SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs

SATHISH PANCHAPAKESAN and ZHENMAN FANG, Simon Fraser University, Canada
JIAN LI, Futurewei Technologies Inc., USA

Compared to conventional artificial neural networks, Spiking Neural Networks (SNNs) are more biologically plausible and require less computation due to their *event-driven* nature of spiking neurons. However, the default asynchronous execution of SNNs also poses great challenges to accelerate their performance on FPGAs.

In this work, we present a novel synchronous approach for rate encoding based SNNs , which is more hardware friendly than conventional asynchronous approaches. We first quantitatively evaluate and mathematically prove that the proposed synchronous approach and asynchronous implementation alternatives of rate encoding based SNNs are similar in terms of inference accuracy and we highlight the computational performance advantage of using SyncNN over asynchronous approach. We also design and implement the SyncNN framework to accelerate SNNs on Xilinx ARM-FPGA SoCs in a synchronous fashion. To improve the computation and memory access efficiency, we first quantize the network weights to 16-bit, 8-bit, and 4-bit fixed-point values with the SNN friendly quantization techniques. Moreover, we encode only the activated neurons by recording their positions and corresponding number of spikes to fully utilize the event-driven characteristics of SNNs, instead of using the common binary encoding (i.e., 1 for a spike and 0 for no spike).

For the encoded neurons that have dynamic and irregular access patterns, we design parameterized compute engines to accelerate their performance on the FPGA, where we explore various parallelization strategies and memory access optimizations. Our experimental results on multiple Xilinx ARM-FPGA SoC boards demonstrate that our SyncNN is scalable to run multiple networks, such as LeNet, Network in Network, and VGG, on various datasets such as MNIST, SVHN, and CIFAR-10. SyncNN not only achieves competitive accuracy (99.6%) but also achieves state-of-the-art performance (13,086 frames per second) for the MNIST dataset. Finally, we compare the performance of SyncNN with conventional CNNs using the Vitis AI and find that SyncNN can achieve similar accuracy and better performance compared to Vitis AI for image classification using small networks.

ACM Reference Format:
Sathish Panchapakesan, Zhenman Fang, and Jian Li. 2022. SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 1 (August 2022), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Neural networks (NNs) have been widely used in many areas such as image classification, speech recognition, and automated control [16]. More recently, SNNs, often referred to as the third-generation NNs, have attracted increasing attention because they are more biologically plausible and have more potential for hardware acceleration [26]. SNNs process spikes based on the membrane potential of the neurons and are modeled in accordance to the actual neural system of the human

Authors' addresses: Sathish Panchapakesan, sathishp@sfu.ca; Zhenman Fang, zhenman@sfu.ca, Simon Fraser University, Canada; Jian Li, jian.li@futurewei.com, Futurewei Technologies Inc., USA.

1936-7406/2022/8-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

brain [18]. Compared to artificial NNs (ANNs)—such as the widely used convolutional NNs (CNNs)—where all neurons in each layer are activated and computed, SNNs only activate those neurons whose membrane potential exceeds the threshold potential [26]. This event-driven nature greatly reduces the computation and communication between neurons.

The process of representing neural spikes (i.e., information) in SNNs is called neural encoding and there are two major approaches: rate encoding and temporal encoding. Temporal encoding requires spike based training mechanisms to train the time interval between spikes along with the network parameters [1, 2, 10, 20, 27, 35]. However, the temporal information processing capability limits the scope of exploring SNNs only for shallow networks [31]. On the other hand, rate encoding—where the number of spikes in an encoding window is proportional to the numerical value to be encoded—does not need any additional trained information and allows to directly convert the trained ANN models to SNN fashion for evaluation [4, 7, 13, 29]. More recently, the CNN-to-SNN converted models have achieved very good accuracy for deep networks such as VGG and ResNet [31]. In this work, we focus on the conversion based rate encoding SNNs to explore deeper networks on FPGAs.

In rate encoding SNNs, the input information (e.g., image pixels) is converted to spikes for an encoding window (i.e., multiple timesteps). At each timestep, the spikes carry the information along the layers in the network. The default asynchronous execution flow of SNNs enables the hardware to run all the layers concurrently in a pipeline fashion at any timestep [9]. However, the asynchronous execution also poses great challenges for implementing deeper networks on hardware. First, the network parameters have to be on chip for all the layers, which makes it not feasible to run deeper networks on edge devices. Second, the layer with the highest workload becomes the bottleneck and the resources allocated to other layers in the network remain idle for most of the time. Lastly, for deeper networks, the encoding window grows much larger to achieve the required accuracy, and thus running the entire network for many timesteps could make the SNN even slower than the original ANN.

In this paper, we propose a novel synchronous SNN, called *SyncNN*, to overcome those challenges. Instead of running the entire network for multiple timesteps, we only run the input layer—that converts the inputs to spikes—for multiple timesteps, and encode the number of spikes for each neuron. After that, we compute the remaining layers of the network in a synchronous layer-by-layer fashion, for just one timestep. SyncNN preserves the event-driven feature of SNNs by only activating those neurons whose aggregated membrane potential exceeds the threshold; for each spiked neuron, we also encode the number of effective spikes based on its aggregated membrane potential. We prove that the proposed SyncNN based approach is mathematically the same in terms of computation and would achieve the same accuracy as the asynchronous approach. SyncNN addresses the aforementioned challenges, and opens more opportunities for hardware acceleration.

To achieve real-time SNN inference, especially for deep SNNs that can achieve better accuracy, we accelerate SyncNN on Xilinx ARM-FPGA System-on-Chips (SoCs) using high-level synthesis (HLS) C++. To reduce the computing operations and memory accesses, we apply two algorithmic optimizations. First, instead of using the general binary encoding that encodes all neurons as 0 and 1, we record only the neurons that have spiked, by encoding their positions and corresponding number of spikes. Second, we quantize the network weights to 16 bits, 8 bits and 4 bits using SNN friendly quantization that puts more priority in weights with higher magnitude. For deeper networks, we also explore the mixed precision technique to safeguard the accuracy of the network in lower precision, where very few layers in the network have 8 bits precision weights and the remaining majority of the layers in the network have 4 bits precision weights. We design configurable and scalable neuron encoding and spike aggregation engines, which address the challenge of dynamic and irregular access patterns due to the event-driven nature of SNNs and explore different combinations of pipeline and parallelization techniques, as well as memory access optimizations. Also, to support

different network and layer sizes on different FPGA devices, we use a hierarchical on-chip buffering strategy. We also use memory coalescing and bursting to optimize the off-chip memory access. Finally, we compare the hardware performance of SyncNN based SNNs with conventional CNNs using Vitis AI [39] on the same Xilinx FPGA platforms.

Unlike prior FPGA studies [9, 12, 14, 23, 24] that only evaluated small networks such as MLP and LeNet, we have evaluated SyncNN for various CNN-based networks including LeNet, Network in Network (NiN) and VGG, for multiple datasets including MNIST, SVHN and CIFAR-10, on multiple ARM-FPGA SoCs, including Xilinx ZedBoard, ZCU104 and ZCU102 boards. Compared to state-of-the-art SNN acceleration work on FPGAs [9], for the same experimental setup—LeNet for MNIST dataset, on Xilinx ZCU102 board—SyncNN achieves 13,086 frames per second, which is 6.16x faster than [9], and 99.3% accuracy, which is higher than 99.2% in [9].

In summary, this paper makes the following contributions:

- A novel synchronous event-driven SNN with quantitative comparison to asynchronous SNN approaches.
- The first configurable and scalable FPGA engine of SNN that supports deep networks on multiple FPGA devices. The SyncNN framework is also open sourced at https://github.com/SFU-HiAccel/SyncNN.
- The first 4-bit SNN on FPGAs (for LeNet) that achieves a very high accuracy of 99.6% for the MNIST dataset.
- The first work that explores mixed precision quantization for SNNs (8 bits and 4 bits) which has negligible drop in accuracy for deeper networks such as NiN and VGG.
- State-of-the-art SNN performance of 13,086 frames per second (FPS) for the MNIST dataset (using LeNet).
- Comparison of SyncNN accuracy and performance with Vitis AI, which shows that SyncNN opens up a door for CNN alternatives.

The remainder of this paper is organized as follows. Section 2 describes SNN background and related work. Section 3 presents the SNN algorithm, limitations of asynchronous SNNs, and our proposed synchronous SNN approach. Section 4 proves the accuracy of SyncNN and discusses its hardware implementation advantage. Section 5 presents our hardware optimizations to implement SyncNN on Xilinx ARM-FPGA SoCs. Section 6 evaluate the accuracy and performance of SyncNN and compares it to state-of-the-art work. Finally, Section 7 concludes this paper.

2 SNN BACKGROUND AND RELATED WORK

In this section, we will describe how SNNs work and their computing advantages, challenges and advancements to improve SNN accuracy, and hardware acceleration techniques to improve SNN performance. Finally, we will summarize the goal of our work.

2.1 How SNNs Work and Their Computing Advantages

SNNs [18, 26] are considered as the third generation neural networks and are well known for their biological plausibility. They operate using *spikes* happening at discrete events, rather than continuous values as in the case of widely used artificial and convolutional neural networks (ANNs and CNNs). For example, in the widely used integrate-and-fire (IF) based SNN model, shown in Figure 1, when an input *spike* comes into a neuron, the *membrane potential* of the neuron is either increased or decreased based on the nature of the spike (excitatory or inhibitory). Once the membrane potential crosses the threshold value, the neuron generates spikes to the connected neurons in the next layer, and its membrane potential is reset. If the connection associated with the spike has a positive weight, then it is an excitatory spike and the membrane potential of the

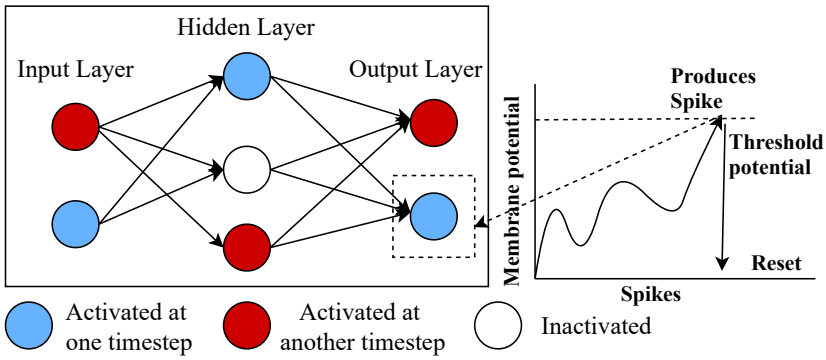


Fig. 1. Overview of the SNN working mechanism

neuron is increased. Otherwise, if the connection associated with the spike has a negative weight, it is an inhibitory spike which reduces the membrane potential of the neuron [26].

There are several models of SNNs, like the IF, leaky IF (LIF), spike response, Izhikevich, and Hodgkin-Huxley models[25][5][7][37]. In this paper, we focus on the IF model: the membrane potential is just added or subtracted by the weight of the connection in which there is an incoming spike, and is reset immediately once it reaches the threshold value.

Event Driven Nature of SNNs. While conventional ANNs and CNNs have had a great success in many areas such as handwriting recognition, image classification, speech recognition, and automated control, they are very hardware hungry and pose great challenges for real-time inference on edge devices. One of the major reasons is that, in ANNs and CNNs, all the neurons in a layer are activated and all the information is passed from one layer to the next layer in a synchronous fashion. In SNNs, unlike ANNs and CNNs, only the neurons whose membrane potential reached the threshold are activated and pass spikes to their connected neurons in the next layer [18], shown in Figure 1. This *event-driven* nature of SNNs greatly reduces the total number of computations and data communications between layers in SNNs, since typically only a small fraction of the neurons generate spikes at any given timestep.

Asynchronous Nature of SNNs. In fact, the neurons in SNNs are not activated layer by layer. Shown in Figure 1, at one timestep, one set of neurons (with membrane potential reached the threshold) are activated among different layers; at another timestep, a different set of neurons among different layers are activated. Overall, at the end of the encoding window, there is a large fraction of neurons that are not activated and computed. Therefore, by default, SNNs have an *asynchronous* execution flow where all the layers in the network can execute in parallel.

In summary, theoretically, SNNs require fewer computation operations and data communications than conventional ANNs and CNNs, and can achieve lower latencies due to their event-driven computing and asynchronous nature. This makes SNNs a great candidate for real-time inference on edge devices.

2.2 Challenges and Advancements for SNN Accuracy

The training mechanisms of SNNs are more computationally intensive and even training a shallow SNN can take orders-of-magnitude more time than training a similar ANN or CNN.

In ANNs and CNNs, the gradient descent algorithm is widely used for training the network: the errors, i.e., the difference between the network's output and desired output, are back-propagated and the network parameters are adjusted accordingly over multiple iterations. In SNNs, the same method is more complex, since the training error is the difference in the spike timings of the actual

and desired spike trains and it is quite unpredictable to achieve the desired output based on input excitation. Many algorithms have been proposed to improve the training of SNNs.

Temporal Coding based Training. One of the earliest proposals was the SpikeProp algorithm [2], which described the cost function in terms of the difference between desired and actual spike times for a single spike. Later studies such as QuickProp[3] and RProp [20] have revised this rule to incorporate learning with multiple spikes for better convergence speed. Other algorithms such as the spike pattern association have their cost functions defined in terms of the difference in spike trains converted to analog signals. The ReSuMe[27] and DL-ReSuMe[35] algorithm use local spike timing dependent plasticity (STDP) rules [1] for weight update. However, temporal training of SNNs is often more computationally intensive to achieve high accuracy and has been studied only on shallow networks [31].

Conversion of CNNs to Rate-based SNNs. An alternative approach is to convert pre-trained CNNs to spiking equivalents, which we use in our work as well. In the conversion technique, the weights obtained from the trained CNNs, are mapped to a network of spiking IF units. And the activation of neurons in CNNs is proportional to the firing rate of SNNs [4, 7, 13, 29]. Various approaches were proposed to reduce the loss in the conversion such as setting the bias value to zero and using ReLu as activation function so that the firing rate of SNNs does not go negative. Also, weight normalization techniques and threshold balancing methods were helpful to decide the input rate of the network and also observe almost lossless conversion from CNNs to SNNs [4, 7, 13, 29]. Recently, the conversion based SNN models have been studied on complex VGG and residual networks [31] with high accuracy.

In this paper, we use the conversion technique and have applied the ReLu activation, weight normalization, zero bias and other optimizations discussed in [7, 13, 29, 31] during the conversion. Our main focus is on the SNN inference stage on edge devices.

2.3 Hardware Acceleration for SNN

As SNNs process the information in an event-driven nature, it is inefficient to implement them on CPUs that process information in a synchronous manner. Also, the substantial computational cost for implementing deep networks has created a need for specialized hardware acceleration. For example, several studies have explored GPU implementations for SNNs and showed significant speedup over CPUs [4, 11, 22].

Lots of efforts have been made to develop neuromorphic hardware for SNN simulation where the communication of spikes is event-driven. BrainScaleS is the mixed signal SNN hardware designed with 384 cores and supports up to 200K Neurons and 45M Synapses [30]. SpiNNaker is an ARM-based processor platform optimized for the simulation of SNNs with up to 1M cores in 130nm CMOS with each core supporting 1K neurons and 1K synapses [34]. The recent upgrade of SpiNNaker2 is a 10M-core machine in 22nm FDSOI with additional numerical accelerators [19]. The TrueNorth chip from IBM contains 4,096 cores supporting 1M neurons and 256M synapses [21].

Apart from the GPU and neuromorphic hardware implementations, FPGAs are also a very attractive alternative especially for the SNN inference on edge devices, as they are commercially-available hardware which can be customized for the SNN computation and provide low power and high energy-efficiency. For example, in [28], a VHDL implementation for a simple pattern recognition problem with temporal coding based SNN is explored. NeuroFlow [5] is another FPGA accelerated SNN architecture for IF and Izhikevich models based on the STDP rule for learning. A fully connected Izhikevich model of SNN for 1,440 neurons has been realized on Xilinx Virtex 6 device [25]. A LIF model of SNN to simulate 20 million to 2.6 billion neurons was realized with Altera Stratix V FPGA [37]. A highly pipelined SNN model using HLS has been realized for 256 neurons which uses DSNN network and Hebbian learning mechanism [15].

Implementation of rate based SNN inference on Xilinx ZC706 SoC FPGA board for MNIST image clasification with LIF model has been studied with an accuracy of 97.06% and a performance of 161 frames per second [12]. Also, an another recent synthesizable Verilog implementation on Xilinx ZCU102 SoC FPGA board for IF model for converted SNNs from ANN and CNN achieves 98.94% accuracy with 164 frames per second at 150MHz clock frequency [14].

More recently, a temporal encoding based SNN inference on Xilinx ZCU102 SoC board for MNIST image clasification achieves an accuracy of 99.2% and a simulation performance of 2,124 FPS [9] at 125MHz. Also, the work compares its performance with Intel Loihi ASIC, Intel i9-9900K CPU, Nvidia RTX 5000 GPU, and Nvidia AGX Xavier Edge GPU implementations.

2.4 Goal of This Work

The dynamic and event-driven nature of SNN operations makes it nontrivial to implement on FPGAs, especially to implement a framework that can run various network models across a range of embedded FPGA boards. One challenge is to resolve the data dependencies and parallelize the computation. Another challenge is to buffer the large weight matrix on chip, where weights are accessed randomly across all the neurons from layer to layer.

The goal of this paper is to design and implement a framework (called SyncNN) to address both computation and memory access challenges to run any deep networks across a range of embedded ARM-FPGA SoCs. Therefore, it can enable more evaluation and optimization of SNN inference on edge devices. To the best of our knowledge, SyncNN is the first FPGA implementation of rate based IF SNN inference to run deeper networks like NiN and VGG, especially designed in high-level synthesis with all the computation and memory access optimizations studied in this paper. We will present the quantitative comparison of our work and prior studies using GPUs, FPGAs, and neuromorphic hardware in Section 6.6.

3 SYNCNN: SYNCHRONOUS SNN APPROACH

In this session, we highlight the challenges in the implementation of rate-based SNN on hardware in asynchronous fashion, and propose a synchronous approach called SyncNN, which is more hardware friendly and is scalabale to run any deeper network on a given Xilinx SoC FPGA board.

3.1 Rate Encoding based SNN Algorithm

Algorithm 1 presents an overview of the conventional rate encoding SNN algorithm based on the IF model and Poisson spike generation [7]. For each input image, it iteratively calls the `SPIKING_NET` function (lines 1-4) for a particular encoding window. The number of simulation steps (*Sims*), i.e., timesteps, is based on the network model and the input. As the encoding window increases, more number of spikes are transmitted in the network. This `SPIKING_NET` function calls the following three major functions (lines 5-12):

(1) *Function* `POISSON_ENCODING` *(lines 13-18).* The input nodes in the network are called *Poissons* as they use Poisson random variables to convert the input amplitudes to a random spike train. It takes the image (img*) as input and generates Poisson spikes (pSp*). These are activated based on the pixel intensity of the image: a Poisson random variable is compared with the pixel intensity, and based on the comparison, the Poisson unit is activated and it spikes. Let the time taken to complete this function be *PE*.

(2) *Function* `SPIKE_AGGREGATE` *(lines 19-25).* The actual computation takes place here. Depending on the layer of the network (convolutional, pooling or dense), we perform the aggregation operation of weights to the membrane potential (Vm*) of the neuron. While ANNs/CNNs perform operation for all the inputs, SNNs compute only for the encoded spiked inputs

Algorithm 1 Pseudo code for conventional rate-based SNN

```
1:  function MAIN
2:      for each image do
3:          for each simulation step do
4:              SPIKING_NET(img*)
5:  function SPIKING_NET(img*)
6:      #Encodes input Poisson spikes: pSp*
7:      POISSON_ENCODING(img*,pSp*)
8:      for each layer in network do
9:          #Update membrane potential: Vm*
10:         SPIKE_AGGREGATE(weights*,pSp*,nSp*,Vm*)
11:         #Encodes neuron spikes: nSp*
12:         NEURON_ENCODING(Vm*,nSp*)
13: function POISSON_ENCODING(img*,pSp*)
14:     pSpiked = 0 #Reset counter for Poisson spikes
15:     for each image pixel do
16:         if pixelValue > PoissonThreshold then
17:             pSp[pSpiked] = pixelIndex
18:             pSpiked++
19: function SPIKE_AGGREGATE(weights*,pSp*,nSp*,Vm*)
20:     #Convolutional, Pooling or Dense
21:     for each spiked inputs in psp*/nsp* do
22:         for each weight w in the kernel do
23:             for each number of feature maps o do
24:                 #Performs the aggregation operation
25:                 Vm[o] += weights[w]
26: function NEURON_ENCODING(Vm*,nSp*)
27:     nSpiked = 0 #Reset counter for neuron spikes
28:     for each neuron n do
29:         if Vm[n] > VmThreshold then
30:             nSp[nSpiked] = n
31:             Vm[n] -= VmThreshold
32:             nSpiked++
```

(Poisson Spikes - pSp*, Neuron Spikes - nSp*) in that simulation step, which is the key advantage of SNNs. Let the time taken to complete this function for the l_{th} layer be *SA[l]*.

(3) *Function* `NEURON_ENCODING` *(lines 26-32).* It takes the aggregated membrane potential as inputs (Vm*), and generates neuron spikes (nSp*). As shown in Figure 1, the membrane potential (Vm*) of the neuron is updated based on the spikes it receives, and once it reaches the threshold value, the neuron is activated with a spike and the membrane potential is set back to Vreset. Let the time taken to complete this function for the l_{th} layer be *NE[l].*

The `SPIKE_AGGREGATE` function takes the encoded spiked inputs generated in the `NEURON_ENCODING` function as input and calculates the membrane potentials of neurons for the `NEURON_ENCODING` function in the next layer. This process is repeated for all the layers (*NL*) in the network. When this algorithm is implemented in a naive synchronous fashion, the time it takes to classify one image is:

$$T_{naive\_sync} = Sims * (PE + \sum_{l=1}^{NL}(NE[l] + SA[l])) \quad (1)$$

3.2 Asynchronous Approach of SNNs

One of the attractive features for SNNs is their asynchronous execution flow. If there is enough hardware resource, all the layers in the network can be computed concurrently in a pipeline fashion. The pipeline throughput is determined by the layer that takes the most of the time at that simulation step. Within the three major functions of SNNs, the SPIKE_AGGREGATE function at any simulation step is the most time consuming one. Therefore, the time it takes to classify one image is:

$$T_{async} = \sum_{s=1}^{Sims} \max(SA_s[1], SA_s[2], ..., SA_s[NL]) \tag{2}$$

However, the asynchronous approach faces several challenges when accelerated on edge devices.

(1) **Network size limitation.** In order to run all the layers in parallel, the network parameters (inputs, weights, and outputs) have to be on-chip independently for every layer. Also, computing resources have to be allocated for all the layers. For edge devices, which has limited resources, it is difficult to implement larger networks in the asynchronous fashion.
(2) **Resource underutilization.** The SPIKE_AGGREGATE function has the highest workload compared to the other functions. In the asynchronous approach, the overall processing time in a simulation step highly depends on the computation unit of the layer that has the longest latency. Therefore, all the resources for other functions have to remain idle until the slowest function finishes. Moreover, since all the layers are implemented on the hardware, the optimizations within each layer is also restricted due to limited resource.
(3) **Impact of encoding window.** For the asynchronous approach, all the layers in the network still run for multiple simulation steps. For small networks like LeNet and MLP, the encoding window is very small to achieve a good accuracy. However, for larger networks like NiN and VGG, the encoding window is very large to achieve a good accuracy as shown in Section 6.2. Based on Equation 2, the large encoding window ($Sims$) limits the performance.

3.3 SyncNN: Synchronous Approach of SNN Acceleration

To address the above challenges, we propose a novel synchronous approach, called SyncNN, to accelerate rate encoding SNNs on hardware. Unlike previous approaches, we do not run the entire network for multiple simulation steps. Only the input layer, i.e., the POISSON_ENCODING function, is run for multiple simulation steps. For the remaining layers, the SPIKE_AGGREGATE and NEURON_ENCODING functions are run only once for each layer in the network and all spikes are aggregated (increment to the membrane potential) together. Therefore, the size of the encoding window has no effect on the performance of the SPIKE_AGGREGATE and NEURON_ENCODING functions. For example, if the aggregated membrane potential value of a neuron is twice the value of the threshold, we consider that the neuron needs to spike twice. The key is that instead of spiking a neuron at multiple timesteps (e.g., timestep 1 and 4), SyncNN only spikes a neuron at the final timestep (and only one timestep) and spikes it with the equivalent number of times. Due to the synchronous layer-by-layer execution, this is equivalent to the original SNN with multiple timesteps; a mathematical proof will be provided in Section 4.1. SyncNN still preserves the event-driven feature of SNN as only those neurons whose membrane potential exceeds the threshold will be activated and computed. The time it takes to classify one image is:

$$T_{SyncNN} = (Sims * PE) + \sum_{l=1}^{NL} (NE[l] + SA[l])) \tag{3}$$

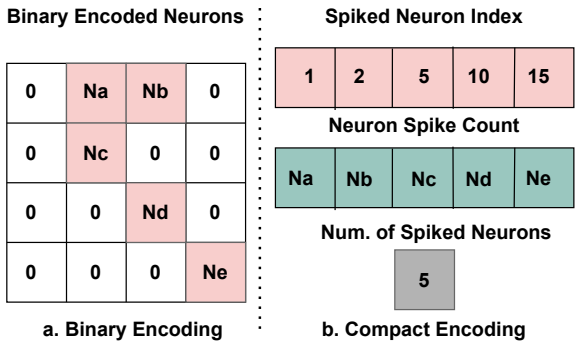


Fig. 2. Neuron encoding method in SyncNN

3.4 Neuron Encoding in SyncNN

The most commonly used neuron encoding technique is the binary encoding (as shown in Figure 2.a), where N_i is used to represent the number of spikes for the activated neuron and 0 is used for neurons that are not activated. Only the neurons that are activated are actually needed for computation. The downside of the binary encoding is that all neurons are now used in the computation. To fully exploit the effectiveness of the event-driven nature, we record only the activated neurons by encoding the position of the activated neuron and its associated number of spikes, and the total number of activated neurons. This encoding for SyncNN is shown in Figure 2.b.

4 WHEN TO USE SYNCNN

In this section, we will first prove that mathematically, our SyncNN is equivalent to the conventional asynchronous SNN and achieves the same accuracy. Then we will compare its required computational operations to conventional CNNs and asynchronous SNNs and discuss its advantage in hardware implementation.

4.1 Accuracy Comparison with Asynchronous Approach

First, we prove that our proposed SyncNN approach is mathematically the same as the asynchronous SNN approach and can reach the same accuracy for any given network. We know that, the Poisson Encoding (PE), Spike Aggregate (SA) and Neuron Encoding (NE) functions are the three major functions in SNNs. Let us compare the operations in each function individually.

Poisson Encoding. In both SyncNN and asynchronous approaches, the image pixel intensity is compared with the Poisson threshold. Based on the comparison, the neuron, corresponding to the pixel intensity of the image, is encoded with a spike. The only difference between the two approaches is, in SyncNN, we repeat the Poisson Encoding function for all the simulation steps *(Sims)* and encode all the spikes together. Therefore, for any neuron, the total number of spikes in *Sync.PE* is equal to the spikes produced in *Asynch.PE* for that neuron after all the simulation steps.

For the encoded neuron n in the PE function,

$$SyncNN.PE_n == \sum_{s=1}^{Sims} (Async.PE_n^s) \tag{4}$$

Spike Aggregate. For the layer that is immediately after the input layer, the neurons encoded in the *Poisson Encoding* function is now the event-driven input to the *Spike Aggregate* function. The Spike Aggregate *(SA)* function can be Convolutional, Pooling or Dense layer depending on the

network. The network topology is the same for both the asynchronous and SyncNN approaches. Let us consider a dense layer for both the cases. A similar proof can also be applied to other layers.

In the dense layer (i.e., fully connected layer), assume the total number of encoded neurons (i.e., spiked neurons) in the current layer is *EN* and the number of neurons in the next layer is *M*. Each encoded neuron *i* in the current layer connects to all *M* neurons in the next layer, i.e., the spikes are passed from neuron *i* to each neuron *j* in the next layer. Let w be the weight associated between *i* and *j*. Then the output for neuron *j* in the next layer is evaluated as the following.

For SyncNN based SNNs:

$$Sync.SA_j = \sum_{i=1}^{EN} Sync.PE_i * w_{ij} \tag{5}$$

For asynchronous based SNNs at any simulation step *'s'*:

$$Async.SA_j^s = \sum_{i=1}^{EN} Async.PE_i^s * w_{ij} \tag{6}$$

With SyncNN approach, the effect of Poisson Encoding for all the simulation steps *(Sims)* is accounted together and the Spike Aggregate function for any layer is executed only once. But in Async. approach, the Spike Aggregate function for any layer is repeated for *Sims* number of times. Therefore, the output for neuron *j* at the end of all simulation steps is:

$$\begin{aligned}
\sum_{s=1}^{Sims} Async.SA_j^s &= \sum_{s=1}^{Sims}\sum_{i=1}^{EN} Async.PE_i^s * w_{ij} \quad (from\ Equation\ 6) \\
&= \sum_{i=1}^{EN}\sum_{s=1}^{Sims} Async.PE_i^s * w_{ij} \\
&= \sum_{i=1}^{EN}\left(\sum_{s=1}^{Sims} Async.PE_i^s\right) * w_{ij} \\
&= \sum_{i=1}^{EN} Sync.PE_i * w_{ij} \quad (from\ Equation\ 4)
\end{aligned}$$

Therefore, for any output neuron *j*,

$$Sync.SA_j == \sum_{s=1}^{Sims} Async.SA_j^s \tag{7}$$

Thus, it is proved that the effect of Spike Aggregate function for any encoded neuron in SyncNN is the same as the effect of Spike Aggregate function for the same encoded neuron in the asynchronous approach at the end of all the simulation steps.

Neuron Encoding. The output of the *Spike Aggregate* function is taken as input for Neuron Encoding function. For every neuron in the layer, we check whether the input is greater than threshold (voltage) *Vth*. In the asynchronous approach, shown in lines 2-6 in Algorithm 2, at any simulation step, if the condition passes, we encode the neuron with 1 and reset the Spike Aggregate output by subtracting it by *Vth*. Whereas, in SyncNN, if the condition passes, we measure the output of the Neuron Encoding as a total number of *Vth* present in the Spike Aggregate output, shown in lines 9-10 in Algorithm 2. By this way, we account all the spikes present in any neuron across all the simulation steps in one shot. Mathematically, we can rewrite the Neuron Encoding function for both the SyncNN approach as shown in lines 12-16 in Algorithm 2.

Algorithm 2 Pseudo code for Neuron Encoding

```
1:  In Asynchronous Approach:
2:  for each neuron 'n' in Layer do
3:      for each simulation step 's' in Sims do
4:          if Async.SA[n]>VmThreshold then
5:              Async.NE[n][s] = 1
6:              Async.SA[n] -= VmThreshold
7:  In SyncNN:
8:  for each neuron 'n' in Layer do
9:      if Sync.SA[n]>VmThreshold then
10:         Sync.NE[n] = int (Sync.SA[n]/VmThreshold)
11: SyncNN can be re-written as:
12: for each neuron 'n' in Layer do
13:     while Sync.SA[n]>0 do
14:         if Sync.SA[n]>VmThreshold then
15:             Sync.NE[n] += 1
16:             Sync.SA[n] -= VmThreshold
```

Since we proved that $Async.SA[n] == Sync.SA[n]$, we can conclude that the condition in asynchronous approach (line 4 in Algorithm 2) and the condition in SyncNN approach (line 14 in Algorithm 2) will be triggered the same number of times. Therefore, we can come to a conclusion that, for any neuron n,

$$Sync.NE_n == \sum_{s=1}^{Sims} Async.NE_n^s \tag{8}$$

Therefore, from Equations 4, 7, and 8, we prove that, the proposed approach *SyncNN* is mathematically the same and will achieve the same accuracy as the conventional asynchronous SNN over the effect of all the simulation steps.

4.2 Hardware Perspective

From the hardware perspective, since a network is executed layer by layer in SyncNN, the computing and memory resources can be reused between layers through time multiplexing. Hence, any deep networks can be implemented and the slowest function will not cause resource underutilization for other functions. Also, the encoding window only affects the performance of the input layer, i.e., the `POISSON_ENCODING` function, where parallelism can be explored among multiple timesteps.

5 HARDWARE DESIGN OF SYNCNN FRAMEWORK

The computation and memory access pattern of SNNs is irregular, because of their event-driven nature. In this section, we discuss the computation and memory-access optimization techniques implemented in SyncNN to address this challenge.

5.1 Quantization

To improve the computing and memory access efficiency of SyncNN, we first apply SNN friendly quantization to represent the weights in low precision fixed-point values. In SNN, after training the CNN network, the weights are normalised between -1 to 1. Therefore, we just need two bits in the integer portion (one for the value of 0 or 1, and the other for the sign bit). In all our experiments (for a wide range of networks and datasets), the remaining 14 bits are enough to represent the

trained weights without any quantization technique involved. For 8 bits and 4 bits, it needs some special quantization to represent the trained weights to reserve the accuracy.

Max scaling and rounding. In SNNs, the weights with high magnitude have more impact on altering the membrane potential of the layer. Hence, priority is given to represent those high magnitude weights for the number of bits chosen. The weight with the maximum magnitude in a layer is represented by the maximum possible fixed point representation for the number of bits chosen. For example, the maximum possible absolute value in fixed point representation for 8 bits and 4 bits with 2 bits allocated for the integer portion (along with sign) is 1.984375 and 1.75 respectively, and let us call it as *max_fixed_point*. Let the absolute max weight in the layer be *max_weight*. Now, we decide the scaling factor *scale* based on this representation.

$$\text{max\_weight} * \text{scale} = \text{max\_fixed\_point} \tag{9}$$

From the obtained scaling factor *scale*, we multiply all the weights with *scale* and then round it to the nearest fixed point representation. The threshold potential of the layer is also multiplied with the same scaling factor in order to maintain uniform number of spikes in the network.

Percentile-based scaling with clipping and rounding. After analyzing the distribution of the weights in a layer, we find that very few weights have large magnitudes and the majority of the weights are small. Deciding the scaling factor *scale* based on the maximum value prevents us from accurately representing the majority of small weights with a non-zero fixed point representation, especially for 4-bit representation. Therefore, based on the weight distribution of the layer, we choose the X-th percentile (e.g., 99-th percentile) of the weight and represent that weight with the maximum possible fixed point representation for the number of bits chosen (*max_fixed_point*). Now, we can decide the scaling factor *scale* based on the following equation:

$$\text{Xth\_percentile\_weight} * \text{scale} = \text{max\_fixed\_point} \tag{10}$$

We then multiply all the weights with the obtained scaling factor (*scale*) and round it to the nearest fixed point value. For the values whose magnitude is greater than the *max_fixed_point*, we clip it to the *max_fixed_point*.

The above two approaches—*max scaling and rounding* and *percentile-based scaling with clipping and rounding*—work good for 8 bits with negligible accuracy drop. And for 4 bits, the later approach works good for smaller networks like LeNet with negligible drop in accuracy, while the prior approach does not. But for larger networks like NiN and VGG, both the approaches result in significant accuracy drop. Figure 9 in Section 6.4 presents more detailed accuracy results for various networks for the *percentile-based scaling with clipping and rounding* quantization that we adopt.

Mixed Precision. To further improve the accuracy of deeper networks like NiN and VGG, we use the *Mixed Precision* quantization. In this approach, we use some layers at 8 bits and the remaining layers at 4 bits. In particular, we observe that, having the first few layers and the last few layers as 8 bits and all the remaining layers as 4 bits help achieve very little drop in accuracy. This decision was concluded after running several experiments. We started with randomly choosing few layers with 8-bit and the rest with 4-bit. But, from the experiments, the first few layers carry the most important information and it is very essential to dedicate some more bits for those layers. And the final layer is going to predict the classification output. Therefore, having the last layer with 8-bits also has a great impact in improving the performance.

Note that, we still use the previous *percentile-based scaling with clipping and rounding* approach to quantize the layers, but just that we use different precisions for different layers in the network. In Section 6.4, we will present the improvement in accuracy for deeper networks with *Mixed Precision* approach.

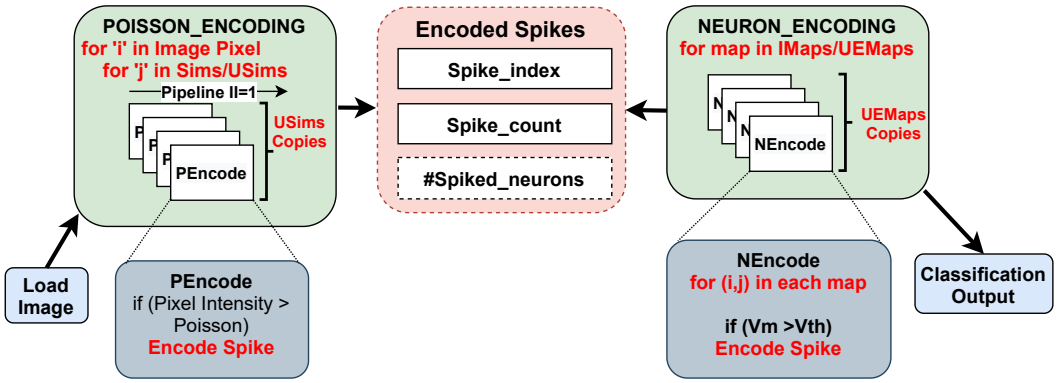


Fig. 3. Hardware architecture of Poisson and neuron encoding units in SyncNN

5.2 Computation Optimization

In SyncNN, we design configurable and scalable compute engines for all major functions with pipeline and parallelization techniques, which can be deployed on different Xilinx FPGA boards based on the available memory and computing resources.

POISSON_ENCODING. In SyncNN, as shown in Figue 3, all the input nodes (i.e., image pixels) in the `POISSON_ENCODING` function is run for multiple simulation steps *Sims*. We partially unroll (parallelize) the simulation step loop with a factor of *USims*, and pipeline the outer loop with pipeline initiation interval as 1 (II=1). The output of this function is the encoded spikes only for the activated neurons in the input layer.

NEURON_ENCODING. First, as shown in Figue 3, we parallelize the neuron encoding for each input feature map of a layer by partially unrolling the input feature map loop with a factor of *EMaps*. Inside each feature map, we pipeline the neural encoding of each neuron with II=1. The output of this function is the encoded spikes only for the activated neurons in that layer. If it is the final layer, it gives the classification output.

For those activated neurons, we encode their original neuron index and aggregated number of spikes in a pair of arrays (*Spike_index* and *Spike_count*); we also record the total number of spiked neurons (*#Spike_neurons*). While such encoding reduces the total number of computations as only activated neurons are encoded and computed, it also creates the dynamic and irregular access pattern challenge.

SPIKE_AGGREGATE. The topology of the function depends on the layer of the network. In SyncNN, we implement the widely used CNN computational units such as Convolutional, Pooling, Dense and Global Average Pooling. The major difference between the conventional CNNs and SyncNN based SNNs is the inputs presented to every layer. In SyncNNs, as shown in Figue 4, only the spiked neurons (random and event-driven) in the previous layer (output from the `NEURON_ENCODING` or `POISSON_ENCODING` function) are presented as inputs for the current layer. That is, the output of this function is now dependent on the randomly spiked inputs, which makes parallelization challenging.

(1) **Convolutional unit.** The `CONV_SPIKE_AGG` function in Figure 4 shows the convolutional unit implementation, which is the most important and complex one. We first design a basic convolutional unit called *ConvPE* to process each input feature map. Inside the *ConvPE*, since the output feature map loop (*OMaps*) has no dependency, we swap it in as the innermost loop and parallelize it by partially unrolling it with a factor of *UOMaps*. Then we pipeline the

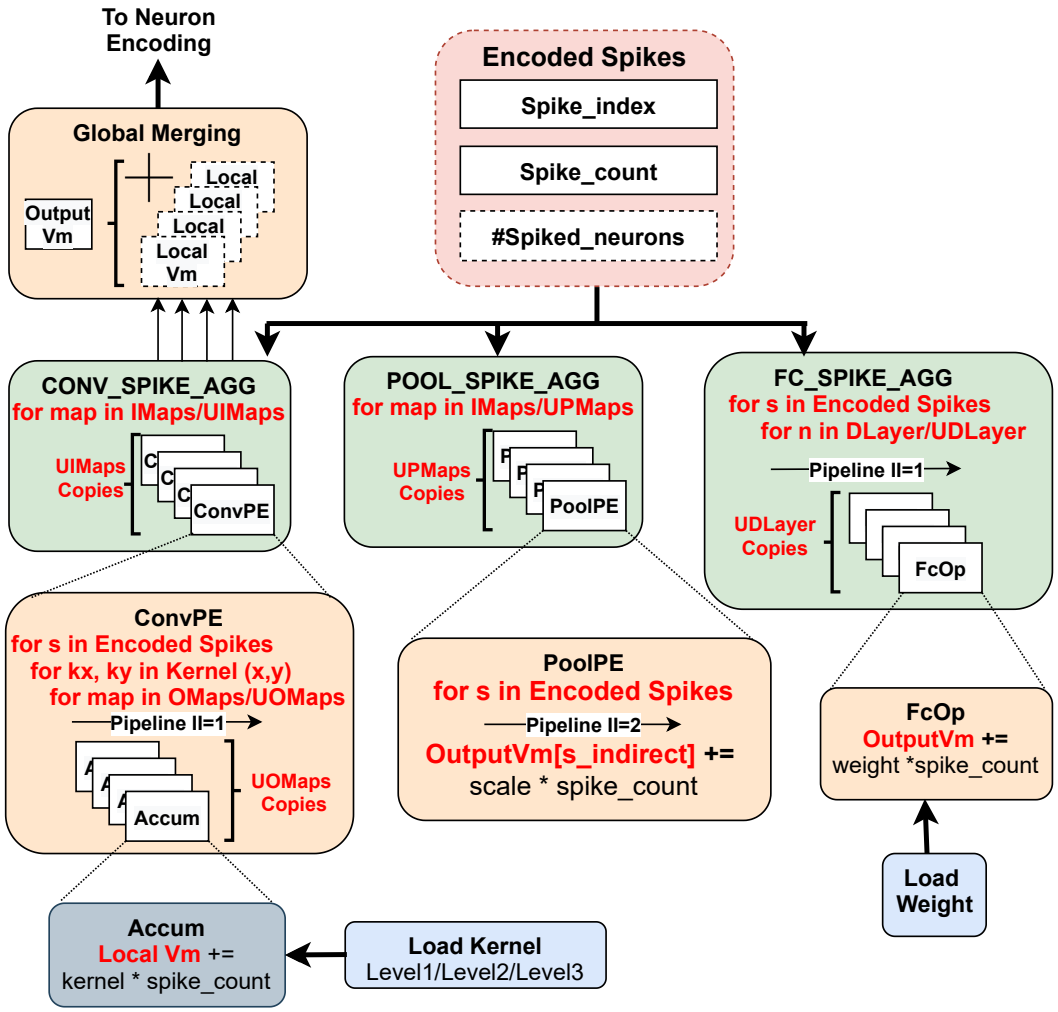


Fig. 4. Hardware architecture of spike aggregation units in SyncNN, including convolutional layer (CONV_SPIKE_AGG), fully connected layer (FC_SPIKE_AGG), and pooling layer (POOL_SPIKE_AGG).

rest of the loop nests (kernel loops, and encoded spikes loop) with II=1. Between the input maps (*IMaps*), there is a data dependency on the output membrane potential array (*Vm*), since multiple neurons from the previous layer can generate input spikes to the same neuron in the current layer and hence cannot be parallelised easily. In SyncNN, we allocate duplicate copies (*UIMaps*) of the output *Vm* array (*local_Vm*) for each group of parallelly accessed input maps to enable parallel processing. Finally, we add a global merging module to aggregate all the *local_Vm* copies to produce the final updated output membrane potential array *Vm*. *UIMaps* is the parameterized factor for parallelizing along the input maps, and *UIMaps* number of local copies are needed to temporally store the output results which is later aggregated together.

(2) **Pooling unit.** For a pooling layer, the number of output feature maps equals to the number of input feature maps for any spiked inputs. Therefore, the number of maps in the layer remains definite even though the spiked inputs for every map is decided in the run time.

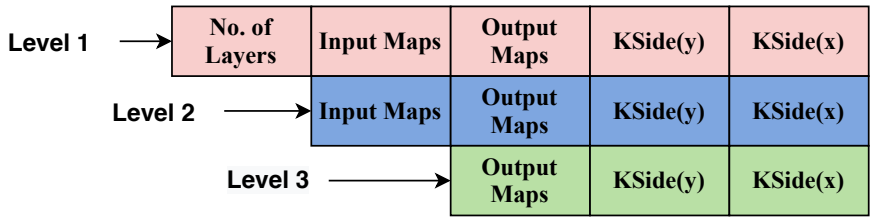


Fig. 5. Different granularity to load kernel depending on the size of the weights and available on-chip memory

Hence, we design a `POOL_SPIKE_AGG` function as shown in Figure 4, where we have *PoolPE* engine for each map and partially unroll the input feature map loop with a factor of *UPMaps*. Inside the PoolPE, we pipeline the loop (spiked_inputs) with II=2. We cannot achieve II=1 because of the data dependency of the output caused between randomly encoded spiked inputs (the index to update *OutputVm* is an indirect variation of the spiked neuron index *s*).

(3) **Dense unit.** The `FC_SPIKE_AGG` function in Figure 4 shows the fully connected dense unit implementation. For dense layers, for any spiked input, all the neurons in the layer are aggregated. For a dense layer of size *DLayer*, we parallelize the *FcOp* units with a unroll factor of *UDLayer*. At each *FcOp* unit, we accumulate the membrane potential with the weight and the spike count. Then for the remaining loop nest (spiked inputs and DLayer/UDLayer), we pipeline it with II=1.

5.3 Memory Access Optimization

The aforementioned event-driven nature of SNNs makes the weight access irregular and it is critical to buffer the weights on chip. In this session, we discuss the various memory access optimizations that are explored in SyncNN.

Hierarchical on-chip buffering. The encoded spikes that act as input to the layer and the membrane potential that is the aggregated output of the layer are relatively small and can be buffered on-chip and reused between the different layers in the network. The biggest challenge is to buffer the randomly accessed network weights on-chip.

In SyncNN, we use a hierarchical on-chip buffering technique to buffer as many weights as possible, depending on the network size and the on-chip memory size available on the FPGA board. We load the weights in a coalesced (widened bus) and burst fashion [17, 40] from the off-chip memory to on-chip memory at different granularity. As shown in Figure 5, for every convolutional layer, the weights are resolved in four dimensions. As we go down in the granularity, more weights have to be accessed from off-chip memory which impacts the performance of the network.

(1) **Level 1:** If the FPGA board can load the weights corresponding to all the layers in the neural network, the weights are prefetched at network level and therefore the entire off-chip access for the network is done only once. This is the best case scenario for loading the network weights and is often possible for smaller networks.
(2) **Level 2:** For bigger networks or smaller FPGA boards, where we cannot load all the weights on-chip, we load them layer-wise so that the weights buffer can be reused for every layer in network.
(3) **Level 3:** For very big networks (like NiN, VGG) or very small FPGA boards, it is not possible to load even one layer's weight on-chip. In this case, we go one step further in granularity and load the weights for every map of the layer and perform the computation for that map. For the next map, we again load the weights from off-chip.

Algorithm 3 Deciding which granularity level to buffer weights for convolutional lalyers

Input: A SNN network, with the total size of weights: $weights\_total$, the maximum size of weights available in any given layer: $weights\_max\_layer$, the maximum size of weights available in any feature map of a given layer: $weights\_max\_FM$, the total size of activations: $activations\_total$; and an FPGA board with the total available resource on-chip buffer size: $buffer\_total$

Output: The granularity level to buffer the weights for convolutional lalyers

```
buffer_avail = buffer_total - activations_total
if buffer_avail > weights_total then
    return Level 1
else if buffer_avail > weights_max_layer then
    return Level 2
else if buffer_avail > weights_max_FM then
    return Level 3
```

More specifically, Algorithm 3 formulates the steps to decide which level of buffering to choose for a given network on a given FPGA board. As shown in Table 1, different networks uses different granularity levels for a given FPGA platform. Detailed experimental setup about the network configuration and FPGA boards is explained in Section 6.1.

Table 1. Different granularity levels for different networks and FPGA platforms

Network/Platform	ZED	ZCU104	ZCU102
LeNet-S	Level 2	Level 1	Level 1
LeNet-L	Level 2	Level 1	Level 1
NiN	Level 3	Level 2	Level 2
VGG	Level 3	Level 3	Level 2

For dense layers, we prefetch and store the maximum number of rows in the weight matrix that can be buffered on-chip. If the spiked input can access the weights from the prefetched weight matrix, it directly reads them from on-chip buffer. Otherwise, that row containing the required weights will be loaded from the off-chip memory.

6 EXPERIMENTAL RESULTS

In this section, we describe our experimental setup and present our experimental results. We will first present the SNN accuracy results, including the impact of encoding window size, CNN-SNN conversion and quantization techniques. Then we will present the SyncNN performance results, and compare it with recent work in SNN acceleration and conventional CNN acceleration.

6.1 Experimental Setup

The network configurations for each network is summarized in Table 2. n*C*s denotes the convolutional layer with kernel size s*s and n is the number of kernels, *P*s denotes the pooling layer with size s*s and GAP denotes Global Average Pooling Layer, and a pure number n denotes the dense layer with n neurons. The first LeNet network *Lenet-S* is the same network configuration used in [9]. The second LeNet network *Lenet-L* is comparatively larger than *Lenet-S*, where it is difficult to hold all the network parameters on-chip, but has better accuracy. Two deeper networks, Network in Network (NiN) and VGG, are also evaluated. Three widely used image classification datasets are evaluated in SyncNN: MNIST, SVHN and CIFAR-10.

Table 2. Neural network configurations

Network	Configuration
Lenet-S	Input-32C3-P2-32C3-P2-256-Output
Lenet-L	Input-32C5-P2-64C5-P2-2048-Output
NiN	Input-(192C5-192C1-192C1-P3)*2 -192C5-192C1-10C1-GAP-Output
VGG	Input-(64C3-64C3-P2)-(128C3-128C3-P2) -(256C3-256C3-P2)-(512C3-512C3-P2)*2 -256-256-Output

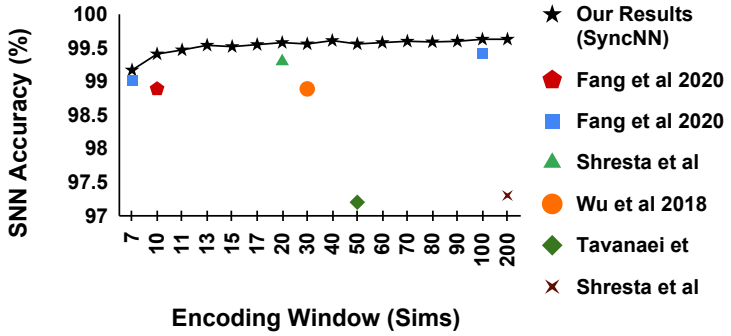


Fig. 6. Impact of encoding window on the accuracy for MNIST dataset

The MNIST dataset is tested for *Lenet-S* and *Lenet-L* networks. SVHN dataset is tested for *Lenet-S* and *VGG* networks. And CIFAR-10 is tested for *NiN* and *VGG* networks. Our SyncNN framework is built using Xilinx SDSoC 2019.1 that integrates Vivado HLS C++. Vivado HLS allows a user to code the hardware with C or C++ and supports pragmas to optimize the hardware. In SyncNN, HLS techniques such as pipelining, fine-grained and coarse-grained unrolling (i.e. parallelization), coalesced (widened bus), burst off-chip memory access, and hierarchical on-chip buffering are explored. Three different Xilinx SoC boards are used to test the SyncNN framework: Xilinx Zynq ZedBoard, Zynq UltraScale+ ZCU104 and ZCU102 boards. The main difference between the boards is the available resources: ZCU102 > ZCU104 > ZedBoard. For SNN, as per the SyncNN framework, the performance between the boards vary because of the difference in the available resources between the boards. Our designs run at 100MHz on ZedBoard, 150MHz on ZCU104, and 200MHz on ZCU102. Further increasing the frequency does not meet timing and the critical path is in the memory control signals and not related to the HLS kernel code.

6.2 Impact of Encoding Window

The encoding window *(Sims)* for a network is dependent on the depth of the network used and the input presented to the network. As the encoding window increases, the network transmits more spikes to predict the output more accurately. For MNIST dataset with smaller networks, as shown in Figure 6, the encoding window required to achieve a good accuracy is relatively small. Shrestha et al. [33] achieve 97.3% accuracy for 200 Sims. Tavanaei et al. achieve [36] 97.2% at 50 Sims. Wu et al. [38] achieve 98.89% accuracy for 30 Sims. Shrestha et al. [32] achieve 99.3% accuracy for 20 Sims. More recently, Fang et al. [9] achieves 99.01% at 7 Sims with a peak accuracy of 99.43%. Our work, SyncNN achieves better accuracy of 99.17% for the same 7 Sims and achieves a peak accuracy of 99.63% with 100 Sims, which is more accurate than all prior FPGA implementations. After 100 Sims, the accuracy saturates and remains the same.

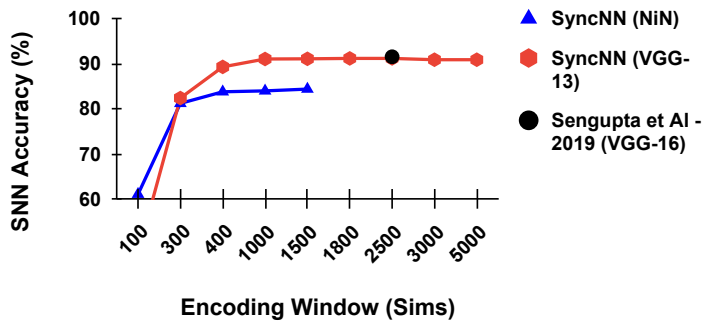


Fig. 7. Impact of encoding window on the accuracy for CIFAR-10 dataset

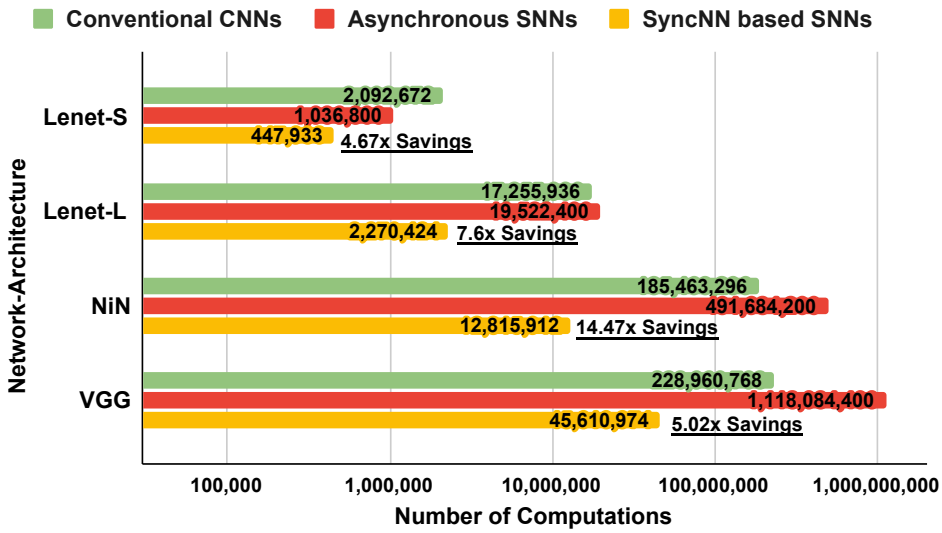


Fig. 8. Comparison of computation operations required in conventional CNNs, asynchronous SNNs, and our SyncNN based SNNs (savings is over CNNs)

For CIFAR-10 dataset, with larger networks like NiN and VGG, as shown in Figure 7, the required encoding window is very large to transmit the needed spikes in the network. In our SyncNN approach, the NiN achieves 88.1% accuracy and the VGG-13 network achieves 91.19% accuracy at 1,800 Sims. The recent study by Sengupta et al. [31], which explores converted SNNs on deeper networks, also confirms the need for a larger encoding window. Their work achieves 91.55% accuracy at 2,500 Sims with the VGG-16 network.

6.3 Computational Comparison for SyncNN

Finally, we compare the number of computing operations required in the conventional CNNs, asynchronous SNNs, and our SyncNNs in Figure 8. The detailed experimental setup will be explained in Section 6.1. Note that for asynchronous SNNs, we assumed an ideal case where all layers can execute in parallel and we only counted the computing operations in the largest layer. For CNNs and SyncNN based SNNs, we counted all computing operations in all layers since they execute layer by layer. Our SyncNN approach has a significant advantage over asynchronous SNNs, especially for deep networks (NiN and VGG) that require a large encoding window, since SyncNN requires only

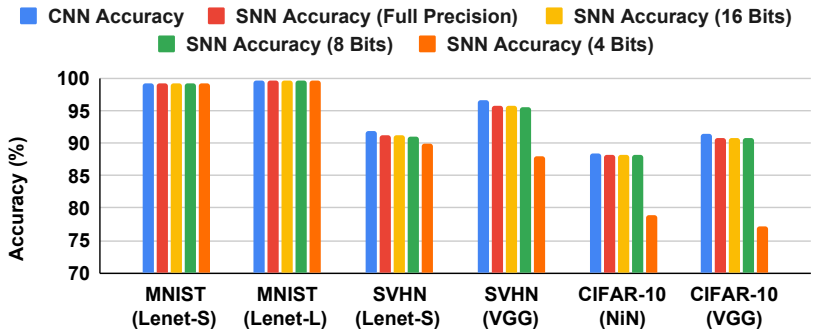


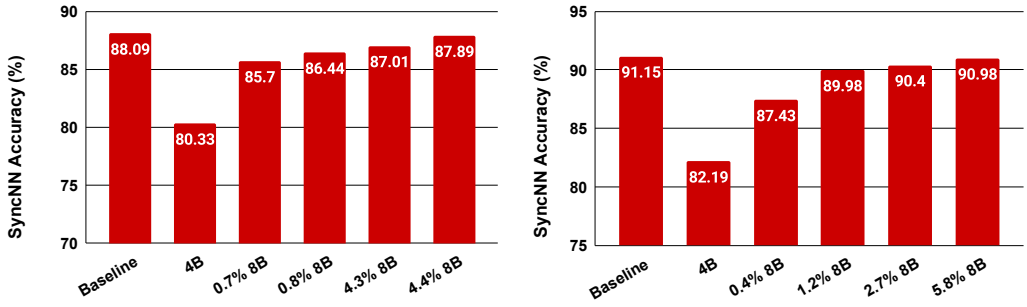
Fig. 9. Inference accuracy comparison of CNN and SyncNN based SNNs with different data precision

one timestep for all layers except the input layer. For CNNs, we do a theoretical estimate for the number of computations needed. This is feasible because, in CNNs, all the neurons are activated and computed. In the case of SyncNN and asynchronous SNNs, since they adapt the event-driven nature, we pass the input-set to estimate the total number of activated neurons in each layer and evaluate the number of computations needed. Also, within the same dataset, between different input images presented, the number of activated neurons varies. Therefore, we take an average value for the total number of neurons activated for the entire dataset and measure the results. Compared to conventional CNNs, our SyncNN based event-driven SNNs can reduce the number of computing operations by 4.67x to 14.47x. The savings in computation has direct relation to the storage needed. The total storage required can be categorized as two parts. One is the storage required for the network weights. Between SyncNN, asynchronous SNNs and equivalent CNN, the storage needed for the weights remain the same. The other is the storage required for the activations. In SyncNN, as discussed in Figure 8, we require reduced number of activations (computations), compared to both conventional CNNs and asynchronous SNN. In SyncNN, only a fraction of neurons are encoded and activated. Also, unlike the asynchronous SNNs, SyncNN does not run the entire network for multiple simulation steps and the encoding window is confined only to the Poisson Encoding stage. Therefore, the encoding window does not affect the storage needed and SyncNN would have a great savings over the equivalent CNN and asynchronous SNNs.

6.4 Accuracy of SyncNN with Quantization

In conversion based SNNs, the accuracy of the SNN network is dependent on the underlying CNN accuracy. The better we train the CNN model, the higher SNN accuracy is achieved. During the CNN training phase, we use data normalization, augmentation, regularization and dropout techniques [7, 13, 29, 31] to improve the accuracy of the network. We do not use batch normalization during training, which is an important CNN optimization, because it negatively affects the SNN accuracy after conversion. As summarized in Figure 9, the accuracy loss from converting CNN to SNN is negligible. When the compute units are offload to hardware, there is no drop in accuracy with 32 and 16 bits representation. For the LeNet networks there is negligible drop in accuracy for 8 and 4 bits representations. For NiN and VGG networks, there is negligible drop for 8 bits, but big drop for 4 bits. Therefore, we use 4 bits for LeNet networks and 8 bits for NiN and VGG networks in SyncNN hardware evaluation.

To further improve the accuracy for deeper networks like *NiN* and *VGG* at lower precision, we use the *Mixed Precision* technique as discussed in Section 5.1, where the first and/or last few layers of the network uses 8 bits weight precision and the remaining majority of the layers uses 4 bits weight precision. In Figure 10, we compare the SyncNN accuracy for NiN and VGG networks under



(a) CIFAR-10 (NiN Network) (b) CIFAR-10 (VGG Network)

Fig. 10. Inference accuracy of SyncNN using mixed 4-bit and 8-bit (8B) quantization

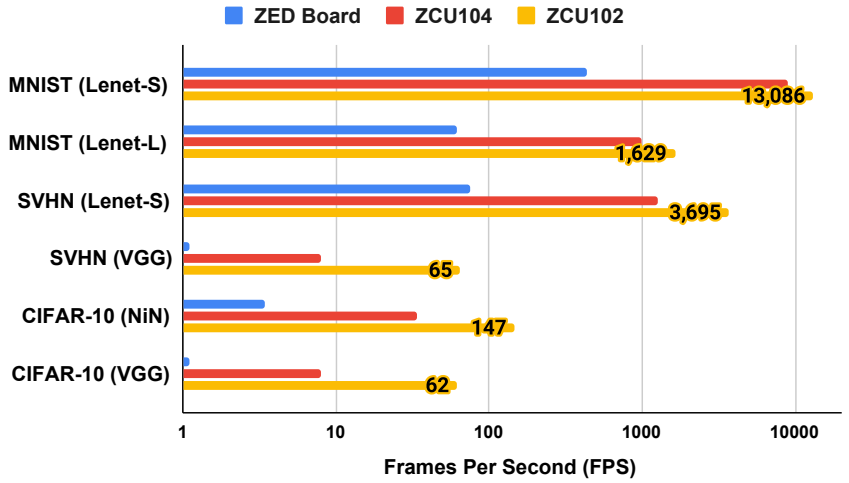


Fig. 11. Frames per second (FPS) for different networks and datasets running on different FPGA boards

1) pure 8-bit quantization, 2) pure 4-bit quantization, and 3) mixed 8-bit and 4-bit quantization, where the percentage of 8-bit weights is gradually increased until it reaches a similar accuracy to that of the pure 8-bit quantization. For NiN network with CIFAR-10 dataset, as shown in Figure 10a, with just 4.4% of the total weights in 8 bits precision and remaining with 4 bits, we achieve 87.89% accuracy, which is close to the accuracy of 88.09% using pure 8-bit quantization. And for VGG network with CIFAR-10 dataset, as shown in Figure 10b, with just 5.8% of the total weights in 8 bits precision and remaining with 4 bits, we achieve 90.98% accuracy, which is close to the accuracy of 91.15% using pure 8-bit quantization.

6.5 Overall Performance

Figure 11 shows the overall performance of each network with the best configuration on each FPGA board (configuration parameters are presented in Section 5.2). The performance of the networks varies between FPGA boards because of the difference between resources available. ZCU102 has the best performance as it is the board having the maximum resources and ZED board has the least performance since it is a very small FPGA board. The ZCU102 board is about 1.46x to 8.4x faster than the ZCU104 board and 26.7x to 60.9x faster than the ZED board for various networks.

On the ZCU102 board, the MNIST dataset for the Lenet-S network achieves a maximum of 13,086 FPS (frames per second) whereas it can achieve 1,629 FPS for the Lenet-L network. The huge difference in the performance is because, it is not possible to prefetch all the network parameters on chip for the Lenet-L network, especially because of its much larger dense layer. Therefore, more off-chip access is involved in Lenet-L which downgrades the performance of network. The SVHN dataset achieves a maximum of 3,695 FPS for the LeNet-S network and 65 FPS for the VGG network. The CIFAR-10 dataset achieves a maximum of 147 FPS for NiN network and 62 FPS for the VGG network.

For all the networks, their parameterized configuration and on-chip buffering is allocated in such a way to not use more than 80% of the computing and memory resources available in the board. For the MNIST dataset, we also present the performance/resource information in the Table 3.

6.6 Comparison with Related Work in SNN Acceleration

Most of the related studies are evaluated on the MNIST dataset with MLP or smaller CNN networks like LeNet. Table 3 summarizes all the recent related work of SNNs on FPGAs and neuromorphic hardware for the MNIST dataset, including the platform, frequency, training method, data precision, network model, inference accuracy, frames per second (FPS), FPS per DSP usage, FPS per 1K LUTs, and FPS per W (dynamic power) usage. The FPGA implementations, Minitaur [24], Han et al. [12], and Ju et al. [14], achieve 92%, 97.06%, and 98.94% accuracy with 108, 161 and 164 FPS, respectively. The prior neuromorphic hardware studies, Darwin [6], Stromatias et al. [34], and Esser et al. [8], achieve 93.8%, 95%, 95% accuracy with 6.25, 50, and 1,000 FPS respectively. More recently, a temporal encoding based SNN inference on Xilinx ZCU102 SoC board for MNIST image clasification achieves an accuracy of 99.2% and a simulation performance of 2,124 FPS [9] at 125MHz. Shown in Table 3, the same work also demonstrates superior performance of FPGA-based SNNs over that on CPU, GPU, Intel Loihi neuromorphic chip [9].

In terms of power measurements, we use a power meter to collect the results. In our experiments, we measure two power values. First, we measure the power of just loading the built SyncNN FPGA bitstream into the SD card and powering on the embedded ZCU102 ARM-FPGA SoC board without running any SyncNN network. We call this value as the *static power*, which is needed to run the embedded peripherals of the board. On average, the static power for SyncNN is 24.1W on the ZCU102 board. When running he network, irrespective of the size of the SyncNN network, the readings increase with an average of 0.4W and record 24.5W. Therefore, the power required to run any SyncNN network is around 0.4W, which we call it as the *dynamic power*. The prior studies [9] and [24] report their power requirements as 4.5W and 1.5W respectively, where [9] uses the same ZCU102 FPGA platform. There are no details about how these prior studies have measured their power; we believe it should be dynamic power in [9] and [24]. In order to have a uniform comparison with the prior studies, we use the dynamic power and calculate the FPS/W in Table 3. As shown in Table 3, in terms of FPS/W, SyncNN is orders-of-magnitude better than prior FPGA designs [9] and [24], and competitive to the True North ASIC design [21].

Our SyncNN framework stands out in terms of all four metrics—accuracy, FPS, FPS per DSP, FPS per 1K LUTs , and FPS per W (dynamic power) usage—for the MNIST dataset, compared to the prior neuromorphic hardware implementations, FPGA and GPU implementations. SyncNN achieves a maximum of 13,086 FPS at 99.3% accuracy for the same *Lenet-S* network and the same FPGA board used in [9]. SyncNN achieves 99.6% accuracy with 1,629 FPS on a bigger network *Lenet-L*. For both networks, the hardware runs at 200MHz and uses 4 bits precision for the network weights. In fact, SyncNN is the first work to explore 4 bits weights precision for SNNs on FPGAs. It is also the first SNN framework on FPGAs that supports deep CNN models like VGG and NiN.

Table 3. Comparison with related work for image classification using SNNs for the MNIST dataset. *FPS/W is measured using the dynamic power required to run the network.

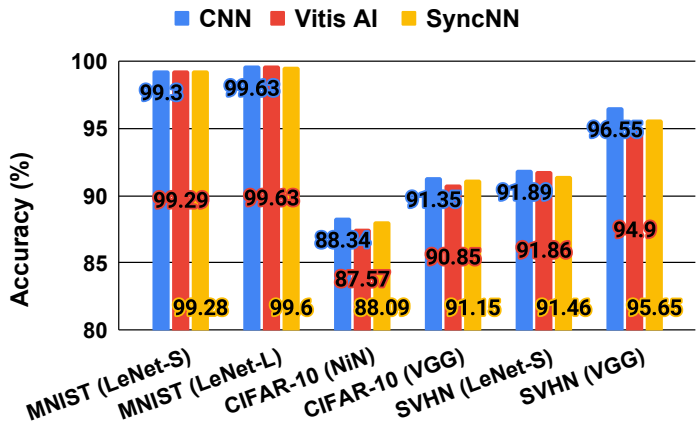
Work	Platform		Frequency	Training Method	Precision (bits)	Network	Accuracy (%)	FPS	FPS/ DSP	FPS per 1k LUTs	FPS/ W*
Stromatias et al. (2015)~ [34]	ASIC	SpiNNaker	150MHz	Spike Based	16	MLP	95	50	-	-	167
Esser et al. (2015) [8]	ASIC	True North	-	Offline Training	Binary	-	95	1,000	-	-	9,259
Darwin (2017) [6]	ASIC	-	25MHz	-	16	MLP	93.8	6.25	-	-	-
Minitaur (2014)[24]	FPGA	Spartan-6 LX150	75MHz	Spike Based	16	MLP	92	108	-	1.22	72.4
Han et al. (2020) [12]	FPGA	Xilinx ZC706	200MHz	Converted SNNs	16	MLP	97.06	161	-	29.92	-
Ju et al. (2020) [14]	FPGA	Xilinx ZCU102	150MHz	Converted SNNs	8	CNN (LeNet)	98.94	164	-	1.52	-
Fang et al. (2020) [9]	ASIC	Intel Loihi	-	Spike Based	16	CNN (Lenet-S)	99.2	671	-	-	178
	CPU	Intel i9-9900K	3.7GHz					100	-	-	1.72
	GPU	Nvidia RTX 5000	1.62GHz					864	-	-	14.1
	Edge GPU	Nvidia AGX Xavier	1.37GHz					211	-	-	15
	FPGA	Xilinx ZCU102 (Simulation)	125MHz					2,124	1.18	13.62	471
SyncNN (Our Work)	**FPGA**	**Xilinx ZCU102**	**200MHz**	**Converted SNNs**	**4**	**CNN (Lenet-S)**	**99.3**	**13,086**	**19.23**	**61.2**	**32,715**
						CNN (Lenet-L)	**99.6**	**1,629**	**2.94**	**7.25**	**4,072**

6.7 Comparison with CNN Acceleration in Vitis AI

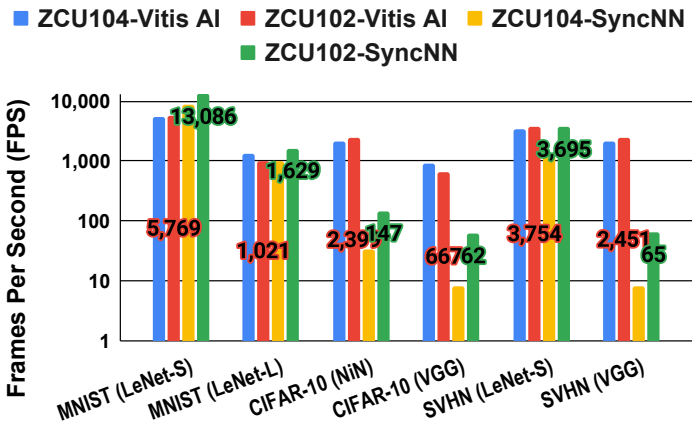
In Section 6.3, we observed that, SyncNN has about 4.67x to 14.47x savings than the conventional CNNs in terms of number of computing operations. Although there are great algorithmic savings, the event-driven nature of SNNs makes it challenging when accelerated on hardware. On the other hand, the memory access pattern and the computation pattern is more regular in CNNs, making the hardware acceleration better and easier. In this subsection, we experimentally compare SyncNN to the FPGA-accelerated CNN (before it is converted to our SyncNN). There are numerous studies involving the acceleration of CNNs on FPGAs. Here we use Vitis AI - Xilinx's state-of-the-art development platform for optimized AI inference on FPGA platforms [39]. Vitis AI supports mainstream frameworks like TensorFlow, PyTorch, Caffe and Keras and provides optimized models (compressed and quantized models at 8-bit precision) that are ready to deploy on Xilinx devices. We implement the same networks discussed in Section 6.1 and evaluate the CNN performance.

First, as shown in Figure 12a, we compare the CNN evaluation accuracy, Vitis AI hardware accuracy for CNN, converted SyncNN hardware accuracy. For smaller networks like *Lenet-S* and *Lenet-L*, both Vitis AI and SyncNN achieves almost the same accuracy as the software CNN accuracy. For deeper networks like *NiN* and *VGG*, for the CIFAR-10 and SVHN dataset, we see that the converted SyncNNs achieves slightly better accuracy than that of Vitis AI.

Next, as shown in Figure 12b, we compare the hardware performance in terms of Frames Per Second (FPS) between Vitis AI and SyncNN on ZCU104 and ZCU102 FPGA boards. In Vitis AI, we see that the performance of ZCU104 and ZCU102 FPGA boards almost remains the same. But in SyncNN, there is a drastic improvement in performance with ZCU102 board over ZCU104 board as the memory and computing resources of ZCU102 board is relatively higher than ZCU104 board. For smaller networks like *LeNet-S* and *LeNet-L* with the MNIST dataset, we see that *SyncNN* performs better with **2.27x** and **1.6x** speedup than the *Vitis AI*, respectively. This is because of two primary reasons. On one hand, SyncNN explores *4 bits* quantization for network weights for *LeNet-S* and



(a) Accuracy



(b) Performance

Fig. 12. Accuracy and performance comparison of CNN in Vitis AI and SyncNN

LeNet-L networks, but Vitis AI explores *8 bits* quantization for network weights (note that the Vitis AI engine is fixed to 8 bits and is not configurable). On the other hand, for smaller networks, most of the network parameters can be buffered on-chip reducing the off-chip access as discussed in Section 5.3. But for deeper networks like *NiN* and *VGG*, Vitis AI has better performance than SyncNN. This is because, for deeper networks, SyncNN is more difficult to buffer the irregularly accessed weights and has more off-chip access.

Moreover, we also compare the detailed power consumption and performance per watt (FPS/W in terms of dynamic power) results between Vitis AI and SyncNN on the ZCU102 FPGA board, as shown in Table 4. First, Vitis AI consumes a higher static power of 31.3W, since it has a larger FPGA bitstream. Second, unlike SyncNN, where the dynamic power usage remains constant for running different networks, Vitis AI has different dynamic power usage for different networks. Specifically, SyncNN runs at a constant dynamic power of 0.4W for all the networks. But, Vitis AI runs between a dynamic power of 1.3W for smaller LeNet network and 15.5W for deeper network like VGG, which is much higher than SyncNN. As shown in Table 4, SyncNN has better performance per watt

Table 4. FPS per (dynamic) Watt comparison of SyncNN and Vitis AI on the ZCU102 platform. We have recorded three different power values: 1) the total power is the power recorded when running the network; 2) static power is the power recorded when the FPGA board powers on and remains idle with the bitstream loaded in the SD card; and 3) the dynamic power is the power difference between the two. Note that FPS/W is calculated using the dynamic power.

Network/ & Dataset	Vitis AI				SyncNN			
	Total Power (W)	Static Power (W)	Dynamic Power (W)	FPS/W	Total Power (W)	Static Power (W)	Dynamic Power (W)	FPS/W
MNIST (LeNet-S)	32.6	31.3	1.3	**4,287**	24.5	24.1	0.4	**32,715**
MNIST (LeNet-L)	34.8	31.3	3.5	**375**	24.5	24.1	0.4	**4,073**
CIFAR-10 (NiN)	37.2	31.3	5.9	**355**	24.5	24.1	0.4	**368**
CIFAR-10 (VGG)	46.8	31.3	15.5	**58**	24.5	24.1	0.4	**155**
SVHN (LeNet-S)	34.1	31.3	2.8	**1,222**	24.5	24.1	0.4	**9,238**
SVHN (VGG)	45.9	31.3	14.6	**145**	24.5	24.1	0.4	**163**

results, which is **1.04x to 10.87x** better than Vitis AI across a wide range of networks and dataset using the same ZCU102 FPGA platform.

Therefore, as a final verdict, SyncNN would perform better than Vitis AI for networks where most of the network parameters can remain on-chip for a given FPGA board, as SyncNN requires less computing operations than CNNs. For larger networks, due to the random-access nature of SNN, it is difficult to buffer the network parameters, making the conventional CNN performance better than SyncNN. However, for all the networks, SyncNN performs better than Vitis AI CNNs in terms of FPS/W. This is mainly because, the SyncNN framework runs at a constantly much lower power for all the networks. Also, SyncNN based SNNs can achieve a similar accuracy to Vitis AI for image classification problems, and has more potential for neuromorphic datasets like N-MNIST, N-Caltech101 and DvsGesture, which we plan to evaluate in future work.

7 CONCLUSION

In this paper, we have proposed a novel synchronous approach called SyncNN, to accelerate event-driven rate encoding SNNs on FPGAs. First, we quantitatively compared the CNNs, asynchronous SNNs, and SyncNNs, to demonstrate the advantage of SyncNNs. Also, we prove that SyncNN is mathematically the same as the asynchronous approach. Second, we applied SNN-friendly quantization techniques, to reduce the computing operations and memory accesses. Moreover, we developed configurable and scalable computing engines on FPGAs to accelerate different network models of SNNs across various Xilinx ARM-FPGA SoCs. SyncNN is capable to run any deep networks on a given Xilinx ARM-FPGA SoC and it is the first work to explore NiN and VGG networks for SNNs on FPGAs. SyncNN achieves the state-of-the-art performance for MNIST dataset with 13,086 FPS, which is 6.16x faster than the previous state-of-the-art implementation for the same network configuration and same FPGA board. SyncNN reports the best accuracy of 99.6% for MNIST, which is so far the highest accuracy recorded for SNNs on any hardware implementation, to the best of our knowledge. Finally, we compare the performance of SyncNN with Vitis AI based CNNs and observed 2.26x speedup for the MNIST dataset. We have also open sourced SyncNN (https://github.com/SFU-HiAccel/SyncNN) to the community to stimulate more researches in the area of FPGA-based SNNs that has a high potential in future deep learning systems.

REFERENCES

[1] Guo-qiang Bi and Mu-ming Poo. 1998. Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type. *Journal of Neuroscience* 18, 24 (1998), 10464–10472.

[2] Sander M. Bohte, Joost N. Kok, and Han La Poutré. 2002. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing* 48, 1 (2002), 17 – 37.

[3] Olaf Booij and Hieu tat Nguyen. 2005. A Gradient Descent Rule for Spiking Neurons Emitting Multiple Spikes. *Inf. Process. Lett.* 95, 6 (Sept. 2005), 552–558.

[4] Romain Brette and Dan FM Goodman. 2012. Simulating spiking neural networks on GPU. *Network: Computation in Neural Systems* 23, 4 (2012), 167–182.

[5] Kit Cheung, Simon R. Schultz, and Wayne Luk. 2016. NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors. *Frontiers in Neuroscience* 9 (2016), 516.

[6] M.A. De, Shen JunCheng, G.U. ZongHua, Zhang Ming, Zhu XiaoLei, X.U. XiaoQiang, Qi Xu, Shen YangJing, and Gang Pan. 2017. Darwin: a Neuromorphic Hardware Co-Processor based on Spiking Neural Networks. *Journal of Systems Architecture* 77 (01 2017), 43–51.

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

[8] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. 2015. Backpropagation for Energy-Efficient Neuromorphic Computing. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc., 1117–1125.

[9] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu. 2020. Encoding, Model, and Architecture: Systematic Optimization for Spiking Neural Network in FPGAs. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[10] Haowen Fang, Amar Shrestha, Ziyi Zhao, and Qinru Qiu. 2020. Exploiting Neuron and Synapse Filter Dynamics in Spatial Temporal Learning of Deep Spiking Neural Network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*. 2771–2778.

[11] A. K. Fidjeland and M. P. Shanahan. 2010. Accelerated simulation of spiking neural networks using GPUs. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 1–8.

[12] J. Han, Z. Li, W. Zheng, and Y. Zhang. 2020. Hardware implementation of spiking neural networks on FPGA. *Tsinghua Science and Technology* 25, 4 (2020), 479–486.

[13] Eric Hunsberger and Chris Eliasmith. 2016. Training Spiking Deep Networks for Neuromorphic Hardware. *arXiv:1611.05141* (2016). arXiv:1611.05141

[14] Xiping Ju, Biao Fang, Rui Yan, Xiaoliang Xu, and Huajin Tang. 2020. An FPGA Implementation of Deep Spiking Neural Networks for Low-Power and Fast Classification. *Neural Computation* 32, 1 (2020), 182–204.

[15] Taro Kawao, Masato Neishi, Tomohiro Okamoto, Amir Masoud Gharehbaghi, Takashi Kohno, and Masahiro Fujita. 2016. Spiking neural network simulation on FPGAs with automatic and intensive pipelining. In *2016 International Symposium on Nonlinear Theory and Its Applications, NOLTA2016*. 202–205.

[16] Yann LeCun, Y. Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521 (05 2015), 436–444.

[17] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 2021. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, 105–115.

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

[19] Christian Mayr, Sebastian Höppner, and Steve Furber. 2019. SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning. *arXiv:1911.02385* (2019). arXiv:1911.02385 [cs.ET]

[20] S. McKennoch, Dingding Liu, and L. G. Bushnell. 2006. Fast Modifications of the SpikeProp Algorithm. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. 3970–3977.

[21] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.

[22] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeff Krichmar, Alex Nicolau, and Alexander Veidenbaum. 2009. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural networks : the official journal of the International Neural Network Society* 22 (08 2009), 791–800.

[23] Morcos, Benjamin. 2019. NengoFPGA: an FPGA Backend for the Nengo Neural Simulator. http://hdl.handle.net/10012/14923

[24] D. Neil and S. Liu. 2014. Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 12 (2014), 2621–2628.

[25] Danilo Pani, Paolo Meloni, Giuseppe Tuveri, Francesca Palumbo, Paolo Massobrio, and Luigi Raffo. 2017. An FPGA Platform for Real-Time Simulation of Spiking Neuronal Networks. *Frontiers in Neuroscience* 11 (2017), 90.

[26] Michael Pfeiffer and Thomas Pfeil. 2018. Deep Learning With Spiking Neurons: Opportunities and Challenges. *Frontiers in Neuroscience* 12 (2018), 774.

[27] Filip Ponulak and Andrzej Kasiundefinedski. 2010. Supervised Learning in Spiking Neural Networks with Resume: Sequence Learning, Classification, and Spike Shifting. *Neural Comput.* 22, 2 (Feb. 2010), 467–510.

[28] A. Rosado-Muñoz, M. Bataller-Mompeán, and J. Guerrero-Martínez. 2012. FPGA implementation of Spiking Neural Networks. *IFAC Proceedings Volumes* 45, 4 (2012), 139 – 144. 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control.

[29] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, and Michael Pfeiffer. 2016. Theory and Tools for the Conversion of Analog to Spiking Convolutional Neural Networks. *arXiv:1612.04052* (2016). arXiv:1612.04052 [stat.ML]

[30] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. 2010. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1947–1950.

[31] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. 2019. Going Deeper in Spiking Neural Networks: VGG and Residual Architectures. *Frontiers in Neuroscience* 13 (2019), 95.

[32] Amar Shrestha, Haowen Fang, Qing Wu, and Qinru Qiu. 2019. Approximating Back-Propagation for a Biologically Plausible Local Learning Rule in Spiking Neural Networks. In *Proceedings of the International Conference on Neuromorphic Systems* (Knoxville, TN, USA) *(ICONS '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 8 pages.

[33] Sumit Bam Shrestha and Garrick Orchard. 2018. SLAYER: Spike Layer Error Reassignment in Time. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., 1412–1421.

[34] Evangelos Stromatias, Dan Neil, Francesco Galluppi, Michael Pfeiffer, Shih-Chii Liu, and Steve Furber. 2015. Scalable Energy-Efficient, Low-Latency Implementations of Trained Spiking Deep Belief Networks on SpiNNaker. In *2015 International Joint Conference on Neural Networks (IJCNN)*. 1–8.

[35] A. Taherkhani, A. Belatreche, Y. Li, and L. P. Maguire. 2015. DL-ReSuMe: A Delay Learning-Based Remote Supervised Method for Spiking Neurons. *IEEE Transactions on Neural Networks and Learning Systems* 26, 12 (2015), 3137–3149.

[36] Amirhossein Tavanaei and Anthony Maida. 2019. BP-STDP: Approximating backpropagation using spike timing dependent plasticity. *Neurocomputing* 330 (2019), 39 – 47.

[37] Runchun M. Wang, Chetan S. Thakur, and André van Schaik. 2018. An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator. *Frontiers in Neuroscience* 12 (2018), 213.

[38] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. 2018. Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks. *Frontiers in Neuroscience* 12 (2018), 331.

[39] Xilinx. 2021. Vitis AI: Adaptable and Real-Time AI Inference Acceleration. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

[40] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2019. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2019), 2072–2085.