You Only Spike Once: Improving Energy-Efficient Neuromorphic Inference to ANN-Level Accuracy

Srivatsa P*, Kyle Timothy Ng Chu*, Yaswanth Tavva*, Jibin Wu†,
Malu Zhang†, Haizhou Li† and Trevor E. Carlson*
*School of Computer Science †Department of Engineering
National University of Singapore

***Abstract*—In the past decade, advances in Artificial Neural Networks (ANNs) have allowed them to perform extremely well for a wide range of tasks. In fact, they have reached human parity when performing image recognition, for example. Unfortunately, the accuracy of these ANNs comes at the expense of a large number of cache and/or memory accesses and compute operations. Spiking Neural Networks (SNNs), a type of neuromorphic, or brain-inspired network, have recently gained significant interest as power-efficient alternatives to ANNs, because they are sparse, accessing very few weights, and typically only use addition operations instead of the more power-intensive multiply-and-accumulate (MAC) operations. The vast majority of neuromorphic hardware designs support rate-encoded SNNs, where the information is encoded in spike rates. Rate-encoded SNNs could be seen as inefficient as an encoding scheme because it involves the transmission of a large number of spikes. A more efficient encoding scheme, Time-To-First-Spike (TTFS) encoding, encodes information in the relative time of arrival of spikes. While TTFS-encoded SNNs are more efficient than rate-encoded SNNs, they have, up to now, performed poorly in terms of accuracy compared to previous methods.**

Hence, in this work, we aim to overcome the limitations of TTFS-encoded neuromorphic systems. To accomplish this, we propose: (1) a novel optimization algorithm for TTFS-encoded SNNs converted from ANNs and (2) a novel hardware accelerator for TTFS-encoded SNNs, with a scalable and low-power design.

Overall, our work in TTFS encoding and training improves the accuracy of SNNs to achieve state-of-the-art results on MNIST MLPs, while reducing power consumption by 1.29× over the state-of-the-art neuromorphic hardware.

I. Introduction

In recent years, Artificial Neural Networks (ANNs) have demonstrated excellent results in a wide range of real-world computational problems such as object detection, speech recognition and image classification. ANNs have been improving in accuracy, and in 2015, crossed an important threshold, beating human accuracy [1] at the ImageNet 2012 Visual Recognition Challenge [2]. However, the effectiveness of ANNs comes at the cost of high power consumption. In short, the performance of these networks rely on an extremely large number of model parameters, requiring a huge number of computational resources. This tends to make large ANNs unsuitable for low-powered applications such as Internet-of-Things (IoT) and mobile devices. To address this issue, there has been an increased focus on developing energy efficient networks, including EfficientNet [3], MobileNet [4] and SqueezeNet [5], to meet the efficiency needs of these low-power systems.

While these more efficient networks are promising for deployment to low powered devices, the use of spiking neural networks (SNNs) allows for even greater power savings. In SNNs, information is represented by binary events called spikes, similar to the way information is communicated in the human brain. This technique of mimicking brain functionality, called neuromorphic computing, makes use of only addition operations, instead of multiply-and-accumulate operations in standard ANNs, which has the capability to significantly reduce the computational power and complexity needed. Furthermore, SNNs can also take advantage of the sparsity of their neuron activations to significantly reduce the movement of data. As a result, SNNs have garnered significant interest over the last few years as a candidate for low-powered accelerators [6].

While there are several encoding methods for SNNs [7], the two most prominent ones are rate-based [8], [9] and temporal encoding [10], [11]. Because temporally encoded networks have not been able to match the state-of-the-art accuracy of rate-based coded networks [12], rate-based encoding has been the most common encoding scheme used in hardware SNN accelerators [13], [14]. In rate-based encoding, information is represented by the average number of spikes over a period of time, and the inference results become more accurate as additional spikes are generated. Because each spike triggers memory accesses (to load parameter information) which needs to be fetched from on- or off-chip memory, the power consumption in rate-encoded networks can be relatively high. Hence, some works have turned to temporal encoding instead to further take advantage of the sparsity of the networks [15]. One type of temporal encoding is known as time-to-first-spike (TTFS) encoding. Under this encoding, information is represented by the relative time of arrival of the spikes with respect to the first spike, not the average number of spikes over a time period. Previously, TTFS encoding had traded off power efficiency for accuracy, unable to match the results of rate-encoded systems. In this work, we propose a training method to leverage the power efficiency TTFS encoded SNNs with minimal loss to accuracy.

Among existing rate-based accelerators, IBM's TrueNorth is the most well known. With the ability to hold up to 1 million neurons and 256 million synapses, and can consume 65mW when running typical computer vision applications [16], TrueNorth still remains the state-of-the-start in terms of power efficiency. However, while TrueNorth is extremely power efficient at implementing rate-based networks, it is unable to take full advantage of the properties of temporally encoded networks. One of the main reasons behind TrueNorth's impressively low power usage is that the number of memory accesses on each core does not scale with the number of spikes it receives and remains constant. While having a constant number of memory accesses per tick works well for rate-based networks which produce large numbers of spikes, it prevents TrueNorth from maximizing efficiency of temporally encoded networks with sparse spikes. Thus, this work introduced the You Only Spike Once, or YOSO, accelerator, a novel architecture specifically designed to leverage the sparsity in the spiking behavior of temporally coded networks.

One of the largest barriers preventing the widespread adoption of SNNs is the fact that SNNs are notoriously difficult to train from scratch. While significant progress has been made in recent years [17], their results are still far behind what has been achieved by state-of-the-art ANNs today for classification problems. To circumvent this problem, we instead chose to convert pre-trained ANN models into SNNs before mapping them onto the proposed hardware. Previous work [18] has introduced a technique that can convert ANNs to SNNs with minimal loss in accuracy for small networks. For larger and deeper networks, this approach does not work as well. Hence we have developed a novel training method that improves the accuracy of the converted SNNs through this technique.

The main goal of this paper is to run highly accurate networks on low power devices, with minimal loss to accuracy. Therefore, this work focuses on (1) optimizations to improve the accuracy of TTFS-encoded SNNs, and (2) optimizations to the hardware running the network. Hence, the contributions of this work are:

- An end-to-end neuromporhic technique that demonstrates state of the art performance and accuracy for TTFS-encoded SNNs
- A new training algorithm that reduces the approximation error which accumulates as a result of converting ANNs to SNNs. In doing so, our TTFS-encoded SNNs reach near ANN-accuracy (within 0.2%), allowing TTFS-encoded SNNs to be considered for traditional ANN tasks, at much higher efficiency.
- An implementation of a novel hardware accelerator for TTFS-encoded SNNs that is configurable and scalable. Our work significantly improves power efficiency with minimal reduction in the accuracy of network performance.

II. Background

Spiking neural networks (SNNs) have garnered significant interest over the last few years, primarily as a candidate for energy-efficient inference on low-powered devices. In SNNs, information is represented by discrete binary events called spikes, similar to the way the brain represents information. This is unlike a standard artificial neural network (ANN) where information is represented as continuous values [19]. The use of SNNs reduces the computational power needed by only requiring cheaper addition operations compared to the more power-intensive multiply-and-accumulate (MAC) operations used in ANNs. Furthermore, SNNs typically have an extremely low activation count, in comparison to their non-spiking counterparts. Activations are low because every neuron is only activated by a strictly positive input, a subset of all possible inputs, above a pre-defined threshold. This translates to just a small subset of all neurons firing for any given inference. A small subset of neurons firing translates into a low memory access count and, hence, a low cost when performing inference tasks.

SNNs are fundamentally different from ANNs. ANNs traditionally involve a synchronous tensor multiplication for each layer while SNNs involve an asynchronous propagation of information. The way information can be propagated through an SNN can vary. The two most prominent methods are rate-based [8], [9] and temporal encoding [10], [11]. In rate-based encoding, information is represented by the mean firing rate of the neurons. Although there exist different definitions of firing rate, it often denotes either spikes averaged over repetitions of an experiment or the average number of spikes over a period of time. This work refers to the latter when referring to rate-encoded networks. Rate-encoded networks become more accurate over time as more spikes are generated. From a power consumption point of view, each spike will require a weight look-up. Because rate-based encoding has many spikes, having a weight look-up for each spike limits the minimum number of memory accesses and the corresponding amount of energy saved.

An alternative form of encoding uses temporal encoding which is based on spike timing [15]. Common temporal encoding schemes include Time to First Spike (TTFS), where information is represented by the relative time of arrival of the spikes with respect to the first spike, and phase-of-firing, where information is encoded using the time at which neurons fire within a periodic cycle [20]. When information is encoded in the TTFS scheme, neurons in an SNN spike at most once during each inference pass and see many fewer spikes compared to their rate-based counterparts. By definition, the rate encoding scheme relies on the generation of multiple spikes over a fixed period of time, while the TTFS encoding scheme relies on the time taken for a single neuron to spike. Therefore, TTFS encoding scheme allows for fewer spikes compared to a rate-based encoding scheme. Assuming a spike corresponds to a memory access, TTFS encoding scheme allows for a low number of memory accesses. In addition, an inference pass of a TTFS network can end once the output layer produces its first output spike instead of waiting for the rest of the inputs to arrive. As a result, a minimal number of computations are performed for any particular inference, making temporal encoding a highly suitable candidate for encoding energy-efficient SNNs.

A. Formalizing TTFS-SNNs

SNNs have been proposed to model the biological neural network of brains that use spikes to represent and communicate information across neurons [9]. As the fundamental information processing units in the biological neural networks, the spiking neurons are composed of dendrite, soma, and axon. Dendrites receive weighted inputs from the preceding neurons, which are further integrated into the membrane potential of the soma. An output spike is generated from the soma once the membrane potential crosses the firing threshold. The output spike is then transmitted to the subsequent neurons through the axonal connections. A number of spiking neuron models have been proposed to describe the internal dynamics and diversified characteristics of biological neurons.

In this work, to properly encode information into spike timings, we use a non-leaky IF neuron model [18]. The membrane potential dynamics of this model can be described by the following equation:

$$\frac{dV_{mem}^{i}(t)}{dt} = \sum_{j} w_{ij} \sum_{n} \kappa_{ij}(t - t_{j}^{n}) + b_{i}t \tag{1}$$

where V_{mem}^{i} is the membrane potential of neuron i, and w_{ij} is the weight of the synaptic connection from the pre-synaptic neuron j to the post-synaptic neuron i. t_j^n is the timing of the nth spike from the pre-synaptic neuron j. Since in TTFS-encoding we are concerned with the time associated with only the first spike ($n = 1$), hereon we ignore subsequent spikes and refer to the time of the first spike of neuron j using t_j. κ_{ij} is the kernel that describes the induced post-synaptic potential (PSP) by the incoming spikes and is defined as follows:

$$\kappa(t - t_j) = [t - t_j]\Theta(t - t_j) \tag{2}$$

where Θ is the heaviside step function. The heaviside step function Θ can be ignored by only considering input spikes that arrive before t_i for each neuron i. Because it is these input spikes that influence the output spike of each neuron i, we consider the pre-synaptic neurons that produce these input spikes as a set of causal neurons. The set of causal neurons $\Gamma_i^<$, can hence be defined as $\Gamma_i^< := \{j | t_j < t_i\}$. The time to first spike for the neuron i can be expressed as follows:

$$t_i = \frac{1}{\mu_i}\left(\theta + \sum_{j \in \Gamma_i^<} w_{ij} t_j\right) \tag{3}$$

where

$$\mu_i := \sum_{j \in \Gamma_i^<} w_{ij} + b_i. \tag{4}$$

As the instantaneous firing rate r_i of the neuron i is the inverse of t_i, the proposed ANN-to-SNN conversion method [18] equates activation a_i in an ANN to the instantaneous rate r_i of the corresponding neuron i in the converted SNN, assuming the use of ReLU activation functions in the ANN. The state of SNNs have changed over the years, and have shown significant progress.

III. Related Work

There are three key approaches to achieve power efficient neural network inference covered by this work. They include spiking neural networks, hardware accelerators and neural network optimizations.

A. Spiking Neural Networks

SNNs can be constructed by either training from scratch or converting from a pre-trained ANN. Although many spike-based training algorithms [21], [22], [23] have shown promising results on the MNIST [18] dataset, these algorithms have not been tested rigorously on larger network architectures and more challenging datasets. As an alternative to SNN-based training, pre-trained ANNs can be converted into SNNs. This method has been shown to be highly successful for rate-encoded SNNs on the CIFAR-10 and ImageNet [24] datasets. These previous works have focused on rate-encoded SNNs, where a large number of synaptic operations are required As a result, rate-based encoding greatly limits the power efficiency of SNN models when deployed onto the neuromorphic hardware.

Compared to rate-encoded SNNs, temporally encoded SNNs are able to run with fewer operations and memory accesses [15] which are highly desirable attributes for low-powered devices (details in Section II). Although significant power savings can be achieved by using TTFS-encoded SNNs instead of rate-encoded SNNs, TTFS-encoded SNNs that are constructed by either training from scratch [15], [17], [25], [26] or converting from the pre-trained SNN [18] did not perform as well as their ANN counterparts in terms of the classification accuracy. As demonstrated in a recent study [18], converting from ANNs to TTFS-encoded SNNs, unfortunately, leads to accumulated approximation errors, which results in significantly lower accuracy in the SNNs as compared to the equivalent ANNs, particularly in larger network architectures. Our work tackles this problem by proposing a novel training approach to refine the network weights after conversion, which improve the performance of the converted TTFS-encoded SNNs.

B. Hardware Accelerators

In the past few years, there have been several architectures that have been proposed for neuromorphic hardware. The most prominent among them is IBM's TrueNorth neuromorphic chip [16], which has an extremely low power density of just 20 milliwatts per square centimeter and has shown results equivalent to state of the art on several benchmarks. However, TrueNorth does not take advantage of sparse activations to reduce the number of memory accesses. Regardless of the number of spikes, cores on the TrueNorth chip will always perform a read for each neuron in their core SRAM. Such an implementation can be attributed to the usage of non-standard networks for inference, making no assumptions about the networks that are run. As a result, TrueNorth needs to handle connections on a per-neuron basis, requiring the hardware to keep track of the connections of every single neuron. There is a high overhead for such an implementation, where half of the data used during read operations (256 out of 410 bits) on TrueNorth are to check for connectivity.

Unlike TrueNorth, our work implements standard networks with an exploitable access pattern. Hence, we are able to make reasonable assumptions about neuron connectivity and can express these connections as a layer-wise access pattern instead of storing them individually for each neuron. As a result, we are able to perform significantly fewer reads per time step

In the space of Time-To-First-Spike (TTFS) based hardware accelerators, the viability of Time-To-First-Spike based systems have been demonstrated by examining the sparsity of TTFS-encoded networks [15]. However, the work did not explore the potential for a low powered accelerator, as the authors have not leveraged the sparsity of activations in a TTFS-encoded SNN in a significant way. Furthermore, the accuracy reported was far below the state-of-the-art.

Other notable SNN architectures include Intel's Loihi [13] and Minitaur [14], which feature online training, and a processor for Deep Neural Networks with Binary/Ternary Weights in 28nm CMOS [27].

C. Quantization

Quantization is one technique that is often applied to SNNs, with layers having 4- or 8-bit precision [28]. In this work, we examine the effects of 8-bit quantization on efficiency and accuracy.

D. Summary

Power efficient inference can be achieved through the use of extremely sparse SNNs. While most works use rate-encoded

SNNs instead of TTFS-encoded SNNs, trading accuracy for power efficiency, we show that is possible to achieve greater power efficiency for comparable accuracy, through the use of TTFS-encoded SNNs.

In this work, we demonstrate a novel hardware accelerator specifically designed for TTFS-encoded SNNs. Along with the energy-efficient accelerator, we propose a method to convert ANNs to SNNs which allows our TTFS-encoded SNNs to leverage network compression techniques for more power savings. In this work, we propose the combination of enhanced training and efficient hardware to demonstrate the potential of TTFS-based platforms.

IV. Training Competitive TTFS-SNNs

TTFS-encoded SNNs have shown better power efficiency and inference speed as compared to their rate-based variants. The converted TTFS-encoded SNNs, however, suffer from quantization errors that accumulate across layers. This significantly deteriorates the classification accuracy, particularly in deeper SNNs, as compared to their equivalent ANNs. Another source of error arises when an input spike, coming from the synaptic connection with a large weight, drives a neuron's internal membrane potential across the firing threshold, before subsequent inhibitory input spikes that targeting the same post-synaptic neuron arrives. This problem can be explained by the different operating mechanism of the spiking neuron and artificial neuron, wherein the input information is distributed and integrated over time by the SNN rather than at the same time instant as happened in the ANNs. Raising the threshold value of the post-synaptic neurons may alleviate this problem. However, it is not a good option in practice since it adversely increases the latency for decision making.

To address these problems, we propose a training method to systematically convert pre-trained ANNs to the TTFS-encoded SNNs. First, we apply a data-driven weight normalization strategy such that the neuron activation is not dominated by a few input spikes with large weights while also ensure timely decision making. Finally, to mitigate conversion errors, we propose a layerwise training methodology. As a whole, the proposed training framework effectively closes the accuracy gap between the pre-trained ANNs and the converted SNNs.

A. Firing Threshold Determination

Determining the right combination of neuronal firing threshold, weight and bias values is crucial to striking a balance between the classification accuracy and latency. Apart from the learnable parameters (weights and biases) that can be directly taken from the pre-trained ANNs, the firing threshold requires extra effort to be determined. An inappropriate threshold value will cause the converted SNN to perform significantly poorer compared to the equivalent ANN. One common approach to this problem would be to set the threshold to 1 and adjust the weights such that the activations are normalized.

B. Weight Normalization

In order to prevent the converted SNNs from underestimating output activation of the corresponding ANNs, this work applies weight normalization. One way to normalize weights is to consider all possible combinations of positive activations that could occur at a particular ANN layer and scale the weights by that maximum quantity. The benefit of such an approach is that it only depends on the weights and biases of the network. However, in reality the maximum activation that determined in this way might be far from the actual activation values for majority of neurons. This leads to weights and biases that are much smaller than they need to be, increasing the time taken for a neuron to get activated. Because the time taken for a neuron to first spike increases, a longer duration will be required to achieve high classification accuracy. This problem will be exacerbated in deeper networks if weights are normalized in this way for all layers.

Instead of this conservative approach, we estimate the maximal activation values of an ANN by making use of the training data [29]. Note that because this algorithm uses data from the training set, a strong performance guarantee cannot be extended to the test set. As long as the training and test sets have a similar data distribution, the activation vectors observed using the training set would be similar to that observed in the test set. The benefit of this method over the former method is that it provides a much better trade off between latency and accuracy. This is because the time taken to spike is shorter to achieve a similar accuracy.

C. Training Network

Errors arising from converting ANNs to SNNs can be further reduced through (1) retraining an ANN with constraints or (2) refining the learnable parameters on the converted temporally-encoded SNN. While retraining a standard ANN with constraints might be feasible for small tasks such as the MNIST dataset, it might be extremely challenging to do so with larger networks on larger tasks such as ImageNet.

We propose coupling each layer in an ANN and the corresponding layer in the converted SNN, and minimzing a layer wise cost function. Unlike traditional SNN training algorithms which utilize a loss computed at the final layer, the algorithm we are proposing is aimed at minimizing the divergence between ANN activations a_{Li} and SNN activations s_{Li} for every neuron with index i in a layer L.

From the ANN-SNN conversion, the analog activation of a neuron in the ANN is equivalent to the instantaneous firing rate of TTFS-encoded SNN. The instantaneous firing rate is given by the inverse of the time taken for a neuron to first spike. It is possible to model the approximation between the activation of a single neuron neuron i in a particular layer l in an ANN and the corresponding neuron in an SNN: $a_i^l = \frac{1}{t_i^l} + \varepsilon$ where the introduction of ε allows for activation between SNN and ANN to deviate by a reasonable margin of error. A potential loss function is the L2-norm, given by $L = \frac{1}{2} * (a_i^l - r_i^l)^2$ where r_{Li} is the instantaneous firing rate of neuron i in layer l given by $r_i^l = \frac{1}{t_i^l}$.

The loss function is minimized by updating synaptic weights as described in Algorithm 1. For each layer, L, the divergence between the ANN activation vector and SNN instantaneous rate vector is computed and minimized. This has the effect of delaying or advancing spike times in the network. In Section VIII, we demonstrate how this improve training method works to increase inference accuracy.

Algorithm 1: *train_network*: SNN training

```
Input: $\{I^1 ... I^n\}$: Set of $n$ input spikes vectors generated
  from randomly sampling $n$ images from training set
Input: $\beta$: Fraction of neurons to keep in each layer
Input: $\eta$: Learning rate
Input: $\varepsilon$: Margin of error
Input: $K$: Number of iterations
Output: Finetuned weights vector $\mathbf{w}$
$\mathbf{w_i} \leftarrow$ get_parameters_from_ann($\beta$);
$\mathbf{w_n} \leftarrow$ normalize_weights($\mathbf{w_i}$, $\{I^1 ... I^n\}$);
k = 0
while $k < K$ and error $> \varepsilon$ do
    for $I^r$ in $\{I^1 ... I^n\}$ do
        // Get activation vectors for each layer in an
          L-layered ann
        $\{\mathbf{a^1} ... \mathbf{a^L}\} \leftarrow$ ann_forward_pass($I^r$);
        // Get vectors of spike times for each layer in an
          L-layered snn
        $\{\mathbf{t^1} ... \mathbf{t^L}\} \leftarrow$ snn_forward_pass($I^r$);
        // Get vectors of instantaneous spike rates for
          each layer in an L-layered snn $\{\mathbf{r^1} ... \mathbf{r^L}\} \leftarrow$
          get_spike_rates($\{\mathbf{t^1} ... \mathbf{t^L}\}$);
        for q=1 to L do
            layer_type $\leftarrow$ get_layer_from_index(q);
            if layer_type in {batch norm, dropout} then
                skip
            else
                error = $L_2(\mathbf{a^q}, r^q)$;
            // update weight
            $\mathbf{w_n}$ -= $\eta * \frac{\partial L}{\partial w} * error$;
    k+=1
```

V. Architecture Description

A. Abstract Hardware Model

This section describes an abstract hardware model which we used to translate the mathematical SNN models described in earlier sections into a model that is easier to translate into actual physical hardware. The abstract model consists of a computational block that loads from and writes to three storage blocks and represents a single layer of neurons i. The hardware unit receives two forms of inputs - (1) incoming spikes from neurons j in the previous layer and (2) End-of-Timestep (EoT) packets used to signal the unit to move onto the next timestep $t+1$.

The Weight block stores the weights of the synaptic connections w_{ij} while the Accumulated Weights block keeps track of the gradient of the neuron potential $\frac{dV^i_{mem}(t)}{dt}$ in equation 1. The Neuron Potentials block stores the neuron potential $V^i mem(t)$ as well as information on whether the neuron has already produced a spike.

The Computational Block contains a set of registers that are used to either keep track of the current state of layer or are used to generate various access patterns of different network types.

1) Processing input spikes: The core implements a spike processing algorithm which uses four registers to implement an access pattern. One register is used to keep track of the address to be accessed while another will be used to keep track of the number of accesses made. The remaining two registers are populated on program time and store the number of accesses P that need to be made and the address increment after each access M. When a spike arrives at the core from neuron j, the address register will be set based on the index of j. The core will then generate P memory accesses, incriminating the address by M after each access. Using this algorithm, our core is able to implement fully connected networks efficiently, as unnecessary computations and memory accesses will be skipped over by the access pattern.

2) Processing End-of-Timestep (EoT) Signals: In SNNs, time is used as an additional mechanism to store information. In YOSO, EoT signals are used to indicate that a timestep is complete and the accelerator can move on to the next one. Unlike the processing of input spikes, no additional information needs to be decoded from the EoT signals. The Computational Block supports two ways of handling EoT signals - the standard Integrate-and-Fire method and the softmax method which is normally used in the final layer of the networks.

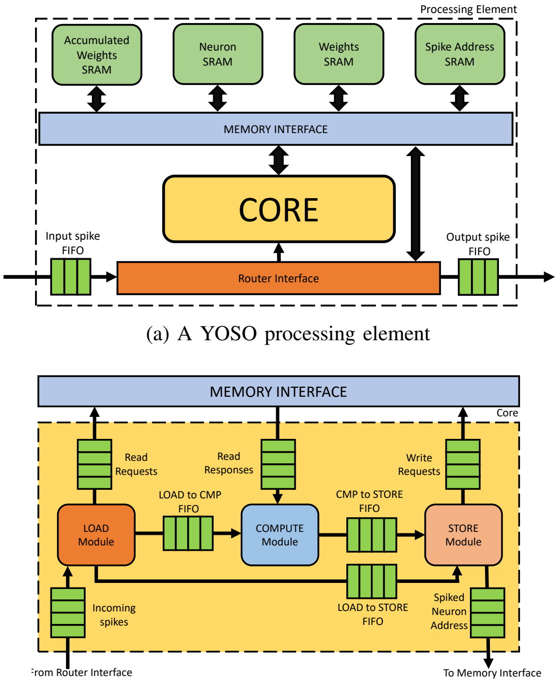
In the standard Integrate-and-Fire method, spikes are generated as long as the neuron potential crosses the threshold and the neuron has not spiked before. However, in the softmax method, only the neuron with the largest neuron potential produces a spike. In both methods, an EoT signal is sent to the next layer after the Computational Block has finished updating the neuron potentials and generating spikes.

B. Architecture Description

In this section, we discuss the detailed design of the YOSO accelerator. The accelerator consists of multiple Processing Elements (PEs) that are connected together through a Network-on-Chip (NoC) with each PE supporting up to 256 neurons. In this work we build on the OpenSMART NoC architecture [30] to implement a lightweight NoC that utilizes x-y routing to send spike packets from one PE to another. Figure 1a shows the layout of the different components of a PE. Under normal operations, the core will update the accumulated weights and the neuron potentials by accessing and updating the appropriate data from the SRAM banks attached to the memory interface.

1) Router Interface: The router interface is responsible for sending input spikes to the appropriate components depending on the current state of the PE. When the PE is in programming mode, incoming spikes may be sent to the memory interface to set the initial values of the SRAM blocks while all spikes will be directed to the core under normal operation.

The router interface also supports two methods of generating output packets that are sent to the router. First, the router may take the output spike from the Spike Address Storage – when a spike is generated by the neuron core, a 8-bit neuron address is sent to the Spike Address Storage SRAM which sends the 32-bit spike to the router interface. The router interface then appends the 8-bit coordinates of the core *output destination* and sends the 40-bit packet to the router. The second method of output packet generation is by forwarding received spikes. Spike forwarding allows a single layer to be mapped across multiple cores without the need for the sender to keep track of the coordinates of all the cores in the layer. If forwarding is active, the router interface sends spikes received from the



(a) A YOSO processing element

(b) A detailed diagram of the core

Fig. 1. A processing element (a) and its core (b), the main components of the YOSO neuromorphic processor.

router to the core while also using it to create a 40-bit packet by appending the 8-bit coordinates of the core *forwarding destination* which is then sent back to the router.

2) Memory Interface: The memory interface consists of four individual SRAM interfaces – one for each of the SRAM blocks in the PE. As YOSO uses single-port SRAMs, only a single read/write request can be processed at one time. The SRAM interfaces alternate between servicing requests from the Write Request FIFO and the Read Request FIFO to ensure that all requests are processed in a reasonable amount of time. SRAM blocks whose write queues are connected to the router interface are populated on program time while SRAM blocks with write queues connected to the core will be updated during run time.

Handling RAW dependencies. To handle RAW dependencies, read requests must contain an additional bit which indicates if the read is done with the intention to alter the current value. Additionally, SRAM interfaces with RAW protection contain a special 256-bit RW Protection Register - 1 bit for each entry in the SRAM. When a read request with the intention to write occurs, the bit in the RW Protection Register indexed by the read address will be set to 1 and will only be set back to 0 after a write request to that same address is processed. Any subsequent reads (regardless of whether they intend to alter the current value) will be stalled until the matching write request is processed. Reads that do not have the intention to write will not cause the RW Protection Register bit to be set. Since the Weight SRAM is only written to during program time, its SRAM interface does not contain this mechanism as RAW dependencies will not occur during runtime.

3) Core: The core is the key computational element of the PE. The design of the core was inspired by traditional deep learning accelerators like VTA [31] and adopts a decoupled access-execute model [32] to memory access hide latency. Each core consists of 3 modules that communicate with each other through FIFOs. This allows for a better utilization of the cores' resources as the other modules can continue execution if one of them encounters a stall.

Load Module. The load module is a finite state machine with two states: an idle state and an active state. In the idle state, the load module waits for an input spike to arrive from the router interface. When an input spike arrives at the core, the information encoded in the spike is decoded and used to populate the initial values of the internal registers. If a register is not set by the decoded spike values, they are set by the values of reference registers which are written to during the programming stage. After the register values have been set, the load module sends the spike type along with the number of addresses to be generated to the compute module and transits to the active state. In the active state, the load module begins to generate read requests according to algorithm described in the previous section . The read requests are sent to different SRAM blocks depending on the spike type. The generated addresses are sent to both the memory interface and the store module. After all the necessary addresses are generated, the load module transits to the idle state and waits for the next input spike.

Compute Module. Like the load module, the compute module is a finite state machine with an idle state and an active state. In the idle state, the compute module waits for the load module to send the spike type and the number of addresses to be generated. The compute module initializes its registers with this data and transits to the active state.

In the active state, the compute module performs addition operations on the data retrieved from the memory interface in response to the requests generated by the load module. Saturating adders are used to handle overflow and underflows. The results are then sent to the store module along with the spike type. After all the data is processed, the compute module transits back to the idle state.

Store Module. The store module is responsible for checking for spikes and storing updated values back to the appropriate SRAM block. The store module takes the values from the compute module and stores them in the address obtained from the load module. Since there is no re-ordering of read requests, we can ensure that the values from the compute module and the store module correspond to one another. If the spike is an EoT signal, the store module will first check if the neuron potential crosses the threshold before storing the value. The store module supports spike generation for both TTFS and softmax layers. In order to support softmax, reserved spike types are used to mark the first and last neuron in the layer. The store module then picks the neuron highest neuron potential within the range of neurons marked out by the two neurons and sends out the spike associated with the neuron, regardless of whether it has crossed the threshold.

VI. Mapping

This section, we explain how SNNs are mapped to the YOSO accelerator. In order to map a $m \times n$ fully connected layer, $C = MAX(\frac{n}{N}, \frac{m \times n}{W})$ PEs are needed where N is the maximum number of neurons that can be mapped to a single core and W

is the maximum number of weights that the core can contain. The PEs are placed within a $\sqrt{C}$ by $\sqrt{C}$ grid.

Parameter Mapping. Mapping a FCN to a YOSO core is a straightforward process. Biases are stored in the neuron SRAM as the initial value of the neuron potentials.

VII. Experimental Methodology

In this section, we outline the details of the experimental setup and algorithms used in evaluating our works.

A. Input & Output representations

Benchmarking SNNs require input data to be encoded as spike trains. For visual datasets, possible techniques include using: (1) Event-based sensors - creating a dataset using event-based cameras to generate spike trains (2) Stochastic methods - conversion of image intensity of images from conventional datasets into Poisson/Bernoulli spike trains (3) Intensity to Latency (ItL) encoding - generating a spike train containing a spike per pixel in the image, where the spike's latency is inversely proportional to the intensity of a pixel in an image. The use of stochastic methods could potentially be useful in improving generalizability of networks at the cost of added training stage complexity. Hence, this work uses the simpler ItL encoding scheme for data from conventional image classification datasets used for ANNs. Although not difficult, the extension of this work to integrate input data from event-based sensors is a potential avenue for future work. The output layer of our SNN implementation is one-hot encoded. The categorical classes of each dataset are encoded such that the index of the neuron that spikes corresponds to one of the output classes.

B. Networks

The MNIST Handwritten Digits dataset [33] contains grey-scale images of 10 handwritten digits of size 28 x 28, with a total training set of 60,000 examples, and a test set of 10,000 examples. We built a fully-connected network with three layers (300-300-10) with hidden layers containing 300 neurons in the hidden layer.

C. Hardware Simulation

The YOSO accelerator was synthesized using Synopsys Design Compiler version P-2019.03-SP5 targeting a 22nm technology node. Gate-level simulation was performed using Synopsys VCS-MX K-2015.09-SP2-9 and power analysis was performed with Synopsys PrimePower version P-2019.03-SP5. The simulations were run at 120KHz and accounts for both programming and inference time.

VIII. Results and Analysis

A. Performance

Prior TTFS-encoding work [18] has shown an accuracy of 98.30% (without quantization) on the MNIST dataset. Our work, in contrast, achieves 98.44% (without quantization) and 98.40% (with quantization). Our proposed method improves the accuracy of fully connected TTFS-encoded SNNs on the MNIST dataset. The chosen input parameters to Algorithm 1 were: (1) $|\{I^1...I^n\}| = 100$ (2) $\beta = 0.99$ (3) $\eta = 10$ (4) $\varepsilon = 0.001$ (5) $K = 100$.

TABLE I
Performance on MNIST Dataset (TTFS-encoding)

Network	Coding	ANN acc(%)	SNN acc(%)
TrueNorth [34]	Rate	-	99.42
Rueckauer et al [24]	Rate	98.56	98.50
Mostafa [25]	Temporal	-	97.55
Comsa et al [17]	Temporal	-	97.96
Rueckauer et al [18]	Temporal	98.56	98.30
YOSO (Our Work)	Temporal	98.56	**98.44**
YOSO (Our Work) + Quantization	Temporal	98.56	**98.40**

Although the accuracy achieved by our TTFS-encoded networks is slightly lower than that of rate-based networks, the power consumed per inference is significantly lower as shown in Table II. Our work pushes the performance and efficiency boundary through the use of TTFS-encoded SNNs. While we demonstrate results on fully connected networks, our future work includes evaluating the performance gains obtained on larger networks and datasets [2], [35].

TABLE II
Comparison of our work with general neuromorphic accelerators on the MNIST dataset sorted by accuracy. Our work demonstrates both low power and high accuracy. Enc. is encoding, Acc. is Top-1 accuracy in percent, fps is frames per second, Tech is in nm, power in mW.

Accelerator	Enc.	Acc.	fps	Tech	Power	uJ/frame
SNNwt [36]	Rate	91.82	-	65	-	214.700
TrueNorth-a [34]	Rate	92.70	1000	28	0.268	0.268
Spinnaker [37]	Rate	95.01	77	130	300.000	3896.000
Tianji [38]	Rate	96.59	-	120	120.000	-
Shenjing [39]	Rate	96.11	40	28	1.260	38.000
YOSO (this work)	Temp.	98.40	30	22	**(0.978*) 0.836**	**(32.604*) 27.867**
TrueNorth-b [34]	Rate	99.42	1000	28	108.000	108.000

* Scaled for 28nm process (×1.17 for half a generation)

B. Choice of output layer

Typically, the output layer of an ANN is chosen to be a softmax layer because it ensures that the final layer's outputs are both normalized and strictly positive. For TTFS-encoded SNNs, the output neurons are one-hot encoded. One problem with such a method is if all neurons in the final layer receive negative inputs or inhibitory input comes later than another neuron spiking, either no neuron spikes or the incorrect neuron will spike.

One solution would be to perform softmax on the membrane potentials of the output layer neurons to determine the predicted output class, instead of choosing the neuron that spikes first. This method has allowed us to realize **1%** improvement in accuracy of the network. This technique can be used when there is no need to use a purely spiking neural network.

IX. Conclusion

In this work, we introduced the YOSO accelerator, and an improved Time-to-First-Spike training algorithm which demonstrates the viability of temporally-encoded SNNs for image classification tasks. To address the limitations of temporally-encoded SNNs, we proposed a novel training algorithm which achieves state of the art accuracy on temporally encoded SNNs. By combining this highly accurate temporal encoding method with our energy-efficient hardware design, YOSO, we demonstrate state-of-the-art temporal encoding results with high efficiency (1.17× better) and a lower power consumption (1.29× better) over other state-of-the-art designs with comparable accuracy.

X. Acknowledgements

We thank the anonymous reviewers for their comments and suggestions to improve this work. This research is supported by A*STAR under its RIE2020 IAF-ICP (Award I2001E0053). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the A*STAR.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," *arXiv:1502.01852 [cs.CV]*, Feb. 2015.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "ImageNet large scale visual recognition challenge," *arXiv:1409.0575 [cs.CV]*, Sep. 2014.

[3] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97, Jun. 2019, pp. 6105–6114.

[4] A. Howard, M. Sandler, G. Chu, L. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for MobileNetV3," *arXiv:1905.02244 [cs.CV]*, May 2019.

[5] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size." *arXiv:1602.07360 [cs.CV]*, Feb. 2016.

[6] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[7] F. Ponulak and A. Kasiński, "Introduction to spiking neural networks: Information processing, learning and applications," *Acta neurobiologiae experimentalis*, vol. 71, pp. 409–33, Jan. 2011.

[8] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, Jan. 2014.

[9] W. Gerstner, Wulfram, Kistler, and W. M., *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, Jan. 2002.

[10] J. Hopfield, "Pattern recognition computation using action potential timing for stimulus representation," *Nature*, vol. 376, pp. 33–36, Aug. 1995.

[11] S. Bohte, "The evidence for neural information processing with precise spike-times: A survey," *Nat. Comput.*, vol. 3, pp. 195–206, Jun. 2004.

[12] M. Pfeiffer and T. Pfeil, "Deep learning with spiking neurons: Opportunities and challenges," *Frontiers in Neuroscience*, vol. 12, p. 774, Oct. 2018.

[13] C.-K. Lin, A. Wild, G. Chinya, M. Davies, N. Srinivasa, D. Lavery, and H. Wang, "Programming spiking neural networks on Intel Loihi," *Computer*, vol. 51, no. 3, pp. 52–61, Mar. 2018.

[14] D. Neil and S. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 12, pp. 2621–2628, Dec. 2014.

[15] H. Mostafa, B. U. Pedroni, S. Sheik, and G. Cauwenberghs, "Fast classification using sparsely active spiking networks," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4.

[16] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "TrueNorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," vol. 34, no. 10, Oct. 2015, pp. 1537–1557.

[17] I. M. Comsa, T. Fischbacher, K. Potempa, A. Gesmundo, L. Versari, and J. Alakuijala, "Temporal coding in spiking neural networks with alpha synaptic function," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020, pp. 8529–8533.

[18] B. Rueckauer and S.-C. Liu, "Conversion of analog to spiking neural networks using sparse temporal coding," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[19] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. Maguire, and T. McGinnity, "A review of learning in biologically plausible spiking neural networks," *Neural Networks*, vol. 122, pp. 253–272, Feb. 2020.

[20] A. Cattani, G. T. Einevoll, and S. Panzeri, "Phase-of-firing code," *arXiv:1504.03954 [q-bio.NC]*, Apr. 2015.

[21] S. B. Shrestha and G. Orchard, "Slayer: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems 31*, 2018, pp. 1412–1421.

[22] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: faster, larger, better," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1311–1318.

[23] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks," *arXiv:1901.09948 [cs.NE]*, Jan. 2019.

[24] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu, "Conversion of continuous-valued deep networks to efficient event-driven networks for image classification," *Frontiers in Neuroscience*, vol. 11, p. 682, Dec. 2017.

[25] H. Mostafa, "Supervised learning based on temporal coding in spiking neural networks," *arXiv:1606.08165 [cs.NE]*, Jun. 2016.

[26] M. Zhang, J. Wang, Z. Zhang, A. Belatreche, J. Wu, Y. Chua, H. Qu, and H. Li, "Spike-timing-dependent back propagation in deep spiking neural networks," *arXiv:2003.11837 [cs.NE]*, Mar. 2020.

[27] Y. Shouyi, O. Peng, Y. Jianxun, L. Tianyi, L. Xiudong, L. Leibo, and W. Shaojun, "An ultra-high energy-efficient reconfigurable processor for deep neural networks with binary/ternary weights in 28nm CMOS," *IEEE Symposium on VLSI Circuits*, pp. 37–38, Jun. 2018.

[28] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges: A survey," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 2, pp. 1–35, 2019.

[29] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2015, pp. 1–8.

[30] H. Kwon and T. Krishna, "OpenSMART: Single-cycle multi-hop noc generator in BSV and Chisel," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2017, pp. 195–204.

[31] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "VTA: an open hardware-software stack for deep learning," *arXiv:1807.04188 [cs.LG]*, Jul. 2018.

[32] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture (ISCA)*, Apr. 1982, p. 112–119.

[33] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[34] S. K. Esser, R. Appuswamy, P. Merolla, J. V. Arthur, and D. S. Modha, "Backpropagation for energy-efficient neuromorphic computing," in *Advances in neural information processing systems (NIPS)*, Dec 2015, pp. 1117–1125.

[35] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Technical report, University of Toronto*, Apr. 2009.

[36] Z. Du, D. D. B.-D. Rubin, Y. Chen, L. Hel, T. Chen, L. Zhang, C. Wu, and O. Temam, "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 494–507.

[37] M. M. Khan, D. R. Lester, L. A. Plana, A. Rast, X. Jin, E. Painkras, and S. B. Furber, "Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *2008 IEEE International Joint Conference on Neural Networks (IJCNN)*, Jun 2008, pp. 2849–2856.

[38] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2018, pp. 448–460.

[39] B. Wang, J. Zhou, W.-F. Wong, and L.-S. Peh, "Shenjing: A low power reconfigurable neuromorphic accelerator with partial-sum and spike networks-on-chip," *Proceedings of Design, Automation, and Test in Europe (DATE)*, Mar 2020.