Article

A Low-Power Spike-Like Neural Network Design

Michael Losh and Daniel Llamocca *

Electrical and Computer Engineering Department, Oakland University, Rochester, MI 48309, USA; mlosh@oakland.edu
* Correspondence: llamocca@oakland.edu

Received: 19 October 2019; Accepted: 28 November 2019; Published: 4 December 2019

Abstract: Modern massively-parallel Graphics Processing Units (GPUs) and Machine Learning (ML) frameworks enable neural network implementations of unprecedented performance and sophistication. However, state-of-the-art GPU hardware platforms are extremely power-hungry, while microprocessors cannot achieve the performance requirements. Biologically-inspired Spiking Neural Networks (SNN) have inherent characteristics that lead to lower power consumption. We thus present a bit-serial SNN-like hardware architecture. By using counters, comparators, and an indexing scheme, the design effectively implements the sum-of-products inherent in neurons. In addition, we experimented with various strength-reduction methods to lower neural network resource usage. The proposed Spiking Hybrid Network (SHiNe), validated on an FPGA, has been found to achieve reasonable performance with a low resource utilization, with some trade-off with respect to hardware throughput and signal representation.

Keywords: spiking neural networks; bit-serial architectures; FPGA

1. Introduction

Deep Learning Convolutional Neural Networks (CNNs) can achieve unprecedented accuracy on state-of-the-art artificial intelligence applications [1]. Most of the active research is focused on the convolutional layers as they account for a large portion of the computational cost [2]. This includes accelerators for deep learning CNNs on a custom chip [1], a methodology to extract optimal hardware realizations from a large design space as well as optimized buffer management and memory bandwidth to optimize hardware resource usage [2], on-the-fly reconfiguration using FPGAs (field programmable gate arrays) [3], and reducing the complexity of the sum-of-product operations [4]. However, hardware implementations of fully connected networks do not receive the same attention.

A combination of advanced neural network architectures, improved training approaches, and widespread availability of modern open-source neural network and machine learning (ML) solution frameworks now makes it feasible to design, train, and deploy artificial neural networks of unprecedented performance and sophistication. As neural network models improve and their accuracy increases, so do network size and computational complexity.

However, the current industry-favored GPU (graphic processing unit) hardware platforms are extremely power-hungry. For example, the state-of-the-art nVIDIA Tesla V100 card consumes 250 W and it is usually teamed with a high-end CPU to support massive data volume flows and handle other application logic [5]. This level of power consumption and associated heat output are major impediments to a wider adoption of neural-network technology, especially for mobile applications such as autonomous aerial drones and ground vehicles, robotics, and wearable technology. Some GPU chip-makers have begun offering more power-efficient versions (e.g.,: the nVIDIA's new Xavier chip family can enable an embedded module to deliver 30 Tera operations per second at 30 W). While this power reduction is welcome, there is a strong interest in alternative implementation methods that

drastically reduce the power consumption and would enable neural networks to hit size, weight, and cost targets while meeting the requirements of real-time applications.

Performance and energy requirements are hardly met by general purpose processors either, effectively shifting the research towards hardware accelerators. A comprehensive overview of hardware implementations (analog, digital, hybrid) for various ANN models is presented in [6]. FPGA-based accelerators have attracted a great deal of attention as they exhibit high performance, energy efficiency, and reconfiguration capability [7,8]. FPGA-based implementation of NNs is an active area of research and there are a large variety of design techniques and hardware optimizations.

The work in [9] describes a parameterized architecture on an FPGA for feed-forward multilayer perceptron networks and for the back-propagation learning algorithm. In the FPGA implementation in [10], only one layer implements all the layers in the network via a time-multiplexing scheme; each neuron is designed via an LUT-based approach to reduce hardware usage. The work in [7] describes a hardware co-processor for a feedforward neural network on a PSoC (Programmable System-on-Chip). A software application is in charge of configuring and enable the hardware. The work in [8] considers whether FPGAs can outperform GPUs as deep neural networks grow in complexity.

A neural network model that offers power consumption benefits is the spiking neural network (SNN) that take the bio-mimicry of natural neural signals a step further, representing neuron activation levels using a series of pulses that mimics the action potential voltage spike train found in biological neurons [11]. This signaling scheme let us simplify connections between neural units by using a single line instead of a multi-bit (8, 16, 32) bus. The extensive survey in [11] lists variations of SNN training and implementation, and their strengths and weaknesses. The work in [12] details an FPGA implementation for a SNN, where the neuron design includes the membrane potential calculation, a weight memory, and a multiplexer. A SNN with Izhikevich spiking neurons in a pipelined fashion is presented in [13]; a large-scale network was implemented on a Cray XD1 (that contains an AMD processor and an array of various FPGAs). The work in [14] presents a fixed-point architecture for a SNN that is validated on an FPGA. An alternative neuron model (VO_2) is presented in [15], where a VO_2 neuron is implemented as an electrical circuit. A SNN (9 inputs and 3 outputs) was built and testing consisted on detecting input patterns in a 3x3 pixel grayscale image.

Based on earlier work, we utilize aspects of SNNs to design an architecture that further optimizes hardware usage, while at the same time allowing training of our SNN-like design with conventional ANN methods. We summarize the main contributions of this manuscript as follows:

- A fully customizable hardware design for a SNN-like neural network. A software application automatically generates the hardware code for the integration of all neurons in the network. In addition, comparisons with a standard multiply-and-add NN version are provided;
- Use of duty-cycle encoding for the signals between neurons. This allows for efficient routing and for hardware usage minimization;
- Implementation of thrifting, i.e., limiting the number of inputs for a neuron. This was first tested in software and then validated on the hardware architecture;
- Pipelined neural network: Each layer is always processing data. Output results are generated as one set per frame, where a frame is defined as the processing time of a layer. The frame time only depends on the hardware parameters, and not on the data;
- A bit-serial neuron design that only requires a counter, 1-bit multiplexors, and comparators.

The rest of the manuscript is organized as follows. Section 2 presents some definitions and positions our work in context. The proposed approach is detailed in Section 3. Section 4 presents the results. Section 5 provide conclusions.

2. Background

Here, we introduce the basic definitions associated with Artificial Neural Networks and Spiking Neural Networks. Then, we describe the contribution of this work.

2.1. Artificial Neural Network Fundamentals

Artificial neural networks (ANNs) take inspiration from nature. Biological neurons, as depicted in Figure 1a, respond to positive and negative stimuli and generate a stimulus to other neurons. The stimulus from one neuron to another crosses a small gap called a *synapse*. The neuron's activity level is an electrical potential across the neuron cell membrane. The signal is collected along the dendrite structures, and it is integrated as a *membrane potential*, until the neuron has a high-enough potential to fire an outgoing pulse. This pulse, called *action potential*, travels along the axon structure to one or more axon terminals at synapses with further neurons [16].

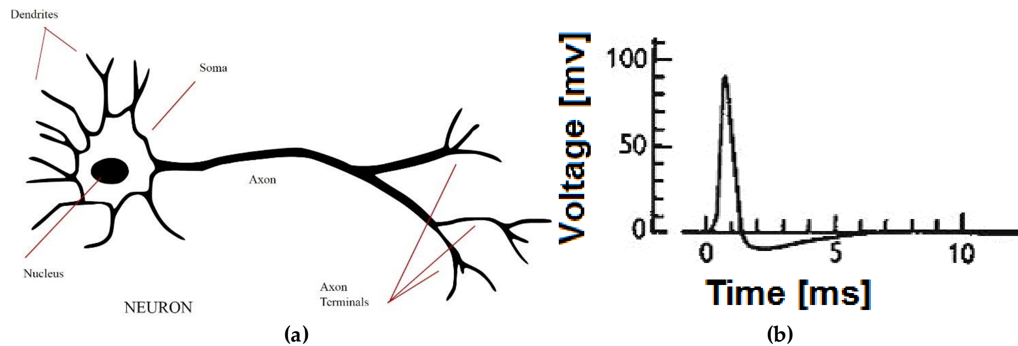

Figure 1. (**a**) Biological neuron. (**b**) Action potential voltage spike. Source: WikiMedia Commons. These files are licensed under the Creative Common Attributions-Share Alike 4.0 International license.

Figure 1b depicts the potential voltage level observed during a firing event. It has a spike-like shape, so action potential pulses are often called spikes. Once the action potential pulse travels down the axon, the potential drops below the nominal baseline level, and there is a time (refractory period), during which the neuron cannot fire another significant pulse. Thus, each neuron has an upper frequency limit on the overall spike rate that can be considered its fully-saturated activity level.

Once the action potential reaches an axon terminal, the signal propagates onto the next neurons. This mechanism can be inhibitory (suppressing the neuron output), or excitatory (increasing the activity level of the downstream neuron). In addition, within a group of synapses, a synapse may be larger/smaller, more/less sensitive, or repeated multiple/fewer times (or not at all) than the other. This can increase or decrease the effect of a neuron's stimulus impulse on the following neuron. Thus, there is a variable gain or *weight* for each connection from one neuron to another, that can be modeled with gain factor (positive, negative, null) relative to a baseline level. A model of the biological neuron, called an artificial neuron is shown with its connections in Figure 2.

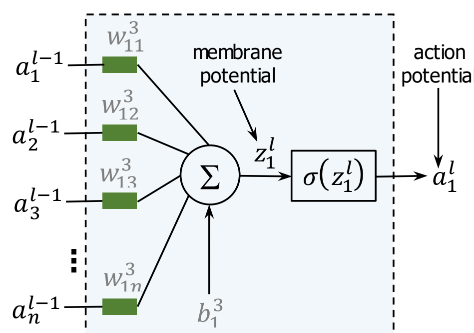

Figure 2. Artificial neuron model. The membrane potential is a sum of products (input activations by weights) to which a bias term is added. The neuron shown here belongs to a layer with index l. The input activations come from a previous layer $(l-1)$. The neuron here is the first neuron (index '1') in layer l.

ANNs are organized into layers, where all the neurons in one layer receive their inputs from only neurons in a prior (upstream) layer, and only provide outputs to the neurons in the subsequent (downstream) layer. Figure 3a shows a neural network layer with 3 layers, and Figure 3b shows the notation for the weights, bias, inputs, and output for a neuron. Note that we can use a vector to denote the set of input activations to a neuron from the connected neurons in the prior layer. Same for the set of weight factors for each of these connections. Here, we denote L as the number of layers, with l being the layer index, $l = 1, 2, \ldots L$. For a neuron j in layer l, the membrane potential is denoted by z_j^l, the bias by b_j^l, and the action potential by a_j^l. Most ANN implementations model z_j^l as the sum of each input connection's signal level a_k^{l-1} weighted by the synaptic gain factor w_{jk}^l from neuron k in an upstream layer (prior layer $l-1$) to neuron unit j in the current layer l [17].

$$z_j^l = \sum_k \left(w_{jk}^l a_k^{l-1} \right) + b_j^l, \quad l > 1 \tag{1}$$

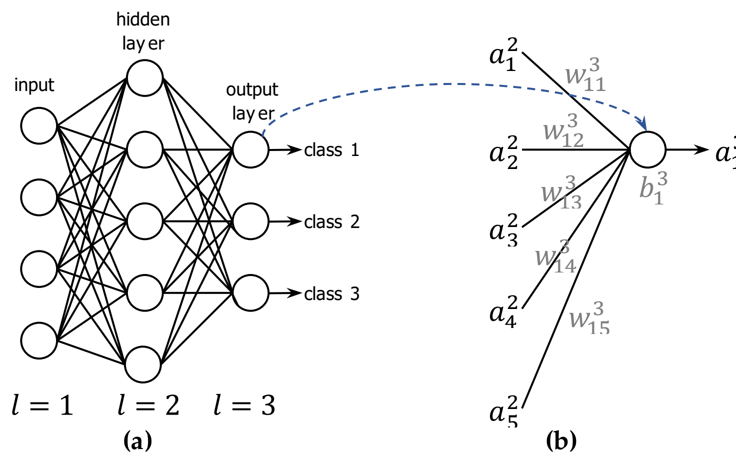

Figure 3. (**a**) Neural network example with 3 layers. The input layer represents the input values to the network. (**b**) A neuron in the third layer. We use a generic notation for the weights, biases, activation inputs, and activation outputs (action potentials) of each neuron in a network.

It is common to represent the computation for all membrane potentials in layer l in matrix form as per Equation (2). Here, z^l is a column vector that includes all the membrane potentials of layer l, a^{l-1} is a column vector that includes all the inputs signals to the layer l, w^l is the weight matrix containing all the synaptic gain factors from layer $l-1$ to layer l, and b^l is a column vector that includes the biases of all neurons of layer l. Figure 4 depicts the matrix operation.

$$z^l = w^l a^{l-1} + b^l, \quad l > 1 \tag{2}$$

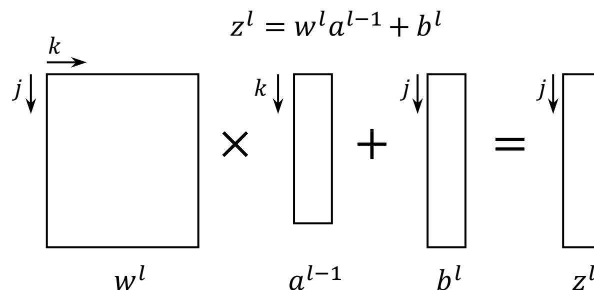

Figure 4. Matrix operation for the computation of all membrane potentials in layer l. The index k is used for the neurons of the previous layer ($l-1$), while the index j is used for the neurons in the current layer l.

The action potential intensity (activation function) of a neuron j in a layer l, denoted by a_j^l, is modeled as a scalar function of the membrane potential z_j^l. We can also denote all action potentials of layer l as a^l, where $a^l = \sigma\left(z^l\right)$. Note that for $l = 1$, a^l represents the outputs of the first layer, which are defined as the inputs to the neural network.

$$a_j^l = \sigma\left(z_j^l\right) = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \quad a^l = \sigma\left(w^l a^{l-1} + b^l\right) = \sigma\left(z^l\right), \ l > 1 \tag{3}$$

Early research into so-called Perceptron neural networks determined that purely linear weighting combiners as in Equation (1) do not have some important processing abilities [18], and that some non-linear response is needed. Natural neurons mostly exhibit not much response to low input signals, a rapidly increasing response in some sensitive portion of the overall range, and then a modest or no further response level for additional input levels (i.e., saturation). Various functions have been proposed based on their computational cost and system effectiveness. Specific examples include biased, thresholded step-function (Equation (4)), Rectified Linear Unit (ReLU) (Equation (5)), and true sigmoids based on the hyperbolic tangent, arc-tangent, or logistic functions (Equation (6)).

$$\sigma_{step}(z_j^l) = \begin{cases} 1, & if\ z_j^l - \theta > 0 \\ 0, & otherwise \end{cases} \tag{4}$$

$$\sigma_{ReLU}(z_j^l) = \max(0, z_j^l) \tag{5}$$

$$\sigma_{tanh}(z_j^l) = \tan\mathrm{h}(z_j^l), \ \sigma_{arctan}(z_j^l) = c.arctan(k, z_j^l), \ \sigma_{logistic}(z_j^l) = \frac{1}{1 + e^{-kz_j^l}} \tag{6}$$

While there are mathematical arguments for smooth sigmoid functions such as full differentiability; in practice, linear rectification (ReLU) tends to work nearly as well and it is much simpler to implement in software and hardware [19], whereas the thresholded step function does not provide much expressiveness, leading to information loss that is unsuitable for many applications.

The action potential between neurons can be encoded with numeric formats such as 16-bit, 32-bit, 64-bit floating-point or variable-size fixed-point (down to 8 bits or less). Likewise, the weights for each synapse-like connection between neurons can have sizes from 64 to just a few bits. With fewer bits to represent the numbers the range and precision of the encoded value is reduced, resulting in loss of accuracy. However, such methods reduce hardware resource usage, thereby increasing performance along dimensions not directly related to accuracy, such as power, end-to-end latency, throughput bandwidth, heat dissipation, and system cost at a given level of performance.

2.2. Spiking Neural Networks

These are ANN designs that encode the neuron activation as a series of pulses and implement an *integrate and fire* neuron action model where the neuron unit receives spike inputs over a period of time and fires once a certain threshold is achieved [11]. For such Spiking Neural Networks (SNNs), the most direct realization of a biological neural signal would be a series of short-duration binary pulses, whose frequency is proportional to the overall excitation level of that neuron. Since natural potential voltage spikes of a given neuron's action potential signal tend to have the same size and shape, using a simple, short 1-bit digital pulse is a reasonable simplification. Encoding of higher-level information onto clusters/trains of spike events in biological neurons is not well-understood, but some researchers suspect that it is more than straightforward frequency encoding. Conceivably, pulse clusters of various durations, spacing, and intra-cluster spike density, or finer-grained spike timing patterns may encode multi-dimensional information in a given neuron's axonal output. However, useful applications are possible with simple spike patterns, as detailed in this work.

Often, the SNN model includes a leakage decay term so that the activation is suppressed when there is a low rate of input activation. The bias term is a first-order approximation of the leakage effect, at least when the SNN network activity is structured into fixed, synchronized time-frames. In an asynchronous SNN, where each unit fires independently at the time it has met its threshold, the loss mechanism can be modeled as a fractional loss rate for the current membrane potential z_j^l.

SNNs have some features that reduce hardware complexity, but at the same time they present some implementation challenges. The features that can be optimized are listed as follows:

- The output of a given unit can be represented by a single wire rather than by an 8-bit or 16-bit bus. This simplifies routing in the FPGA and reduces the size of registers and other elements;
- Spikes (as pulses) can be easily counted (up/down counter), making the integration a straightforward process rather than utilizing multi-bit accumulators;
- Weights (gain factors) for input connections can be realized by repeated sampling of the input pulse as per the gain level or by 'step counting' after receiving each input pulse;
- In the analog domain, specially-designed transistors may be used to better implement the *integrate and fire* behavior with very few active components by using a special charge-accumulating semiconductor layer. Also, analog signals are more robust to modest amounts of signal pulse collision (two signals hit the bus at the same time), within the limits of saturation.
- However, there exist some challenges to applying SNNs in applications, such as:
- Many existing processors, input, and output devices are designed to work with multi-bit digitized signals. Spike-oriented encoding methods are uncommon among current electronics components and system designs, although there are some examples of duty-cycle encoding (variable pulse width for a set pulse frequency) or pulse-rate encoding. Pure analog-domain transducers exist, and components such as voltage-controlled oscillators (VCOs) may be a useful front-end interface. If SNNs become more prominent, interfacing mechanisms will improve;
- If spikes are represented by 1-bit pulses, if two of more outputs feeding a single input are firing at the same time, it will be received as a single spike pulse, and any multiplicity of spikes will be lost. For a large and robust enough network, the signal loss due to spike collision may not impact the accuracy significantly. This collision effect can be minimized if the pulses are kept short and each unit producing an output delays the timing of the output by a variable amount within a longer time window. This delay could be random, pre-configured to avoid conflict with other units, or externally-triggered and coordinated to minimize conflict. Alternatively, the multiple output spikes can be held over a longer period so the receiving unit may sample each one using a series of sub-spike sampling period. These techniques, however, introduce latency;
- Some types of neural network layers, such as the softmax layer that emphasizes one output among a set of outputs (the best classification out of multiple possibilities) and normalizes them in a likelihood-based ration, are difficult to implement in a SNN. It is possible to make do without such layers, but interpretation of SNN output becomes somewhat less intelligible;
- Perhaps the most fundamental challenge is that training of SNNs is not as straightforward as the conventional ANNs. Back-propagation of network error compared to the expected results is a gradient descent operation where the local slope of an error surface is expected to be smooth due to differentiability of the neuron internal operation. Depending on the SNN design, there may be internal variables that provide a smooth error surface for gradient descent. Alternatively, we can train an ANN proxy of the intended SNN with gradient descent and map the gain and bias values to the SNN representation with reasonable overall performance. We also note that there are training approaches unique to SNNs such as Spike-Timing-Dependent Plasticity (STDP) that reinforces connections in the network where the input spike is temporally coincident or proximal to a rewarded output spike, while weakening others [15].

2.3. Our Implementation

Based on the advantages and drawbacks of SNNs, we propose an SNN-like design that features: (i) Fixed-frequency duty-cycle encoding of excitation level inputs rather than frequency encoding; (ii) sub-pulse sampling over all connections via a control circuit; (iii) use of back-propagation of a proxy ANN with corresponding encoding precision for weights, biases, and excitation levels.

Our work seeks to address the design of a neural network (SNN-like) that optimizes resource usage and that it is trainable with conventional back-propagation techniques. There exist fully analog neural network implementations that offer high efficiency in power and performance. There are also custom chip digital designs for a neural network. We preferred an FPGA implementation as its flexibility lets us explore various hardware designs. We can model and train a neural network using a software application, then map it into a hardware design in an FPGA that could be successfully synthesized, simulated, and validated. In addition, while some SNNs use an alternative to classic gradient-descent back-propagation such as spike-timing-dependent plasticity (STDP), we prefer to make use of back-propagation techniques for which a large body of work already exists.

Finally, for our target application, we selected the problem of classification of images of hand-written numeric digits in the MNIST data set [20]. This is a well-known and popular neural network test application because the data set is easy to access and use; input test images are of modest size; the possible digit classes 0 to 9 are well-bounded to a small, definite set, allowing correspondingly modest-sized neural networks to be designed, trained, implemented and evaluated in hardware. However, the digit-classification problem offers a set of non-trivial challenges, including a wide range of digit styles, various levels of neatness/clarity, different sizes, different orientations, etc., and requires learning generalization in order to perform robustly [21].

3. Methodology

The hardware design, called Spiking Hybrid Network (SHiNe), includes a combination of SNN and ANN aspects, allowing back-propagation to be used for training. We will use the neural network model depicted in Figure 5a. A SHiNe neuron design is depicted in Figure 5b along with its connections and control signals. Details of this design are explained next.

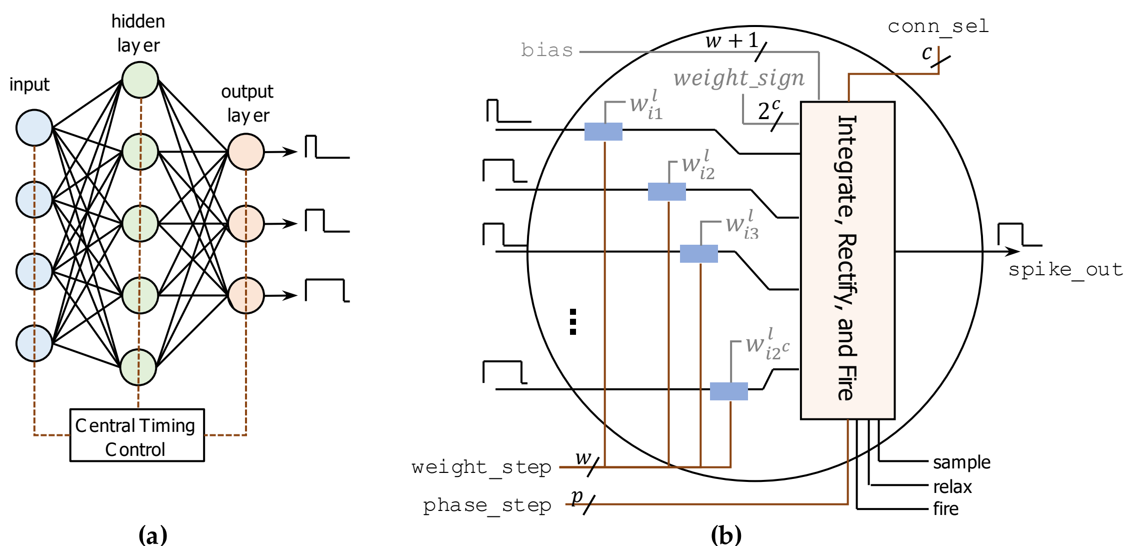

Figure 5. SHiNe design. (**a**) Neural network example. (**b**) SHiNe neuron with 2^c inputs and w bits for weight magnitude. Inputs: Duty-cycle encoded. Central Timing Control Circuit: It controls all the neurons.

3.1. Design Considerations

The proposed SHiNe design works under the following considerations.

3.1.1. Single-Wire Connection between Neurons in Adjacent Layers

This feature significantly simplifies routing in a digital hardware implementation. We use only one wire per connection between one neuron and the other, as opposed to most designs that use multi-bit signals. The maximum number of input connections per neuron (fan-in) is given by 2^c, with c being a small number (e.g., 5). As a result, we might not have a fully connected layer if the prior layer has more than 2^c units. Not having full connections can impact network accuracy, but this limit allows each unit to sample each connected unit sequentially in a reasonable time, completely avoiding the spike collision effect that otherwise requires more hardware resources to address.

3.1.2. Duty-Cycle Encoding for the Neuron Inputs

In duty-cycle encoding, the signal line is high for a phases, and low for $2^p - a$ phases. The frequency of this signal (or spikes) is constant, and the excitation level is equal to a. We use 2^p phases (or steps) for duty-cycle encoding, with p being a small number (e.g.,: 5, which yields 32 possible excitation levels). Note that we restrict the excitation levels (input signals values) to be unsigned values in the range $a \in \left[0, 2^{p-1}\right]$. These levels might have a fixed-point equivalence.

Each duty-cycle phase is defined to last 2^{w+c} cycles of a fast clock (called *clock*), where w is the bit-width of the weights' magnitude. A duty-cycle encoded signal then lasts a total of 2^{w+c+p} clock cycles. Figure 6 depicts examples of three duty-cycle encoded signals ($input_0$, $input_1$, $input_2$) where $p = 4$, $w = 2$, $c = 5$. Here, the duty-cycle encoded signal lasts 2^{11} cycles. For $input_0$, the excitation level is $a = 2$. For $input_1$, $a = 1$; and for $input_2$, $a = 14$.

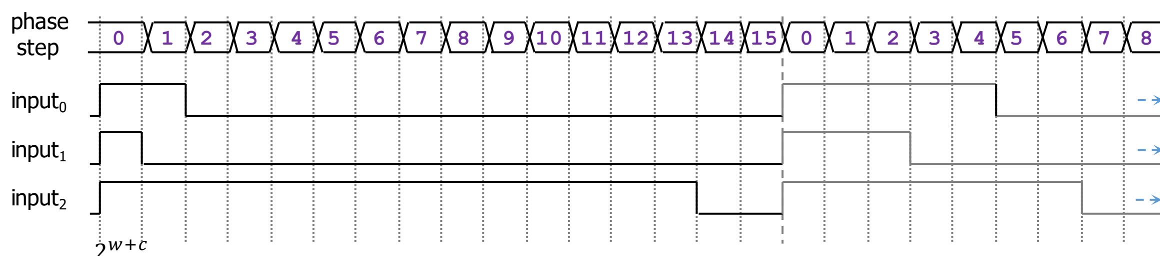

Figure 6. Examples for three duty-cycle encoded signals for $p = 4$, $w = 2$, $c = 5$. Each phase step of the duty-cycle encoded signals lasts $2^{w+c} = 2^7 = 128$ cycles of clock (the fast clock).

3.1.3. Weights (Connection Gain)

They are defined as the number of times to sample (with the *clock* signal) the duty-cycle-encoded input signal; this increments the SHiNe unit's membrane potential z_j^l each time. Weights can be negative, and we use the sign-and-magnitude representation with $w + 1$ bit (sign bit encoded separately) bits. The w-bit magnitude can take on 2^w different values, w is a small number (e.g.,: 3).

We use the fixed-point representation for the weights. We let $[n\ p]$ denote a fixed-point format with n bits and p fractional bits. Then, we use the unsigned fixed-point format $[w\ w]$ for the magnitude, resulting in weights being in the range $[-(1-2^{-w}), 1-2^{-w}]$ with a step size of 2^{-w}. For example, for $w = 3$, the weight range is $[-0.875, 0.875]$ with a step size of 0.125. This means that a signal going through only one neuron will lose strength due to the under-unity maximum gain. However, neural networks include numerous units that tend to combine their effect, and some units might become fully saturated. The weight values are fixed constants in the hardware implementation, and they are based on the results obtained during supervised learning.

3.1.4. Bias

The bias term is represented by a signed (2's complement) value in the fixed-point format $[w+1\; w-1]$. This results in the range $\left[-2, 2-2^{-w+1}\right]$. This slightly wider range was found in experiments to work better than the one associated with the weights.

3.1.5. Computation of the Membrane Potential (z_j^l)

This is accomplished via an up/down signed (2's complement) counter with $p+c+w+1$ bits. These number of bits avoid overflow even if each input unit has the maximum possible gain. This counter is included in every SHiNe neuron and it operates at the speed of the fast clock (*clock*).

A Central Timing Control circuit generates index signals for enabling sampling at each level of weight, connection, and possible step of the overall excitation duty cycle. The circuit also generates control signals to trigger the unit to *fire* the new excitation level, and to *relax* back to its baseline (bias).

3.1.6. Output Activation Function

The ReLU (Rectified Linear Unit) was selected as the activation function for each neuron. This introduces non-linearity in the neuron output. As rectification causes the output values of a neuron to be non-negative, the excitation signal going to the neurons of the next layer can be represented with only 1-bit using duty-cycle encoding. Otherwise, an extra wire for the sign would be needed; handling these two wires and a more complex circuitry.

3.2. Implementation Details

Figure 5a depicts a Fully Connected Network that consists of several layers, each layer having a number of neurons. Figure 5b depicts the hardware architecture for a neuron. Note the different components: the synapse per connection circuits (blue boxes, up to 2^c), as well as the 'Integrate, Rectify, and Fire circuit'. The control signals that allow for proper processing of the duty-cycle encoded input signals across the Fully Connected Network are generated by a Central Timing Control circuit, depicted in detail in Figure 7. This circuit generates signals to every single neuron in the neural network. Note that the *clock* signal is the fast clock mentioned before.

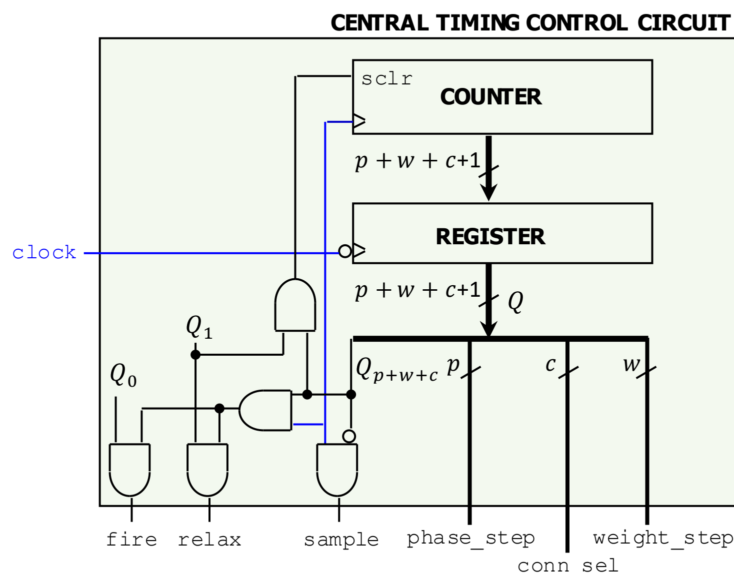

Figure 7. Central Timing Control Circuit. This circuit generates control and synchronization signals for all neurons in a network. For example, if $p = 5$, $c = 5$, $w = 3$, the counter is 14-bit wide.

3.2.1. Central Timing Control Circuit

This circuit consists of an unsigned counter with $p + c + w + 1$ bits and an associated register. The counter output bits are clustered into several groups that act as the index signals for the weights, connections, and duty-cycle phases as shown in Figure 7. The Central Timing Control Circuit generates: (i) The *weight_step* index that counts up through each possible weight magnitude, (ii) the *conn_sel* index that indicates which input signal $input_i$ we should be sampling, and (iii) the *phase_step* index that indicates with phase (out of 2^p) we are currently in of the duty-cycle encoded input signal.

The most significant bit (MSB) enables the special *fire* and *relax* signals, and briefly triggers the count reset so the next overall timing frame can start anew. The counter output is captured on a falling clock edge so the derived index values are ready for the next rising edge event.

3.2.2. SHiNe Neuron—Multiplicative Effect of Each Connection Weight

While the neuron effectively implements the product of weights by the input signal values, the Central Timing Circuit generates all the required synchronization signals.

The neuron includes a 'synapse per connection circuit', depicted in Figure 8. The input signal is called $input_i$. The associated constant weight is called $weight_i$. The resulting output signal is called $weighted_input_i$. The associated weight sign ($weight_sign_i$) is passed through. The circuit operates as follows: if the weight is greater than the *weight_step* index, then the input signal is allowed to pass onto the output. If the weight is lower or equal than the *weight_step* index, the output is set to '0'.

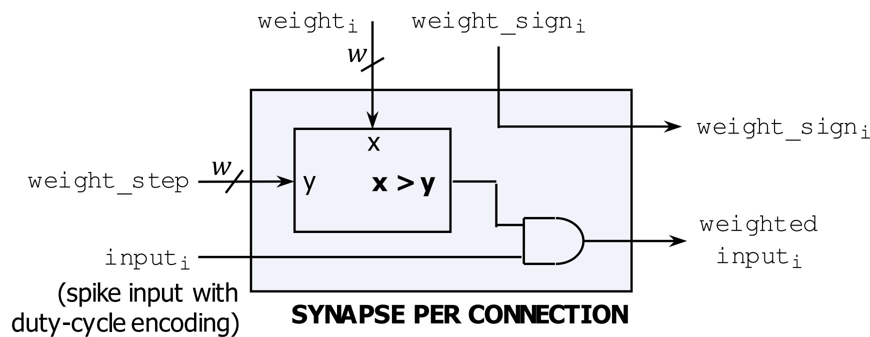

Figure 8. 'Synapse per connection' circuit. A neuron has up to 2^c instances of this circuit. The collection of signals $weigthed_input_i$ are fed into the Integrate, Rectify, and Fire circuit, as per Figure 5b.

The collection of signals $weight_sign_i$, $weighted_input_i$ ($i = 0, \ldots, 2^c - 1$) is passed to the Integrate, Rectify, and Fire Circuit (also called post-synaptic processing). This circuit samples (at the fast clock rate) each of the $weighted_input_i$ signals and accumulates the instances where the signal is high into a count value (this is called integration). A signed (2's complement) up/down counter is used; whether to count up or down depends on the $weight_sign_i$.

For $w = 2, c = 5, p = 4$, Figure 9 illustrates the multiplicative effect for two input signals shown in Figure 6: $input_0 = 2$ (signal is high for 2 phases out of 16) with $weight_0 = 2$, and $input_1 = 1$ with $weight_1 = 1$. Note that the $weighted_input_0$ lasts 2 clock cycles (out of 4) for every lap of *weight_step* (0 to 3) and for the first 2 duty cycle phases ($input_0 = 2$). The up/down counter will sample this signal 4 times (only when $conn_sel = 0$, see shaded portion), hence implementing the product $2 \times 2 = 4$. Likewise, the $weighted_input_1$ last 3 clock cycles (out of 4) for every lap of *weight_step* and only for the first duty cycle phase ($input_1 = 1$). The up/down counter will sample this signal 3 times (only when $conn_sel = 1$, see shaded portion), implementing the product $1 \times 3 = 3$.

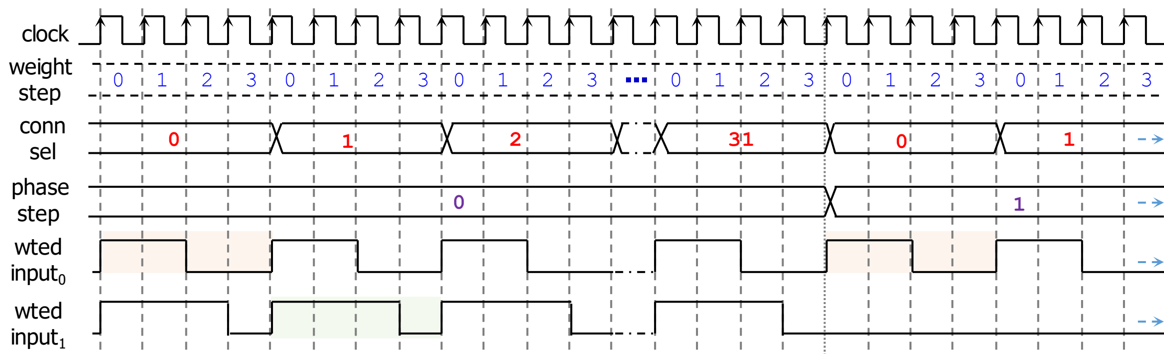

Figure 9. Multiplicative effect implemented by the 'synapse per connection' circuit with $p = 4, c = 5, w = 2$. The input signals $input_0$ and $input_1$ are depicted in Figure 6.

3.2.3. SHiNe Neuron—'Integrate, Rectification and Fire' Circuit

The integration consists on collecting the weighted signals from each input synapse. This is achieved this by using the connection selection (*conn_sel*) index from the Central Timing Control circuit as the selector of a 2^c *to* 1 multiplexor for both the weighted input signal pulses and weighted signs. The selected signal for the currently-sampled connection is passed to a signed (2's complement) up/down counter that holds the integrated membrane potential z ('0': count up, '1': count down). Note that at the end of a prior frame (frame: 2^p phases or 2^{p+c+w} clock cycles), the *relax* signal from the Central Timing and Control Circuit will reset the counter to the signed $(w+1)$-bit neuron bias value (defined in supervised learning). This Integration Circuit that generates the membrane potential z is depicted in the top portion of Figure 10.

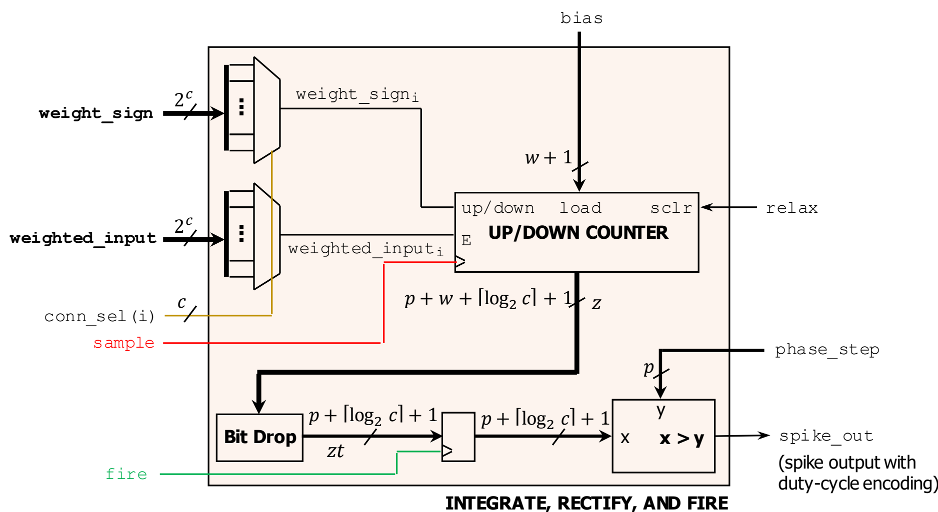

Figure 10. Integrate, Rectify, and Fire Circuit included in every SHiNe neuron.

The membrane potential z (value before the activation function) is a sum of c terms, where each term is a product of a weight and an input with $w+1$ (weight magnitude and sign) and p bits respectively. Thus, the total number of bits required is $p + w + \log_2 c + 1$ bits. For the counter, z appears as a signed integer. Using the fixed-point representation for the weights, z is a signed fixed-point number with $p + w + \log_2 c + 1$ total bits and w fractional bits.

Next, we need to rectify this value (ReLU) and generate the duty-cycle encoded output. This is accomplished by the circuit depicted in the bottom portion of Figure 10.

To prevent the signal bit-width from growing too rapidly, it is typical in fixed-point operations to truncate the fractional bits. In our SHiNe design, the duty-cycle encoded inputs are defined as

unsigned integers. Thus, we need to drop w fractional bits to get integer values. This re-scaling operation (division by 2^w) results in an integer value zt with $p + \lceil \log_2 c \rceil + 1$ bits.

To apply the activation function (ReLU) to the re-scaled value zt, a common approach is to use the sign of zt (its MSB). If $zt < 0$ then the rectified value is 0. If $zt \geq 0$, then the rectified value is equal to zt. After that, we need to encode the rectified value as a duty-cycle encoded output.

The SHiNe design uses a simpler approach, where we combine the rectification with the duty-cycle encoded output generation in a single step. In Figure 10, when zt is ready to be captured, the *fire* control signal is activated, and zt is latched onto the register. The rectification and duty-cycle encoding are implemented with a comparator, that compares the signed, re-scaled potential zt against the positive-only *phase_step*. If the latched zt is negative, it will always be lower than the current *phase_step*, causing the output to be always 0. On the other hand, it the potential is positive and greater than the current *phase_step*, the output is 1. As soon as the current *phase_step* matches or exceeds the latched potential, the output drops to 0.

As an example, consider a SHiNe neuron with $w = 2$, $p = 4$, $c = 4$. Here, the weights are in the range $[-0.75, 0.75]$ and the input signals are in the range $\left[0, 2^4 - 1\right]$. The input data is given by [2, 5, 1, 7]. The weights are given by $[-0.5, 0.75, -0.25, 0.5]$. Then, the potential z is $-2 \times 0.5 + 5 \times 0.75 - 1 \times 0.25 + 7 \times 0.5) = 7.75$. The up/down counter sees the value as $7.75 \times 2^w = 31$. The bit-drop of w bits amounts to the integer division of $31/2^w = 7$. Applying ReLU, the result is 7.

If the weights had been $[0.5, -0.75, 0.25, -0.5]$, the potential z would be -7.75. With the bit drop, we get -7. Applying ReLU, we get 7. Figure 11 shows some examples: -7, 7, 13.

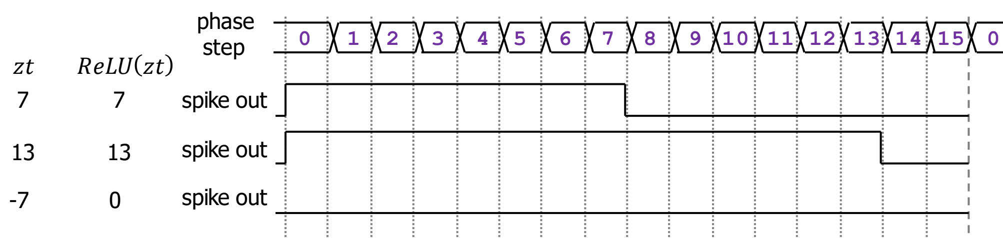

Figure 11. Examples of output spikes.

In theory, the final unsigned value after rectification requires $p + \lceil \log_2 c \rceil$ bits. This means that the duty-cycle encoded output signal needs $2^{p + \lceil \log_2 c \rceil}$ phases. However, as a design consideration, we limit all duty-cycle encoded signals to 2^p phases. So, if the final unsigned value requires more than p bits, our comparator will saturate the value, i.e., it will set the output value equal to 2^{p-1}.

3.3. Hardware Processing Time

A frame is defined as the length of the duty-cycle encoded input, i.e., 2^p phases or 2^{p+w+c} cycles. Due to the processing scheme at each neuron, a layer processes a duty-cycle encoded input in a frame. As a result, the processing time for a network layer in our SHiNe design is given by 2^{w+c+p} clock cycles, i.e., the number of cycles needed to sample all external duty cycle encoding phases, connection fan-in, and effective bit-width of connection weights. Figure 12 depicts the number of clock cycles for different values of w, c, p.

Note that the integration function occurs during one frame (the duration of a duty-cycle-encoded signal), and ends at the end of the frame, after sampling all the *weighted_input*$_i$ signals. At the end of this integration process, the *fire* signal triggers the circuit to forward the result as a duty-cycle encoded output, simulating the biological actional potential traveling down the axon. So, for a layer, the generation of the duty-cycle-encoded outputs occur in the next frame after the integration frame so that the next layer can process them.

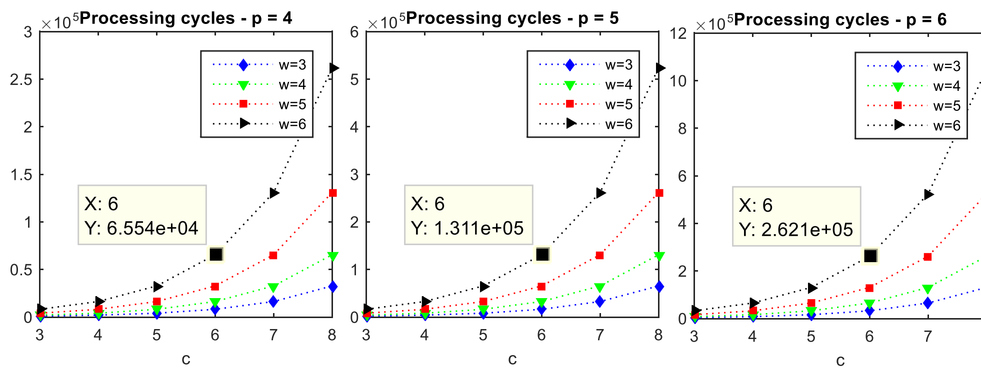

Figure 12. Processing cycles vs. w, c, p.

Thus, the generation of the output of a prior frame overlaps with the integration of the current frame. This amounts to a pipelining effect, where each network layer adds one frame to the initial latency for input processing and reading out the final output requires an extra frame. This pipelining effect is depicted in Figure 13 for the neural network of Figure 5a. If we continuously feed a new set of inputs at every frame, the neural network generates a new output every frame. The throughput of the hardware design is given by Equation (7).

$$Network\ Throughput = \frac{(\#\ of\ outputs\ of\ layer\ L) \times (\#\ of\ bytes\ per\ output)}{2^{w+c+p}\ clock\ cycles} \tag{7}$$

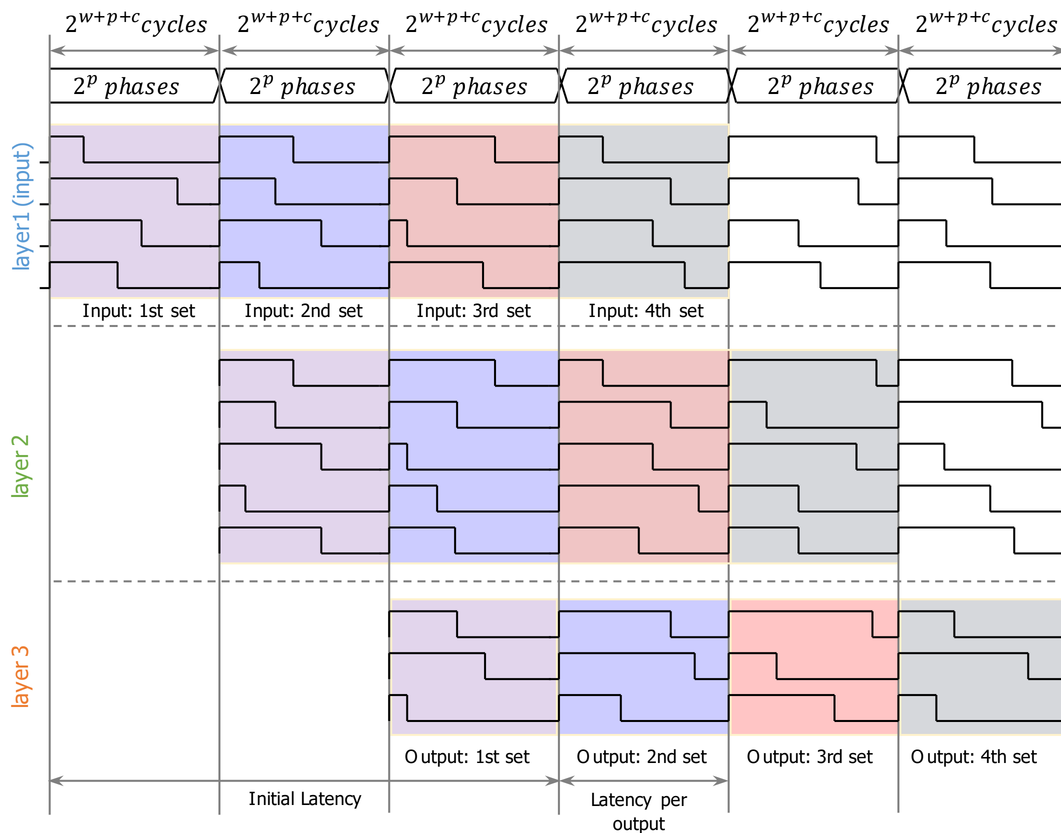

Figure 13. Pipelining effect for a 3-layer network with 4 units in the first layer, 5 units in the second layer, and 3 units in the third layer. A frame lasts 2^{w+p+c} clock cycles.

Figure 14 shows an example of this timing pattern (in FPGA simulation of the synthesized circuit). We have a 3-layer network with 10 neurons in the last layer, so the results are seen in the 3rd frame

relative to the feeding of a particular input. As per their encoded value, the duty-cycle-encoded outputs in all neurons in a single layer rise to '1' at the same time but drop to '0' at various times within the frame. For a neural network performing a classification task, we need to look at which unit in the output layer has the largest duty cycle to determine the 'correct' output (the recognized classification).

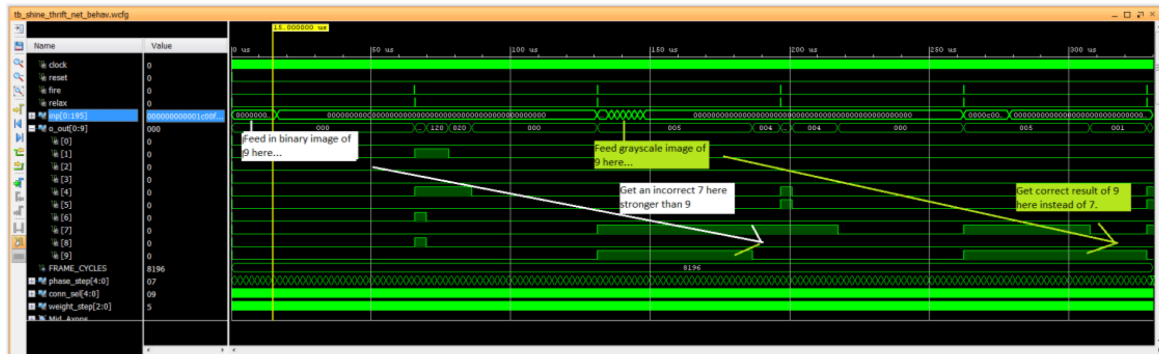

Figure 14. SHiNe signal waveforms over multiple frames.

3.4. Training Approach and Software Tools

The accuracy performance of a neural network is critically related to the arrangement and organization of the neuron units and the specific values of neuron unit parameters such as bias and connection weights. Certain network architectures tend to work well in some applications and not as well in others, so the designer should select the best suited for a particular application. On the other hand, the neuron bias and connection weight values cannot be computed by hand in any but the most trivial examples. Therefore, various methods of training (Machine Learning) are used to determine a usable set of neuron parameters throughout the overall configured network architecture. These methods can start with an untrained network, perhaps configured with default or random values, and adjust it through a series of training examples toward a more optimal set of values that achieves required results. Some methods are considered unsupervised where the direction of adjustment is based on a reward/punishment factor that is inferred by the problem domain (hit something in a virtual world, lose a game, etc.), defined in a general similarity to an exemplar (what a human does in a similar situation), or tied to some useful internal metric (ability to cluster observations into distinct types). Other methods are considered supervised where there is a specific right answer for each given input, and other answers are considered wrong. This is called labeled data, where the correct class of input is identified, or the location of a significant item in the data input is defined, or other representation of the correct answer is specifically available. However, it is not always possible or practical to provide labeled data in enough quantity or diversity, especially if the cost to provide the labeled data is prohibitive, which is why unsupervised methods have attracted attention in the research community. However, since the supervised learning case has precise right and wrong answers, the notion of accuracy is more well-defined and easier to quantify. Also, the computation speed of supervised learning to a usable level is generally faster. In this work, we used supervised learning, as it is more straightforward to apply and because of our prior experience with it.

A software application (in C) for a neural network training and testing has been developed. It can load the MNIST data set. It can train and evaluate a configurable number of network layers, network layer sizes, and network neuron types. It can also generate hardware description code for the top-level file of the hardware code. This software application is detailed in Appendix A.

3.5. Circuit Synthesis and Simulation

We use the Xilinx Vivado 2016.3 development tool to synthesize, simulate, and estimate power consumption for each design. The target device was the Zynq XC7Z010-1CLGC400C that is housed in the ZYBO Development Board. As a result, the fast clock is set up to be 125 MHz (8 ns).

3.5.1. Synthesis Details

The SHiNe architecture was described in VHDL at the Register Transfer Level (RTL). As hinted in the command-line options for the C software application (Appendix A), *cnn* can generate a VDHL description of the top-level hardware file that implements a configured, trained network. All the VHDL lines of code are printed from the console output into a text file (.vhd file).

The automatically-generated VHDL code has a straightforward structure. First, some boilerplate statements for the IEEE libraries and main top component definitions, customized according to the name of the network configuration file. Next, the generic control and neuron component statements, a large set of constants and signals for the trained weights, biases, and axons. Then, the instantiation of the control circuit and each neuron, grouped into layers, with each neuron's inputs, weights, and bias connections tied to the correct constant or top-level circuit signal. Our current implementation of this feature is limited to two active layers that follow both the initial input layer and a max-pooling down-sampling layer.

A similar approach is used to generate a fixed-point implementation (called *FXP_N*) of neural network that uses more conventional multiply-and-accumulate (MAC) dot product type calculations [22]. This common architecture for one neuron is depicted in Figure 15. As the weights and biases are constant, the multiplier is constant. This still requires relatively large resources for a single neuron. This allows for hardware usage comparisons, which will be detailed in the results section.

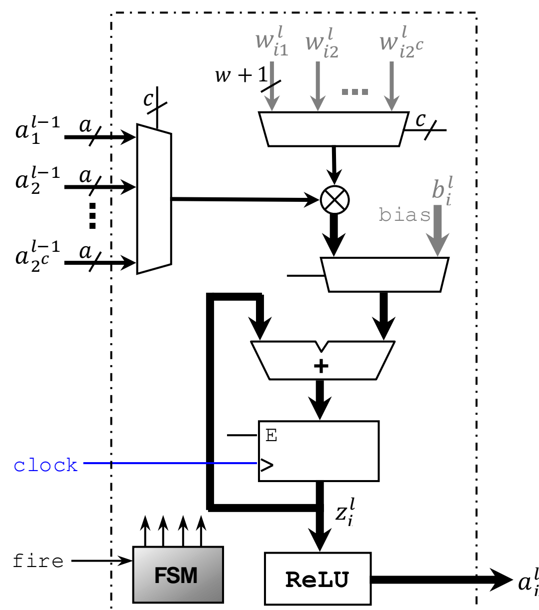

Figure 15. FXP_N neuron with 2^c inputs. Input excitation levels are multi-bit signals. This common architecture will be used to compare resources and power to those of the SHiNe design.

The VHDL code for implementing a trained SHiNe instance was organized as follows:

- **SHiNe_neuron.vhd**: This code describes one neuron, assuming certain timing and control inputs, a set of input lines, input weights (sign and magnitude), and a signed bias value. All of these inputs are passed in from a higher-level module once it is instantiated. Bit-widths for several

parameters are defined in a generic fashion. Once the fire input arrives, the new version of the output signal is prepared, and a relax resets the neuron to its biased initial value;

- **ShiNe_ctrl.vhd**: This is the Central Timing and Control Circuit that generates a set of multi-bit index signals, and some additional trigger signals, including sample, fire and relax. It includes a reset input so the control indexes can be reset to allow a frame to be processed at power-up;
- **top_{net name}_net.vhd**: Automatically generated by *cnn*. This is the top-level circuit for a neural network. It instantiates the control circuit configured to specific bit-widths for the control indices, a set of neurons organized into layers according the network configuration file. The trained weight and bias are set as constant values that are fed into each neuron instance. If a connection is not needed because a prior layer has fewer units than the maximum allowed number of connections, the unused neurons inputs are tied to '0';
- **tb_{net name}_net.vhd**: This is the testbench for simulation. It feeds the input images (binary or grayscale) to the circuit so that the output lines can be retrieved at simulation. For example, in hand-digit recognition, we have an output line per each numeric digit we can recognize.

We will use a preferred set of hardware parameters: $w = 3$ (weights); $c = 5$, which allows for up to 32 input connections per neuron; and $p = 5$, which defines 32 different excitation input levels.

3.5.2. Simulation

Behavioral simulations were run for many examples of MNIST training and testing images, where the input data was prepared by running *cnn* for grayscale images or binary images (with the *-imN* option to print a binary-threshold version of an image). Input data is placed in the testbench file to feed the top-level circuit's image input. For example, there is a set of more than 40 test case images in the **tb_thrift_net.vhd** file; testing all of them is a laborious process (loading of test data from the MNIST dataset, simulation, and output retrieval and verification) and it could be automated.

The simulation time was configured to allow several frames. The input image was modified after a few frames to ensure different results. For the preferred set of bit-width parameters, a total of $2^{w+c+p} = 8192$ fast clock cycles or 8192×8 ns $= 65.536$ ns are needed per frame. For two active layers, we need two processing frames, plus one additional frame to read out the result. For a second input, it can be fed at the start of the second frame and be pipelined to appear in the fourth frame. In our testing, we usually held the same input for two frames before moving on to a new input test case, this results in about 5 frames (327.68 us) of simulation time span.

4. Results

There have been many approaches among researchers to design and test efficient artificial neural network design. Here, we explore some known options, selected ones that were predicted to work well with a spiking-type neural network, and then evaluated the overall network performance.

4.1. Neural Network Trade-Off Evaluation

Neural networks with limited bit-width values are impacted by quantization noise, due to error introduced by truncation or rounding up/down. For our activation outputs, we perform a bit drop. Every bit introduces up to 1 bit of error (0.0 to 1.0 bits for truncation, or −0.5 to +0.5 for rounding).

We will use a simplified model for the quantization noise. Using Equation (1), and ignoring any trained bias, we can say that for each neuron j, its membrane potential z_j^l is proportional to the sum of inputs and weights. Since each of the multiplicands has a quantization error, we can model the resulting product fractional variance as:

$$\left(\frac{\sigma_{z_j^l}}{2^b}\right)^2 = \left(\frac{\sigma_{a_k^{l-1}}}{2^b}\right)^2 + \left(\frac{\sigma_{w_{jk}^l}}{2^b}\right)^2 = \frac{1}{12(2^{2b})} + \frac{1}{12(2^{2b})} \quad (8)$$

where 2^b is the number of quantization levels for a given number of bits b, which we assume to be the same for weights and input activations. The right-most side of the Equation (8) results from the fact that the variances $\sigma^2_{a_k^{l-1}}$ and $\sigma^2_{w_{jk}^l}$ are known to be 1/12 each because the quantization error is uniformly distributed over a range of 1. In a neural network, the membrane potential z_j^l in layer l is a function of the previous neural network layer's activation signal a_j^{l-1}. Assuming that the variance of the activation output a_k^{l-1} approximates that of the membrane potential z_k^{l-1}, we can replace $\left(\frac{\sigma_{a_k^{l-1}}}{2^b}\right)^2$ by $\left(\frac{\sigma_{a_k^{l-2}}}{2^b}\right)^2 + \left(\frac{\sigma_{w_{jk}^{l-1}}}{2^b}\right)^2$. Applying this recursively, the quantization terms can be grouped together, obtaining L of them. Then, we can add the fractional variance of the input quantization, which we assume to be of similar scale (bit-width wise) as the weight factors, resulting in $L+1$ times the weight quantization terms. So, after L layers, the overall noise to signal ratio is estimated to be:

$$\left(\frac{\sigma_{z_j^L}}{2^b}\right)^2 = (L+1)\left(\frac{\sigma_{w_{jk}^L}}{2^b}\right)^2 = (L+1)\left(\frac{1}{12(2^{2b})}\right) \tag{9}$$

To make a network accuracy prediction, we can use $Q(1-F)$ where F is the overall fractional noise to signal ratio, which is the square root of the fractional variance as in Equation (9). This results in Equation (10), where Q is a maximum expected accuracy assuming no quantization noise.

$$Accuracy = Q(1-F) = Q\left(1 - \frac{\sigma_{z_j^L}}{2^b}\right) = Q\left(1 - \sqrt{\frac{L+1}{12}}.\frac{1}{2^b}\right) \tag{10}$$

4.1.1. Experiment with a Small Neural Network

In our experiment, we defined a small neural network for the MNIST handwritten digits data set that had 28 × 28 input units immediately followed by 2 × 2 max-pooling to produce a 14 × 14 downsampled image (this is the max-pool layer). From the hardware perspective, this max-pool layer is the input layer, which is then fully connected to an inner (hidden) layer of 16 neurons, and finally fully-connected to the 10-neuron output layer. The training and performance evaluation were done with full double floating-point connection weights and activation signals. Then the *cnn* application was run with various levels of quantization to estimate the performance of low bit-width integer implementation. Each level of quantization was tested with a 1000-example test round drawn from the 10,000-image test set and repeated 20 times to determine an average value and an estimate of the variation to produce the plot in Figure 16. Results were encouraging as this small network was able to achieve well over 90% recall (correct categorizations to all tested input test cases) even down to fairly small numbers of bits, as per Table 1 and Figure 16.

Table 1. Low bit-width recall accuracy in a small, shallow network.

Bits Per Connection	Average Accuracy (%)
0	26.67
1	70.42
2	86.17
3	89.94
4	93.17
6	94.57
8	94.54
64	94.60

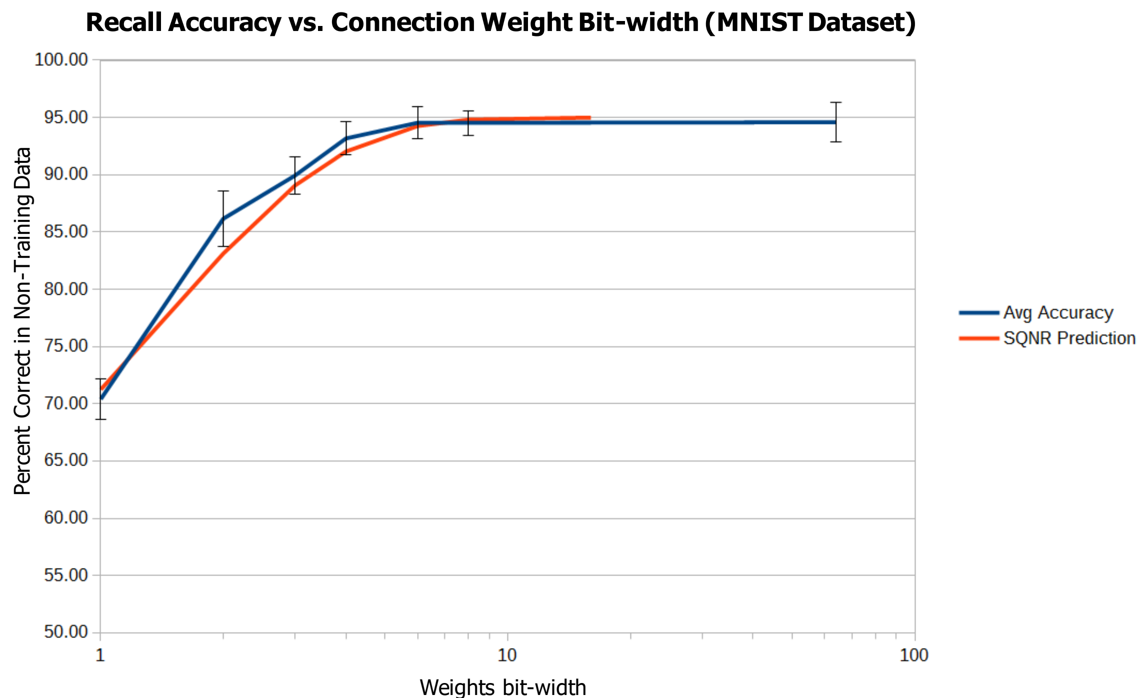

Figure 16. Recall (classification) accuracy for various bit-widths.

Figure 16 shows that the accuracy falls off significantly for very low bit-widths, but it also shows that there is no practical benefit to using more than 6 bits for connection weights, bias values, and activation levels in this small network. This finding enables a reduction in hardware resources.

4.1.2. Thrifting to Reduce Hardware Resources

Hardware resources can be further reduced if we limit the number of allowed connections from neurons in a layer to a neuron in the next layer. We call this approach '*thrifting*' after the practice of depopulating electronic components from hardware boards that seem to provide little to no extra value once a particular design is debugged and well-vetted. However, the term *pruning* is more widely known in the ML community. This connection limit can also be thought of as fan-in (like the limit of input lines a digital gate can directly support). Reducing fan-in helps minimize the circuits by having far fewer signals being routed. In the SHiNe design, having fewer connections means fewer clock cycles to iterate through the connections, whereby halving the allowed connections per neuron also halves the total time needed to sample and count incoming signal spikes.

We tested a few levels of thrifting using *cnn* with the *-thrift{n}* command-line option which implements a strict sorting of weight magnitudes and replaces all weights smaller than the nth-largest magnitude with a zero. At each thrifting level, the accuracy assessment was repeated 20 times to produce a more reliable average and variance. We anticipated thrifting to improve the network performance; this makes the remaining connections more important and more sensitive to fine tuning. The network was re-trained with the connection fan-in limits and performance retested at the limit shown in Table 2 and Figure 17. Note a modest increase of accuracy at the same level of thrifting in most cases, ranging up to about 7% (and an average of 2.8% higher) at the same fan-in limit.

The overall shape of the thrifting plot in Figure 17 is consistent with the pattern formed by reducing connection weight bit widths. Notice that a fan-in limit of 32 connections achieves essentially the same accuracy as higher levels of connectivity. However, a fan-in limit of 16 causes a moderate drop-off of performance, and a limit of 8 connections per neuron restricts performance to about 60% accuracy, a level probably far too low to be usable in most applications.

Table 2. Thrifted recall percentage with normal and thrifted training.

Fan-In	Normal Training (%)	Thrifted Training (%)
1	11.78	9.67
2	20.69	27.01
4	37.61	36.33
8	54.28	60.79
16	80.79	85.50
32	88.78	92.65
64	92.51	92.54
128	93.24	
256	93.12	

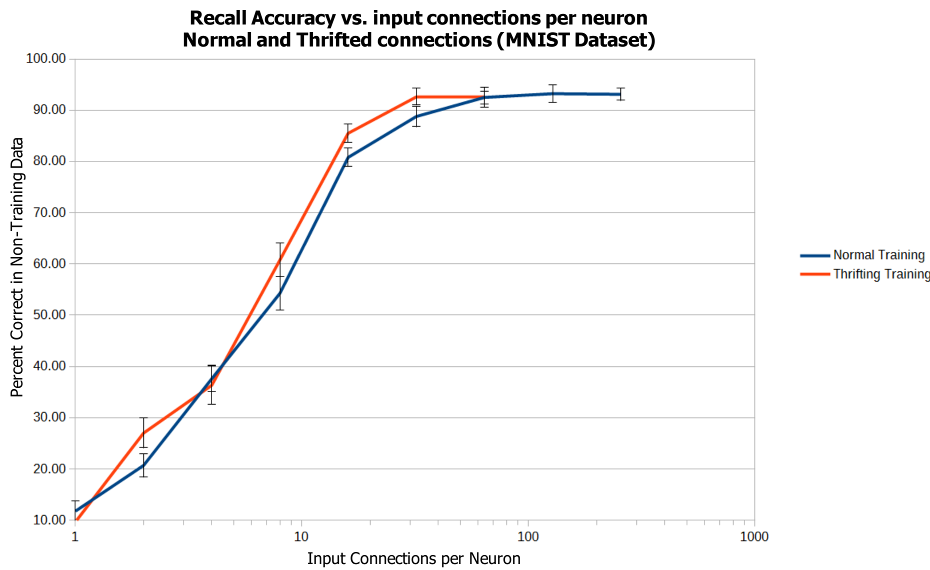

Figure 17. Thrifted Recall % with normal and thrifted training.

To better understand the impact and potential for thrifting, we generated histograms of the connection weights in Figure 18. In the hidden layer, we see that lower-value weights predominate. With conventional network training, the histogram is more spread out and has a longer tail. After re-training with the *cnn* program, the histogram for thrifted network connections shows relatively fewer small-weight values, except the zeroed (thrifted-out) connections. This was not the case in the output layer, where the weights' distribution is fairly uniform. However, after thrifting, the values drop slightly.

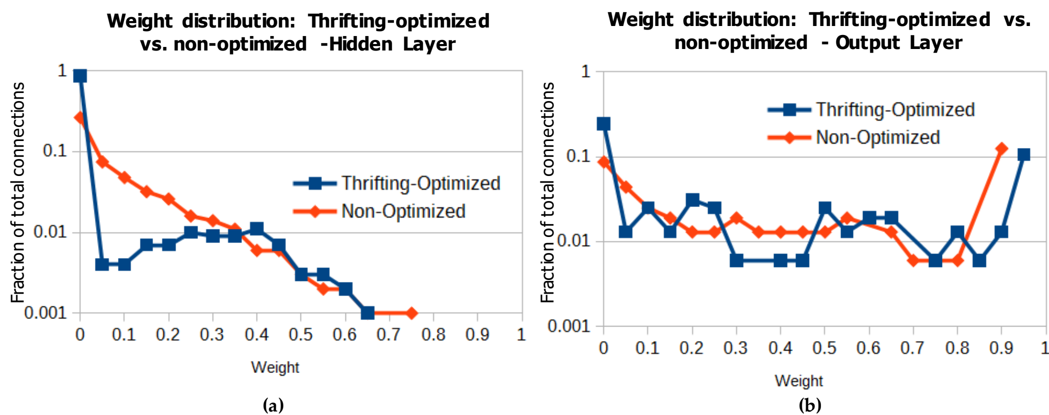

Figure 18. Histogram plot of thrifted vs. normal connection weights. (**a**) Weights in Hidden Layer. (**b**) Weights in Output Layer.

This cost-benefit analysis allowed us to conclude that we should use 3-bit connection weights ($w = 3$), and activation signals thrifted to a fan-in limit of 32 ($c = 5$). This will provide decent accuracy and performance with the SHiNe design but at a much smaller hardware cost than a fully-connected network, and/or one with floating-point or even byte-wide fixed-point signal paths.

4.2. Hardware Accuracy (or Network Performance) Evaluation

The SHiNe design, implemented in VHDL, uses a combination of manual coding and machine-generated top-level circuit model from the *cnn* application. We did not implement an automated testing framework that could use the full MNIST dataset to exercise the synthesized VHDL design, but we used *cnn* to print VHDL-compatible binary representations of the hand-written digit images that could be pasted into the SHiNe test bench file to generate the needed input signals.

Note that our test case is the one with the following hardware parameters for all the neurons in our different networks: $w = 3, c = 5$, and $p = 5$. Hardware test is carried out with an input layer of 14x14 pixels (down-sampled layer), a hidden layer with 16, 32, or 64 neurons, as well as an output layer with 10 neurons (for hand-written digit recognition). These networks are listed in Table 3. Note that the initial down-sampling using max-pooling units is not part of our hardware design.

Table 3. SHiNe design: 3 network layers that were used for testing.

Network	Name	Input Layer	Hidden Layer	Output Layer
1	SHiNe_f16_m10_net	196	16	10
2	SHiNe_f32_m10_net	196	32	10
3	SHiNe_f64_m10_net	196	64	10

Many tests were run with binary images (thresholded grayscale images). We also performed testing with 5-bit grayscale images, where higher intensity portions of the input image had a high duty-cycle waveform, while less intense portions had a lower duty-cycle waveform. This helped the network correctly identify marginal classification cases, where the binary-only version of the input failed. Note that these input signals in the testbench along with a clock and reset signals feed into top-level circuit at the 'fully-connected' (but thrifted) layer.

For binary images, the simulation would read this hard-coded data and start a spike for each 1-valued pixel and no spike for each 0-valued pixel. These values where held constant through all of the sub-frames for duty-cycle encoding. Note that these are still encoded as 5-bit signals, where '0' is 0, and '1' is 31). A binary image example is shown in Figure 19 that also shows the corresponding VHDL constant initialization statement.

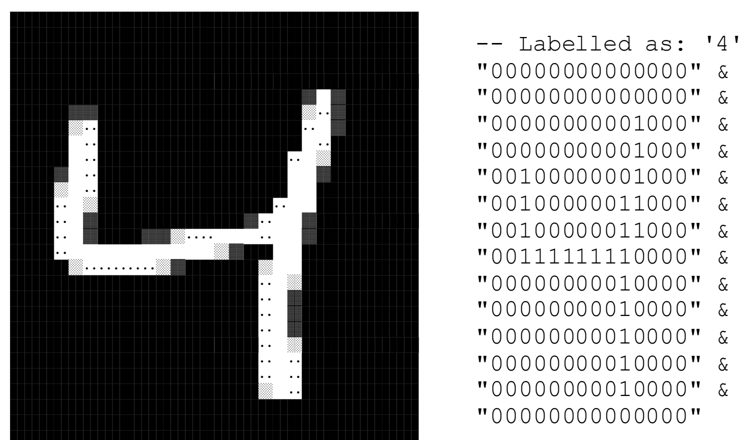

Figure 19. Input image (binary) example used in our simulation.

Hardware verification was carried out by comparing the software *cnn* implementation and the SHiNe design for 43 test images that were fed into the testbench VHDL files and simulated one at a time. The simulation was examined to determine if the neural network output was correct (whether it generated the longest duty-cycle signal). Due to the large quantization steps in the output, it is possible to have 'ties', where multiple neurons, each representing a different classified digit, generate the same output value. For a suite of tests with 43 images, Table 4 lists results on correct classification of the SHiNe design (network 1) with respect to the corresponding *cnn* C-based simulation.

Table 4. SHiNe design—network 1 accuracy evaluation: pass, tie, and fail count with respect to the *cnn* software implementation.

	Pass	Tie	Fail	Total
Tallies	39	1	3	43
SHiNe (vhdl) %	90.70%	2.33%	6.98%	
C-Sim %	91.97%	For grayscale input images		
C-Sim %	89.60%	For Binary-thresholded input images		

Once can see that the C-Sim result using binary thesholded values was about 89.6%, whereas the SHiNe design scored about 90.7%. The difference between these two numbers is not statistically significant.

4.3. Hardware Processing Time

As for processing speed of the SHiNe design, even with our modestly-valued parameters of $w = 3, c = 5, p = 5$ (8 weight levels, 32 connection fan-in, 32 duty cycle levels), the total number of cycles per frame is relatively high (8192). The hardware design is fairly simple, which may allow it to be clocked at high speeds. In our setup, the clock rate is 125 MHz, which results in 8192×8 ns $= 65.536$ us per frame. Due to the pipelined implementation, a new result can appear at each output frame.

This processing time is very quick compared to large and/or deep networks, but it would be slower than highly optimized DSP-type MAC (multiply-and-accumulate) operations. We note that the fixed-point network implementation (*FXP_N*) used for comparison purposes requires fewer processing cycles per processing frame. However, due to the operations involved, the frequency of operation might not be as high as in our SHiNe design.

4.4. Hardware Resources

Hardware resource requirements in terms of 6-to-1 LUTs (Look-Up Tables) and flip-flops (FFs) were extracted from synthesis results. The network modeling and synthesis was repeated for networks of differing sizes so that common parts of the baseline FPGA synthesis and deployment not related to network scale could be factored out. Recall that the target device was the Zynq XC7Z010-1CLGC400C. Table 5 shows the resource costs for the SHiNe neuron version and a similar fixed-point math neuron version with the same bit widths and general constraints.

Table 5. Resource requirements for simple fixed-point design vs. our SHiNe design.

	Neurons	FXP_N		SHinE		SHiNe Hardware (LUT) Savings (%)
		LUTs	FFs	LUTs	FFs	
1	16 + 10 = 26	1635	720	814	646	50.1%
2	32 + 10 = 42	1955	930	1091	824	44.1%
3	64 + 10 = 74	2402	1260	1335	1003	44.2%

The hardware usage trend upward shows a slight decreasing rate. We speculate the FPGA routing complexity would eventually dominate the resource demands, leading to an increasing slope again for larger circuits.

Table 5 shows a large reduction in combinational hardware resources (−45%) compared to a standard neural network implementation. In terms of flip flops, the reduction is slight (−10%).

The large reduction in hardware resources is due to two factors: (i) Neuron implementation: while each SHiNe neuron includes a counter and a set of comparators, a standard neuron implementation (Figure 15) includes a multiply-and-add accumulator (even when only constant multipliers are needed) and multi-bit multiplexors, and (ii) neural network: The SHiNe network is significantly simpler than the standard neural network as it requires only one bit per signal.

Our SHiNe design shifts the hardware complexity from routing to logic. However, even at the logic level, for each neuron, the architecture is simpler than that of a standard neuron (Figure 15).

4.5. Hardware Power Consumption

On FPGAs, the on-chip power is the sum of the device static power, the design static power, and the dynamic power. The design static power is a function of the utilized resources. The dynamic power is consumed when the circuit is operating. The device static power depends on the environment, the device family and size (for all practical purposes, it is assumed to be constant). For example, the device static power of the Zynq XC7Z010 is 92 mW at 25 °C. For a larger chip like the XC7Z045, the device static power is 201 mW at 25 °C.

We utilize the Vivado Power Analysis tool to provide estimates of the power consumption of our designs. These estimates are reported in Table 6. We provide the total power and the design power (the sum of the design static power and dynamic power). Note that the SHiNe design saves about 25% in power consumption with respect to *FXP_N*.

Table 6. Power consumption (mW) for fixed-point design vs. our SHiNe design. Results for the Zynq XC7Z010 PSoC, where the device static power is 92 mW at 25 °C.

Neural Network	Neurons	FXP_N		SHinE		% of Design Power Savings of SHiNe
		Design Power	Total Power	Design Power	Total Power	
1	16 + 10 = 26	90	182	69	161	23.3%
2	32 + 10 = 42	79	171	60	152	24.1%
3	64 + 10 = 74	66	158	50	142	24.2%

4.6. Power Consumption—Comparison with Other Implementations

Table 7 lists the power consumption of various implementations. Providing a fair comparison across different platforms is challenging. First, we have to ensure that the neural network is of comparable size to the one presented here. We attempted to do this, but we note that in most of the works the neural network included convolutional layers.

For FPGAs, some works report the power consumed by the specific circuitry of the neural network, while others include the power of external interfaces, memory, or the testing board. In addition, the device static power in FPGAs makes across different devices and families very difficult. In terms of CPU and GPU implementations, note that what is reported is the power consumed by the computer and the GPU card respectively, irrespective of the load. All in all, Table 7 is useful to provide a big picture of the space of implementations for neural networks.

Table 7. Power comparisons with FPGA, CPU, and GPU implementations of Neural Networks.

Work	Platform	Power	Details	
[23]	ASIC	18.3 mW	40 nm CMOS technology	CNN, fixed point
Ours	FPGA	161 mW	Estimated power of the design	ANN (SHiNe_f64), 196 × 196 image
[24]	FPGA	4000 mW	Device power: Zynq XC7Z045 Zynq (ARM-Cortex + FPGA) + memory	CNN, fixed point, 500 × 350 image
[25]	FPGA	15 W	Power of a custom board: Virtex-4 SX35 FPGA + QDR-SRAM.	CNN 500 × 500 image
[26]	FPGA	7.3W	Power of the XC7Z045 device	Binarized Neural Network (BNN)
[25]	CPU	90 W	Core 2 Duo CPU	CNN
[27]	CPU	95 W	Core i7 860 CPU	CNN
[28]	GPU	133 W	Tesla C2075 GPU	ANN
[29]	GPU	240 W	K40 GPU	CNN
[27]	GPU	216 W	GeForce GTX 275 GPU	CNN

As expected, the FPGA designs consume far less power than the GPU or CPU implementations. We note however, that ASIC designs can consume even less power than FPGAs. Our design consumes less power than comparable FPGA designs [24], even when accounting for the convolutional layers and larger images.

5. Conclusions

A bit-serial hardware-optimized implementation for a Spiking Neural Network (SNN) was presented. The approach included (i) duty-cycle-encoding method for signals, (ii) dot product implementation by repeated sampling of the signal duty cycle for weight products and a counter to sum of products, thereby avoiding dedicated multiply-and-add hardware, (iii) combination of non-linear ReLU operation with the generation of the duty-cycle-encoded output using a comparator. Further simplifications were enabled by reducing the effective bit width of connection weights, biases, and activation signals and by thrifting (pruning) connections that are small in magnitude. The approach, named SHiNe, allows for the hardware implementation to achieve reasonable accuracy (>90%) for the MNIST dataset, which is very close to results from a C neural network software application. When compared to a similarly strength-reduced and thrifted fixed-point Multiply and Accumulate (MAC) approach, the SHiNe design has significantly lower FPGA resource utilization (about 35% less) and thus less power consumption.

Our design is tailored to embedded applications, especially reconfigurable embedded systems that include a reconfigurable fabric and a microprocessor [22]. Due to the inherent serialization of input sampling per neuron units, this bit-serial design works well for signals with small bit-widths, that as demonstrated in this work achieved reasonable accuracy. For larger bit-widths, the design requires many clock cycles. However, the bit-serial nature of the architecture allows for fast clock rates, and the aggregate time should still be fast enough to process many real-world problems. While the SHiNe design was verified with an FPGA implementation, the design can also be targeted onto an ASIC (application-specific integrated circuit) that can achieve very high clock rates.

Author Contributions: Conceptualization, methodology, software, hardware design, validation, and original draft preparation, M.L.; supervision, writing—review and edition, D.L.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Here, we detail a C software application that can train and evaluate a configurable number of network layers, layer sizes, and network layer neuron types. It can also generate specific VHDL code.

Appendix A.1. Available Neuron Types

- *Input*—always outs a specific value taken from a test data set, or real-world phenomenon.
- *Convolution*—a unit that applies convolution to a portion of a prior layer. A convolutional layer (not addressed in this work) may be arranged into multiple portions called feature maps.
- *Pooling*—a unit that passes a reduced version of the connected inputs, usually a compact set (e.g., 2 × 2). The most common pooling function is the max-pool, that achieves a down-sampling effect.
- *Fully-Connected*—a unit that can receive input from any unit in the prior layer, according to the weights. Weights are allowed to be zero, effectively meaning that it is disconnected.
- *Soft-Max*—a fully-connected unit with normalized output relative to all other units in the layer to emphasize a unit's output in the layer and assign a likelihood-type value to each unit's output.

The C-software application was named '*cnn*', due to its ability to support Convolutional Neural Networks (convolutional layers and the fully connected network). such as LeNet developed to perform hand-written digit classification in the MNIST data set [20].

The input to *cnn* is a text file that specifies a network configuration, layer organization (layer count, layer types, dimensions for each layer), and optionally, the specific weights and bias values for the units in the layers. The naming convention is a shorthand description of the network structure, such as *f16m10* for a fully-connected layer of 16 units followed by a soft-max layer of 10 units. An example of a small blank network configuration is shown in Figure A1, where the weights and bias values are left unspecified. Lines starting with dash (-), equals (=) and hash (#) characters are ignored.

```
------------------------------------------------------------------
Num layers: 4


Layer[0] type: input
Layer[0] shape: [28 28] # [ rows cols ]
============================================================
Layer[1] pooling: [2 2 2] # [ height width stride ]
# NOTE: Pooling results in a [14 14] size image,
#       196 total pixels/neural units
============================================================
Layer[2] type: hidden   # fully connected
Layer[2] shape: [1 16]  # [ rows columns ]
Layer[2] weights shape: [196 16]  # [ prev_layer_units
#                                     this_layer_units ]
============================================================
Layer[3] type: softmax  # fully connected to prior layer,
#                         but output adjusted by soft-max to
#                         provide likelihood values
Layer[3] shape: [1 10]  # [ rows columns ]
Layer[3] weights shape: [16 10]  # [ prev_layer_units
#                                    this_layer_units ]
============================================================
End.
```

Figure A1. Example of network configuration file for cnn application.

Some layer types such as input and pooling do not require any weight parameters. Fully-connected and soft-max layers need weights and bias values. If no weights and bias values are provided, *cnn* will compute random weight values in a pseudo-normal distribution around zero with a standard deviation of about 0.2 and bias values with a standard deviation of about 0.3. Figure A2 shows a network where weights and bias values are indicated.

The term *meta-parameter* is used to refer to a parameter that is not a detailed weight or bias value but something else such as network layer size, the learning (gradient-descent) rate during training.

Some of these parameters are defined in the network configuration file as previously described. Other parameters affect the training process and are made tunable when *cnn* is executed. The *cnn* program has several command-line options to control whether training is done, and whether to constrain the tunable values like weight and bias, or how to constrain the calculations in various ways.

```
 ...
 ===========================================================
 Layer[2] type: hidden   # fully connected
 Layer[2] shape: [1 16]  # [ rows columns ]
 Layer[2] weights shape: [196 16]  # [ prev_layer_units
 #                                   this_layer_units ]
 Layer[2] weight [0]:
 [ 0.03851341 -0.02019609 -0.00180569  0.00915824  0.06892495  0.01320399
  -0.00319309  0.01246490  0.01465623  0.00000352 -0.00000618  0.05062256
   0.04643634  0.02970700 -0.00000568  0.04300897]
 Layer[2] weight [1]:
 [ 0.03849308 -0.02020839 -0.00180264  0.00915570  0.06892211  0.01321017
  -0.00318822  0.01246519  0.01465665  0.00001089  0.00001186  0.05062851
   0.04642271  0.02968511  0.00000058  0.04301001]
 ...


 Layer[2] weight [195]:
[ 0.03852068 -0.02019071 -0.00179249  0.00916033  0.06892192  0.01321872
 -0.00318490  0.01247167  0.01466084  0.00002307  0.00000062  0.05062604
  0.04641144  0.02969721  0.00000852  0.04301187]


Layer[2] bias values:
array([-0.01247983  0.08926019  0.12111333 -0.04132422 -0.30204476  0.07710016
  0.13315082 -0.07699171 -0.07780880 -0.38037932  0.09788831  0.14122878
 -0.17630519 -0.21480234 -0.02220427 -0.64331198])
...
=============================================================
 End.
```

Figure A2. Example of network weights and biases values for cnn application.

Additional command-line options control the training process, such as how long the process should proceed before stopping, how rapidly to try to descend the gradient decent slope, or to generate special types of output. In detail, the command-line options are:

- *-p*: Whether to print the network configuration before and after the training process (needed if one wants to capture training to a file).
- *-d*: Print additional intermediate values useful in debugging.
- *-err*: For each wrong classification result, print input image and classification info.
- *-thrift{N}:* Keep only the N largest-magnitude weights for each neuron (others are zeroed out) after loading a network configuration and before performance evaluation and optional training, and again after each optional training batch). N defaults to 32 if not specified.
- *-pos*: Whether to scale test inputs to non-negative values and assume non-negative only when printing graphical representations of layer activation levels.
- *-q{n}*: Whether to quantize weights, biases, and excitation/activation signal values so that it can be represented by n fractional bits. Fractional step size is 2^{-n}, and a real-value can be scaled up to an integer-equivalent by a factor of 2^n. Defaults n to 4 if not specified.
- *-m*: Whether to limit weights to $-1 + 2^{-n}$ to $1 - 2^{-n}$ and biases to -2 to $2 - 2^{-n+1}$. Assumes quantization level set by *-q* or a default value of 1/16 quantization steps (4 fractional bits).
- *-bininp{I}*: Down-sample 2×2 patches of pixels and threshold the input pixel values at value of I to produce a binary image. 'I' should be an integer 0 to 255, or defaults to 200.
- *-weightbins*: Print a tabular histogram of the distribution of network weights, 0.05 per bin.
- *-im{N}*: Print a picture of the specified image number (0 to 59999 for the MNIST data), and a binary-thresholded version, according to the default threshold or the threshold set with the *-bininp{I]* option. N defaults to 0 if not provided.
- *-one*: Test only the image specified in the *-imN* option, print the expected (labeled) result and actual network-classified result, then quit.

- *-train*: Perform training. The best-performing overall weight and bias set from each training and evaluation batch is remembered then used during the network configuration printing if enabled by the *-p* option. Print summary metrics of the training and evaluation at each batch.
- *-bN*: Train up to N batches, each of which has 500 trials of an image selected randomly from the 50000 training examples in the MNIST data set, followed by back-propagation. The batch of training is followed by a full 10000 image performance assessment. N must be specified, but if *-b* is not used, the default batch limit is 5000.
- *-etaF*: Set the training rate eta to the floating-point value F (0.005 by default). *Eta* is how much of the full back-propagation error correction to apply to each weight after each training image is presented, and can be thought of as how 'fast' to descend the local error surface gradient.
- *-etafactorF*: How much of *eta* to keep after each training batch once the overall performance is at least 80% correct. *F* should be a factor between 0.0 and 1.0. The default *Eta* factor is 0.999.
- *-weightdecayF*: At each training step where the root mean square (RMS) of weights are >0.25, weights will be decayed by this much (multiplicative factor of 1.0 −F). *F* can be thought of fractional loss at each training step, and it is 0.004 (0.4%) by default, and is intended to keep weights relatively small and bounded through the training period.
- *-vhdl*: Generate a VHDL top-level circuit with the proper layers of SHiNe neurons, connected as per the thrifting limit, with connections weights and biased according to the trained parameters. The current implementation is limited to two active layers after both an input and down-sampling (max-pool) layer.
- *-fxp_vhdl*: Generate a VHDL top-level circuit with the correct layers of fixed-point multiply-and accumulate neurons, connected according to the thrifting limit (e.g., fan-in of 32), with connections weighted and biased according to the trained parameters.
- *-xray{I},{J}*: Print details about how each connection is contributing to unit J's activation in layer I. Note: only fully-connected or soft-max layers are supported.

Appendix A.2. Compile-Time Options

The code was initially designed to work with a generic *num_t* numeric type defined at compile-time, e.g., double-precision (64-bit) float (DFLOAT_MATH), single-precision (32-bit) float (SFLOAT_MATH), or a specific 16-bit fixed-point (FP_16_MATH). A custom 'integer' configuration (INT_MATH) was defined that implements the neuron calculations with repeated counting and otherwise mimics the planned SHiNe circuit design for evaluation of network performance when operating in an integer-based mode.

Note that the back-propagation training process using gradient-descent assumes floating point math (double preferred) and will not work properly with fixed-point or integer math compile configurations. The compile time option determines which version of the helper functions such as *to_num( )*, *to_double( )*, *nmul( )*, and *nadd( )* gets compiled. These functions allow a large number of other functions such as the ones used for network feed-forward to be written generically.

References

1. Chen, Y.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid State Circuits* **2017**, *52*, 127–138. [CrossRef]
2. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; ACM: New York, NY, USA, 2015; pp. 161–170.
3. Chakradhar, S.; Sankaradas, M.; Jakkula, V.; Cadambi, S. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Comput. Archit. News* **2010**, *38*, 247–257. [CrossRef]

4. Hardieck, M.; Kumm, M.; Möller, K.; Zipf, P. Reconfigurable Convolutional Kernels for Neural Networks on FPGAs. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; ACM: New York, NY, USA, 2019; pp. 43–52.
5. Markidis, S.; Chien, S.; Laure, E.; Pong, I.; Vetter, J.S. NVIDIA Tensor Core Programmability, Performance & Precision. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, Vancouver, BC, Canada, 21–25 May 2018.
6. Misra, J.; Saha, I. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing* **2010**, *74*, 239–255. [CrossRef]
7. Renteria-Cedano, J.; Rivera, J.; Sandoval-Ibarra, F.; Ortega-Cisneros, S.; Loo-Yau, R. SoC Design Based on a FPGA for a Configurable Neural Network Trained by Means of an EKF. *Electronics* **2019**, *8*, 761. [CrossRef]
8. Nurvitadhi, E.; Venkatesh, G.; Sim, J.; Marr, D.; Huang, R.; Hock, J.O.G.; Liew, Y.T.; Srivatsan, K.; Moss, D.; Subhaschandra, S.; et al. Can FPGAs beat GPUs in accelerating next-generation Deep Neural Networks? In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; ACM: New York, NY, USA, 2017; pp. 5–14.
9. Gomperts, A.; Ukil, A.; Zurfluh, F. Development and Implementation of Parameterized FPGA-Based General-Purpose Neural Networks for Online Applications. *IEEE Trans. Ind. Inform.* **2011**, *7*, 78–89. [CrossRef]
10. Himavathi, S.; Anitha, D.; Muthuramalingam, A. Feedforward Neural Network Implementation in FPGA using layer multiplexing for effective resource utilization. *IEEE Trans. Neural Netw.* **2007**, *18*, 880–888. [CrossRef] [PubMed]
11. Tavanaei, A.; Ghodrati, M.; Kheradpisheh, S.R.; Masquelier, T.; Maida, A.S. Deep Learning in Spiking Neural Networks. *Neural Netw.* **2019**, *111*, 47–63. [CrossRef] [PubMed]
12. Iakymchuk, T.; Rosado, A.; Frances, J.V.; Batallre, M. Fast Spiking Neural Network Architecture for low-cost FPGA devices. In Proceedings of the 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), York, UK, 9–11 July 2012.
13. Rice, K.; Bhuiyan, M.A.; Taha, T.M.; Vutsinas, C.N.; Smith, M. FPGA Implementation of Izhikevich Spiking Neural Networks for Character Recognition. In Proceedings of the 2019 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 9–11 December 2009.
14. Pearson, M.J.; Pipe, A.G.; Mitchinson, B.; Gurney, K.; Melhuish, C.; Gilhespy, I.; Nibouche, N. Implementing Spiking Neural Networks for Real-Time Signal Processing and Control Applications. *IEEE Trans. Neural Netw.* **2007**, *18*, 1472–1487. [CrossRef] [PubMed]
15. Belyaev, M.; Velichko, A. A Spiking Neural Network Based on the Model of VO_2-Neuron. *Electronics* **2019**, *8*, 1065. [CrossRef]
16. Arbib, M.A. *The Handbook of Brain Theory and Neural Networks*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2002.
17. Nielsen, M.A. *Neural Networks and Deep Learning*; Determination Press: San Francisco, CA, USA, 2015.
18. Minsky, M.L.; Papert, S.A. *Perceptrons: An Introduction to Computational Geometry*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2017.
19. Glorot, X.; Bordes, A.; Bengio, Y. Deep sparse rectifier neural networks. In Proceedings of the 14th International Conference on Artificial Intelligence and Statistics, Ft. Lauderdale, FL, USA, 11–13 April 2011; pp. 315–323.
20. Deng, L. The MNIST database of handwritten digit images for machine learning research [best of web]. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
21. Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
22. Llamocca, D. Self-Reconfigurable Architectures for HEVC Forward and Inverse Transform. *J. Parallel Distrib. Comput.* **2017**, *109*, 178–192. [CrossRef]
23. Reagen, B.; Whatmough, P.; Adolf, R.; Rama, S.; Lee, H.; Lee, S.; Hernandez-Lobato, J.; Wei, G.; Brooks, D. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016.
24. Gokhale, V.; Jin, J.; Dundar, A.; Martini, B.; Culurciello, E. A 240 G-Ops/s mobile coprocessor for deep neural networks. In Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops, Columbus, OH, USA, 23–28 June 2014.

25. Farabet, C.; Martini, B.; Akselrod, P.; Talay, S.; LeCun, Y.; Culurciello, E. Hardware accelerated convolutional neural networks for synthetic vision systems. In Proceedings of the 2010 IEEE International Symposium on Circuits and Systems, Paris, France, 30 May–2 June 2010.
26. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. FINN: A framework for Fast, Scalable Binarized Neural Network Interface. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; ACM: New York, NY, USA, 2017; pp. 65–74.
27. Strigl, D.; Kofler, K.; Podlipnig, S. Performance and scalability of GPU-based convolutional neural networks. In Proceedings of the 2018 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, 17–19 February 2010.
28. Song, S.; Su, C.; Rountree, B.; Cameron, K.W. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Boston, MA, USA, 20–24 May 2013.
29. Hauswald, J.; Kang, Y.; Laurenzano, M.A.; Chen, Q.; Li, C.; Mudge, T.; Dreslinski, R.; Mars, J.; Tang, L. DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers. In Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 13–17 June 2015.

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).