An FPGA Implementation of Deep Spiking Neural Networks for Low-Power and Fast Classification

Xiping Ju
xpju1104@163.com
Biao Fang
afetsc2008@sina.com
Rui Yan
ryan@scu.edu.cn
College of Computer Science, Sichuan University, Chengdu 610065, China

Xiaoliang Xu
xxl@hdu.edu.cn
School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China

Huajin Tang
huajin.tang@gmail.com
College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China, and College of Computer Science, Sichuan University, Chengdu 610065, China

A spiking neural network (SNN) is a type of biological plausibility model that performs information processing based on spikes. Training a deep SNN effectively is challenging due to the nondifferention of spike signals. Recent advances have shown that high-performance SNNs can be obtained by converting convolutional neural networks (CNNs). However, the large-scale SNNs are poorly served by conventional architectures due to the dynamic nature of spiking neurons. In this letter, we propose a hardware architecture to enable efficient implementation of SNNs. All layers in the network are mapped on one chip so that the computation of different time steps can be done in parallel to reduce latency. We propose new spiking max-pooling method to reduce computation complexity. In addition, we apply approaches based on shift register and coarsely grained parallels to accelerate convolution operation. We also investigate the effect of different encoding methods on SNN accuracy. Finally, we validate the hardware architecture on the Xilinx Zynq ZCU102. The experimental results on the MNIST data set show that it can achieve an accuracy of 98.94% with eight-bit quantized weights. Furthermore, it achieves 164 frames per second (FPS) under 150 MHz clock frequency and obtains 41× speed-up compared to CPU implementation and 22 times lower power than GPU implementation.

https://doi.org/10.1162/neco_a_01245

1 Introduction

Neuroscience has provided lots of inspiration for the advancement of artificial intelligence (AI) algorithms and hardware architecture. Highly simplified abstractions of neural networks are now revolutionizing computing by solving difficult and diverse machine learning problems (Davies et al., 2018). A spiking neural network (SNN) is a type of biologically inspired neural network that processes information based on spikes. The spiking neuron integrates input spikes over time and fires a spike when its membrane potential crosses a threshold. It is challenging to train a deep SNN effectively to achieve high precision (Lee, Delbruck, & Pfeiffer, 2016) due to the lack of efficient learning algorithms. Discrete spike signals produced by a spiking neuron are not differentiable; thus, the backpropagation (BP) rule can not be directly applied to the SNN.

One solution is to train deep SNNs based on variants of the backpropagation algorithm. Samadi, Lillicrap, and Tweed (2017) showed that deep SNNs can be trained by applying a variant of the feedback alignment algorithm (Lillicrap, Cownden, Tweed, & Akerman, 2016) and using an approximation of neuron dynamics. A four-layer network was trained and achieved 97% accuracy on the MNIST. Wu, Deng, Li, Zhu, and Shi (2018) proposed a spatiotemporal backpropagation algorithm and achieved 99.42% accuracy on the MNIST with a convolutional spiking neural network.

An alternative way to obtain high-performance SNNs is to convert traditional artificial neural networks (ANNs). Cao, Chen, and Khosla (2015) converted a convolutional neural network into an SNN architecture that is suitable for mapping to spike-based neuromorphic hardware. They used a rectified linear unit (ReLU) as the activation function to avoid negative values and set all biases to zero. Furthermore, the max-pooling operations were replaced by average-pooling operations. Their method was improved by Diehl et al. (2015) to minimize performance loss through weight normalization, which regulates firing rates to reduce the errors due to the probabilistic nature of the spiking input. Rueckauer, Lungu, Hu, Pfeiffer, and Liu (2017) expanded the class of networks that can be converted, which enables networks with max-pooling, softmax, batch normalization, and biases to be converted as well. However, deep SNNs are poorly served by conventional architectures (e.g. CPU) due to the dynamic nature of neurons (Davies et al., 2018). Therefore, it is necessary to accelerate SNN applications through dedicated hardware.

There are various ways to improve SNN computation efficiency based on different hardware platforms. The IBM TrueNorth processor of digital logic implementation integrates 1 million spiking neurons and 256 million configurable synapses (Merolla et al., 2014). Real-time multiobject detection and classification consumed only 63 milliwatts. The ROLLS processor is a mixed-signal very large scale integration (VLSI) with neuromorphic learning circuits, which supports a 128K analog synapse and 256 neurons (Qiao et al., 2015). There are also attempts to implement neuromorphic computing based on emerging devices such as memristors (Querlioz, Bichler, & Gamrat, 2011). These neuromorphic platforms, which are based on event-driven computation, can be several orders of magnitude more efficient in terms of power consumption than conventional accelerators or GPUs. However, neuromorphic hardware faces some limitations on the neural network to be implemented (such as maximum fan-in/fan-out of a single neuron, synaptic precision, and type of neuron models; Ji et al., 2016).

Field-programmable gate array (FPGA) acts as a programmable device that allows the development of custom logic, which can relax restrictions on neural networks to be implemented. It has rich computing resources and provides a shorter development period than ASICs. Furthermore, a successful FPGA implementation can be seen as a step toward lower power custom chips. In this work, we use FPGA to accelerate the simulation of deep SNNs derived from network conversion, enabling high-performance SNN applications.

We implement a deep SNN on the Xilinx Zynq ZCU102 board using synthesizable Verilog. In network conversion, we propose a new spiking max-pooling operation, whose basic idea is similar to the approach mentioned in Hu (2016) but with lower implementation complexity. In the hardware architecture, a hardware layer corresponds to each layer in the network (i.e., all the layers are mapped on one chip). These layers could form different computing stages that work concurrently. Thus, the computation of different time steps can be parallel to reduce latency. Furthermore, a shift register–based convolution operation and a coarsely grained parallel approach are adopted to improve the data reuse rate. The FPGA implementation of SNN achieves 164 frames per second (FPS) under 150 MHz clock frequency and obtains 41 times speed-up compared to CPU implementation and 22 times lower power than GPU implementation. Compared to the Minituar (Neil & Liu, 2014) and Darwin chip (Ma et al., 2017), it achieves 24.9 times and 26.2 times speed-up, respectively.

2 Methods

2.1 Neuron Model. In a traditional ANN, the continuous function is used as the activation of the neuron to realize the processing of real numbers. A whole input vector is presented at one time and processed layer by layer, producing output values. However, spiking neural networks use the timing of spikes to encode information. The current activation level is normally considered to be the neuron's state, with incoming spikes pushing this value higher and then either firing or decaying over time (Maass, 1997). The neuron model used in our work is an integrate-and-fire (IF) model without a refractory period; the membrane potential V of each neuron evolves according to the following equations:

$$V(t) = V(t-1) + \sum_i x_i(t-1)w_i + L, \tag{2.1a}$$

$$\text{if } V(t) \geq V_{thr}, \text{ fire and reset } V(t) = V(t) - V_{thr}, \tag{2.1b}$$

$$\text{if } V(t) < V_{\min}, \text{ reset } V(t) = V_{\min}, \tag{2.1c}$$

where $x_i(t-1)$ denotes the input value of the ith neuron at time $t-1$, w_i is the corresponding weight; and L is the constant leakage parameter. When the membrane potential V crosses the threshold V_{thr}, the neuron fires a spike, and V is subtracted by the threshold V_{thr}. The reset-by-subtraction mechanism makes the conversion scheme suitable for deeper networks (Rueckauer et al., 2017). If the membrane potential is less than $V_{\min}$, the $V(t)$ will be reset to $V_{\min}$. We set $V_{thr} = 1$, $V_{\min} = 0$ and $L = 0$ here.

2.2 Network Conversion. The basic idea of conversion is to approximate the activation value of the neuron in ANN by the firing rate of the same neuron in SNN, that is, from a rate-based model to a spike-based model (Rueckauer et al., 2017). The network conversion has three steps. First, a traditional ANN is trained with the backpropagation algorithm. Because it is hard for SNNs to represent negative values by spikes, we take ReLU as the activation functions in the original ANN to avoid negative values (Cao et al., 2015). The ReLU function is defined as

$$a_i^l = \max\left(0, \sum_j w_{ij}^l a_j^{l-1}\right), \tag{2.2}$$

where a_i^l is the activation value of neuron i in layer l and $\sum_j w_{ij}^l a_j^{l-1}$ is the weighted sum of the input. Second, the weight normalization (Diehl et al., 2015) operation is used to normalize output values of each layer in the trained ANN into [0, 1], which are relative to firing rates of spiking neurons in the converted SNN. The range of the output values of different layers varies a lot in the trained ANN and would cause large conversion accuracy loss, so weights are scaled to normalize the output values. Third, the scaled weights are used as weights of the corresponding spiking neural network while maintaining the connection mode. Finally, the converted SNN is obtained after replacing all neurons with spiking neurons.

In this work, a seven-layer convolutional neural network has been converted into a spiking neural network. The structure of the network is 28×28-64c5-2s-64c5-2s-128f-10o, as shown in Figure 1. The input layer consists of 784 neurons representing pixels in a 28×28 input image. The first convolutional layer filters the 28×28 input image with 64 kernels with a size of 5×5, followed by a max-pooling layer with a kernel size of 2×2.

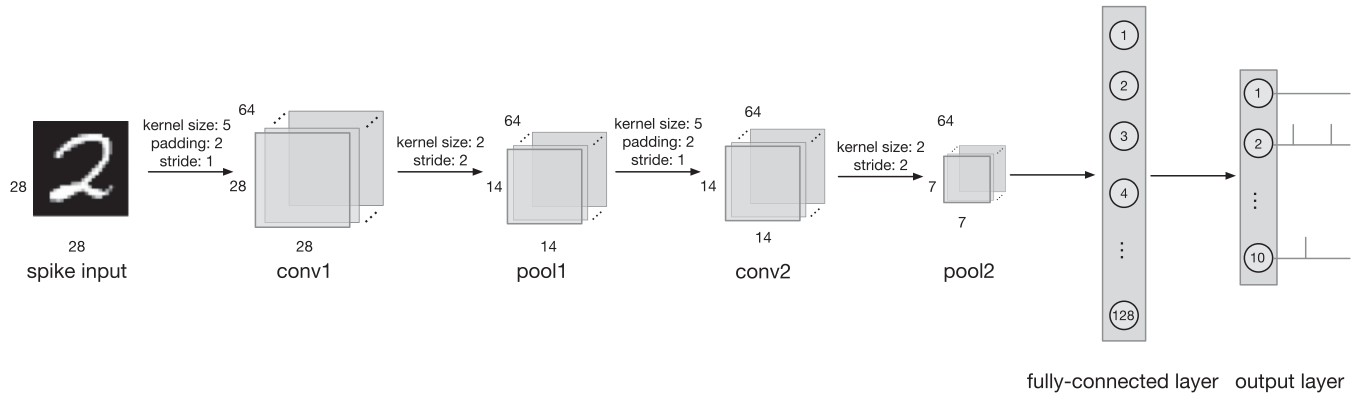

Figure 1: An illustration of the SNN architecture. Input images are encoded to spike trains and then fed into the SNN. The input is 28×28, and the number of neurons in the remaining layers is given by 50176-12544-12544-3136-128-10. All neurons in the network are spiking neurons.

The second convolutional layer filters the first pooling layer with 64 kernels of size $5 \times 5 \times 64$, followed by a max-pooling layer that does 2×2 pooling. The fully connected layer has 128 neurons. The 128 neurons are fully connected to 10 output neurons, each representing a digit of 0 to 9.

Max-pooling, which acts as a form of nonlinear subsampling, not only reduces the computation complexity of subsequent layers by decreasing the dimension of feature maps but also provides a slight translation invariance (Goodfellow, Bengio, & Courville, 2016). However, implementing it in hardware is challenging because the firing rates of neurons need to be evaluated at every time step. Rueckauer et al. (2017) proposed a gating function–based mechanism for spiking max-pooling, which estimates presynaptic firing rates at every moment and lets only spikes from the maximally firing neuron pass. Specifically, Hu (2016) proposed three implementations of the max-pooling operation. One of the approaches, Fir.Max, calculates the absolute spike rate and accumulates it over time. The firing rate for this method is estimated as follows,

$$f_i(t) = f_i(t-1) + \frac{x_i(t)}{t}, \tag{2.3}$$

where $f_i(t)$ denotes the firing rate of the neuron i at time t. $x_i(t)$ represents the presence (1) or absence (0) of a spike at time t. If the neuron i fires a spike, its firing rate will be updated. However, the formula includes division part $\frac{x_i(t)}{t}$, and the corresponding hardware circuit is complex. Motivated by the simplicity, we propose a hardware-friendly approach for estimating the firing rate. The firing rate is updated as follows,

$$f_i(t) = f_i(t-1) + x_i(t)(T-t), \tag{2.4}$$

where T is the length of the time window for one presentation. After each presentation, the firing rate will be reset to 0. The basic idea of this method is similar to Fir.Max. The later-arriving spikes have smaller impacts on firing rates of neurons. It takes inspiration from the time domain winner-take-all (WTA) operation (Zhao, Chen, & Tang, 2014; Zhao, Ding, Chen, Linares-Barranco, & Tang, 2015). Besides, the firing rates of other neurons may exceed that of the neuron fired earlier when they fire several times although they fire later. In our formula, there are only addition and subtraction operations, leading to simpler hardware implementation. Moreover, the conversion loss of our method is almost the same as the Fir.Max (Hu, 2016).

2.3 Input Encoding. After conversion, input values of typical deep ANNs need to be encoded into spike patterns to fit the spiking neuron dynamics. One commonly used method to generate input spike trains is the Poisson-distributed spike encoding (Cao et al., 2015; Diehl et al., 2015). At time t, the input neuron generates a spike if (we call Poisson random)

$$rand() < cx \tag{2.5}$$

where $rand()$ is a random number generator uniformly distributed on (0,1), c is the scaling factor used to adjust neurons' firing rates, and x is the normalized pixel value of the input image.

There is an alternative way to generate Poisson-distributed spike trains based on the interspike interval (ISI)—the interval time between two successive spikes. This ISI irregularity can reflect a rich bandwidth for information transfer (Heeger, 2000). The interval time between every two successive spikes can be produced by the following equation (we call it Poisson-ISI):

$$I = \frac{-ln(U)}{\lambda}, \tag{2.6}$$

where I is the interval time between spikes, U is a random number generated from uniform distribution on (0,1), and λ is the firing rate. The result of cumulating the Is is the sequence of time in which the neuron fires.

These methods, however, introduce variability into the firing of the network and impair network performance (Rueckauer et al., 2017). The number of spikes and the time of spikes generated are uncertain due to the randomness. Here we use the fixed uniform encoding method to eliminate input uncertainty (Xu, Tang, Xing, & Li, 2017), as shown in algorithm 1. f_{rate} denotes the normalized input pixel value. N_{fire} is the total number of spikes that a neuron will fire. T is the length of the time window. $f_{interval}$ stores the interval time between two successive spikes. F_{time} includes the firing time of each spike. The encoding steps are as follows: (1) the pixel values of the image are normalized to values between 0 and 1 as firing rates of neurons; (2) we compute the total number of spikes fired according to the length of the time window T; (3) these spikes are uniformly distributed along the time window. Obviously the number of spikes fired is determinate, and the distribution of the spikes is uniform for one input neuron. The network using the fixed uniform encoding method has the highest accuracy among the three encoding methods (see section 4.2).

We analyze the effect of different encoding methods based on the reconstruction results of input spike trains as shown in Figure 2. For example, the digit 9, which is randomly selected from the MNIST data set, is encoded using the three encoding methods already noted. Setting the time window to 10 ms and the time step to 1 ms, the size of encoding results will be $10 \times 28 \times 28$. Encoding results can be reconstructed by the spikes fired at the same moment, so there are 10 reconstruction results for every input spike train. Compared with the reconstruction results of Poisson-ISI and Poisson-random (see Figures 2A and 2B), the digits from the reconstruction results of fixed uniform spike trains (see Figure 2C) are the sharpest and smoothest, leading to higher similarity with the original MNIST image (displayed in the right-most column of each row in Figure 2). We found that

Algorithm 1: Fixed Uniform Spike Trains Generation Algorithm.

```
Input: the vector of normalized pixel values $I$, the length of time window $T$;
Output: the encoding result;
for each input $f_{rate} \in I$ do
    initialize a spike sequence in which no spike fires;
    if $f_{rate} > 0$ then
        $N_{fire} = round(T \times f_{rate})$ ;    // round to nearest integer
        $f_{interval} = \frac{1}{f_{rate}}$;
        for j in 1 to $N_{fire}$ do
            $F_{time}[j] = round(j \times f_{interval})$;
        end
        set the neuron to fire spikes according to values in $F_{time}$;
    end
end
```

the higher the similarity is between the reconstructed results and the original MNIST image, the better classification result will have. The basic idea of network conversion is to approximate the output of the neuron in ANN by the firing rate of the same neuron in SNN; we think the input stimuli can be more stable if the reconstructed result is more similar to the original MNIST digit at each time step. Thus, the average firing rates of neurons in the entire time window is more similar to the output of neurons in the ANN, and the time window required to approximate the output of neurons is also smaller. Furthermore, the precision gap among three encoding methods can be extended to more complicated tasks or deeper networks (Xu et al., 2017).

3 System Architecture

The overall architecture diagram for the deep SNN is shown in Figure 3. Multiple time steps are needed to carry out the inference of an image. The controller module controls the advancement of the time step and parallel computation of the different time steps (mentioned in section 3.4). The SNN

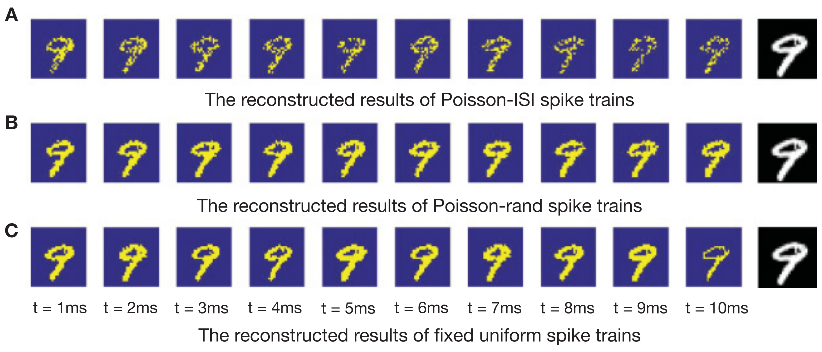

Figure 2: Reconstruction results of the input spike trains. We set the time window to 10 ms and the time step to 1 ms, so the reconstruction results consist of 10 reconstruction images, each representing input data at different moments, from left to right. The right-most column of each row is the original MNIST digit 9. (A) The reconstruction results of Poisson-ISI spike trains; the pixels that make up the digit 9 are very scattered and not smooth enough. (B) The reconstructed results of Poisson-rand spike trains. (C) The reconstructed results of fixed uniform spike trains.

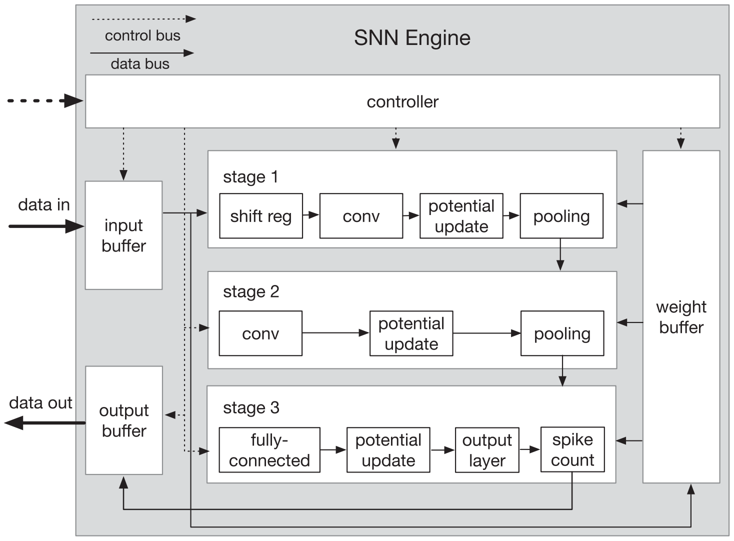

Figure 3: The hardware architecture of the SNN engine. The inference is divided into three stages: the first convolution and pooling operation (stage 1), the second convolution and pooling operation (stage 2), and the fully connected and output layer computation (stage 3). Three stages can work concurrently when the data are ready.

engine can be integrated with the CPU (e.g., the ARM processor in the Zynq processing system) to form a complete system on chip (SoC).

3.1 Fixed-Point Calculation. Most contemporary deep neural networks typically use high precision (e.g., 32-bit) neurons and synapses to provide continuous derivatives and support incremental changes to network state (Esser et al., 2016). However, such high precision for inference is not necessary due to the redundancy of networks. To reduce computational complexity and save hardware resources, the float-point variables need to be converted to fixed-point variables. We perform floating-to-fixed-point conversion by defining $v(t) = V(t) \cdot 2^{\beta}$ as the neuron status, $W_i = w_i \cdot 2^{\beta}$ as the synaptic weights, and $v_{thr} = V_{thr} \cdot 2^{\beta}$ as the neuron threshold in equation 2.1a. β is the integer that represents the scaling factor. Equation 2.1 is converted into the following fixed-point equations:

$$v(t) = v(t-1) + \sum_i x_i(t-1) W_i, \tag{3.1a}$$

$$\text{if } v(t) \geq v_{thr}, \text{ fire and reset } v(t) = v(t) - v_{thr}, \tag{3.1b}$$

$$\text{if } v(t) < 0, \text{ reset } v(t) = 0. \tag{3.1c}$$

In our work, we use eight-bit fixed-point numbers to represent weights. Thus, the value of β is 7 because the highest bit of the eight-bit number represents the sign (0 for positive values, 1 for negative values). The approach to convert values and operations from floating point to fixed point can be seen as an 8-bit uniform quantization (Sze, Chen, Yang, & Emer, 2017).

3.2 Convolution Parallel. There exist many convolution operations in the converted deep spiking neural network, so we optimize the architecture for convolution operations. We apply different parallel methods for the two convolutional layers due to the constraints of hardware resources.

3.2.1 Shift Register. The convolution computation of the first convolutional layer combines the shift register and the pipeline. After the pipeline is filled, a result is produced in every clock period. When the membrane potential of adjacent neurons on the output feature map is updated, the input values of their corresponding convolution windows overlap. As shown in Figure 4A, the size of the input feature map (fmap) is 5×5, the size of the filter is 3×3, and the stride is 1, so the size of the output feature map is also 3×3. To calculate the value of $A1$, we have to get input values from its corresponding convolution windows (the values in the red box) and then the next neighboring convolution window (the values in blue box) for calculating $A2$, where some values have already been received in the previous computation (the orange elements). In order to reduce the time of data transfer and improve the data reuse rate, a shift register is used to store the values

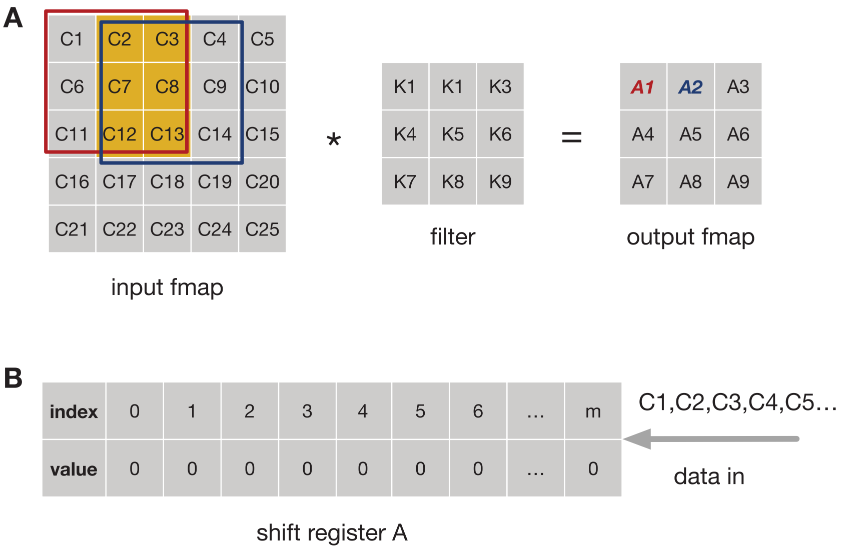

Figure 4: The basic convolution operation and the shift register. (A) The basic convolution operation. (B) The values of the input feature map are expanded by row and then fed into the shift register A. The length of the shift register A is $m + 1$.

of the input feature map. It receives new value at the input while removing old data every clock cycle.

The values in the shift register are initialized to 0. Then the values of the two-dimensional input feature map are expanded by row and fed into the shift register, as shown in Figure 4B. Suppose the size of the input feature map is $i \times i$ and the size of the filter is $k \times k$; then the number of computing units (CUs) used for multiplication is determined by the size of filter ($k \times k$). These CUs form a two-dimensional computing array. Every computing unit completes a multiplication of convolution operation. The input values of the right-most column of the computing array are provided by the first, $(i \times 1 + 1)$th , . . . , $(i \times (k - 1) + 1)$th elements in the shift register A, respectively. The input values of CUs will move to the left in the computing array every clock cycle.

For example, for the convolution operation shown in Figure 4A, the input values of the right-most column of the computing array are provided by the first, sixth, and eleventh elements in the shift register A. In clock T, the sliding window corresponding to CUs is shown in Figure 5A. The computing results are invalid because values in the sliding window are incomplete at this time. The shift register A will shift in the data presented at input and shift out the last bit in the array, and data in the computing array will

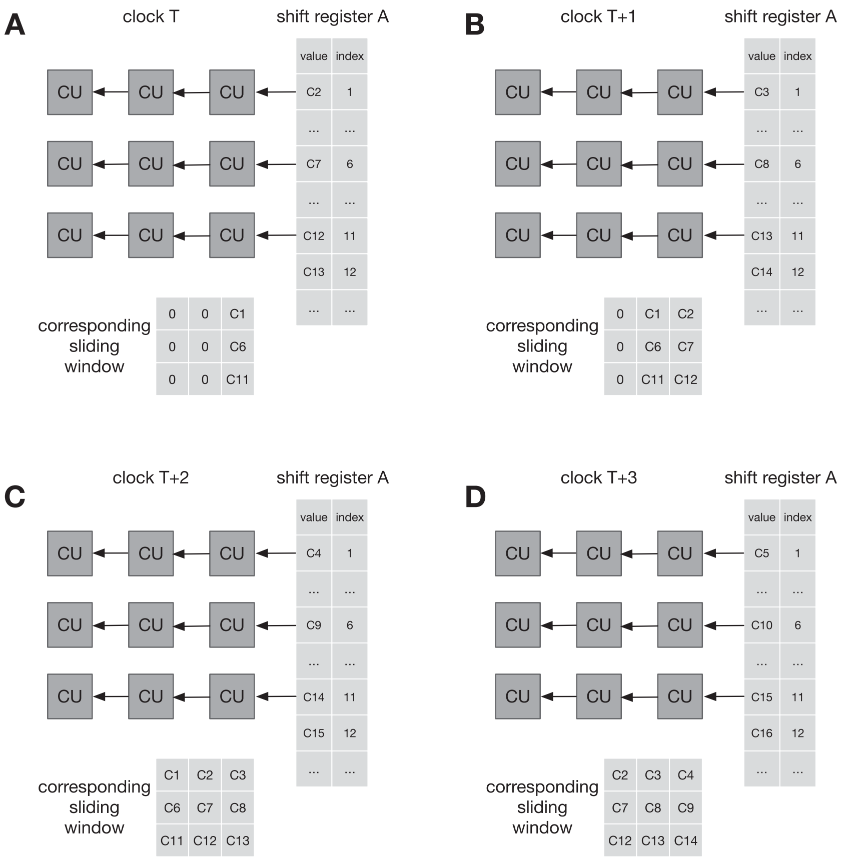

Figure 5: Convolution operations with the shift register. (A) The values in the shift register and CUs in clock T. (B) The values in the shift register and CUs in clock T+1. (C) The values in CUs correspond to the first convolution window in clock T+2. (D) The values in CUs correspond to the second convolution window in clock T+3.

move one position to the left in the same clock cycle. After moving, the values in the shift register A and computing array are as shown in Figure 5B. The results are also invalid, and the reason is the same as above. In clock T+2, the input values of the computing array correspond to the first convolution window of the input feature map. The computing results in these CUs will be summed up as a weighted sum used to update the membrane potential of the corresponding neuron. In the next clock, the input values in the computing array will be updated, corresponding to the second convolution window. When sliding to the last convolution window of every row, the shift register will keep shifting in and out, but the corresponding convolution windows are invalid. The corresponding results should be discarded. In this computing scheme, the filter's weights are stationary, so it can minimize the energy consumption of reading weights.

3.2.2 Coarse-Grained Parallel. If a convolutional layer has N input feature maps and M output feature maps, then all $N \times M$ convolutions can, in theory, be performed in parallel. However, due to the limitations of hardware resources and memory bandwidth, keeping all convolvers busy is impractical, so we apply coarse-grain parallelism, mentioned in Chakradhar, Sankaradas, Jakkula, and Cadambi (2010), to the second convolutional layer. The convolution operation architecure is shown in Figure 6. Because there are only two possible values (1 or 0) for the output value of a neuron, we use only multiplexers and adders to compute the weighted sum of the input. Coarse-grain parallelism, which refers to parallel computing between convolvers, includes intra-output and inter-output parallelism. There are N multiply accumulated convolution windows for each output neuron, and the convolution operation of different input feature maps can be performed in parallel. The spikes from neurons of different input feature maps are used as the SEL signal of the multiplexer. If there is a spike (eg., 1 in SEL pin), the corresponding weight will be accumulated into the membrane potential of the neuron. The modules in a dashed box denote the intra-output parallelism. n represents the number of channels that can be computed in parallel in the N input feature maps. Neurons located in the same position of different output feature maps, which have the same convolution windows on the input feature maps, can also be computed in parallel. Thus, the input of neurons can be reused among neurons of different output feature maps. The dashed boxes indicate the inter-output parallelism. m is the number of output neurons that can be computed in parallel in the M output feature maps. The calculation time of this layer dominates the inference latency of images. To simplify the logic control circuit and maximize the degree of parallelism, we set $n = 64$, and $m = 4$ in this layer. It takes 16 (64/4) loops to complete the computation of all output feature maps.

3.3 Fully Connected and Output Layer Computation. The fully connected (FC) layer can be seen as a special convolutional layer. The size of filters is the same as that of input feature maps and the size of output feature maps is 1×1. Thus, only one convolution operation is needed for every input feature map to compute the weighted sum of neurons of the FC layer. In order to speed up the computation of the FC layer, we apply the parallelism strategy of the second convolutional layer to it where $n = 64$, and $m = 2$. While it takes 64 (128/2) loops to complete computing of the FC layer, the time per loop is short due to the simplicity of the computation.

The output layer has 10 neurons, each representing a digit of 0 to 9. We update membrane potentials of 10 output neurons in parallel and count the

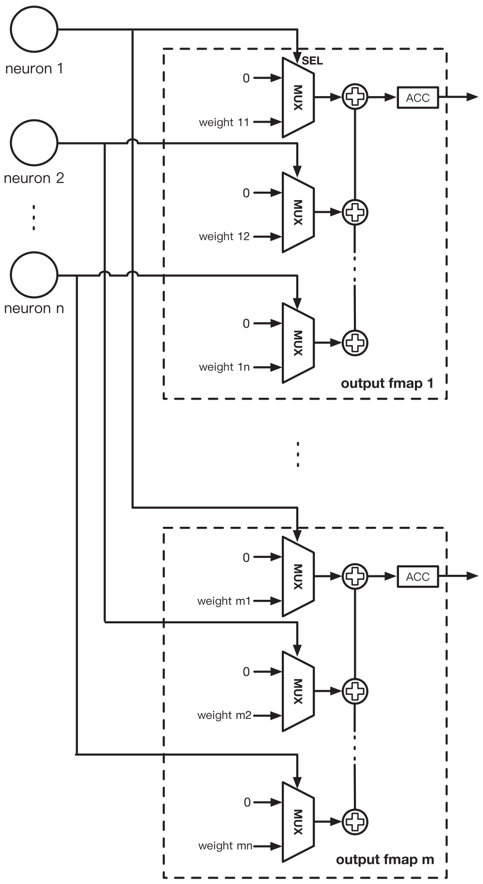

Figure 6: The convolution operation architecture. n represents the number of channels that can be computed in parallel in the N input feature maps. m is the number of output neurons that can be computed in parallel in the M output feature maps. (Adapted from Chakradhar et al., 2010.)

number of spikes fired by every neuron. The classification result is determined by the neuron that has the maximum number of spikes in the entire time window.

3.4 Parallel Computation among Different Time Steps. The entire network performs multiple time step computations to complete the inference, and the computation of different time steps has some independence. For example, there is independence between the second convolutional layer computation at time t and the first convolutional layer computation at time

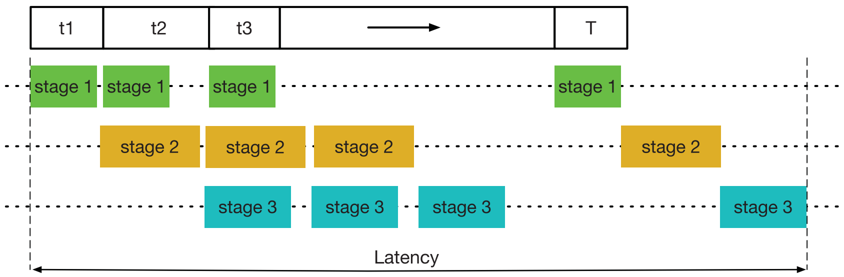

Figure 7: Parallel computation among different time steps. Once stage 1 of the current time step finishes, stage 1 of the next time step can start with stage 2 of the current time step. T denotes the length of time window. Latency is the time to classify an image.

$t + 1$, so they can be executed concurrently. In the hardware architecture, a hardware layer corresponds to each layer in the SNN. These hardware layers could form different computing stages that work concurrently. As shown in Figure 3, the process of inference is divided into three stages: the first convolution and pooling operation form stage 1, the second convolution and pooling operation form stage 2, and the fully connected and output layer computation form stage 3. Once stage 1 of the current time step finishes, stage 1 of the next time step can be started with stage 2 of the current time step, as shown in Figure 7. The mechanism, which allows different stages of adjacent time steps to run simultaneously, can keep these hardware layers busy to reduce the inference latency. Three stages can run together when all data are ready. There is a trade-off between inference speed and hardware resource cost. The system architecture can be divided into more stages to speed up inference, but this also requires more registers to store intermediate results, leading to a more complex control logic.

4 Results

4.1 System Prototype. A complete system verification platform for classification is implemented using the Xilinx Zynq ZCU102 board. The board consists of the Zynq processing system (PS) and programmable logic (PL), which implements the SNN engine, as shown in Figure 8. The Zynq PS contains a quad-core ARM processor and 4 GB of DDR4 RAM. A PYNQ framework is running on the Zynq PS to control the operation of the entire system. The process of classification is as follows. The input data are first loaded to the DDR4 RAM on the PS side. The Zynq PS initializes the SNN engine through the AXI Lite bus, which is responsible for transmitting control instructions. The input data are then fed to the SNN engine through the high-speed AXI DMA bus, which is used for data transmission. Finally,

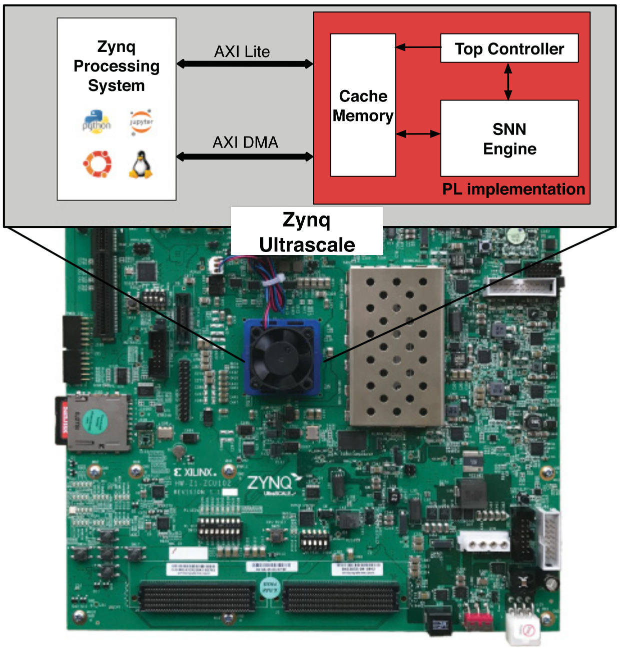

Figure 8: A prototype system for classification. The Zynq processing system is responsible for communicating with the outside world and controlling the operation of the entire system, and PL implementation is used to inference. They communicate via AXI buses. The cache memory is used to store data to be processed and the results of the SNN engine.

the PS side collects the classification results from the SNN engine. This separate structure of control signals and data transmission makes the system more efficient. Furthermore, all the weights are stored using on-chip memory to reduce the latency caused by DRAM accesses.

4.2 System Evaluation. The seven-layer spiking neural network mentioned in section 2.2 has been obtained through network conversion. We first evaluate the performance of network on the MNIST data set, which has 60,000 training images and 10,000 testing images. The performance loss due to conversion and eight-bit quantization is negligible (it is within 1%) compared to the accuracy of trained CNN (99.47%) as shown in Table 1. The network using the fixed uniform encoding method has the highest accuracy of the methods. It is noteworthy that the accuracy results are obtained based on an extremely short time window of only 10 ms (i.e, 10 time steps). As a result, we can get satisfactory classification results with short inference latency. There is a large accuracy gap between the two Poisson methods.

Table 1: Classification Accuracy on the MNIST Test Data Set.

Encoding Method	Accuracy(%)
Poisson-ISI	84.15
Poisson-rand	98.82
Fixed uniform	98.94

Table 2: Resource Utilization on the Xilinx ZCU102.

	8-Bit		16-Bit	
Resource	Utilization	Percent	Utilization	Percent
Look-up table	107,273	39.14	140,537	51.28
Look-up table random access memory	17,457	12.12	19,524	13.56
Flip-flops	67,278	12.27	81,453	14.95
Block RAMs (36 Kb)	264.5	29.00	457	50.11

Poisson-ISI randomly generates interspike intervals that may be too large or even exceed the length of the time window. Thus, the practical firing rate of a neuron is less than the given value, which leads to worse classification results. Two Poisson encoding method results are obtained through 10 experiments due to the random.

In order to explore the impact of different quantization levels on hardware implementation, we implement the 8-bit and 16-bit versions of SNN on the FPGA using synthesizable Verilog to compare resource utilization. As shown in Table 2, a significant reduction in resource utilization is achieved, with the Block RAM used to store weights decreasing the most, which approximates 42%. Resource-constrained or -embedded devices will benefit from neural networks with low-precision weights.

4.3 Performance Comparisons. In this section, we analyze the performance of our design through comparing different computing platforms. We implement the SNN using Matlab on NVIDIA GTX 1080 GPU, where serial computing is converted to parallel computing based on the compute unified device architecture (CUDA) framework. The membrane potential update of neurons in one layer is independent of each other, so each neuron can be assigned a thread to compute the activity. In existing parallel method (Brette & Goodman, 2012), each thread computes the activity of a neuron at one time step, as shown in Figure 9A. Then multiple computations are required to complete the update of the network state over the entire time window. In this work, we adopt the structure-time parallel approach developed in Wu, Wang, Tang, and Yan (2019). Each thread computes all the activities of a neuron over the time window in one computation, as shown in Figure 9B.

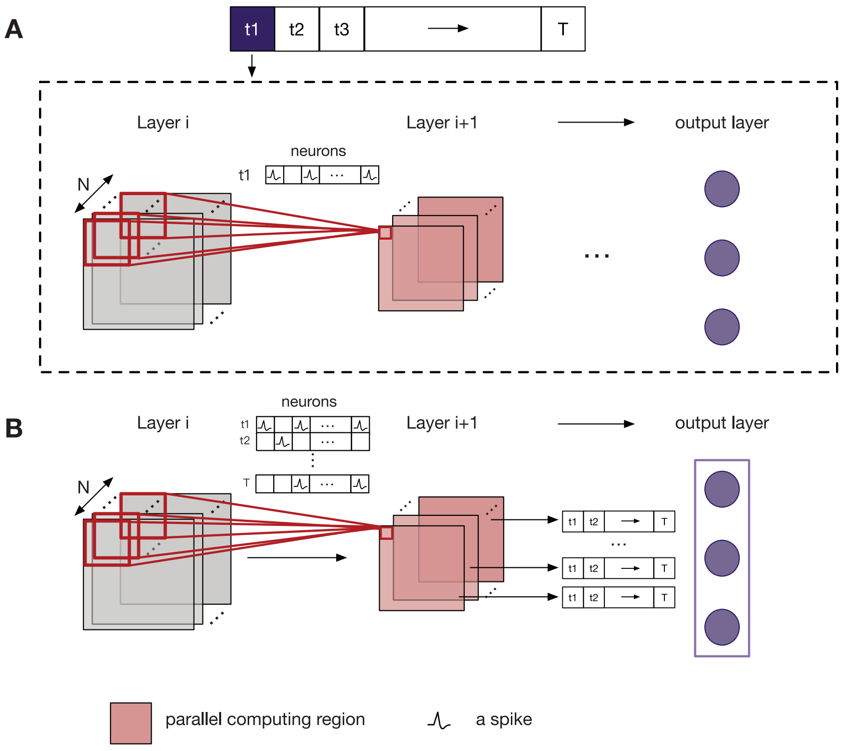

Figure 9: Two implementations of GPU parallel computing. (A) At every time step, the activities of all neurons in the same layer are calculated concurrently, and the computation is down layer by layer. The neuron inputs are the values in the corresponding convolution windows at this time step. The operation in the dashed box continues until the end of the time window. (B) All activities of the neurons in one layer are computed over the entire time window (from $t1$ to T). The previous layer inputs are spikes fired in the entire whole time window. The operation continues until the output layer. (Adapted from Wu et al., 2019.)

Thus, the inference latency can be reduced by half, which benefits from the reduction of context switches. The processing time per image is shown in Table 3.

We first compare our design with implementations on traditional computing platforms. The results are in Table 3. In our design, by using the shift register and the coarsely grained parallel strategy, the time of processing one image is 6.11 ms under 150 MHz clock frequency. Thus, the corresponding FPS is 164, which is 41 times higher than the implementation on Intel i7-6700K CPU (4.0 GHz). Although the throughput achieved by the well-optimized GPU implementation is similar to that of the FPGA

Table 3: Comparison of Hardware Platforms on the MNIST 10,000 Test Images.

Platform	Intel i7-6700K 4.0 GHz	NVIDIA GTX 1080	Xilinx ZCU102
Processing time/image (ms)	252	6.19	6.11
FPS	4	162	164
Speed-up	1×	40×	41×
Power (W)	54	100	4.6
Energy/image (J)	13.61	0.62	0.03

Table 4: Comparison with Previous FPGA and ASIC Implementations.

	Minitaur (Neil & Liu, 2014)	Darwin (Ma et al., 2017)	This Work
Clock (MHz)	75	25	150
Platform	Spartan-6 LX150	ASIC	Zynq ZCU102
Method	DBN	DBN	CNN
Weight precision	16-bit fixed	16-bit fixed	8-bit fixed
Weight sum	647 K	647 K	506 K
Power	1.5 W	21 mW	4.6 W
Processing time/image (ms)	152	160	6.11
Energy/image (J)	0.228	0.003	0.03
Classification accuracy (%)	92	93.8	98.94

implementation, it consumes 22 times more power. The FPS of our design is 24.9 times higher than the Minitaur (Neil & Liu, 2014), 26.2× higher than Darwin (Ma et al., 2017) neuromorphic coprocessor as shown in Table 4. Our design also has highest classification accuracy on the MNIST test data set. Compared to fully connected spiking deep belief networks (DBNs) implemented on the Minitaur and Darwin, our network has the least number of weights due to sparse connections and shared weights of CNN. Schmitt et al. (2017) converted a deep feedforward neural network to a spiking network on the BrainScaleS wafer-scale neuromorphic system and compensated conversion loss by in-the-loop training. They achieved 95% on a modified subset of the MNIST data set that has only five digit classes. Besides, our conversion loss (0.53%) is lower than theirs (2%). The total on-chip power evaluated using the Xilinx Power Estimator (XPE) tool is 4.6 W, of which the Zynq processing system accounts for 2.954 W. The power of our system is higher than the Minitaur yet consumes 7.6 times lower energy to process one image. The Zynq processing system power is high because it contains lots of components, such as the quad-core ARM Cortex-A53 processor and the Mali-400 MP processing unit. The energy consumption of inference in PL implementation is relatively low. Thus, the SNN engine can be integrated with the other low-power RISC CPU to further reduce total energy consumption.

5 Discussion

In this work, we obtain a high-performance SNN through network conversion and accelerate its computing using FPGA. The network achieves the accuracy of 98.94% on the MNIST data set. In network conversion, input values of typical deep ANNs need to be encoded into spike patterns to fit the spiking neuron dynamics. The basic idea of conversion is to approximate the activation value of neurons in ANN by the firing rate of the same neurons in SNN, so previous work (Cao et al., 2015; Diehl et al., 2015) generally uses a rate-coding scheme (i.e., Poisson firing rate), while Rueckauer et al. (2017) argued that this method introduced variability into the firing rate of the network and impaired its performance. Rueckauer et al. (2017) directly used analog values as input and found this to be particularly effective in the low-activation regime of ANN units. In our work, we adopt the fixed uniform encoding method to eliminate randomness and achieve the highest performance among three encoding methods. We analyze the difference in accuracy based on the reconstruction results of input spike trains. We found that reconstructed results that have a higher similarity with the original MNIST image have better classification.

The SNN model and hardware are codesigned to jointly maximize accuracy and throughput while minimizing energy and cost. The SNN accelerator in this work can achieve a performance of 164 FPS under 150 MHz clock frequency. Max-pooling is a common choice in most successful CNNs, which spatially downsample feature maps. It is challenging to implement it in hardware because the firing rates of neurons need to be estimated at every time step. Thus, we propose a hardware-friendly max-pooling method to evaluate the firing rates of neurons in the previous layer. Compared with the gating function-based mechanism for spiking max-pooling (Rueckauer et al., 2017), division is not included in our formula of the firing rate, which leads to lower circuit complexity. In addition, we found that the network with max-pooling layers spends less time integrating spike activities to reach the best performance compared with average pooling layers. In our experiment, the network with average pooling achieves only 93.63% on the MNIST data set when the length of the time window is 10 ms.

In the hardware architecture, a hardware layer corresponds to each layer in the SNN. The different computing stages of these hardware layers can work concurrently. Thus, the different time steps can be computed in parallel to reduce the latency. However, as the size of network expands, this direct mapping approach will consume a deal of the available hardware resources or even exceed them. One solution is to quantize weights or compact network to reduce its size. Esser et al. (2016) presented an approach called Eedn, which creates CNNs suitable for neuromorphic hardware, and then mapped deep convolutional networks to TrueNorth by restricting network precision to trinary weights $\{-1, 0, 1\}$. Amir et al. (2017) applied the Eedn algorithm to implement an event-based gesture recognition system, which combined dynamic vision sensor (DVS; Lichtsteiner, Posch, & Delbruck, 2008) to implement end-to-end computation. In future work, we will focus on BinaryConnect (Courbariaux, Bengio, & David, 2015), binarize the weights (-1 and 1), and then investigate the new conversion method.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under grant 61673283 and also supported by the Fundamental Research Funds for the central universities.

References

Amir, A., Taba, B., Berg, D., Melano, T., McKinstry, J., Di Nolfo, C., . . . Kusnitz, J. (2017). A low power, fully event-based gesture recognition system. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 7243–7252). Piscataway, NJ: IEEE.

Brette, R., & Goodman, D. F. (2012). Simulating spiking neural networks on GPU. *Network: Computation in Neural Systems*, *23*(4), 167–182.

Cao, Y., Chen, Y., & Khosla, D. (2015). Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, *113*(1), 54–66.

Chakradhar, S., Sankaradas, M., Jakkula, V., & Cadambi, S. (2010). A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, *38*(3), 247–257.

Courbariaux, M., Bengio, Y., & David, J.-P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems* (pp. 3123–3131). Red Hook, NY: Curran.

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., . . . Liao, Y. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, *38*(1), 82–99.

Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C., & Pfeiffer, M. (2015). Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *Proceedings of the 2015 International Joint Conference on Neural Networks* (pp. 1–8). Piscataway, NJ: IEEE.

Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., Andreopoulos, A., . . . di Nolfoa, C. (2016). Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, *113*(41), 11441–11446.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA: MIT Press.

Heeger, D. (2000). *Poisson model of spike generation*. Handout at Standford. https://www.cns.nyu.edu/~david/handouts/poisson.pdf

Hu, Y. (2016). *Max-pooling operations in deep spiking neural networks*. Neural Systems and Computation project report. https://dgyblog.com/projects-term/res/Max-Pooling%20Operations%20in%Deep%20Spiking%20Neural%20Networks.pdf

Ji, Y., Zhang, Y., Li, S., Chi, P., Jiang, C., Qu, P., . . . Chen, W. (2016). Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (p. 21). Piscataway, NJ: IEEE Press.

Lee, J. H., Delbruck, T., & Pfeiffer, M. (2016). Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, *10*, 508.

Lichtsteiner, P., Posch, C., & Delbruck, T. (2008). A 128 × 128 120 dB 15 mu s latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, *43*(2), 566–576.

Lillicrap, T. P., Cownden, D., Tweed, D. B., & Akerman, C. J. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*, *7*, 13276.

Ma, D., Shen, J., Gu, Z., Zhang, M., Zhu, X., Xu, X., . . . Pan, G. (2017). Darwin: A neuromorphic hardware co-processor based on spiking neural networks. *Journal of Systems Architecture*, *77*, 43–51.

Maass, W. (1997). Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, *10*(9), 1659–1671.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., . . . Brezee, B. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673.

Neil, D., & Liu, S.-C. (2014). Minitaur, an event-driven FPGA-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *22*(12), 2621–2628.

Qiao, N., Mostafa, H., Corradi, F., Osswald, M., Stefanini, F., Sumislawska, D., & Indiveri, G. (2015). A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in Neuroscience*, *9*, 141.

Querlioz, D., Bichler, O., & Gamrat, C. (2011). Simulation of a memristor-based spiking neural network immune to device variations. In *Proceedings of the 2011 International Joint Conference on Neural Networks* (pp. 1775–1781). Piscataway, NJ: IEEE.

Rueckauer, B., Lungu, I.-A., Hu, Y., Pfeiffer, M., & Liu, S.-C. (2017). Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, *11*, 682.

Samadi, A., Lillicrap, T. P., & Tweed, D. B. (2017). Deep learning with dynamic spiking neurons and fixed feedback weights. *Neural Computation*, *29*(3), 578–602.

Schmitt, S., Klähn, J., Bellec, G., Grübl, A., Guettler, M., Hartel, A., . . . Karasenko, V. (2017). Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system. In *Proceedings of the 2017 International Joint Conference on Neural Networks* (pp. 2227–2234). Piscataway, NJ: IEEE.

Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295–2329.

Wu, X., Wang, Y., Tang, H., & Yan, R. (2019). A structure–time parallel implementation of spike-based deep learning. *Neural Networks*, *113*, 72–78.

Wu, Y., Deng, L., Li, G., Zhu, J., & Shi, L. (2018). Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience*, *12*.

Xu, Y., Tang, H., Xing, J., & Li, H. (2017). Spike trains encoding and threshold rescaling method for deep spiking neural networks. In *Proceedings of the 2017 IEEE Symposium Series on Computational Intelligence* (pp. 1–6). Piscataway, NJ: IEEE.

Zhao, B., Chen, S., & Tang, H. (2014). Bio-inspired categorization using event-driven feature extraction and spike-based learning. In *Proceedings of the 2014 International Joint Conference on Neural Networks* (pp. 3845–3852). Piscataway, NJ: IEEE.

Zhao, B., Ding, R., Chen, S., Linares-Barranco, B., & Tang, H. (2015). Feedforward categorization on AER motion events using cortex-like features in a spiking neural network. *IEEE Transactions on Neural Networks and Learning Systems*, *26*(9), 1963–1978.

Received June 25, 2019; accepted August 12, 2019.