape	# Params	Delay
MNIST	Conv 1	$(28 \times 28 \times 8)$	80	6
	Maxpool	$(14 \times 14 \times 8)$	-	-
	Conv 2	$(14 \times 14 \times 8)$	584	4
	Maxpool	$(7 \times 7 \times 8)$	-	-
	FC 1	128	50,304	2
	Output	10	1,290	0
N-MNIST	Conv 1	$(32 \times 32 \times 8)$	152	6
	Maxpool	$(16 \times 16 \times 8)$	-	-
	Conv 2	$(16 \times 16 \times 8)$	584	4
	Maxpool	$(8 \times 8 \times 8)$	-	-
	FC 1	32	16,416	2
	Output	10	330	0
DVS128 Gestures	Conv 1	$(64 \times 64 \times 32)$	608	14
	Maxpool	$(32 \times 32 \times 32)$	-	-
	Conv 2	$(32 \times 32 \times 64)$	18,496	14
	Maxpool	$(16 \times 16 \times 64)$	-	-
	Conv 3	$(16 \times 16 \times 128)$	73,856	12
	Conv 4	$(16 \times 16 \times 128)$	147,584	8
	Maxpool	$(8 \times 8 \times 128)$	-	-
	Conv 5	$(8 \times 8 \times 256)$	295,168	4
	Conv 6	$(8 \times 8 \times 256)$	590,080	2
	Maxpool	$(4 \times 4 \times 256)$	-	-
	FC 1	128	524,416	0
	Output	11	2,827	0
SHD	FC 1	256	$179,456$	4
	FC 2	256	$65,792$	2
	Output	20	$5,140$	0

TABLE II
ACCURACY OF THE NETWORKS WHEN TRAINED WITH SGD AND ADAM OPTIMIZERS, WITH AND WITHOUT DELAYED GRADIENTS. MEAN AND STANDARD DEVIATION OF ACCURACIES OVER 10 DIFFERENT WEIGHT INITIALIZATIONS ARE REPORTED.

Dataset	Optimizer	Delayed Gradients	
		No	Yes
MNIST	SGD	$96.37\% \pm 0.32\%$	$96.38\% \pm 0.25\%$
	Adam	$98.64\% \pm 0.13\%$	$98.59\% \pm 0.05\%$
N-MNIST	SGD	$96.54\% \pm 0.32\%$	$96.34\% \pm 0.52\%$
	Adam	$98.17\% \pm 0.15\%$	$98.13\% \pm 0.12\%$
DVS128 Gestures	SGD	$79.03\% \pm 5.71\%$	$73.35\% \pm 2.71\%$
	Adam	$85.42\% \pm 2.47\%$	$76.39\% \pm 3.52\%$
SHD	SGD	$67.65\% \pm 0.53\%$	$69.41\% \pm 0.80\%$
	Adam	$73.08\% \pm 1.01\%$	$73.90\% \pm 0.98\%$

The obtained accuracies are compared against prior SNN works, which have targeted the same four datasets, and tabulated in Table III. The networks trained using the Adam optimizer with delayed gradients have comparable accuracy to the prior works on the MNIST and N-MNIST datasets. The proposed method achieves competitive accuracy on the SHD dataset using feed-forward SNNs with simple LIF models. The use of recurrent SNNs or specialized adaptive delays [39] has been shown to achieve higher accuracies but is beyond the scope of this work. The accuracy on the DVS128 Gestures dataset is lower than the prior works. However, the network trained in this work has fewer parameters and timesteps than the prior works. Moreover, the training method used in this work is a generic method without any dataset-specific tuning such as data preprocessing. There is also no fine-tuning done

TABLE III
COMPARISON OF THE ACCURACIES OBTAINED IN THIS WORK WITH PRIOR WORKS.

Work	Dataset	Accuracy (%)	# Params	Timesteps
[6]	MNIST	98.89	636,010	30
[6]		99.42	606,740	30
[9]		99.31	517,780	-
[10]		98.42	22,662	-
[17]		99.59	517,780	50
This Work		**98.46**	**52,258**	**8**
[6]	N-MNIST	98.78	1,858,410	30
[9]		98.66	1,858,410	-
[10]		97.23	37,044	-
[17]		99.09	750,780	50
This Work		**98.06**	**17,482**	**30**
[37]	DVS128 Gestures	95.83	2,081,866	500
[38]		93.40	2,332,891	-
This Work		**81.25**	**1,653,035**	**40**
[36]	SHD	48.1	92,308	100
[39]		92.42	109,076	150
This Work		**75.97**	**250,388**	**40**

without delayed gradients to further improve the accuracy. Therefore, the obtained accuracy is reasonable.

IV. ACCELERATORS FOR TRAINING SNNS

Recent years have seen significant progress in developing efficient hardware accelerators for training neural networks. Most accelerators developed for training neural networks are based on systolic arrays [40]–[42], like the Google Tensor Processing Unit (TPU) [43]. Systolic arrays are well-suited for training neural networks because of their regular structure and high compute density. Figure 10 shows the structure of a typical 4×4 systolic array. Each element in the array is a processing element (PE) that performs a single multiply-and-accumulate (MAC) operation every cycle. The structure in Fig. 10 is an output-stationary architecture performing the convolution operation. The input is a 5×5 feature map with 3 channels, which is convolved with four 3×3 filters. The output is a 3×3 feature map with 4 channels. The first element in every channel of the output feature map is computed by multiplying the first 3×3 patch of the input feature map with the four filters. This operation is done in the first row of the systolic array, where the 3×3 patch of the input feature map is streamed through the row and the filters are streamed down the four columns. In the structure shown in Fig. 10, only four elements of the output feature map can be computed per channel as there are only four rows in the systolic array. The remaining elements of the output feature map are computed after the first four are computed and streamed out.

A. Modified systolic array for SNNs

The systolic array structure and the data streaming scheme need to be modified to train SNNs. Both the forward pass and backward pass of the LIF neuron model require an IIR filter. Therefore, a bank of IIR filters should be present along with the systolic array. The output of each column of the systolic array is passed through an IIR filter and sent to the output SRAM. The number of IIR filters in the filter bank is S_C, where S_C is the number of columns in the systolic array. Figure 11 shows the architecture of the modified systolic array processor for SNNs. The interconnect bus in Fig. 11 connects adjacent processors and facilitates data transfer between them.

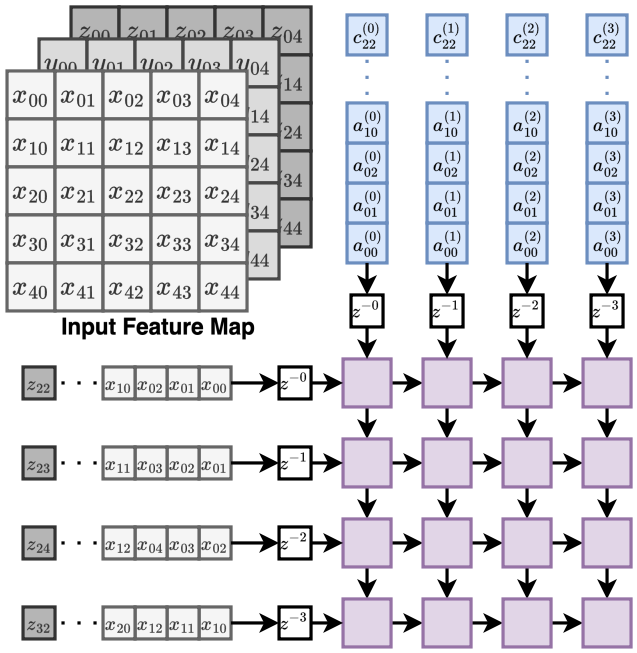


Fig. 10. Structure of a regular 4×4 systolic array in output-stationary configuration for computing a convolution. The input is a 5×5 feature map with 3 channels, which is convolved with four 3×3 filters. $a_{ij}^{(f)}$ is the coefficient of filter f at the i, j^{th} location for the first channel. The second and third channels are represented by $b_{ij}^{(f)}$ and $c_{ij}^{(f)}$, respectively.

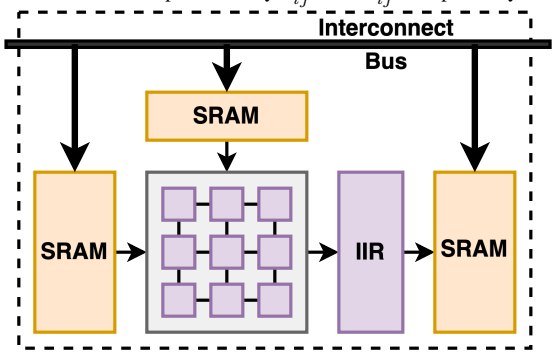


Fig. 11. Modified systolic array processor for training SNNs. The interconnect bus connects adjacent processors and facilitates data transfer between them.

B. Estimating clock cycles for computation

The discussed processor has to be modeled to accurately estimate the number of clock cycles required to compute a given task [44]. Since the target problem is image classification, only two types of computations are required: convolution and matrix multiplication. For each of these, there are three cases to consider: the forward pass, gradient with respect to weights, and gradient with respect to inputs. The mapping of the computations to the systolic array for each of these cases is discussed in this section.

1) Convolutional layer: In the forward pass of convolution, the input image is convolved with a set of filters. Assuming the image to be $H \times W \times C$ including padding, and the filter to be $K \times K \times C$, the output of the convolution is $H_{out} \times W_{out} \times 1$, where $H_{out} = (H - K + 1)$ and $W_{out} = (W - K + 1)$. The convolution operation is performed by sliding the filter over the input image and computing the dot product of the filter and the image patch. The dot product is computed by multiplying the corresponding elements of the filter and the image patch and summing them up. This is repeated for T timesteps and F filters.

For each of the three tasks, *i.e.,* the forward pass, gradient with respect to weights, and gradient with respect to inputs, the number of rows and columns necessary for computation, and the number of MACs per PE are different. Table IV summarizes the number of rows and columns and the number of MACs per PE for each of the three tasks. In most practical cases, the systolic arrays are not large enough to accommodate all the rows and columns necessary for computation. Therefore, the computations are tiled to fit the systolic array. The number of tiles required in an $(S_R \times S_C)$ systolic array is given by (26).

$$N_{tiles} = \left\lceil \frac{N_{rows}}{S_R} \right\rceil \cdot \left\lceil \frac{N_{cols}}{S_C} \right\rceil \tag{26}$$

where, $\lceil x \rceil$ is the ceiling function. For each tile, the number of clock cycles is the number of cycles required for accumulating the output, and the input and output skews, given by (27).

$$N_{cycles_{tile}} = N_{MAC} + (S_R - 1) + (S_C - 1) \tag{27}$$

The total number of clock cycles required for each task is given by (28) when N_{tiles} and $N_{cycles_{tile}}$ are computed using the values from Table IV.

$$N_{conv} = N_{tiles} \cdot N_{cycles_{tile}} \tag{28}$$

TABLE IV
NUMBER OF ROWS AND COLUMNS, AND NUMBER OF MACS PER PE FOR EACH OF THE THREE TASKS FOR CONVOLUTION.

Task	N_{rows}	N_{cols}	N_{MAC}
Forward pass	$H_{out} \cdot W_{out} \cdot T$	F	$K \cdot K \cdot C$
Weight Gradient	$K \cdot K \cdot C$	F	$H_{out} \cdot W_{out} \cdot T$
Input Gradient	$H \cdot W \cdot T$	C	$K \cdot K \cdot F$

2) Fully-connected layer: In the forward pass of a fully-connected layer simulated over T timesteps, the input feature map is a vector of size Q_{in}, and the output feature map is a vector of size Q_{out}. The number of tiles and number of clock cycles per tile are given by (26) and (27), respectively. Similar to the convolutional layer, the total number of clock cycles required for the fully-connected layer is given by (29) when N_{tiles} and $N_{cycles_{tile}}$ are computed using the values from Table V.

$$N_{fc} = N_{tiles} \cdot N_{cycles_{tile}} \tag{29}$$

TABLE V
VALUES OF N_R, N_C, AND N_{MAC} FOR EACH OF THE TASKS IN THE FULLY-CONNECTED LAYER.

Task	N_{rows}	N_{cols}	N_{MAC}
Forward pass	T	Q_{out}	Q_{in}
Weight Gradient	Q_{in}	Q_{out}	T
Input Gradient	T	Q_{in}	Q_{out}

C. Extending to complex neuron models

The cycle estimate per tile in (27) is determined only by the systolic array and not the neuron model computation, as it is assumed that the neuron model is run on a dedicated block that can be pipelined with the systolic array, owing to its simplicity. However, such pipelining may not be feasible for more complex neuron models, resulting in additional cycles per tile. Assuming the number of cycles required by dedicated hardware to compute the neuron model is given by N_{neuron}, the new value of $N_{cycles_{tile}}$ is given by (30).

$$N_{cycles_{tile}} = \max\left(N_{MAC} + (S_R - 1) + (S_C - 1), N_{neuron}\right) \tag{30}$$

The total number of cycles in (28) and (29) would be computed using the new $N_{cycles_{tile}}$. However, in most practical situations, the value of $N_{MAC} + (S_R - 1) + (S_C - 1)$ is in the order of 100-1000, while the value of N_{neuron} would be at most 100 for the most complex models, implying the effect of N_{neuron} on the throughput can be ignored.

D. Clock cycles for the MNIST network

For the four-layer network trained on MNIST detailed in Table I, the number of clock cycles required for each layer when mapped to a 32×32 systolic array is given in Table VI. For most practical applications, the input gradient of the first layer is not required. Therefore, the total number of clock cycles needed for one weight update of the entire network is $46,956$.

TABLE VI
NUMBER OF CLOCK CYCLES REQUIRED FOR EACH LAYER OF THE MNIST NETWORK WHEN MAPPED TO A 32×32 SYSTOLIC ARRAY.

Layer	Forward pass	Weight gradient	Input gradient
Conv1	13,916	6,334	26,264
Conv2	6,566	4,890	6,566
FC1	1,816	3,640	2,470
Output	190	280	288

V. SCHEDULING TO MULTIPLE PROCESSORS

When the training is done with more than one processor, the workload should be distributed evenly among the processors to minimize the total number of clock cycles required for the entire network. Ideally, the training throughput should be increased by a factor equal to the number of processors used. However, achieving this in practice is difficult. This section explores various scheduling algorithms that can distribute the workload among multiple processors and compares their performance in terms of throughput. The number of cycles required for the forward pass, weight gradient, and input gradient of a layer indexed l is denoted by N^l_{FP}, N^l_{WG}, and N^l_{IG}, respectively. The number of processors used for training is denoted by P. The total number of clock cycles required for the entire network is represented by N_{total}. The network trained on MNIST detailed in Table I is used for the simulations in this section. For this network, N_{total} is 46,956 clock cycles. The values of N^l_{FP}, N^l_{WG}, and N^l_{IG}, for all l, are tabulated in Table VI. N^1_{IG} corresponds to the computation of $\partial\mathcal{L}/\partial\mathbf{x}[\mathbf{n}]$ in Fig. 8, and is ignored in the simulations since it is not required for training.

The basis for all the discussed scheduling algorithms is the ability to split the training dataflow graph shown in Fig. 8 into various subgraphs. As discussed in Section III-D, this can be achieved with the help of delayed gradients. The DFG can be retimed and split into subgraphs, shown by the dashed lines in Fig. 9. Each subgraph can be mapped onto a processor. The number of batches by which each batch is delayed depends on the processor to which the corresponding weight gradient computation task is mapped. For each weight gradient task, the number of delays would be twice the number of processors following the processor to which the task is mapped. For example, if a weight gradient task is mapped to the third processor, and there are a total of 5 processors, the number of delays would be $2 \times (5 - 3) = 4$.

Throughout this section, the speedup of training is compared to the single processor implementation. If the training is

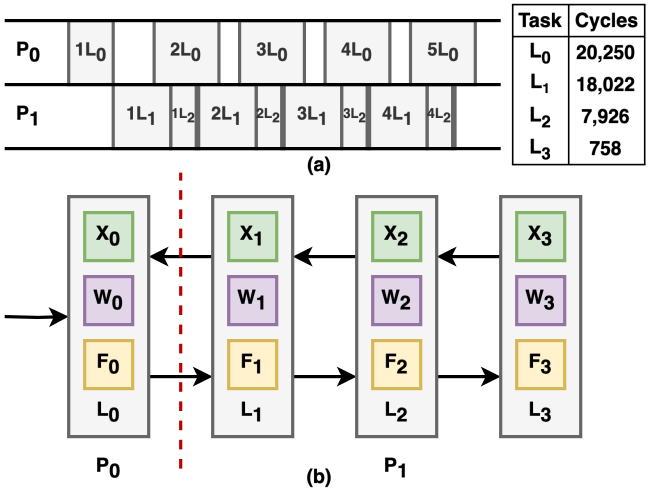
done on a single processor, the number of clock cycles per weight update is N_{total}, or $46,956$ for the network under consideration. The number of required clock cycles reduces when the training is done on multiple processors. The speedup of training is the ratio of the number of clock cycles per weight update on a single processor to the number of clock cycles per weight update on multiple processors.

A. Layer-wise scheduling

The most straightforward way to distribute the workload among multiple processors is to consider an entire layer as a single task and schedule the tasks to the processors. With this approach, the speedup in computation, σ, is upper bounded by the number of clock cycles required for the longest layer, as given by (31).

$$\sigma_{layerwise} \leq \frac{N_{total}}{\max_l(N^l_{FP} + N^l_{WG} + N^l_{IG})} \tag{31}$$

For the network trained on MNIST detailed in Table I, the longest layer is the first layer, with $20,250$ clock cycles[2]. $\sigma_{layerwise}$ for this network is 2.32. Therefore, using more than three processors does not improve training throughput. Figure 12(a) shows the schedule map for the layer-wise scheduling algorithm when $P = 2$ along with the number of clock cycles necessary for each task. The schedule map shows the processor to which each task is assigned. After the pipeline is full, the processor P_0 is active for $20,250$ cycles and the processor P_1 is active for $26,706$ cycles. The number of cycles per weight update is $26,706$, resulting in a speedup of $1.8\times$ over the single processor implementation. Figure 12(b) shows the split of the DFG into two subgraphs that are mapped to the corresponding processors.



Task	Cycles
L_0	20,250
L_1	18,022
L_2	7,926
L_3	758

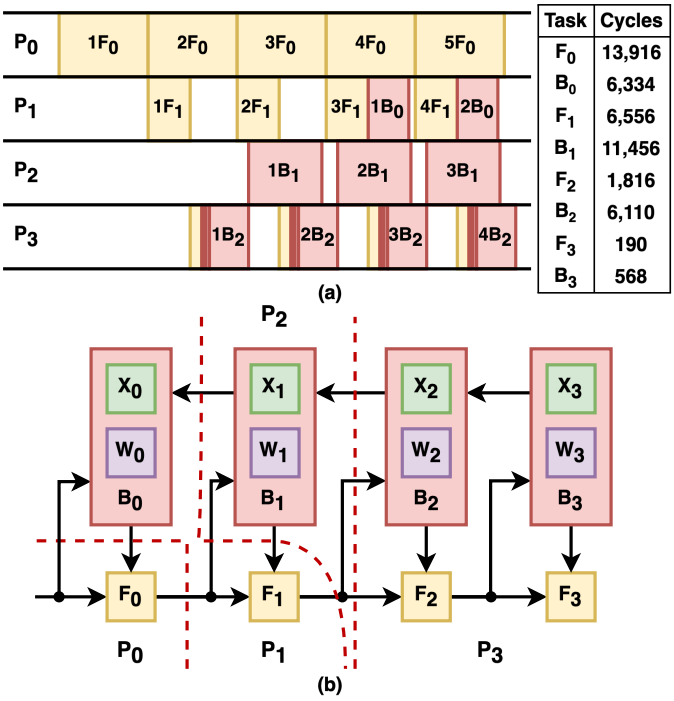
Fig. 12. (a) Schedule map for the layer-wise scheduling algorithm when $P = 2$ and (b) split of the DFG to two subgraphs. xL_i represents the tasks of layer i for input batch index x.

B. PipeDream-based scheduling

Scheduling based on the PipeDream algorithm [23] assumes the layer's tasks are split into forward and backward passes. The backward pass contains both input gradient computation and weight gradient computation. This method of splitting increases the upper bound of speedup by considering the maximum of the longest forward pass and longest backward pass. The speedup bound is given by (32).

$$\sigma_{pipedream} \leq \frac{N_{total}}{\max(\max_l(N^l_{FP}), \max_l(N^l_{WG} + N^l_{IG}))} \tag{32}$$

For the values in Table VI, the longest forward pass takes $13,916$ cycles, and the longest backward pass takes $11,456$ cycles, resulting in $\sigma_{pipedream}$ of 3.37. Figure 13(a) shows the schedule map for the PipeDream-based scheduling algorithm when $P = 4$. The number of cycles per weight update is $13,916$, resulting in a speedup of $3.37\times$ over the single processor implementation. Figure 13(b) shows the split of the DFG into four subgraphs.



Task	Cycles
F_0	13,916
B_0	6,334
F_1	6,556
B_1	11,456
F_2	1,816
B_2	6,110
F_3	190
B_3	568

Fig. 13. (a) Schedule map for the PipeDream-based scheduling algorithm when $P = 4$ and (b) split of the DFG to four subgraphs.

C. Split backward pass

The PipeDream-based scheduling algorithm considers the entire backward pass of a layer as a single task. In most cases, this becomes a bottleneck. The backward pass can be split into weight gradient and input gradient tasks. This increases the upper bound of speedup by considering the longest task. The upper bound of speedup is given by (33).

$$\sigma_{split} \leq \frac{N_{total}}{\max(\max_l(N^l_{FP}), \max_l(N^l_{WG}), \max_l(N^l_{IG}))} \tag{33}$$

However, for the example network, this does not offer any further speedup as the longest task is the forward pass of the first layer.

D. Fine-grained pipelining

The speedup of training might not be close to ideal even when the number of processors is less than the upper bound of speedup. In Fig. 12, the speedup is only $1.8\times$ instead of the ideal speedup of $2\times$. This is because the tasks require a different number of clock cycles, which results in some

[2]Note that the input gradient task is ignored for the first layer.

processors being idle. Fine-grained pipelining can be used to balance the pipeline stages such that the number of clock cycles required for each stage is the same, thus reducing the idle time of the processors.

Fine-grained pipelining is done by splitting the tasks into multiple subtasks wherever necessary and assigning them to different processors. The forward pass of a layer needs outputs of the previous layer and weights of the current layer. The output of the forward pass is used as input to the forward pass and weight gradient of the next layer. Therefore, moving a part of the forward pass to the next processor requires more communication of moving part of previous outputs and current weights to the next layer. However, it reduces the cost of moving the outputs of the current forward pass to the next processor. Figure 14 shows the process of moving part of the forward pass to the next processor. The task F_0 is split into F_0 and F'_0. The task F'_0 is moved to the next processor.

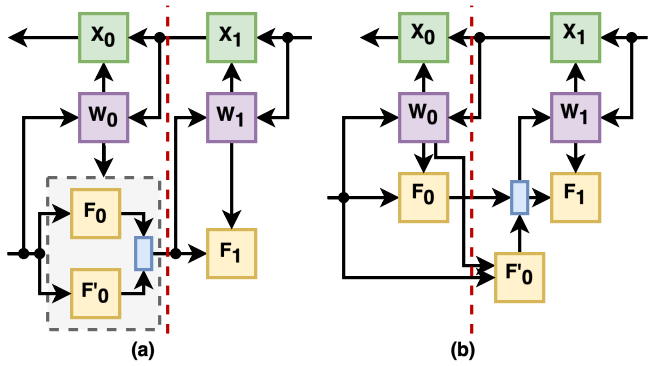
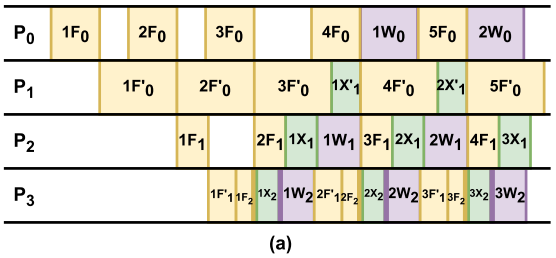


Fig. 14. Mapping two layers to two processors by (a) splitting the forward pass in the first layer and (b) moving one part of the forward pass to the next processor.



Task	Cycles
F_0	5,396
F'_0	8,520
W_0	6,334
F_1	3,350
F'_1	3,216
W_1	4,890
X_1	3,216
X'_1	3,350
F_2	1,816
W_2	3,640
X_2	2,470
F_3	190
W_3	280
X_3	288

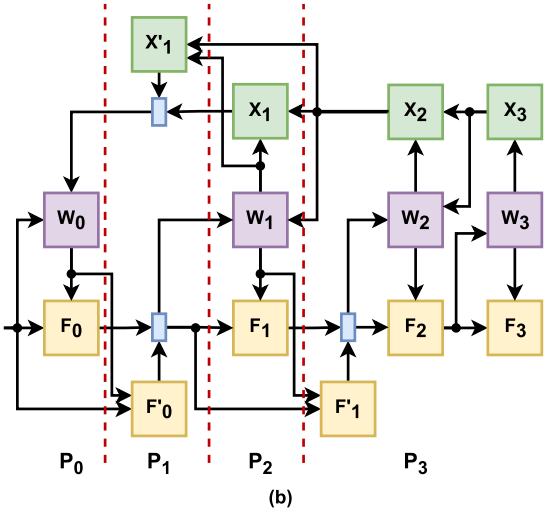


Fig. 15. (a) Schedule map for the proposed fine-grained scheduling scheme when $P = 4$ and (b) split of the DFG to four subgraphs.

Algorithm 1 First-to-last allocation scheme

▷ *Compute clock cycles for each task and store in a matrix of size* $L \times 3$
$N_C \leftarrow$ COMPUTECYCLES(Network)
▷ *Allocate tasks to* P *processors using first to last allocation scheme*
$N_{tot} \leftarrow sum(N_C[l,i])$ for $l \leftarrow 0$ to $L-1, i \leftarrow 0$ to 2
$N_{ideal} \leftarrow \frac{N_{tot}}{P}$ ▷ *Compute ideal time per processor*
$\alpha \leftarrow 1.1$
flag $\leftarrow$ True
while flag **do**
  $N_{alloc}[p] \leftarrow 0$ for $p \leftarrow 0$ to $P-1$
  Layer index, $l \leftarrow 0$
  Processor index, $p \leftarrow 0$
  Task index, $i \leftarrow 2$
  while True **do**
    if $N_{alloc}[p] + N_C[l,i] \leq N_{ideal}$ **then**
      Allocate task i of layer l to processor p
      $N_{alloc}[p] \leftarrow N_{alloc}[p] + N_C[l,i]$
      if $i = 0$ **then**
        $l \leftarrow l + 1$
        $i \leftarrow 2$
      else
        $i \leftarrow i - 1$
    else
      if $i = 1$ **then** ▷ *Cannot split weight gradient*
        if $N_C[l,i]/2 \leq (N_{ideal} - N_{alloc}[p])$ **then**
          Allocate task i of layer l to processor p
          $N_{alloc}[p] \leftarrow N_{alloc}[p] + N_C[l,i]$
          $i \leftarrow 0$
        else
          $N_{rem} \leftarrow N_{ideal} - N_{alloc}[p]$
          Split task into N_{rem} and $N_C[l,i] - N_{rem}$
          Allocate N_{rem} of task i to processor p
          $N_{alloc}[p] \leftarrow N_{alloc}[p] + N_{rem}$
          $N_C[l,i] \leftarrow N_C[l,i] - N_{rem}$
      $p \leftarrow p + 1$
    if $p = P - 1$ **then**
      if $l = L - 1$ **then**
        flag $\leftarrow$ False
      else
        $N_{ideal} \leftarrow \alpha \cdot N_{ideal}$
      break
return layer_map, N_{alloc}

Similarly, the input gradient task can be split into subtasks and moved to the previous processors. Both the forward pass and input gradient splits involve a small overhead of communicating appropriate weights to the respective processor. Splitting the weight gradient task, however, would result in a large overhead of moving either the outputs of the previous layer or the input gradient of the next layer. Therefore, the weight gradient task is not split. Using these conditions, the tasks are allotted to processors using a first-to-last allocation scheme detailed in Algorithm 1. The first-to-last algorithm allocates the tasks in the order of input gradient, weight gradient, and forward pass. The algorithm starts with assigning the first layer's tasks to the first processor and continues allocating the

tasks to the next processor until all the processors are full. If all the layers are not assigned, the ideal time constraint is relaxed, and the algorithm is rerun until all the layers are allocated. Similarly, the tasks can be allocated in the order of forward pass, input gradient, and weight gradient from the last layer to the first layer, resulting in a last-to-first allocation scheme.

For a given network, both allocation schemes are run, and the one with the better speedup is chosen. Since the weight gradient task is not split, this becomes the bottleneck for speedup. The maximum speedup for this scheme is given by (34).

$$\sigma_{finegrained} = \frac{N_{total}}{\max_l(N^l_{WG})} \tag{34}$$

For the example network, the longest weight gradient task takes $6,334$ cycles, resulting in a $\sigma_{finegrained}$ of $7.41\times$, which is more than twice that of the speedup from the PipeDream-based scheduling scheme. Figure 15(a) shows the schedule map for the proposed fine-grained scheduling scheme for $P = 4$. Figure 15(b) shows the modified DFG with the tasks split and moved to execute the schedule given by Fig. 15(a). With this split of the DFG, the number of cycles per weight update is $11,900$, resulting in a speedup of $3.95\times$, which is very close to the ideal speedup of $4\times$.

VI. Simulation Results

The proposed fine-grained scheduling scheme is simulated on the networks described in Table I and compared with the PipeDream-based scheduling scheme for a varying number of processors. Figure 16 shows the speedup of both scheduling schemes for the example network with a batch size of 1 when mapped to 32×32 systolic arrays.

To test the effect of batch size and systolic array size on the speedup, both the scheduling algorithms are simulated for different batch sizes and systolic array sizes. Figure 17 shows the mean and standard deviation of the speedup when batch size is varied from 1 to 128, and the systolic array size is varied from 16×16 to 256×256.

Figure 18 shows the individual effect of batch size and systolic array size on the speedup. Batch size does not have a significant impact on the speedup for both the scheduling schemes, as evident in Fig. 18(a), (c), (e), and (g), which show the speedup with varying batch sizes and a 32×32 systolic array. Figure 18(b), (d), (f), and (h), however, show that the systolic array size affects the speedup of both algorithms. Having smaller systolic arrays is beneficial for the PipeDream algorithm but the proposed algorithm benefits from a few specific array sizes, based on the network structure.

Table VII summarizes the speedup of both algorithms on the four networks for various processors, averaged over all batch sizes and systolic array sizes. It also shows the additional overhead incurred by the fine-grained scheduling scheme. On average, the proposed fine-grained pipelining and scheduling algorithm achieves a speedup improvement of 73.41% over the PipeDream-based algorithm. The proposed algorithm achieves more than $2\times$ speedup over the PipeDream-based algorithm when using a higher number of processors.

The overhead incurred by the fine-grained scheduling scheme is negligible compared to the total memory requirement for the MNIST, N-MNIST, and DVS Gestures networks. For the network trained on MNIST, the communication

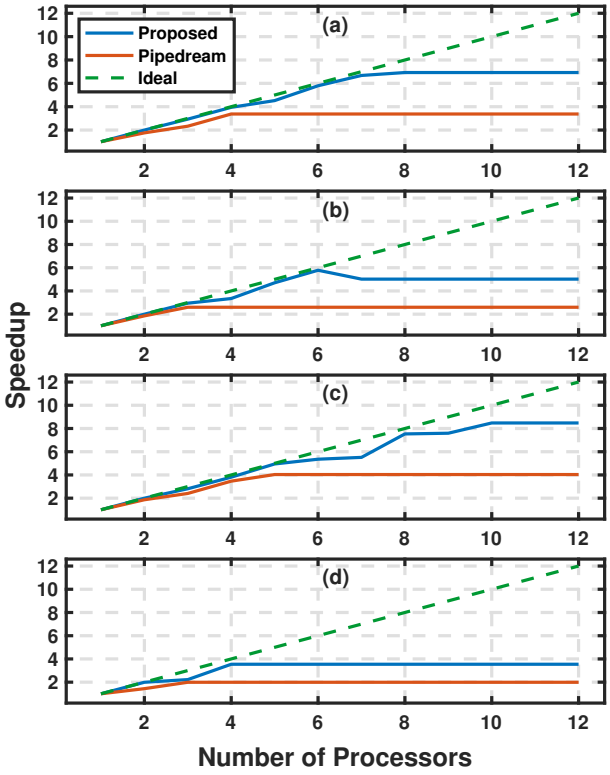


Fig. 16. Comparison of the proposed fine-grained scheduling with the PipeDream-based scheduling over single-processor implementation for the (a) MNIST, (b) N-MNIST, (c) DVS128 Gestures networks, and (d) SHD . The results are with a batch size of 1 and a 32×32 systolic array.

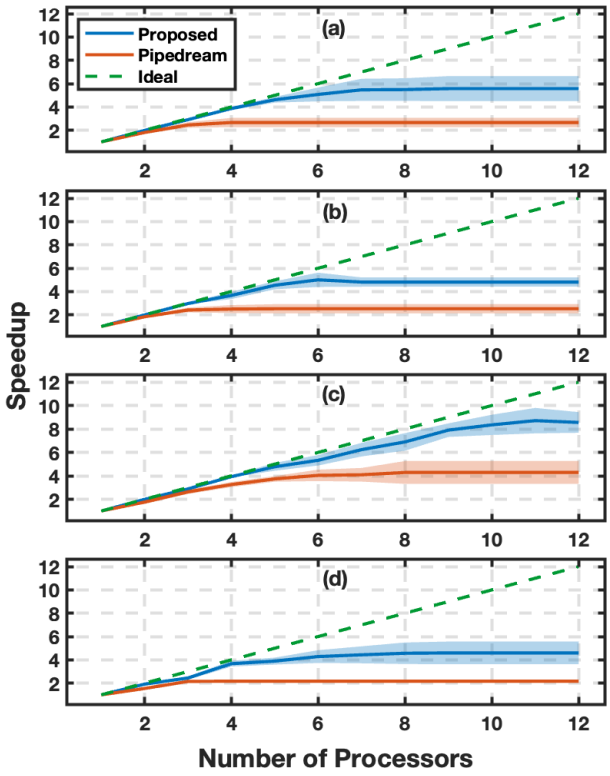


Fig. 17. Mean and standard deviation of the speedup with different batch sizes and systolic array sizes for (a) MNIST, (b) N-MNIST, (c) DVS128 Gestures, and (d) SHD networks.

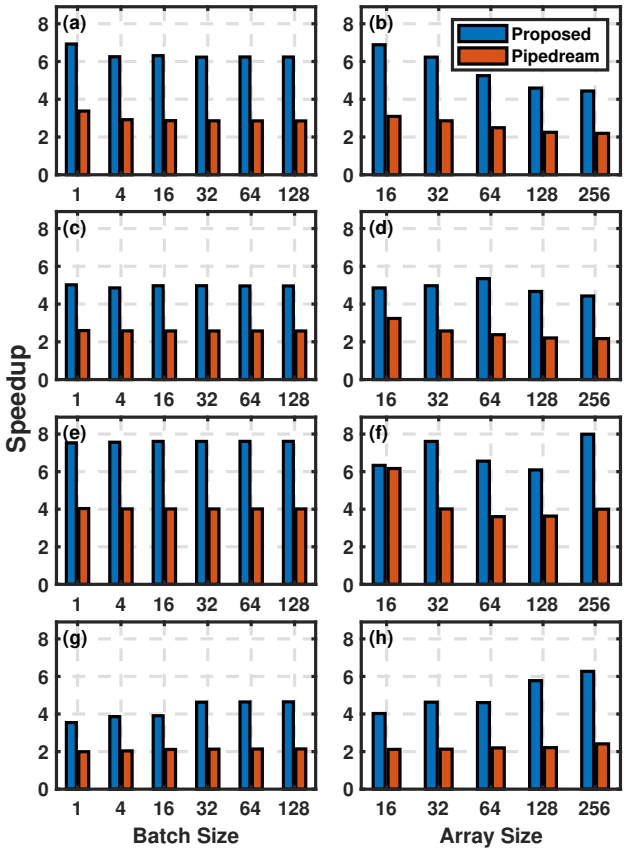


Fig. 18. Speedup with 8 processors with (a) varying batch sizes of the MNIST network, (b) varying systolic array sizes of the MNIST network, (c) varying batch sizes of the N-MNIST network, (d) varying systolic array sizes of the N-MNIST network, (e) varying batch sizes of the DVS128 Gestures network, (f) varying systolic array sizes of the DVS128 Gestures network, (g) varying batch sizes of the SHD network, and (h) varying systolic array sizes of the SHD network. Results with varying batch sizes are with a systolic array of size 32×32, and varying array sizes are with a batch size of 32. The speedup is normalized to one processor.

required to transfer intermediate outputs and gradients is 3.22 MB on average. The maximum overhead is 16.86 KB, which is 0.51% of the total communication. For the larger network trained on the DVS128 Gestures dataset, the total communication requirement is 982.98 MB on average, while the maximum overhead is $\approx$ 260 KB, which is only 0.03% of the total communication. However, the overhead for the SHD network is $\approx 10\%$ of the total communication requirement. The reason for such high overhead is the nature of the network. The network used for SHD contains only fully connected layers. The overhead incurred by the proposed algorithm is the communication cost of transferring weights to the adjacent processors, and the number of weights in a fully connected layer is much higher than in a convolutional layer. Therefore, the proposed method incurs a lower overhead in the networks with convolutional layers such as the MNIST, N-MNIST, and DVS Gestures.

The size of the SRAM required in the processors increases for large networks to store the inputs and intermediate computed variables. However, assuming the worst-case scenario of having a single processor and storing all intermediate outputs for the backward pass, the memory requirement is 11.568 MB, 44.217 MB, 2.092 GB, and 10.769 MB for the MNIST, N-MNIST, DVS128 Gestures, and SHD networks, respectively, for a batch size of 32. This requirement reduces almost linearly with an increase in number of processors.

TABLE VII
SUMMARY OF RESULTS FOR THE TWO SCHEDULING SCHEMES. RESULTS AVERAGED OVER ALL BATCH SIZES AND SYSTOLIC ARRAY SIZES. THE NETWORKS USED FOR THE DATASETS ARE DESCRIBED IN TABLE I.

Network (Memory Required)	No. of Procs.	Speedup			Overhead (KB)
		PipeDream	Fine Grain	Improv. (%)	
MNIST (11.57 MB)	1	1.00	1.00	0.00	0.00
	2	1.81	1.97	8.94	0.73
	4	2.67	3.78	41.84	2.62
	6	2.67	5.04	88.84	9.83
	8	2.67	5.49	105.86	10.40
	10	2.67	5.57	108.92	16.86
	12	2.67	5.57	108.92	16.86
N-MNIST (44.22 MB)	1	1.00	1.00	0.00	0.00
	2	1.83	2.00	8.92	0.56
	4	2.49	3.67	50.67	2.08
	6	2.52	5.00	101.55	4.03
	8	2.52	4.81	94.02	3.92
	10	2.52	4.81	94.02	3.92
	12	2.52	4.81	94.02	3.92
DVS128 Gestures (2.09 GB)	1	1.00	1.00	0.00	0.00
	2	1.76	1.99	14.31	9.00
	4	3.26	3.93	21.35	54.60
	6	4.04	5.33	33.70	96.64
	8	4.29	6.89	67.60	133.95
	10	4.29	8.35	102.75	194.10
	12	4.29	8.56	106.39	260.18
	14	4.29	9.04	113.98	252.56
	16	4.29	9.87	134.73	254.70
SHD (10.77 MB)	1	1.00	1.00	0.00	0.00
	2	1.55	1.92	25.23	396.67
	4	2.17	3.67	69.74	668.67
	6	2.17	4.28	96.88	803.07
	8	2.17	4.57	108.84	992.00
	10	2.17	4.60	110.01	992.00
	12	2.17	4.60	110.01	992.00

VII. CONCLUSION

This paper discusses the general modeling of Spiking Neural Networks based on the Leaky Integrate-and-Fire model, and their training using the backpropagation algorithm. The dataflow graph of training is then pipelined and retimed using delayed gradients in an attempt to map it to multiple processors. The design of typical systolic array-based processors is discussed, along with their modeling to estimate clock cycles necessary for executing various tasks. Using the estimated clock cycles, the dataflow graph is split in various ways using pre-existing algorithms. A fine-grained pipelining and scheduling scheme is then proposed to improve the throughput of training over conventional methods. The proposed scheme is evaluated on four networks, and the results show an average of $\approx 73\%$ improvement in throughput with upward of $> 100\%$ improvement in some cases, with a small drop in accuracy for larger networks. The overhead incurred by the proposed scheme is $\leq 0.5\%$ compared to the total communication requirement of the network for networks with convolutional layers. The proposed scheme assumes that the neural networks have only convolutional or fully-connected layers. The future scope of this work includes extending the proposed scheme to

networks with other types of layers, such as normalization and residual layers, which can improve the accuracies of deeper networks. Further, a hybrid multiprocessor training and single processor fine-tuning approach can be explored to improve the accuracy of the networks trained using the proposed scheme.

REFERENCES

[1] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659-1671, 1997.

[2] W. Gerstner and W. Kistler, "Spiking Neuron Models: Single Neurons, Populations, Plasticity," *Cambridge University Press*, 2002.

[3] M. Pfeiffer and T. Pfeil, "Deep Learning With Spiking Neurons: Opportunities and Challenges," *Frontiers in Neuroscience*, vol. 12, 2018.

[4] K. K. Parhi and N. K. Unnikrishnan, "Brain-Inspired Computing: Models and Architectures," *IEEE Open J. of Circuits and Systems*, vol. 1, pp. 185-204, 2020.

[5] A. Tavanaei et al., "Deep learning in spiking neural networks," *Neural Networks*, vol. 111, pp. 47-63, 2019.

[6] W. Yujie et al., "Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks," *Frontiers in Neuroscience*, vol. 12, 2018.

[7] H. Fang et al., "Encoding, Model, and Architecture: Systematic Optimization for Spiking Neural Network in FPGAs," *IEEE/ACM Intl. Conf. On Computer Aided Design*, pp. 1-9, 2020.

[8] N. Anwani and B. Rajendran, "Training multi-layer spiking neural networks using NormAD based spatio-temporal error backpropagation," *Neurocomputing*, vol. 380, pp. 67-77, 2020.

[9] J. H. Lee, T. Delbruck, and M. Pfeiffer, "Training Deep Spiking Neural Networks Using Backpropagation," *Frontiers in Neuroscience*, vol. 10, 2016.

[10] E. Stromatias, M. Soto, T. Serrano-Gotarredona, and B. Linares-Barranco, "An Event-Driven Classifier for Spiking Neural Networks Fed with Synthetic or Dynamic Vision Sensor Data," *Frontiers in Neuroscience*, vol. 11, 2017.

[11] J. C. Thiele, O. Bichler, and A. Dupret, "Event-Based, Timescale Invariant Unsupervised Online Deep Learning With STDP," *Frontiers in Neuroscience*, vol. 12, 2018.

[12] L. F. Abbott, "Lapicque's introduction of the integrate-and-fire model neuron (1907)," *Brain Research Bulletin*, vol. 50, nos. 5/6, pp. 303-304, 1999.

[13] E. M. Izhikevich, "Simple model of spiking neurons," in *IEEE Trans. on Neural Networks*, vol. 14, no. 6, pp. 1569-1572, Nov. 2003.

[14] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The J. of Physiology*, vol. 117, no. 4, pp. 500-544, Aug. 1952.

[15] B. D. Sahoo, "Ring oscillator based sub-1V leaky integrate-and-fire neuron circuit," *IEEE Intl. Symp. on Circuits and Systems*, pp. 1-4, Baltimore, MD, USA, 2017.

[16] S. Sanjeet, R. K. Meena, B. D. Sahoo, K. K. Parhi, and M. Fujita, "IIR Filter-Based Spiking Neural Network," *IEEE Intl. Symp. on Circuits and Systems*, pp. 1-5, Monterey, CA, USA, 2023.

[17] C. Lee, S. S. Sarwar, P. Panda, G. Srinivasan, and K. Roy, "Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures," *Frontiers in Neuroscience*, vol. 14, 2020.

[18] Y. Dan and M. M. Poo, "Hebbian depression of isolated neuromuscular synapses in vitro," *Science*, vol. 256, no. 5063, pp. 1570-1573, 1992.

[19] C. Lee, P. Panda, G. Srinivasan, and K. Roy, "Training Deep Spiking Convolutional Neural Networks With STDP-Based Unsupervised Pre-training Followed by Supervised Fine-Tuning," *Frontiers in Neuroscience*, vol. 12, 2018.

[20] C. Lee, G. Srinivasan, P. Panda, and K. Roy, "Deep Spiking Convolutional Neural Network Trained With Unsupervised Spike-Timing-Dependent Plasticity," *IEEE Trans. on Cognitive and Developmental Systems*, vol. 11, no. 3, pp. 384-394, Sept. 2019.

[21] T. Ben-Nun and T. Hoefler, "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis," *ACM Computing Surveys*, vol. 52, no. 4, Aug. 2019.

[22] Y. Huang et al., "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in Neural Information Processing Systems*, vol. 32, pp. 103-112, 2019.

[23] D. Narayanan et al., "PipeDream: Generalized Pipeline Parallelism for DNN Training," *Proc. of the 27^{th} ACM Symp. on Operating Systems Principles*, pp. 1-15, Oct. 2019.

[24] L. Zhao et al., "BaPipe: Exploration of Balanced Pipeline Parallelism for DNN Training," *arXiv:2012.12544,* Dec. 2020.

[25] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt, "FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters," *IEEE Trans. on Computers*, vol. 69, no. 8, pp. 1143-1158, Aug. 2020.

[26] N. K. Unnikrishnan and K. K. Parhi, "LayerPipe: Accelerating Deep Neural Network Training by Intra-Layer and Inter-Layer Gradient Pipelining and Multiprocessor Scheduling," *IEEE/ACM Intl. Conf. On Computer Aided Design*, pp. 1-8, Nov. 2021.

[27] G. Long, F. Ling, and J. G. Proakis, "The LMS algorithm with delayed coefficient adaptation," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 37, no. 9, pp. 1397-1405, Sept. 1989.

[28] L. Liang et al., "H2Learn: High-Efficiency Learning Accelerator for High-Accuracy Spiking Neural Networks," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4782-4796, Nov. 2022.

[29] S. Singh et al., "Skipper: Enabling efficient SNN training through activation-checkpointing and time-skipping," *IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 565-581, Chicago, IL, USA, 2022.

[30] R. Yin, A. Moitra, A. Bhattacharjee, Y. Kim, and P. Panda, "SATA: Sparsity-Aware Training Accelerator for Spiking Neural Networks," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 6, pp. 1926-1938, June 2023.

[31] N. R. Shanbhag and K. K. Parhi, "Relaxed look-ahead pipelined LMS adaptive filters and their application to ADPCM coder," *IEEE Trans. on Circuits and Systems II*, vol. 40, no. 12, pp. 753-766, Dec. 1993.

[32] D.P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980*, Dec. 2014.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proc. of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.

[34] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades," *Frontiers in Neuroscience*, vol. 9, 2015.

[35] A. Amir et al., "A Low Power, Fully Event-Based Gesture Recognition System," *IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 7388-7397, Honolulu, HI, USA, 2017.

[36] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke, "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 7, pp. 2744-2757, 2022.

[37] X. Zhu, B. Zhao, D. Ma, and H. Tang, "An Efficient Learning Algorithm for Direct Training Deep Spiking Neural Networks," *IEEE Trans. on Cognitive and Developmental Systems*, vol. 14, no. 3, pp. 847-856, Sept. 2022.

[38] W. He et al., "Comparing SNNs and RNNs on neuromorphic vision datasets: Similarities and differences," *Neural Networks*, vol. 132, pp. 108-120, 2020.

[39] P. Sun, E. Eqlimi, Y. Chua, P. Devos, and D. Botteldooren, "Adaptive Axonal Delays in Feedforward Spiking Neural Networks for Accurate Spoken Word Recognition," *IEEE Intl. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1-5, Rhodes Island, Greece, 2023.

[40] H. T. Kung and C. E. Leiserson, "Systolic Arrays for (VLSI)," *Technical Report*, Carnegie-Mellon University, 1978.

[41] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation.* Wiley, 1999.

[42] X. Wei et al., "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," *Proc. of the 54^{th} Annual Design Automation Conf.*, pp. 1-6, Jun. 2017.

[43] N.P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," *ACM/IEEE 44^{th} Annual Intl. Symp. on Computer Architecture*, pp. 1-12, Jun. 2017.

[44] A. Samajdar et al., "A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim," *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pp. 58-68, Boston, USA, 2020.

Sai Sanjeet received the Dual Degree (B. Tech. and M. Tech.) in Electronics and Electrical Communication Engineering from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 2021 with specialization in the area of Visual Image Processing and Embedded Systems. He is currently pursuing his PhD at the University at Buffalo, State University of New York, Buffalo, in the area of analog and mixed-signal hardware accelerators for machine learning applications. His research interests include machine learning, neuro-inspired computing, analog and mixed-signal circuit design, and signal processing. He was part of a collaborative group from the University of Tokyo which received 3rd place in the 30th Intl. Workshop on Logic and Synthesis (IWLS) Design Contest 2021. He is also a recipient of the National Talent Search Examination (NTSE) fellowship and the Kishore Vaigyanik Protsahan Yojana (KVPY) fellowship.

Bibhu Datta Sahoo received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 1998, the M.S.E.E. degree from the University of Minnesota, Minneapolis, MN, USA, in 2000, and the Ph.D.E.E. degree from the University of California, Los Angeles, CA, USA, in 2009. From 2000 to 2006, he was with Broadcom Corporation, Irvine, CA, USA, From December 2008 to February 2010, he was with Maxlinear Inc., Carlsbad, CA, USA, where he was involved in designing integrated circuits for CMOS TV tuners. From March 2010 to November 2010, he was a Post-Doctoral Researcher with the University of California, Los Angeles, CA, USA. From December 2011 to April 2015, he was an Associate Professor with the Department of Electronics and Communication Engineering, Amrita University, Amritapuri, India. From January 2016 to April 2017 he was on sabbattical from Amrita University and was a Research Scientist at University of Illinois at Urbana-Champaign. From August 2017 to August 2023 he has been Associate Professor in the Department of Electronics and Electrical Communication Engineering at Indian Institute of Technology Kharagpur, Kharagpur, India. Since September 2023 he has been a Professor in the Department of Electrical Engineering at University at Buffalo, State University of New York. His research interests include data converters, signal processing, and analog and mixed signal circuit design. He received the 2008 Analog Devices Outstanding Student Designer Award and was the co-recipient of the 2013 CICC Best Paper Award. He was the Associate Editor of IEEE Transactions on Circuits and Systems-II from August 2014 to December 2015. Since Dec. 2018 he has been the Associate Editor of IEEE Open Journal of Circuits and Systems.

Keshab K. Parhi (Fellow, IEEE) received the B.Tech. degree from the Indian Institute of Technology (IIT), Kharagpur, in 1982, the M.S.E.E. degree from the University of Pennsylvania, Philadelphia, in 1984, and the Ph.D. degree from the University of California, Berkeley, in 1988. He has been with the University of Minnesota, Minneapolis, since 1988, where he is currently the Erwin A. Kelen Chair and a Distinguished McKnight University Professor in the Department of Electrical and Computer Engineering. He has published over 740 papers, is the inventor of 36 patents, and has authored the textbook VLSI Digital Signal Processing Systems (Wiley, 1999). His current research addresses VLSI architecture design of machine learning and signal processing systems, hardware security, and data-driven neuroengineering and neuroscience with applications to computational neurology and psychiatry. Dr. Parhi is the recipient of numerous awards including the 2003 EEE Kiyo Tomiyasu Technical Field Award; and the 2017 Mac Van Valkenburg award, the 2012 Charles A. Desoer Technical Achievement award, and a Golden Jubilee medal in 1999, from the IEEE Circuits and Systems Society. He served as the Editor-in-Chief of the IEEE Trans. Circuits and Systems, Part-I during 2004 and 2005, and currently serves as the Editor-in-Chief of the IEEE Circuits and Systems Magazine. He is a Fellow of the American Association for the Advancement of Science (AAAS), the Association for Computing Machinery (ACM), the American Institute of Medical and Biological Engineering (AIMBE), and the National Academy of Inventors (NAI).