Evaluation and Acceleration of Spiking Neural Networks using FPGAs

by

Sathish Panchapakesan

BE, Anna University, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

© **Sathish Panchapakesan** 2021
SIMON FRASER UNIVERSITY
Fall 2021

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name: **Sathish Panchapakesan**

Degree: **Master of Applied Science**

Thesis title: **Evaluation and Acceleration of Spiking Neural Networks using FPGAs**

Committee:

Chair: Jie Liang
Professor, Engineering Science

Zhenman Fang
Supervisor
Assistant Professor, Engineering Science

Jian Li
Committee Member
Sr. Director, Futurewei Technologies Inc.

Arrvindh Shriraman
Examiner
Associate Professor, Computing Science

Abstract

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 thesis, we present a novel synchronous approach for rate encoding based Spiking Neural Networks (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 the same 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.

Keywords: Spiking Neural Networks; Hardware Acceleration; High-Level Synthesis; Image Classification; Machine Learning

Acknowledgements

SyncNN is the result of my years of hard-work at Simon Fraser University and abundance support from various people. Firstly, I am extremely thankful to my supervisor Dr. Zhenman Fang. Without him, it would have been impossible to reach this goal. He has not only helped me with technical doubts/guidance at any point of time, but also provided me constant support and motivation. I am proud to have such a supervisor and would always be happy to get his advice for any bigger decisions that I make further. Secondly, I take this opportunity to thank my co-supervisor Dr. Jian Li from Futurewei Technologies, for helping with various valuable feedback and suggestions amidst his busy schedule.

Most of my Master's program was in the Covid-19 time frame and was always working from home. Just to keep all the negatives a step away, it gave more flexibility and more productive time than working from lab. Thanks to the technology for keeping us all connected. Coming to the downside, during the first two terms at campus, I took interesting and challenging courses from Dr. Arrvindh Shriraman and Dr. Lesley Shannon. In person, I had the opportunity to discuss a lot with them about my work and get their perspective as well. Thanks to them. Covid-19 limited the chances to connect with them and also working from the beautiful campus up the mountain.

I would like to thank the lab group who joined along with me: Alec, Xingyu and Weihua, as well as the new: Kartik, Moazin, Kenny and Junzhe for supporting and helping at different times. Also, I would like to thank my best friend Spoorthy Gunda for helping and being there for me in this journey. Also, I would like to thank Mary, the graduate program assistant of the ENSC department, for helping me to clarify with even tiny doubts with her quick responses.

I acknowledge the support from NSERC, Huawei, Xilinx, and Nvidia for funding the work. I also thank Dr. Abhronil Sengupta and his group from Penn State University for helping us better understand training techniques for SNNs, and Kunpeng Xie from Nankai University for helping us collect data on Xilinx ZCU102 FPGA board.

Not but not the least, I would like to thank my parents: J.Panchapakesan and P.Vijaya for always being there for me. Thanks for fulfilling my dreams to pursue my Masters in a foreign land. Thanks for the belief you had in me. Thanks for cheering me up always whenever I go down. And thanks for being the best parents in the world.

Table of Contents

Declaration of Committee ii

Abstract iii

Acknowledgements v

Table of Contents vi

List of Tables ix

List of Figures x

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objective . . . 2
- 1.3 Contributions . . . 2
- 1.4 Summary . . . 3
- 1.5 Organization of Thesis . . . 3

2 Prelimnaries 5
- 2.1 Introduction to Neural Networks . . . 5
  - 2.1.1 Artificial Neural Networks . . . 6
  - 2.1.2 Convolutional Neural Networks . . . 7
- 2.2 Field Programmable Gate Arrays (FPGA) . . . 7
  - 2.2.1 Hardware Acceleration using FPGAs . . . 8
- 2.3 High-Level Synthesis . . . 9
  - 2.3.1 Optimizations in HLS . . . 9

3 SNN Background and Related Work 14
- 3.1 How SNNs Work and Their Computing Advantages . . . 14
- 3.2 Challenges and Advancements for SNN Accuracy . . . 15
- 3.3 Hardware Acceleration for SNN . . . 16
- 3.4 Goal of This Work . . . 17

4 Role of Convolutional Neural Networks **19**
4.1 Training CNNs 19
4.2 Network Conversion to SNNs 21
4.3 Vitis AI 22

5 SyncNN: Synchronous SNN Approach **24**
5.1 Rate Encoding based SNN Algorithm 24
5.2 Asynchronous Approach of SNNs 26
5.3 SyncNN: Synchronous Approach of SNN Acceleration 27
5.4 Neuron Encoding in SyncNN 28

6 When to Use SyncNN **29**
6.1 Accuracy Comparison with Asynchronous Approach 29
6.2 Computational Comparison for SyncNN 32
6.3 Hardware Perspective 32

7 Hardware Design of Asynchronous SNNs **33**
7.1 Computational Optimization 33
7.2 Memory Access Optimization 35
7.3 Design Automation 35
7.4 Challenges of Asynchronous Approach 38
7.4.1 Results from Automation Algorithm 40
7.4.2 Resource Utilization 41
7.4.3 Comparison with Related Work 41
7.4.4 Challenges of Asynchronous Approach 42

8 Hardware Design of SyncNN Framework **43**
8.1 Quantization 43
8.2 Computation Optimization 44
8.3 Memory Access Optimization 47

9 Experimental Results for Synchronous Approach **49**
9.1 Experimental Setup 49
9.2 Impact of Encoding Window 50
9.3 Accuracy of SyncNN with Quantization 51
9.4 Overall Performance 52
9.5 Comparison with Related Work in SNN Acceleration 53
9.6 Comparison with CNN Acceleration in Vitis AI 56

10 Conclusion **59**
10.1 Future Work 59

11 Publications **60**
11.1 Journal (Under Submission) 60
11.2 Full Conference Paper (Published) 60
11.3 Conference Poster (Published) 60
11.4 Open Source Software 60

Bibliography **61**

List of Tables

Table 4.1 Impact of CNN training methods on SNN 20

Table 7.1 Automation results breakdown for latency balancing, loop unrolling, and effective weight buffering 39
Table 7.2 Resource utilization (%) on ZCU104 board 41
Table 7.3 Comparison with related work 41

Table 9.1 Neural network configurations 49
Table 9.2 Comparison with related work for image classification using SNNs for the MNIST dataset 55

List of Figures

Figure 2.1	Hardware acceleration of neural networks using FPGA [33]	6
Figure 2.2	Image classification using ANN	6
Figure 2.3	Image classification using CNN	8
Figure 2.4	Flexiblity and efficiency of FPGAs	9
Figure 2.5	Working flow of HLS [49]	10
Figure 2.6	Function parallelism using HLS	11
Figure 2.7	Loops parallelism using HLS	11
Figure 2.8	Loops pipelining using HLS	12
Figure 2.9	Overall optimizations using HLS	12
Figure 3.1	Overview of the SNN working mechanism	15
Figure 4.1	Retrieving weights for CNN-SNN conversion	21
Figure 4.2	Vitis AI flow for FPGA implementation	22
Figure 5.1	Neuron encoding method in SyncNN	27
Figure 6.1	Comparison of computation operations required in conventional CNNs, asynchronous SNNs, and our SyncNN based SNNs (savings is over CNNs)	32
Figure 8.1	Overview of the hardware architecture for SyncNN	46
Figure 8.2	Different granularity to load weights depending on the size of the weights and available on-chip memory	48
Figure 9.1	Impact of encoding window on the accuracy for MNIST dataset	50
Figure 9.2	Impact of encoding window on the accuracy for CIFAR-10 dataset	51
Figure 9.3	Inference accuracy comparison of CNN and SyncNN based SNNs with different data precision	52
Figure 9.4	Inference accuracy of SyncNN using mixed 4-bit and 8-bit (8B) quantization	52
Figure 9.5	Frames per second (FPS) for different networks and datasets running on different FPGA boards	53
Figure 9.6	Accuracy and performance comparison of CNN in Vitis AI and SyncNN	57

Chapter 1

Introduction

Neural networks (NNs) have been widely used in many areas such as image classification, speech recognition, and automated control [19]. More recently, Spiking Neural Networks (SNNs), often referred to as the third-generation Neural Networks (NNs), have attracted increasing attention because they are more biologically plausible and have more potential for hardware acceleration [33]. SNNs process spikes based on the membrane potential of the neurons and are modeled in accordance to the actual neural system of the human brain [21]. 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 [33]. 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 [12, 2, 23, 34, 42, 1]. However, the temporal information processing capability limits the scope of exploring SNNs only for shallow networks [38]. 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 [9, 15, 36, 4]. More recently, the CNN-to-SNN converted models have achieved very good accuracy for deep networks such as VGG and ResNet [38]. In this work, we focus on the conversion based rate encoding SNNs to explore deeper networks on FPGAs.

1.1 Motivation

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 [11]. 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.

1.2 Objective

In this thesis, 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.

1.3 Contributions

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 [47] on the same Xilinx FPGA platforms.

Unlike prior FPGA studies [14, 16, 27, 11, 26] 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 [11], 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 [11], and 99.3% accuracy, which is higher than 99.2% in [11].

1.4 Summary

In summary, the main contributions of the thesis can be summarized as follows:

- 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 (8bits 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.

1.5 Organization of Thesis

The remainder of the thesis is organized as follows.
Chapter 2 gives prelimnary information about Neural Networks, FPGA and Hardware Accleration using High-Level Synthesis.
Chapter 3 describes SNN background and related work.
Chapter 4 explains the training mechanism followed for CNNs, how the model is converted

to SNNs, and finally evaluates the CNN performance using Vitis AI.

Chapter 5 presents the SNN algorithm, limitations of asynchronous SNNs, and our proposed synchronous SNN approach.

Chapter 6 proves the accuracy of SyncNN and discusses its algorithmic computation advantage and hardware implementation advantage.

Chapter 7 discusses the hardware optimizations and limitations to implement Asynchronous SNNs on Xilinx ARM-FPGA SoCs.

Chapter 8 presents the hardware optimizations to implement SyncNN on Xilinx ARM-FPGA SoCs.

Chapter 9 evaluates the accuracy and performance of SyncNN and compares it to state-of-the-art work.

Finally, **Chapter 10** concludes the thesis and also discusses about the possibilities of future work in this area.

Chapter 2

Prelimnaries

This chapter gives basic level information on Neural Networks, Field Programmable Gate Array (FPGA), and hardware design using High-Level Synthesis. Readers who are already aware of these information may skip this section. Figure 2.1 gives a high level overview of hardware acceleration of Neural Networks using FPGA. The different terms used in the figure would be explained in this section.

2.1 Introduction to Neural Networks

Artificial Intelligence is one of the common geek word that we frequently hear around. We all might have wondered about the ability of human beings; how fast we recognize things, do maths, hear something and process information. Artificial Intelligence is a stream in Engineering which poses challenges to human beings by creating a machine that can think itself and process information like a human being [19]. Though artificial intelligence has not reached the stage as they show the robots in the movies, it is taking a promising milestone to do several interesting tasks. Machine Learning is one subset of Artificial Intelligence, where the machine is made to learn for some particular application [33]. The application includes Image recognition, audio and video recognition, medical data centres, weather forecasting, autonomous driving and many. Unlike traditional application programming, we don't code the machine to do a specific job. In Machine learning, we make the machine learn about the application by presenting the training data from the application and the model parameters are adjusted while learning [21]. For example, if I have to add two numbers, I do not say, add 2 and 3. I just present the input data as 2 and 3 and the output as 5. When a lot of data is presented to learn in this fashion, the machine finally gets the ability to predict the addition function between these two input numbers. Once the machine is finished with learning, it is evaluated with a separate set of new testing data and the accuracy is measured. The more you make the machine learn, the more the machine performs. Neural Network is one such state-of-art model of Machine Learning which has been proven for multiple applications [19]. They are inspired from our actual brain in which there are millions of neurons connected

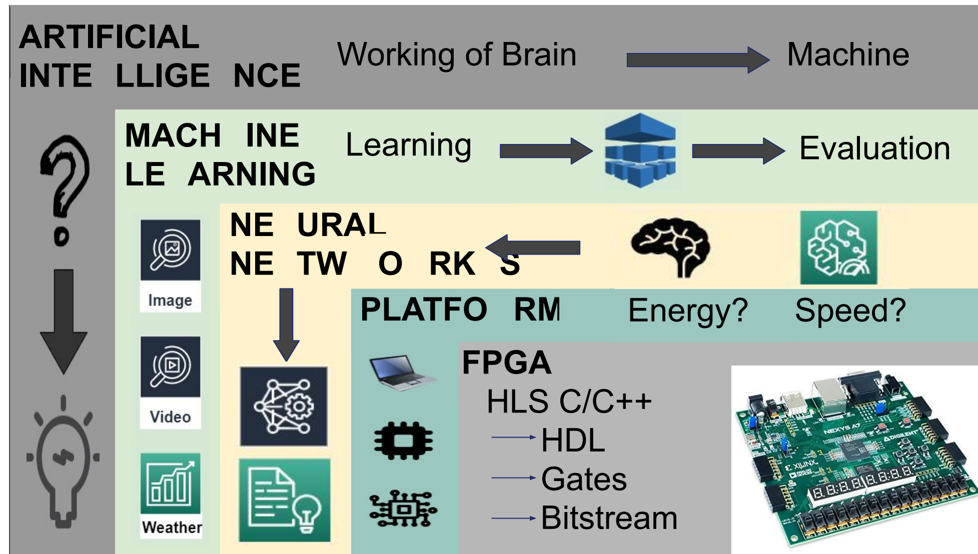


Figure 2.1: Hardware acceleration of neural networks using FPGA [33]

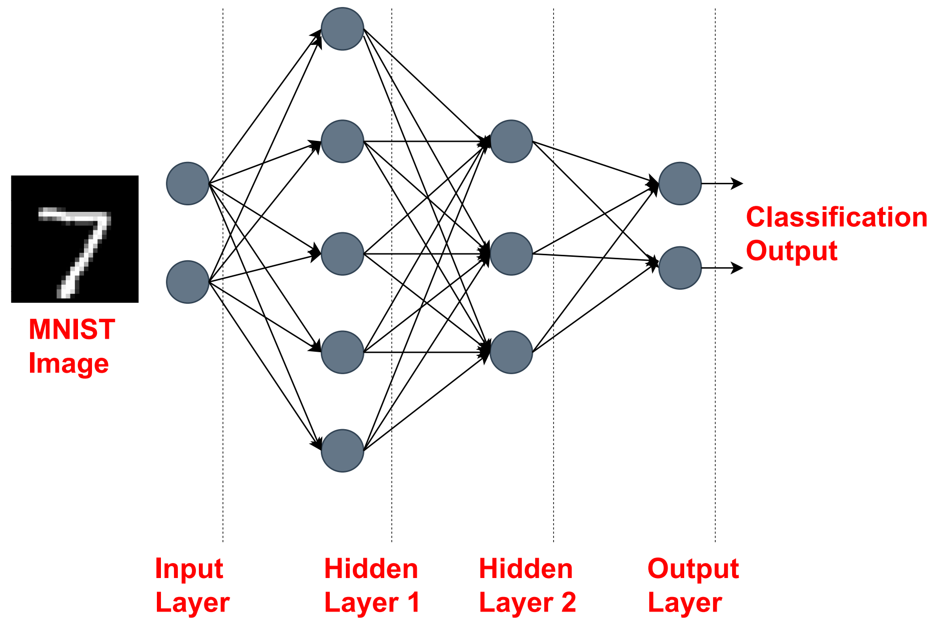


Figure 2.2: Image classification using ANN

by trillions of synapses [33]. The artificial neural network, takes a small scale model of these neurons and synapses and implements the machine learning model.

2.1.1 Artificial Neural Networks

The Artificial Neural Network (ANN) are the first generation of Neural Networks which takes a small scale model of these neurons and synapses and implements the machine learning model. There are also called as Multi-Layer Perceptrons (MLPs) and all the neurons in one layer are connected to the neurons in the next layer. This is in correlation with the hu-

man brain where each connection i.e. the synapses in the brain can transmit signal to other neurons. The signal in ANN is a real number and the output of each neuron is a non-linear function (activation function) of the accumulated values of the inputs. There are multiple types of Activation Functions including ReLu, Sigmoid and TanH. The synapse between the neurons is associated with a weight which is the learning parameter of the neural network. The strength of the connection is defined by the magnitude of the weight associated with the connection. As the neurons are arranged in layers, different layers performs different transformations on their inputs. Signals are transferred from input layer to the output layer and the neuron having the maximum value associated with it in the final layer corresponds to the output. As shown in Figure 2.2, the network uses a 28*28 image from the MNIST dataset as input and the pixel intensities are presented to the input layer. The network has two hidden layers and finally the output layer corresponds to the the classification output.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks commonly called as CNNs or ConvNet are the next generation of Neural Networks where the input features slides along the shared weights architecture of the filters or convolutional kernels and produces the output feature maps. In the multi-layer perceptrons, due to the full connectivity of the neurons, it often learn to overfitting of data which means that the network is over trained to perform well only for the training set and would not perform as expected in the testing dataset. To resolve this issue, the convolutional networks resembles the organization of the animal visual cortex where the individual cortical neurons responds to only a restricted region of the visual field called as the receptive field. Therefore, CNNs also have input layers, hidden layers and the output layer and the kernels are the trained parameters in the network. As shown in Figure 2.3, CNNs not only has convolutional layers, but also has pooling and fully connected layers. Convolutional layers convolve the input and the kernel to produce the output to the next layer. Pooling layers reduces the dimensions of data by combining the outputs of neurons at one layer into a single neuron in the next layer. Typically the pooling layer combines small clusters of tile size 2*2. Finally, the fully connected layers are the same multi-level perceptron or the artificial neural network. Overall, CNNs are widely used neural networks and have seen great success in the area of image and video recognition and classification, recommender systems, natural language processing (NLP) medical image analysis and image segmentation.

2.2 Field Programmable Gate Arrays (FPGA)

Field Programmable Gate Arrays, commonly called as FPGAs, are customizable and can be reconfigured *in the field*, without any modifications to the device or returning to the manufacturer. The important components of a FPGA chip are Programmable logic blocks, Pro-

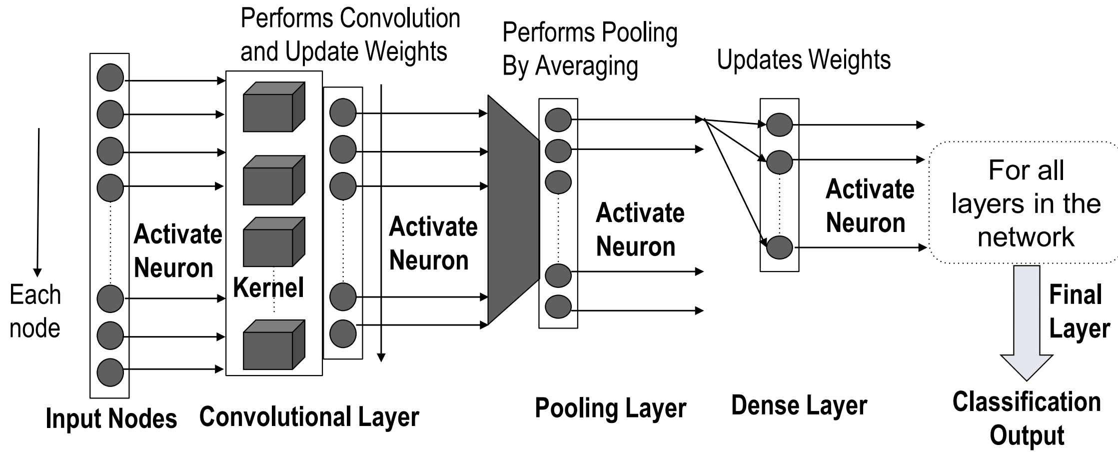


Figure 2.3: Image classification using CNN

grammable I/Os, and Programmable interconnects. Also FPGA consists of other systems like DSP for specific specialized Digital Signal Processing operations, embedded memory like BRAM/URAM, High Speed I/O, CPU an Clock Circuitry.

The logic blocks can be configured to implement and perform any simple logic function like AND, OR, or XOR to complex combinational functions. The logic block consists of logic cells such as look up tables, flip-flops and MUX. Every vendor uses different name for their logic blocks. Xilinx calls it CLB and Altera calls it LAB. For example, each CLB in Xilinx Ultrascale+ VU9P consists of eight 6-input LUTs and sixteen flip-flops [46]. For a k-input LUT, we can implement 2^k logics. Due to reconfigurable nature of FPGAs, it has been widely used in a wide range of applications from embedded computing to data-center computing. Generally implementing on DSP is much faster than implementation on LUT. In Ultrascale+ platforms, Block RAMs (BRAMs) has 36K bits of storage with two separate read/write ports with configurable bit width of 36K x 1, 18K x 2, 9K x 4, 4K x 9, 2K x 18 or 1K x 36 [46]. If there is a need for more ports, then the arrays is typically scattered to multiple BRAM banks. On the other hand, UltraRAMs (URAMs) has 288K bits of storage in a single block, but has a fixed bit width of 4K x 72 bits [46].

2.2.1 Hardware Acceleration using FPGAs

CPU/Microprocessor performance has been drastically increasing based on the size of the transistor being used. As the scaling of transistors reduces every two years, according to Moore's law, it not only doubles the transistor density but also increases the energy and power saving drastically, which lead to many architecture innovations and has been majorly used for running many complex architectures on various domains like Neural Networks. Unfortunately, as the transistor size reduces every generation according to Moore's law, the supply voltage scaling could not be further reduced, since the leakage power kept increasing. This impact created a slow down of the transistor scaling according to Moore's law. The need

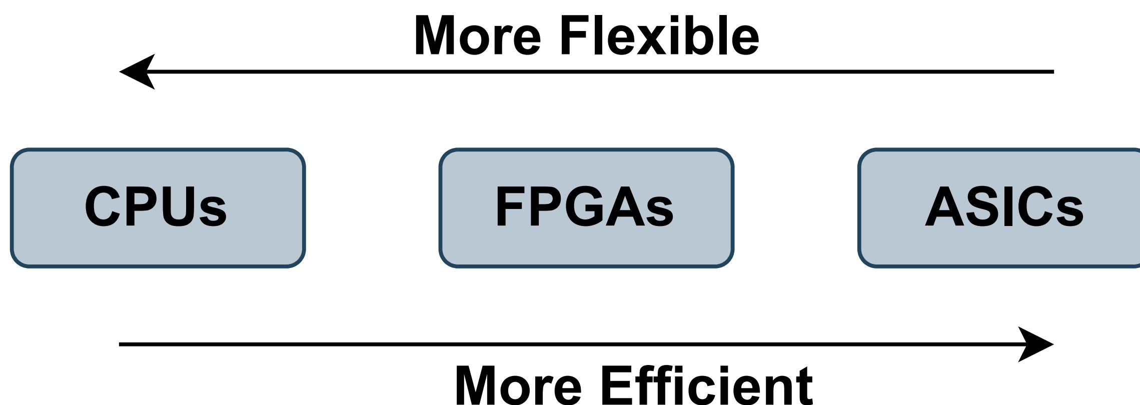


Figure 2.4: Flexiblity and efficiency of FPGAs

for better improvement in performance is never ending, where we have dedicated hardware called ASICs which is more efficient. But, ASICs do not have the flexibility to re-configured. Finally, as shown in Figure 2.4, FPGAs have more flexibility than ASICs as they can be reconfigured and are more efficient than CPUs. Therefore, FPGAs stands out as a right candidate for Customizable Computing using Hardware Acceleration.

2.3 High-Level Synthesis

In general, hardware design for FPGAs are written using Hardware Description Language (HDL) like VHDL and Verilog. For larger designs, it poses challenge in design and verification at the HDL level, especially when the target is to accelerate the design time and time to market [49]. On the other hand, High-Level Synthesis (HLS) raises the abstraction level from Register-Transfer level (RTL) to Software-Defined Hardware (SDH). HLS leverages C/C++ programming languages to design and program FPGAs which translates to HDL languages.

As shown in Figure 2.5, the algorithm specification is done in C, C++ and/or OpenCL languages. Vivado HLS explores the microarchitecture and converts to RTL Implementation. Finally the RTL implementation is translated to System IP comprising of gates and connections.

2.3.1 Optimizations in HLS

Therefore, the High-Level Synthesis (HLS) allows the hardware accelerator to be designed entirely in high-level language like C/C++ and synthesized to a pipeline hardware. The main advantage of HLS is that the accelerator permits a wide variety of hardware implementations, with software changes alone. For example, the neural network accelerator solution is

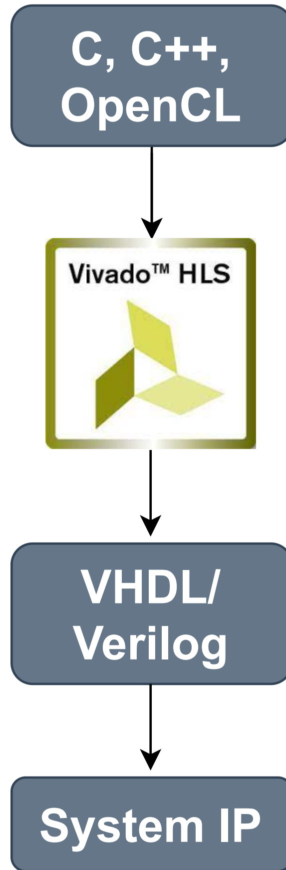


Figure 2.5: Working flow of HLS [49]

not tied to a particular neural network architecture; rather, it is configurable in software, thereby permitting its use for acceleration in a range of neural network architectures.

All the optimizations discussed in this chapter are specific to Vivado/Vitis HLS [6],[7],[18]. The optimizations using HLS, can be categorized as:

- Computation Optimization
  - Latency Optimization
  - Throughput Optimization
- Memory Optimization
  - Data Access Optimization
  - Data Packing Optimization

Latency Optimization of Functions. The high level code consists of various functions and loops. Firstly, by default, Vivado/Vitis HLS allows multiple functions to run in parallel if they do not have data dependency between them. As shown in Figure 2.6, the functions (FunctionA, FunctionB, FunctionC) runs in parallel since there is no data dependency

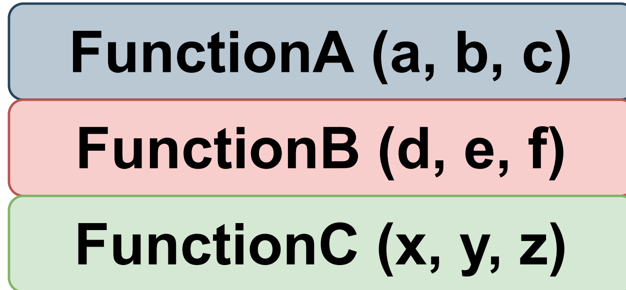


Figure 2.6: Function parallelism using HLS

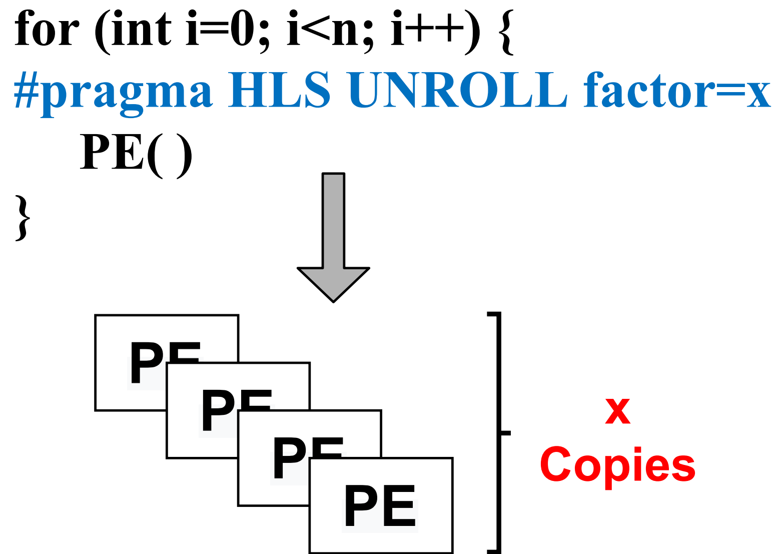


Figure 2.7: Loops parallelism using HLS

between them. Also, to make sure that the function doesn't gets inlined to the calling function, we use #pragma HLS inline off.

Latency Optimization of Loops. To explore parallelism between different iterations of the loop, Vivado/Vitis HLS uses #pragma HLS UNROLL factor=x, where x denotes the number of PEs out of the total iterations of the loop that are running in parallel. The factor is decided based on the available resources in the board. As shown in Figure 2.7, we parallelize a portion *'x units'* of the loop *'n iterations'* by generating multiple PEs running in parallel.

Throughput Optimization of Loops. To improve the throughput of the loops, we pipeline the loops using #pragma HLS pipeline II=1. As shown in Figure 2.8, before we pipeline the loop, once the read, compute and store for iteration i completes its operation, iteration i+1 begins. After we pipeline the loop. iteration i+1 begins its operation while iteration i is performing. The best initiation interval (II) is 1, where the next iteration begins

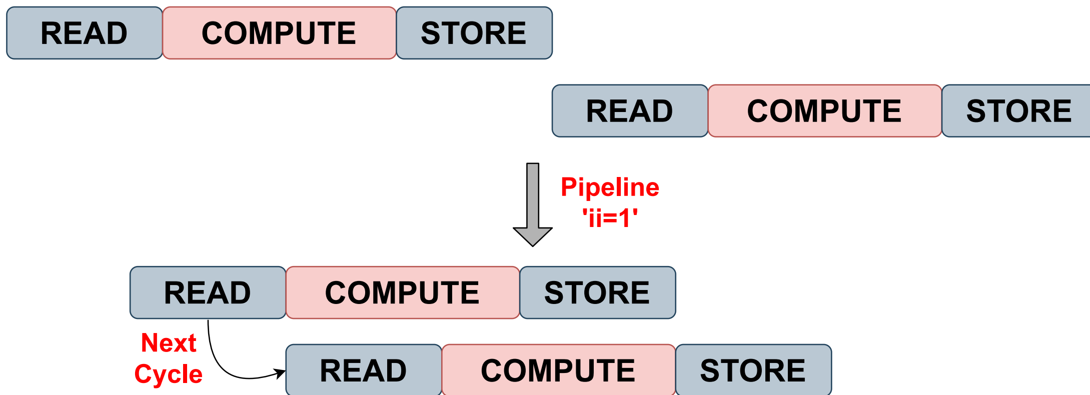


Figure 2.8: Loops pipelining using HLS

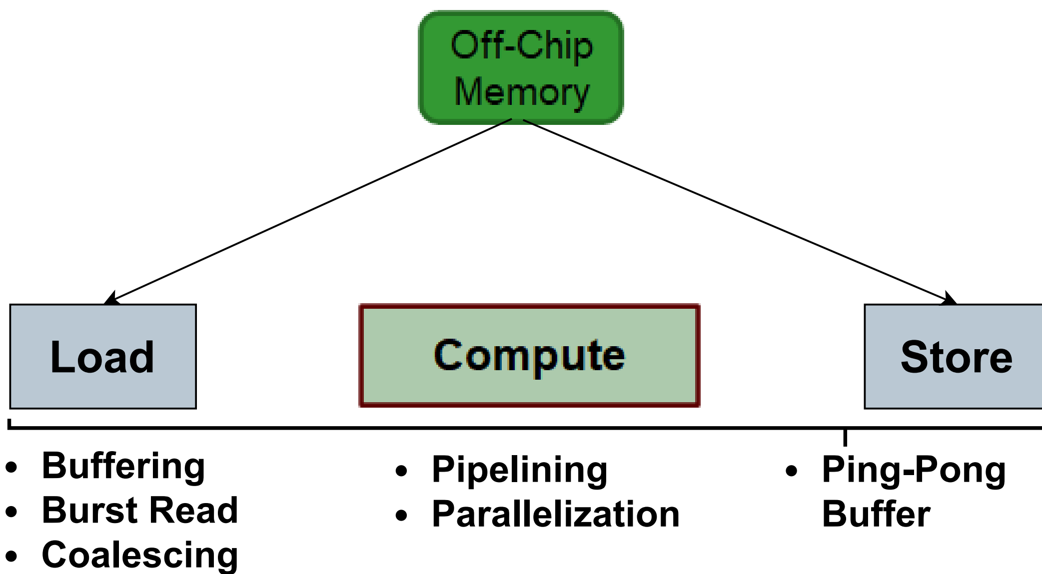


Figure 2.9: Overall optimizations using HLS

the operation immediately at the next cycle. When there is no data dependency between the iterations, we achieve the best II as 1.

Data Access Optimization using Array Partitioning. To successfully unroll the loops, as discussed in Latency Optimization of Loops, we have to make sure the PEs at the different loop iterations have no data dependency and therefore can run in parallel. Suppose, if use arrays and access the array at different iterations, we should make sure the access is happening from different on-chip banks (BRAMs/URAMs). To ensure that, we use array partitioning pragma to partition the array to different BRAM/URAM banks, so that we can access them in parallel. The partitioning can be done in cyclic or block access pattern based on the data access in the loop.

Data Access Optimization using Burst Read. We know that when the off-chip access increases, it reduces the performance. For random access, there is no way to skip the off-chip access. But, for access from continuous memory locations, we burst-read from off-chip memory by pipelining the loop which achieves II as 1.

Data Packing Optimization. (Coalescing) We pack the data for two reasons. Firstly, to fully utilize the off-chip bandwidth, we pack the data to maximum off-chip bandwidth and then read the data to the on-chip memory. Also, for the on-chip UltraRAM (URAM) memory, it has a fixed bit size of 72bits. To effectively utilize the URAMs, we pack the data and store it. Vivado/Vitis HLS supports *range()* function to pack the data.

Ping-Pong Buffering. As shown in Figure 2.9, the problem is considered as three stages - *Load, Compute and Store.*At load stage, we buffer the data to the on-chip memory with burst reading and Coalescing. Similarly, at the store stage, we write back the data from the on-chip memory in the similar fashion. At compute stage, we explore pipelining and parallelization. Between the stages, we parallelize the load, compute and store units by using ping-pong buffering. At each iteration, the stages operate using different copies of arrays to enable to parallelism between them.

Chapter 3

SNN Background and Related Work

In this chapter, 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.

3.1 How SNNs Work and Their Computing Advantages

Spiking neural networks (SNNs) [33, 21] 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 3.1, 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 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 [33].

There are several models of SNNs, like the integrate-and-fire (IF), leaky-integrate-and-fire (LIF), spike response, Izhikevich, and Hodgkin-Huxley models[32][5][9][44]. In this work, 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 chal-

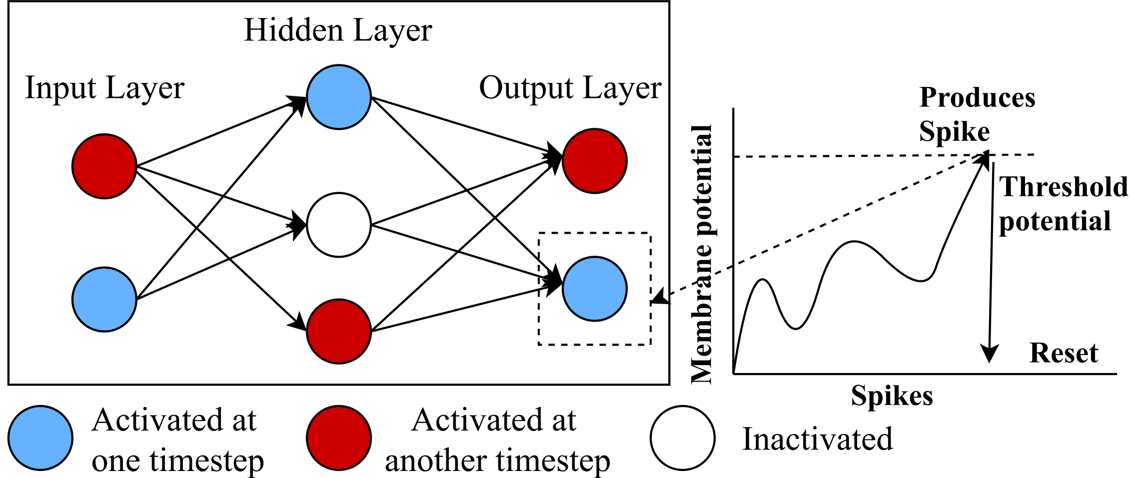


Figure 3.1: Overview of the SNN working mechanism

lenges 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 [21], shown in Figure 3.1. 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 3.1, 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.

3.2 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 [23] 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[34] and DL-ReSuMe[42] 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 [38].

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 (integrate-and-fire) units. And the activation of neurons in CNNs is proportional to the firing rate of SNNs [9, 15, 36, 4]. 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 [9, 15, 36, 4]. Recently, the conversion based SNN models have been studied on complex VGG and residual networks [38] with high accuracy.

In this work, we use the conversion technique and have applied the ReLu activation, weight normalization, zero bias and other optimizations discussed in [38, 9, 15, 36] during the conversion. Our main focus is on the SNN inference stage on edge devices.

3.3 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, 25, 13].

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 implementation designed with 384 cores and can support up to 200K Neurons

and 45M Synapses [37]. 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 [41]. The recent upgrade of SpiNNaker2 is a 10M-core machine in 22nm FDSOI with additional numerical accelerators [22]. The TrueNorth chip from IBM contains 4,096 cores supporting 1M neurons and 256M synapses [24].

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 [35], 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 spike-timing-dependent-plasticity (STDP) rule for learning. A fully connected Izhikevich model of SNN for 1,440 neurons has been realized on Xilinx Virtex 6 device [32]. A LIF model of SNN to simulate 20 million to 2.6 billion neurons was realized with Altera Stratix V FPGA [44]. A highly pipelined SNN model using HLS has been realized for 256 neurons which uses DSNN network and Hebbian learning mechanism [17].

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 [14]. 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 [16].

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 [11] 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.

3.4 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 thesis 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 integrate-and-fire (IF) SNN inference to run deeper networks like NiN and VGG, especially designed in high-level synthesis (HLS) 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 9.5.

Chapter 4

Role of Convolutional Neural Networks

As discussed earlier, the rate encoding based SNN uses Covolutional Neural Networks (CNN) in training phase and we use the same network topology for SNN based encoding mechanism in the evaluation stage. In this chapter, we discuss the training of the network using CNN and how to obtain the trained parameters for the evaluation phase. Finally, also to compare the performance of CNN with SNN, we use Xilinx Vitis AI to implement CNN model on FPGA.

4.1 Training CNNs

During training, in our work, we cover the feed forward networks, i.e. one layer is connected to next layer and there is no feedback from the next layers back to the previous layer. We take a batch of training data and pass it forward to the network. Once the output is obtained in the final later, we then compare the output predicted to the actual output and check how close the predicted values are from the actual labels. Based on this difference, we change the weights value in the network such that, in the next feed forward pass, the network prediction gets closer to the actual values. By repeating the process for all the batches for multiple times, the network trains to predict output closer to the actual output. When we pass all the batches i.e. the training set once through the network, then *one epoch* is completed. Based on the network size and the input presented, the number of epochs used in the training stage varies.

There are several methods and optimizations available for training the Convolutional Neural Networks which impacts the accuracy of the network. To use the same model in SNN inference, it is not possible to use any method or optimization in CNN training. We have to follow certain architectural constraints studied in SNN literature [38, 9, 15] while training CNNs in order to use it for SNN inference.

Table 4.1: Impact of CNN training methods on SNN

S.No	**Optimizations**	**CNN Code**	**Impact on SNN**
1	Kernel Initializer	HeNormal	Working
		Glorot Uniform	
2	Image Generator	Random input addition	Working
		Folding	
		Shuffling	
3	Optimizers	SGD	Working
		Adam	
4	Loss	Cross Entropy	Working
5	Activation	ReLu (from SNN Literature)	Working
6	Final Layer Activation	Softmax	Working
7	Dropout Layer	After Conv Layer	Working
8	Data Normalization	Input Data/Max. Intensity	Working
		Input Data-Mean/SD	Not Working
9	Bias Support	Bias=False	Working
		Having Bias	SNN literature says no.
10	Batch Normalization	Having BN Layers	

Bias in Neural Networks. In rate-encoding based SNNs, for ANN-SNN conversion schemes, the networks are trained without bias term [9]. This is because of the activation format of SNN, where we accumulate the neuron threshold with weight and check whether it has reached threshold potential. Having an extra bias term would expand the parameter space exploration making the ANN-SNN conversion process difficult. Also, since we have bias less neural networks, it is not possible to use Batch Normalization technique as a regularizer. This is because, batch normalization technique biases the input to the every layer of the network such that every layer has input with zero mean. Instead, dropout is used as regularizer technique.

Pooling Layer. Typically Convolutional Neural Networks consists of Pooling layers after a convolutional layer or a bunch of convolutional layer to reduce the size of the convolution output maps. The two widely used options for pooling are either max-pooling or average-pooling. Max-pooling takes the maximum neuron output over the pooling window and the average-pooling takes the average pooling operation over the pooling window. Since the neuron activation in SNNs are binary, taking max-pooling would affect the accuracy as there is significant loss of information to the next layer [9]. Therefore, we use average pooling mechanism during the training stages.

In our work, we follow the literature and use the same. Apart from that, we summarize the other different methods and optimizations explored in the CNN training, as shown in Table: 4.1 and observe its impact on SNN inference. We explored different kernel initializa-

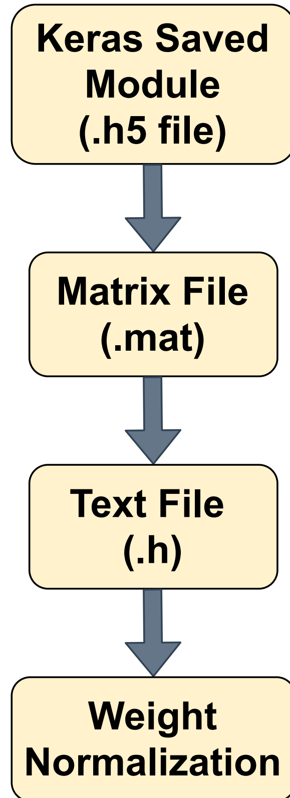


Figure 4.1: Retrieving weights for CNN-SNN conversion

tion methods, modification of image dataset (creating random inputs, folding and shuffling), optimizer function to update the weights after measuring the loss, dropout layers to skip some connections during the training, data normalization to normalize the input data before presenting to the network and batch normalization to normalize the output data of every layer before presenting to the next layer. In our observation, most of the methods in CNN training worked on SNN. But, few methods like data normalization using (Input Data - Mean /Standard Deviation) does not work on SNN with Poisson input generation.

4.2 Network Conversion to SNNs

After we train the CNN network, the next step is to retrieve the trained parameters from the network. Since the networks are bias less [9], network weights are the only trained parameters of the network. For the training of CNNs, we use Keras framework in Python and save the module as (.h5) file. From the .h5 file, as shown in Figure 4.1, we retrieve the network weights to a text file, so that we can use it in our C/C++ version of inference code. Once we obtain the weights, we do not use the weights directly on SNN inference. We follow some "weight-normalization" technique as used in SNN literature to reduce the conversion loss [38, 9, 15]. The approach followed is to set the neuron threshold of the layer equal to

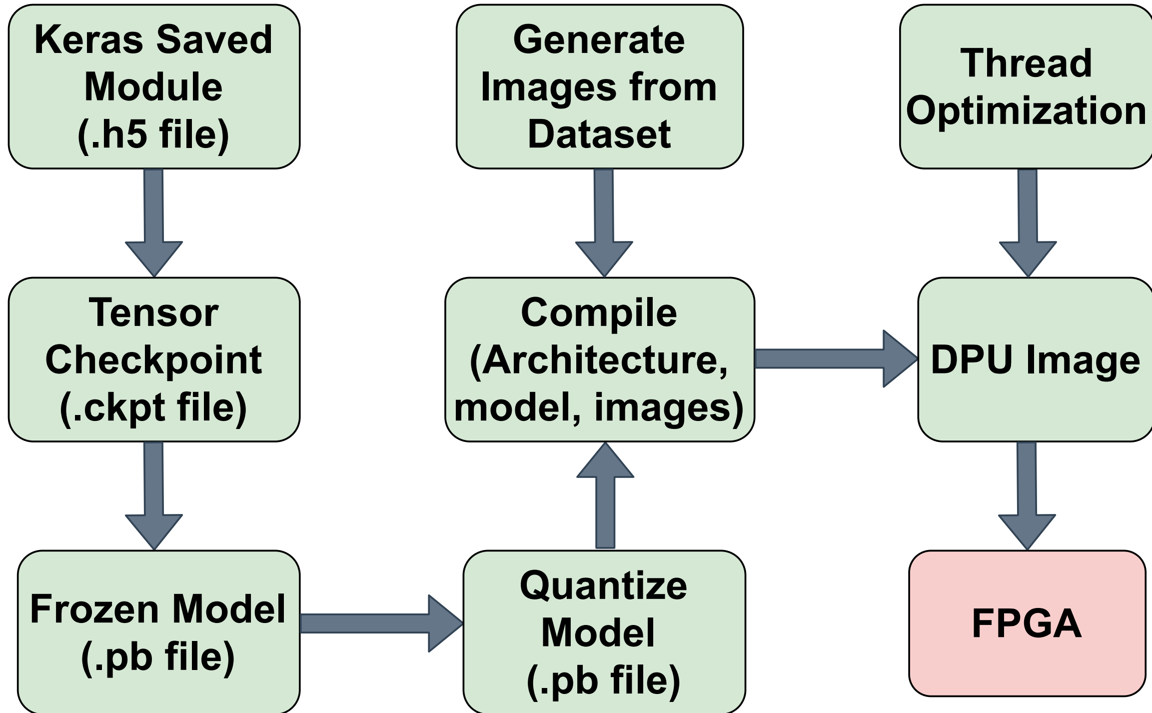


Figure 4.2: Vitis AI flow for FPGA implementation

maximum activation of all the ReLUs in that corresponding layer. The maximum activation is obtained by passing the training set through the trained CNN. On the other hand, we can keep the threshold as 1 and normalize the weights of the layer by the maximum activation. Both the approaches are mathematically the same.

4.3 Vitis AI

After we discussed the role of CNNs in training for SNN inference, the next role of CNN is to measure the inference performance and compare it with SNN. 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 [47]. Vitis AI supports mainstream frameworks like TensorFlow, PyTorch, Caffe and Keras and provides optimized models that are ready to deploy on Xilinx devices. As we used Keras framework in our training, we used the same saved model and target to run it on FPGA platform using Vitis AI.

Vitis AI has multiple in-built scripts which can be run in the Vitis AI docker. As shown in Figure 4.2, it takes the saved Keras model (.h5 file) and generate the tensor checkpoint (.ckpt file). From the checkpoint, we port the input and output node and generate the frozen model (.pb file). Next, the frozen model is quantized to lower precision and generate quantize frozen model. Vitis AI allows to quantize up to 8-bits precision. Next, we present

the image dataset, quantized frozen model and the architecture of FPGA to the compiler script and generate the DPU image. We format the SD card and flash the DPU image to it. Finally, we run the DPU image on the target FPGA board. While running it on FPGA, we can control the number of threads and obtain the performance of the network. The number of threads is varied from 1 to 10 and depending on the network, the thread number that reports the best performance varies.

Chapter 5

SyncNN: Synchronous SNN Approach

In this chapter, 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.

5.1 Rate Encoding based SNN Algorithm

Algorithm 1 presents an overview of the conventional rate encoding SNN algorithm based on the integrate and fire (IF) model and Poisson spike generation [9]. 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 (Poisson Spikes - pSp*, Neuron Spikes - nSp*) in that

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++
```

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 3.1, 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 previous layer's `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:

$$T_{naive_sync} = Sims * (PE + \sum_{l=1}^{NL}(NE[l] + SA[l])) \tag{5.1}$$

5.2 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:

$$T_{async} = \sum_{s=1}^{Sims} \max(SA_s[1], SA_s[2], ..., SA_s[NL]) \tag{5.2}$$

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 9.2. Based on Equation 5.2, the large encoding window ($Sims$) limits the performance.

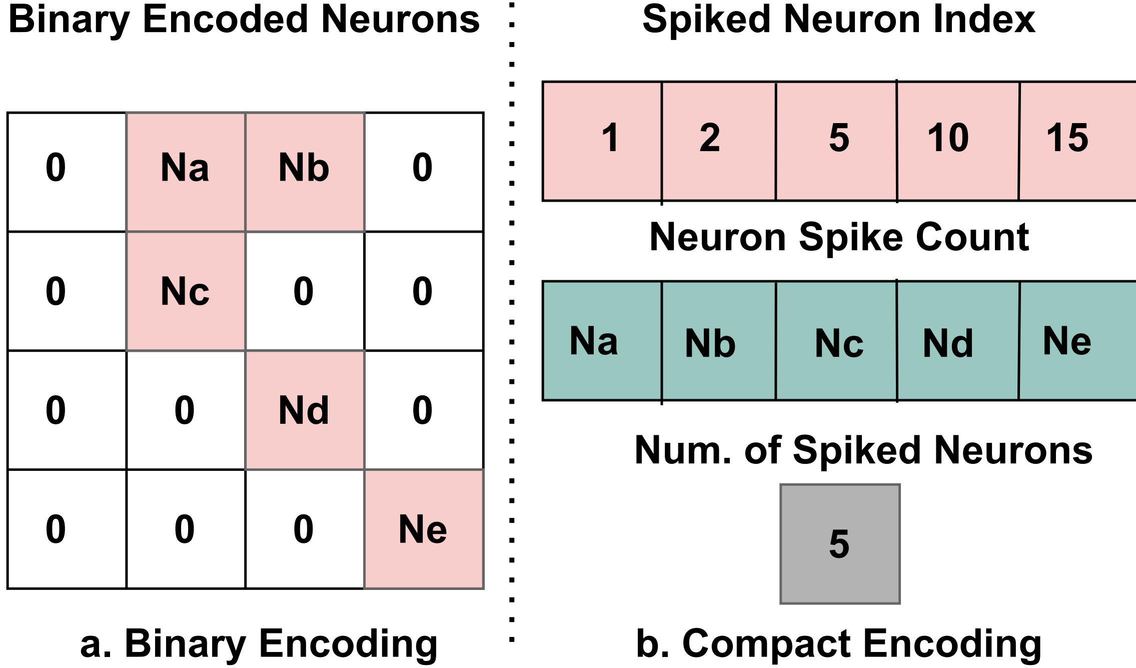


Figure 5.1: Neuron encoding method in SyncNN

5.3 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 `POISSON_ENCODING` input layer 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. 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 6.1. 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:

$$T_{SyncNN} = (Sims * PE) + \sum_{l=1}^{NL}(NE[l] + SA[l])) \tag{5.3}$$

5.4 Neuron Encoding in SyncNN

The most commonly used neuron encoding technique is the binary encoding (as shown in Figure 5.1.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 5.1.b.

Chapter 6

When to Use SyncNN

In this chapter, 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.

6.1 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,

$$SyncNN.PE_n == \sum_{s=1}^{Sims} (Async.PE_n^s) \tag{6.1}$$

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:

$$Sync.SA_j = \sum_{i=1}^{EN} Sync.PE_i * w_{ij} \tag{6.2}$$

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

$$Async.SA_j^s = \sum_{i=1}^{EN} Async.PE_i^s * w_{ij} \tag{6.3}$$

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:

$$\begin{aligned}
\sum_{s=1}^{Sims} Async.SA_j^s &= \sum_{s=1}^{Sims} \sum_{i=1}^{EN} Async.PE_i^s * w_{ij} \quad (from\ Equation\ 6.3) \\
&= \sum_{i=1}^{EN} \sum_{s=1}^{Sims} Async.PE_i^s * w_{ij} \\
&= \sum_{i=1}^{EN} \left(\sum_{s=1}^{Sims} Async.PE_i^s \right) * w_{ij} \\
&= \sum_{i=1}^{EN} Sync.PE_i * w_{ij} \quad (from\ Equation\ 6.1)
\end{aligned}$$

Therefore, for any output neuron j,

$$Sync.SA_j == \sum_{s=1}^{Sims} Async.SA_j^s \tag{6.4}$$

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.

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
```

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.

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,

$$Sync.NE_n == \sum_{s=1}^{Sims} Async.NE_n^s \tag{6.5}$$

Therefore, from Equations 6.1, 6.4, and 6.5, 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.

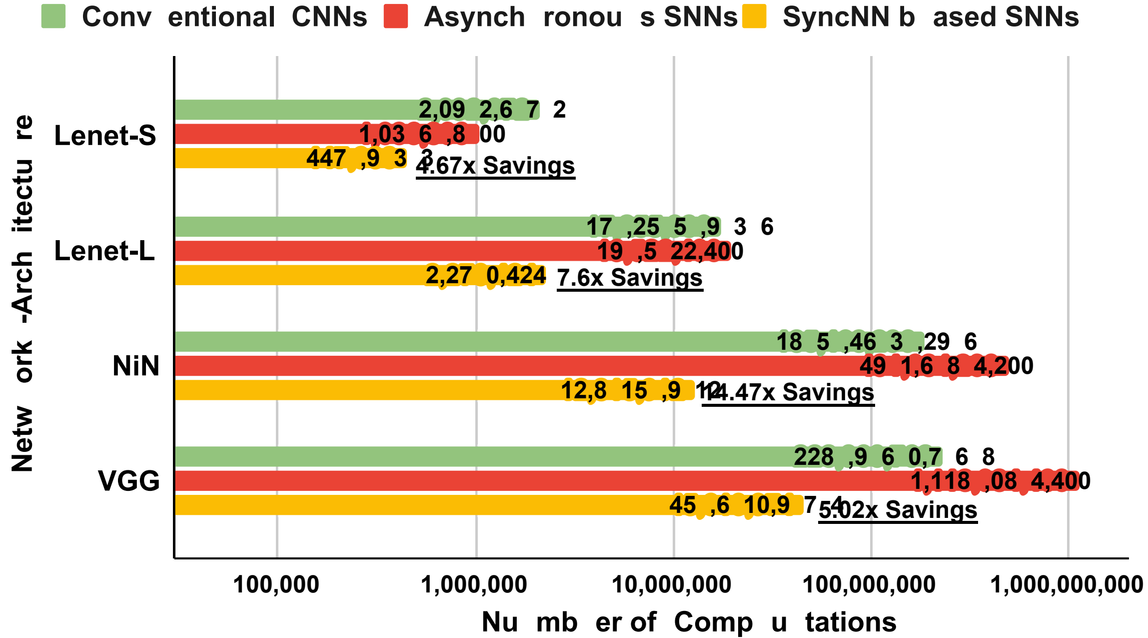


Figure 6.1: Comparison of computation operations required in conventional CNNs, asynchronous SNNs, and our SyncNN based SNNs (savings is over CNNs)

6.2 Computational Comparison for SyncNN

Finally, we compare the number of computing operations required in the conventional CNNs, asynchronous SNNs, and our SyncNNs in Figure 6.1. The detailed experimental setup will be explained in Section 9.1. 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 one timestep for all layers except the input layer. Compared to conventional CNNs, our SyncNN based event-driven SNNs can reduce the number of computing operations by 4.67x to 14.47x.

6.3 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.

Chapter 7

Hardware Design of Asynchronous SNNs

In the previous chapters, we discussed the disadvantages of the Asynchronous approach and proposed the Synchronous approach called SyncNN, to run deeper networks on FPGA. Before we look at the hardware implementation of SyncNN, in this chapter, we implement the hardware design with various computational and memory access optimizations for the Asynchronous SNNs and prove the above discussions in a more practical manner.

7.1 Computational Optimization

To improve the overall performance of the SNN algorithm presented in Algorithm 1, we offload its computation to the FPGA on an embedded ARM-FPGA SoC using Xilinx SDSoC and HLS support. There are a few design choices that we need to consider.

Hardware function selection. In the SDSoC design flow, it allows users to select the top-level hardware functions and then it will automatically offload those selected functions to the FPGA part as separate hardware accelerators. From Algorithm 1, we decide to select the `SPIKING_NET` function as one big hardware accelerator, instead of selecting `POISSON_ENCODING`, `NEURON_ENCODING`, and `SPIKE_AGGGREGATE` functions as individual hardware accelerators. The reason is to improve the data exchange efficiency—such as the Poisson spikes (pSp*), neuron spikes (nSp*), and aggregated spikes (Vm*)—between the functions. With small hardware accelerators, they have to exchange data through a shared memory, which will default to the shared off-chip DRAM. On the other hand, using one big hardware accelerator, these data exchange can be efficiently done using on-chip BRAM and/or URAM blocks.

Also, to minimize the hardware call overhead, the hardware function which was accelerated for each simulation step of an image was further improvised to one master accelerator for each image. The MNIST dataset consists of 10,000 images and each image is run for multiple simulation steps. Let us take an example of 40 simulation steps. When calling

the hardware function for each simulation step, the number of hardware calls was about 400,000. For each call, there is a hardware function call overhead of send and receive commands for the minimal hardware workload. In order to eliminate this overhead, we increase the hardware workload by calling it for each image and therefore the number of hardware calls is minimized by 40x.

Coarse-Grained Parallelization. Within the top-level hardware function, for the Asynchronous SNNs, the `POISSON_ENCODING`, `NEURON_ENCODING` and `SPIKE_AGGGREGATE` functions for every layer have the ability to run in parallel. As shown in Algorithm 1, there are data dependencies among them:

1) the `NEURON_ENCODING` function is dependent on the aggregated spikes (Vm*) generated by the `SPIKE_AGGREGATE` function;

2) the `SPIKE_AGGREGATE` function is dependent on the encoded spikes (pSp* and nsp*) generated by the `POISSON_ENCODING` and the `NEURON_ENCODING` functions;

To resolve this and enable parallelism, we use a ping-pong buffer mechanism so that the `SPIKE_AGGREGATE` function works on one copy of the encoded spikes (pSp* and nSp*) respectively to aggregate spikes (Vm*) buffers, while the (`POISSON_ENCODING` and the `NEURON_ENCODING`) functions work on another buffer copy. Note that this requires one extra timestep to achieve the same simulation, which is negligible. Next we present how to optimize each sub-function.

As shown in Algorithm 1, there are data dependencies among them:

1) the `NEURON_ENCODING` function is dependent on the aggregated spikes (Vm*) generated by the `SPIKE_AGGREGATE` function;

2) the `SPIKE_AGGREGATE` function is dependent on the encoded spikes (pSp* and nsp*) generated by the `POISSON_ENCODING` and the `NEURON_ENCODING` functions; Note that this requires one extra timestep to achieve the same simulation, which is negligible.

Loop Pipelining. We pipeline all the loops in the `POISSON_ENCODING`, `NEURON_ENCODING`, and `SPIKE_AGGGREGATE` functions with a pipeline II as 1.

Combining Loop Pipelining and Unrolling. To further improve their performance, we unroll the loops in the `POISSON_ENCODING`, `NEURON_ENCODING`, and `SPIKE_AGGREGATE` functions to multiple copies and make them run in parallel. The loops in first two functions can easily achieve very low latency as they can easily be unrolled, since the loops have a fixed number of iterations and there are no nested loops. However, it is quite challenging to efficiently unroll the loops in the `SPIKE_AGGREGATE` function. First, the size of the weights matrix is very big to be buffered on-chip. Second, due to the event-driven nature, the access to the weights matrix and aggregated spikes (Vm*) array becomes random. If a `SPIKE_AGGREGATE` function is a dense layer, it has nested loops, where the outer loop is the encoded neurons in the previous layer which does not have fixed bound and the inner loop is the number of neurons in the next layer. Due to data dependencies, the outer loop cannot be unrolled whereas the inner loop can be unrolled. These unrolling factors will be

chosen based on the available computing and memory resources in the FPGA board and also targetting load balancing between the functions as they are executed in parallel.

7.2 Memory Access Optimization

For smaller networks, the arrays that we use in Algorithm 1—such as Poisson spikes (pSp*), neuron spikes (nSp*), aggregated spikes (Vm*), and membrane potential (totalVm*)—are relatively small and can be buffered on chip using BRAM. The challenge is to buffer the relatively larger weights matrix. Next we discuss the several techniques applied for effective memory access from the weights matrix.

Memory Bursting from Off-chip DRAM. Data access from DRAMs is much slower than data access from on-chip memories because of the overhead of the AXI protocol and the long access latency of DRAM itself.To amortize this overhead, we burst read/write all the off-chip memory accesses.

On-chip Memory Buffering. Due to the event-driven nature of SNNs, for a dense network, the access to a row of the weights matrix is random. In order to minimize the off-chip access, in the EASpiNN framework, we store the maximum number of rows in the weight matrix that can be buffered on-chip before starting the process. This buffering is only done once for the entire image classification process and is very negligible. For the remaining rows that are not buffered, we use off-chip burst read access.

Effective URAM Buffering. The maximum number of rows that can be buffered depends on the available on chip memory in the board. In the SoC FPGA devices like ZED, ZC706, ZCU102, the BRAMs are the only on-chip memory available and on devices like ZCU104, on-chip memory includes URAMs in addition to the BRAMs. Each bank of the URAM can store up to 4K x 72 bits of data where-as each BRAM bank can store only 1K x 36 bits of data. If the weight used is a 32-bit floating point number and is stored in a BRAM, there is only 4 bits of wastage per weight which is negligible. But in the case of URAMs, if the same 32-bit number is directly stored on an URAM, there is a wastage of 40 (72-32) bits for each weight and making it inefficient. To solve this issue, two weights (each of 32 bits) are packed so that every entry in the URAM is 64 bits, so that the wastage of URAM is minimized to 4 bits per weight. Based on the size of each element used for the weights array, we pack the weights together in the URAM, such that the wastage in each bank is minimum, and therefore can accommodate more weights in the available on chip memory.

7.3 Design Automation

Finally, we automate the design to run any SNN inference network with fully connected topology on any embedded Xilinx ARM-FPGA SoCs. In order to run different networks on different boards, a user just needs to provide a configuration file based on the network and the board used. From the network, we get the Poisson information, neurons in each

layer and their corresponding weights. Based on the board, we get the maximum number of weights that can be stored on chip (URAM and BRAM). This value is dependent on the data type we choose for the weights (32bits, 16bits, and 8bits). The framework automatically decides the best design possible for the network in the available board considering the load balancing between the functions, buffering weights as much as possible and thus reducing the number of off-chip memory access to the maximum.

Deciding Unrolling Factors. As discussed earlier, the unroll factor that we use in the `SPIKE_AGGREGATE` and function would decide the load balance between the functions. Deciding the unroll factor for each inner loop is done offline using the network and board parameters. The network model is profiled once on CPU to estimate the average value of spikes, including the Num. of Spiked Neurons as discussed in the Chapter: 5.4. The *TargetLatency* is obtained based on the function with the least workload (generally with the `POISSON_ENCODING` or `NEURON_ENCODING` functions). Now, the unroll factors have to be decided such that, the `SPIKE_AGGREGATE` function has similar load balancing to that of the remaining functions. Unroll1 is for the nested loop of activated neurons in input layer *(pSpiked)* and first hidden layer. Unroll2 is for the nested loop of activated neurons in first layer *(nSpiked1)* and second hidden layer *(Layer2)* and Unroll3 is for the nested loop of activated neurons in second layer *(nSpiked2)* and output layer *(Output)*. Using all these values — avgPSpiked, avgNSpiked1, avgNSpiked2, TargetLatency, we estimate the ideal unroll factor using the following formula, assuming there is unlimited resource available and all weights are on chip:

1. Unroll#i = Latency#i / TargetLatency; i = 1, 2, 3
2. Latency1 = avgPSpiked * Layer1
3. Latency2 = avgNSpiked1 * Layer2
4. Latency3 = avgNSpiked2 * Output

If the available computing resource or on-chip memory does not support the estimated unroll factors, we scale them down in equal ratio till the design fits into the FPGA resource.

Allocating On-chip Buffer. After we determine the unroll factors, even if the latency of the `SPIKE_AGGREGATE` functions is reduced to the target latency, the data loading will still not be balanced since the unroll factors are calculated with the assumption that all the weights are laying on chip. If the weights are laying off chip, the overhead to burst read would also account for the latency.

1. Latency1 = avgPSpiked * Layer1 * (1 + avgBurstLatency)
2. Latency2 = avgNSpiked1 * Layer2 * (1 + avgBurstLatency)
3. Latency3 = avgNSpiked * Output * (1 + avgBurstLatency)

Algorithm 3 Automation Algorithm for Buffering of Weights

```
1:  while avlURAM[dt]!=0 && avlBRAM[dt]!=0 && Target[0:2]!=0 do
2:      URAMPartition = AvailablURAM[dt]/3
3:      BRAMPartition = AvailablBRAM[dt]/3
4:      for each Layer do
5:          if Target[i] < URAMPartition then
6:              row = Target[i]/Layer[i]
7:              Push weights_i[row][Layer[i]] to URAM
8:              avlURAM[dt] = avlURAM[dt]-Target[i]
9:              Target[i] = 0
10:         else
11:             row = URAMPartition/Layer[i]
12:             Push weights_i[row][Layer[i]] to URAM
13:             avlURAM[dt] = avlURAM[dt]-URAMPartition
14:             Target[i] = Target[i]-URAMPartition
15:         if Target[i] < BRAMPartition then
16:             row = Target[i]/Layer[i]
17:             Push weights_i[row][Layer[i]] to BRAM
18:             avlBRAM[dt] = avlBRAM[dt]-Target[i]
19:             Target[i] = 0
20:         else
21:             row = BRAMPartition/Layer[i]
22:             Push weights_i[row][Layer[i]] to BRAM
23:             avlBRAM[dt] = avlBRAM[dt]-BRAMPartition
24:             Target[i] = Target[i]-BRAMPartition
```

Thus, the next aim in the framework is to automatically calculate the maximum number of weights that can be buffered on chip. The automation is done in such a way that, equal weightage is given for all the three weight accesses (weight_1, weight_2, and weight_3 for the three unrolled loops) such that the ratio of off chip access is reduced for all the three weights in a similar fashion. To do this, we set a minimum target for buffering the average spiked number of rows on chip:

1. minTarget[0] = avgPSpiked*Layer[0] (Layer[0] = Layer1)
2. minTarget[1] = avgNSpiked1*Layer[1] (Layer[1] = Layer2)
3. minTarget[2] = avgNSpiked2*Layer[2] (Layer[2] = Output)

We set the maximum target for buffering all the weights on chip:

1. maxTarget[0] = Poisson*Layer[0]
2. maxTarget[1] = Layer[0]*Layer[1]
3. maxTarget[2] = Layer[1]*Layer[2]

Achieving the best on chip buffer space allocation is similar to winning a lottery in which everyone gets equal amount of share to participate in the contest. This motives our Algorithm 3 to get the maximum number of rows that be buffered on chip for each layer, given the available on chip memory in the board. We run Algorithm 3 for two rounds.

In the first round, we set Target[i] as minTarget[i] and run recursively for each layer until all Target[i] resources have been allocated on-chip or the on chip memory is used up, as shown in Algorithm 3. The available on-chip memory (URAM and BRAM) is partitioned into three segments, so that each weight segment gets equal importance to fill in the weights. If the target weight of any layer is greater than the partitioned available URAM or BRAM (we allocate to URAM first), we buffer the weights equal to partitioned memory in that layer for that recursive iteration. Suppose if another target of a different layer is less than the partitioned available on-chip memory, we buffer the complete target weights on chip. In this case, the remaining on-chip memory is put back for the next recursive iteration. In the next recursion call, the layer that was not able to reach the target gets another chance to fill more weights on chip. By this way, we keep filling the weights to reach the target until there is no more on-chip memory available.

Once we reach the minimum target for all the layers and there is still on-chip memory left, we run the same algorithm again with the remaining on-chip memory resources and set Target[i] to be maxTarget[i]-minTarget[i]. Basically, the first round makes sure the minimum target is achieved for all three weights. It avoids the case that one weights matrix overuses the on-chip memory to satisfy its maximum target, while other weights matrices do not get enough share. And the second round makes sure if there is more resource, each weights matrix gets another chance to get more on-chip buffer allocation. By this approach, all the weights in each layer gets a fair share of on-chip memory based on their usage weightage.

7.4 Challenges of Asynchronous Approach

The framework to implement Asynchronous SNNs is evaluated with well established handwritten image recognition MNIST benchmark. The dataset has 28*28 size image as inputs, i.e, a total of 784 Poisson inputs. Two MNIST networks were tested. The first network has two hidden layers with 500 and 100 neurons in each layer, and the second network also has two hidden layers with 1,200 neurons in each of them. Their detailed configuration is summarized in the first portion of Table 7.1.

Table 7.1: Automation results breakdown for latency balancing, loop unrolling, and effective weight buffering

Parameters	**Poissons**	**Neurons**	**Layer1**	**Layer2**	**Output**	**Datatype (Weights)**
MNIST (610)	784	610	500	100	10	32 or 16 or 8
MNIST (2410)	784	2,410	1,200	1,200	10	32 or 16 or 8
Latency	**POISSON_SPIKES**	**NEURON_SPIKES**	**POISSON_SPIKE_AGG**		**NEURON_SPIKE_AGG**	
			before unrolling	**after unrolling**	**before unrolling**	**after unrolling**
MNIST (610)	784	1,220	50,500	1,256	3,200	1,067
MNIST (2410)	784	4,820	158,400	4,800	106,800	4,855
	AvgPSpiked	**AvgNSpiked**	**Unroll1**	**Unroll2**	**Unroll3**	
MNIST (610)	101	32	41	3	1	
MNIST (2410)	132	89	33	22	1	
Buffered Weights (%)	**16 Bits Quantization**			**8 Bits Quantization**		
	weights_1	**weights_2**	**weights_3**	**weights_1**	**weights_2**	**weights_3**
MNIST (610)	100%	100%	100%	100%	100%	100%
MNIST (2410)	74%	51%	100%	100%	100%	100%

The conversion from ANN to SNN was done and tested for the two MNIST networks. We use single-precision floating point as the data type for weights in the software versions The framework was built using Xilinx SDSoC 2019.1 and Vivado HLS C++. Vivado HLS allows a user to code the hardware with C or C++ and supports pragmas to optimize the hardware easily. The framework was evaluated using Xilinx Zynq UltraScale+ ZCU104 board where the FPGA accelerator ran at 100 MHz. The baseline CPU version used to compare the hardware performance is the 1.2GHz ARM processor on ZCU104.

7.4.1 Results from Automation Algorithm

As explained in Section 7.3, based on a user configuration file that contains the network information and board information, our automation algorithm aims to obtain the best configuration—including the unroll factors for the loops in the `SPIKE_AGG_POISSON`, and `SPIKE_AGG_NEURON` functions, and the percentage of each weights that are to be buffered on chip—to balance the four major functions. These derived configurations will be automatically feed into the tool to build the FPGA design. Table 7.1 shows the final automated results for both the MNIST networks used.

1. The second portion of Table 7.1 summarizes the average latency of each function. The latency of the `POISSON_SPIKES` and `NEURON_SPIKES` value is based on the number of Poisson and neurons, where `NEURON_SPIKES` latency dominates and it is the target latency that guides our automation. After unrolling, the latency of `SPIKE_AGG_POISSON` and `SPIKE_AGG_NEURON` becomes pretty close to the target latency, which confirms that our automation is quite effective to balance the four major functions and achieves the optimal performance.

2. In the third portion of Table 7.1, we first summarize the average number Poisson and neuron spikes (per simstep) that is profiled in the software run. In MNIST(610), the average number of Poisson spikes is 101, which is is about 12.9% of the total number of Poissons; the average number of neuron spikes is 32, which is is about 5.2% of the total number of neurons. In MNIST(2410), these two percentages are 16.8% and 3.7%. This is one main reason that SNNs are more efficient than ANNs and CNNs.

3. The final unroll factors are also summarized in the third portion of Table 7.1. For MNIST(610), the three unroll factors are 41, 3, and 1; while for MNIST(610), the three unroll factors are 33, 22, and 1. For both networks, even after we unroll the loops, there is sufficient computing resource to hold all unrolled operations. Moreover, there is sufficient on-chip memory to buffer the average spiked number of rows of the weights matrix. So we do not need to scale them down.

4. The fourth portion of Table 7.1 summarizes the percentage of weights that are buffered on chip for 16-bits and 8-bits quantization. For MNIST(610), we are able to buffer

all the weights (maximum target achieved) both 16 bits and 8 bits precision. For MNIST(2410), with 8 bits precision, all the weights are buffered on chip; with 16 bits precision, 74%, 51%, and 100% of the weights_1, weight_2, and weights_3 are buffered on chip.

7.4.2 Resource Utilization

Table 7.2 summarizes the resource utilization of EASpiNN on the ZCU104 board. From the table, it is evident that memory plays a vital role than the the DSP, Flip Flops and LUT usages for bigger networks. This is mainly because the integrate-and-fire (IF) based SNNs are not computationally complex and are mainly limited by memory accesses. Thus, for applications like SNNs, more resource invested on the on-chip memory rather than the computing resource like DSPs and LUTs will give better performance reward.

Table 7.2: Resource utilization (%) on ZCU104 board

Implementation	BRAM	URAM	LUT	FF	DSP
MNIST(610)-32	34%	52%	12%	6%	6%
MNIST(610)-16	27%	26%	7%	9%	0%
MNIST (610)-8	24%	12%	7%	5%	0%
MNIST (2410)-32	78%	95%	28%	48%	13%
MNIST (2410)-16	87%	97%	15%	19%	1%
MNIST (2410)-8	70%	87%	20%	36%	0%

7.4.3 Comparison with Related Work

Table 7.3: Comparison with related work

Implementation	Precision	Accuracy	FPS
Minitaur [28]	16 bits	92%	6.6
Darwin [8]	16 bits	93.8%	6.25
FPGA Reference [14]	16 Bits	97.06%	161
FPGA Reference [16]	8 Bits	98.94%	164
EASpiNN (610)	16 Bits	97.77%	2,654
EASpiNN (610)	8 Bits	96.77%	2,660
EASpiNN(2410)	16 Bits	98.07%	175
EASpiNN(2410)	8 Bits	97.94%	801

The EASpiNN framework stands out in the number of frames per second (FPS) when compared to the similar neuromorphic hardware implementations [28, 8] and FPGA implementations [14, 16] as shown in Table 7.3. All of them use the same MNIST dataset for handwritten image recognition. EASpiNN achieves a maximum of 2,654 FPS at 97.77% accuracy for MNIST(610) using 16 bits and a maximum of 801 FPS at 97.94% accuracy for MNIST(2410) using 8 bits. The two prior neuromorphic hardware studies [14, 16] only achieved 6.6 and 6.25 FPS. The two prior FPGA studies [14, 16] only achieved 161 and 164 FPS.

In terms of accuracy, except for the FPGA implementation in [16], which achieves about 98.94% accuracy, our two implementations that run at 2,654 FPS and 801 FPS also achieve higher accuracy than the other three prior studies.The accuracy difference in our implementations to the one in [16] is because of two reasons. First, even with the ANN implementation that we converted to SNN, we observe an accuracy of 98.55% only and the ANN algorithm is not optimized to improve the accuracy. Second, we do not use CNN layers in the network as used in [16]. Our main focus in the Asynchronous SNN implementation is develop an automated and effective framework to enable SNN acceleration and evaluation on FPGAs.

7.4.4 Challenges of Asynchronous Approach

The challenges discussed in Chapter: 5.2, is practically observed in our hardware implementation of the Asynchronous implementation framework. Firstly, from Table 7.2, we can see that the on-chip memory resources for the MNIST(2410) network has already reached the maximum. Therefore, it is not possible to implement further deeper networks using this framework. Secondly, from Table 7.1, we can see that the latency between the functions vary widely and therefore there would be resource under utilization. To resolve all these challenges, as discussed in Chapter: 5.3, we implement the SyncNN algorithm in hardware and discuss about the implementation in the next chapter.

Chapter 8

Hardware Design of SyncNN Framework

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

8.1 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. For 16 bits, the number of bits available is enough to represent the weights without any accuracy loss. 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.

$$\text{max_weight} * \text{scale} = \text{max_fixed_point} \tag{8.1}$$

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:

$$\text{Xth_percentile_weight} * \text{scale} = \text{max_fixed_point} \tag{8.2}$$

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 9.3 in Section 9.3 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. 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 9.3, we will present the improvement in accuracy for deeper networks with *Mixed Precision* approach.

8.2 Computation Optimization

In SyncNN, we design configurable and scalable compute engines for all major functions with pipeline and parallelization techniques, as shown in Figure 8.1, which can be deployed on different Xilinx FPGA boards based on the available memory and computing resources.

POISSON_ENCODING. In SyncNN, 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, 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, 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.

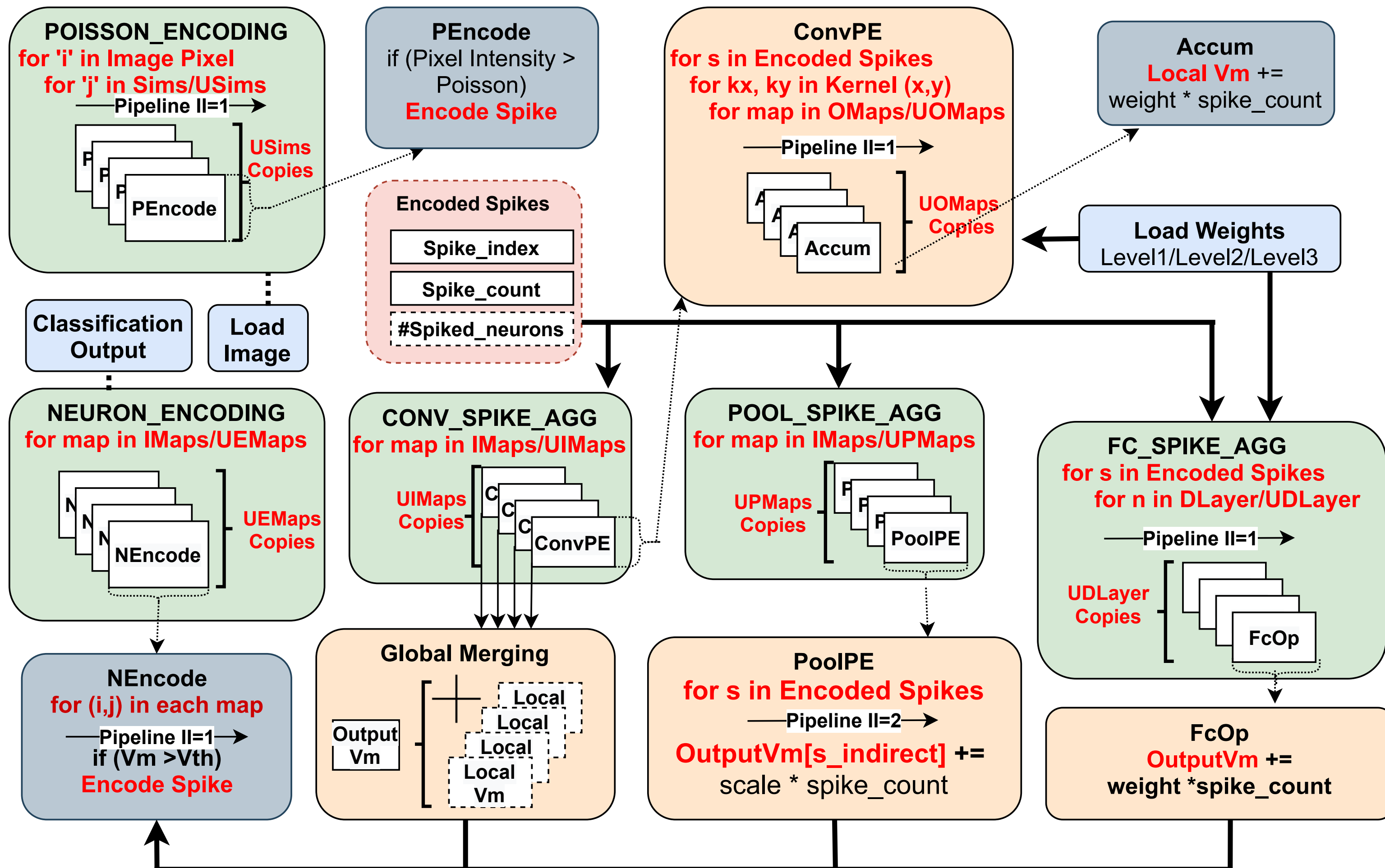


Figure 8.1: Overview of the hardware architecture for SyncNN

1. **Convolutional unit.** The `CONV_SPIKE_AGG` function in Figure 8.1 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 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. Hence, we design a `POOL_SPIKE_AGG` function as shown in Figure 8.1, 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 8.1 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.

8.3 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.

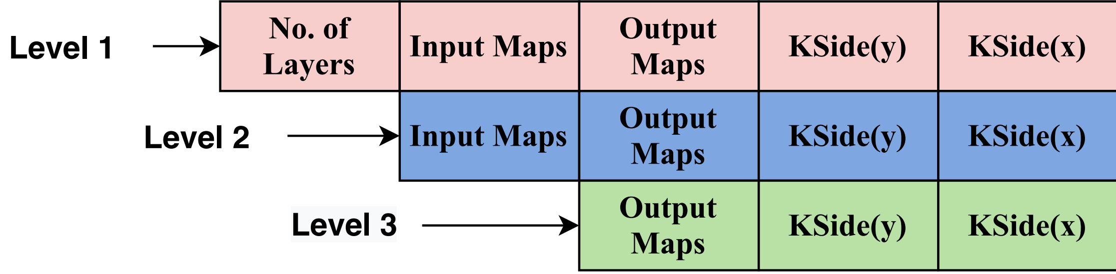


Figure 8.2: Different granularity to load weights depending on the size of the weights and available on-chip memory

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 [20, 48] from the off-chip memory to on-chip memory at different granularity. As shown in Figure 8.2, 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.

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.

Chapter 9

Experimental Results for Synchronous Approach

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.

9.1 Experimental Setup

The network configurations for each network is summarized in Table 9.1. 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 [11]. 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 9.1: 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

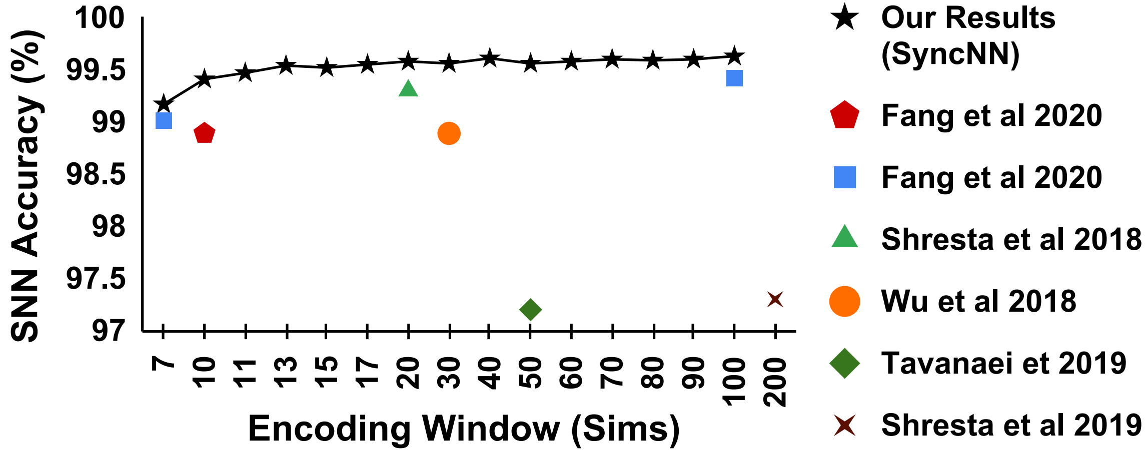


Figure 9.1: Impact of encoding window on the accuracy for MNIST dataset

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. 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.

9.2 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 9.1, the encoding window required to achieve a good accuracy is relatively small. Shrestha et al. [40] achieve 97.3% accuracy for 200 Sims. Tavanaei et al. achieve [43] 97.2% at 50 Sims. Wu et al. [45] achieve 98.89% accuracy for 30 Sims. Shrestha et al. [39] achieve 99.3% accuracy for 20 Sims. More recently, Fang et al. [11] 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.

For CIFAR-10 dataset, with larger networks like NiN and VGG, as shown in Figure 9.2, 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

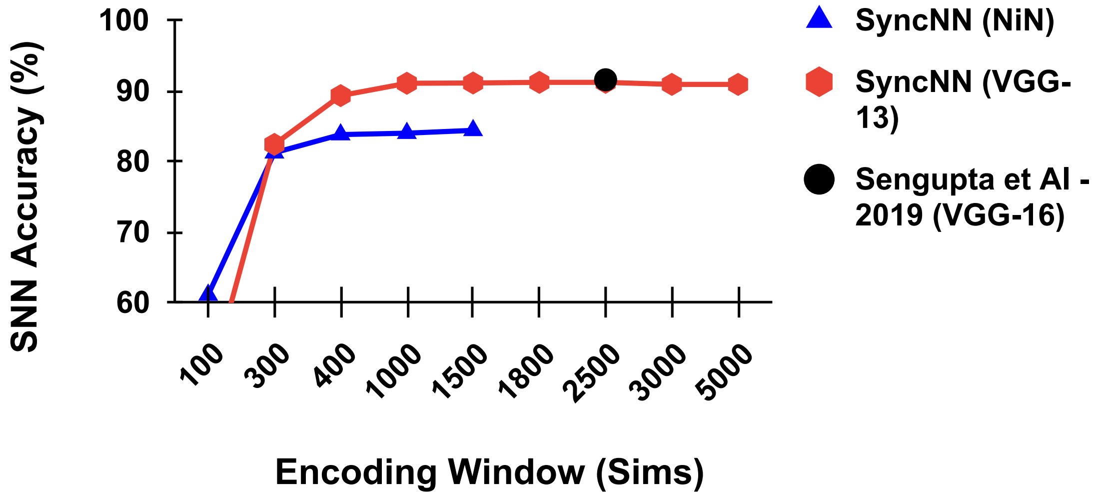


Figure 9.2: Impact of encoding window on the accuracy for CIFAR-10 dataset

91.19% accuracy at 1,800 Sims. The recent study by Sengupta et al. [38] also confirms the need for a larger encoding window as they achieve the peak accuracy at 2,500 Sims with the VGG-16 network.

9.3 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 [38, 9, 15, 36] 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 9.3, 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 8.1, 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 9.4, we compare the SyncNN accuracy for NiN and VGG networks under 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

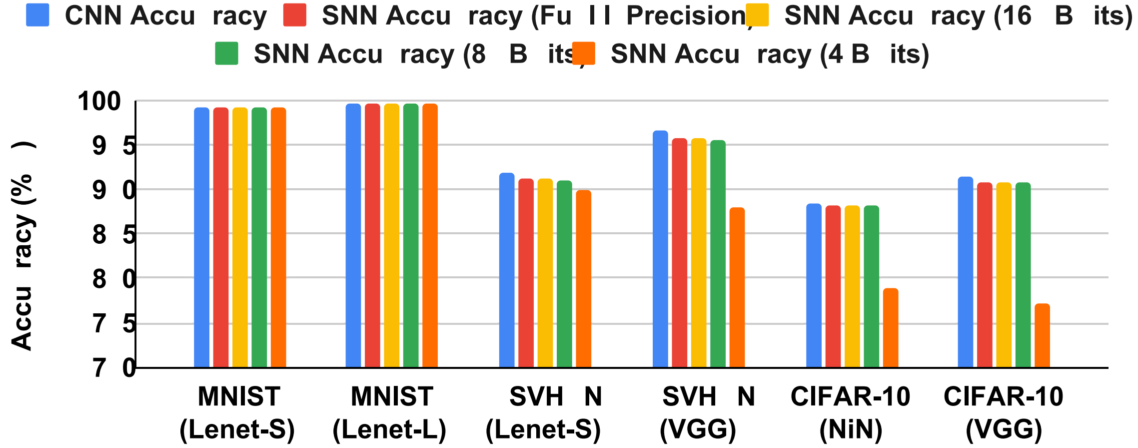
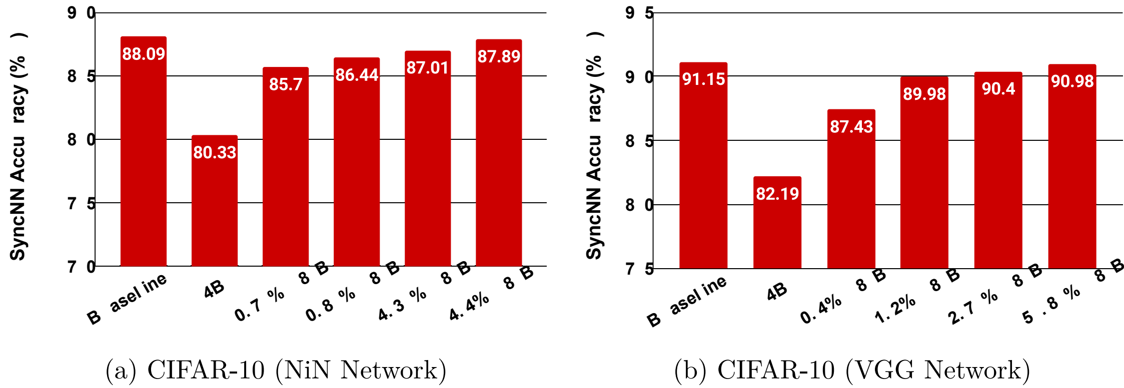


Figure 9.3: Inference accuracy comparison of CNN and SyncNN based SNNs with different data precision



(a) CIFAR-10 (NiN Network)

(b) CIFAR-10 (VGG Network)

Figure 9.4: Inference accuracy of SyncNN using mixed 4-bit and 8-bit (8B) quantization

NiN network with CIFAR-10 dataset, as shown in Figure 9.4a, 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 9.4b, 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.

9.4 Overall Performance

Figure 9.5 shows the overall performance of each network with the best configuration on each FPGA board (configuration parameters are presented in Section 8.2). 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

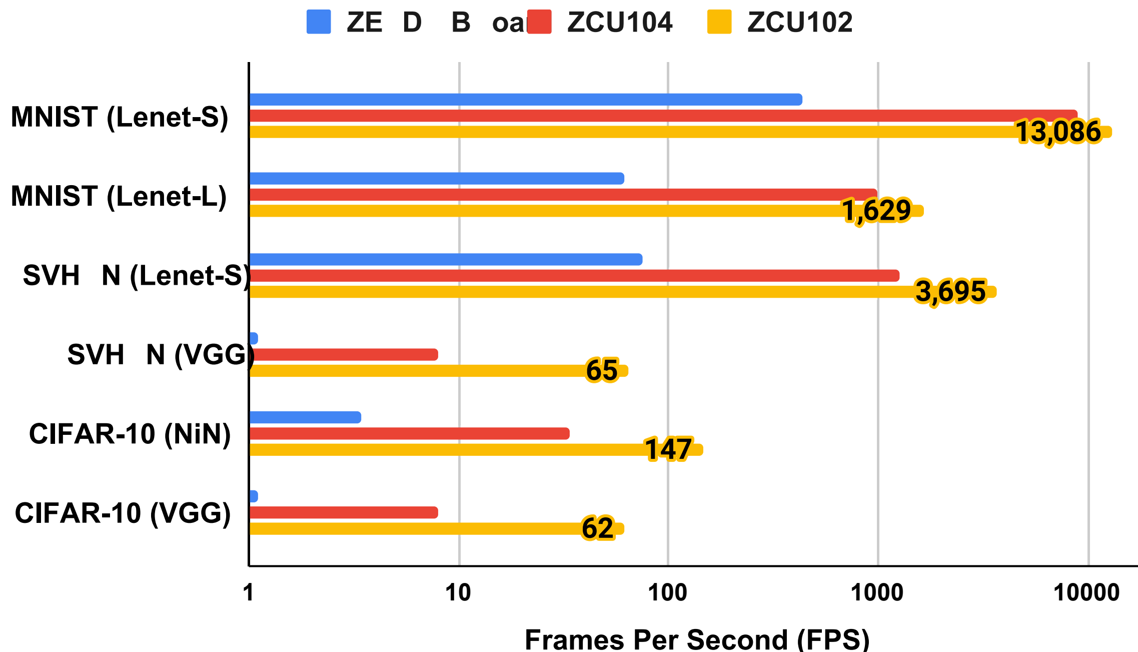


Figure 9.5: Frames per second (FPS) for different networks and datasets running on different FPGA boards

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 9.2.

9.5 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 9.2 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, and FPS per 1K LUTs usage. The FPGA implementations, Minitaur [27], Han et al. [14], and Ju et al. [16], achieve 92%, 97.06%, and 98.94% accuracy with 108, 161 and 164 FPS, respectively. The prior neuromorphic hardware studies, Darwin [8], Stromatias et al. [41], and Esser et al. [10], 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 [11] at 125MHz. Shown in Table 9.2, the same work also demonstrates superior performance of FPGA-based SNNs over that on CPU, GPU, Intel Loihi neuromorphic chip [11].

Table 9.2: Comparison with related work for image classification using SNNs for the MNIST dataset

Work	**Platform**		**Frequency**	**Training Method**	**Precision (bits)**	**Network**	**Accuracy (%)**	**FPS**	**FPS/ DSP**	**FPS per 1k LUTs**
Stromatias et al. (2015)~ [41]	ASIC	SpiNNaker	150MHz	Spike Based	16	MLP	95	50	-	-
Esser et al. (2015) [10]	ASIC	True North	-	Offline Training	Binary	-	95	1,000	-	-
Darwin (2017) [8]	ASIC	-	25MHz	-	16	MLP	93.8	6.25	-	-
Minitaur (2014)[27]	FPGA	Spartan-6 LX150	75MHz	Spike Based	16	MLP	92	108	-	1.22
Han et al. (2020) [14]	FPGA	Xilinx ZC706	200MHz	Converted SNNs	16	MLP	97.06	161	-	29.92
Ju et al. (2020) [16]	FPGA	Xilinx ZCU102	150MHz	Converted SNNs	8	CNN (LeNet)	98.94	164	-	1.52
Fang et al. (2020) [11]	ASIC	Intel Loihi	-	Spike Based	16	CNN (Lenet-S)	99.2	671	-	-
	CPU	Intel i9-9900K	3.7GHz					100	-	-
	GPU	Nvidia RTX 5000	1.62GHz					864	-	-
	Edge GPU	Nvidia AGX Xavier	1.37GHz					211	-	-
	FPGA	Xilinx ZCU102 (Simulation)	125MHz					2,124	1.18	13.62
SyncNN (Our Work)	**FPGA**	**Xilinx ZCU102**	**200MHz**	**Converted SNNs**	4	**CNN (Lenet-S)**	**99.3**	**13,086**	**19.23**	**61.2**
						CNN (Lenet-L)	**99.6**	**1,629**	**2.94**	**7.25**

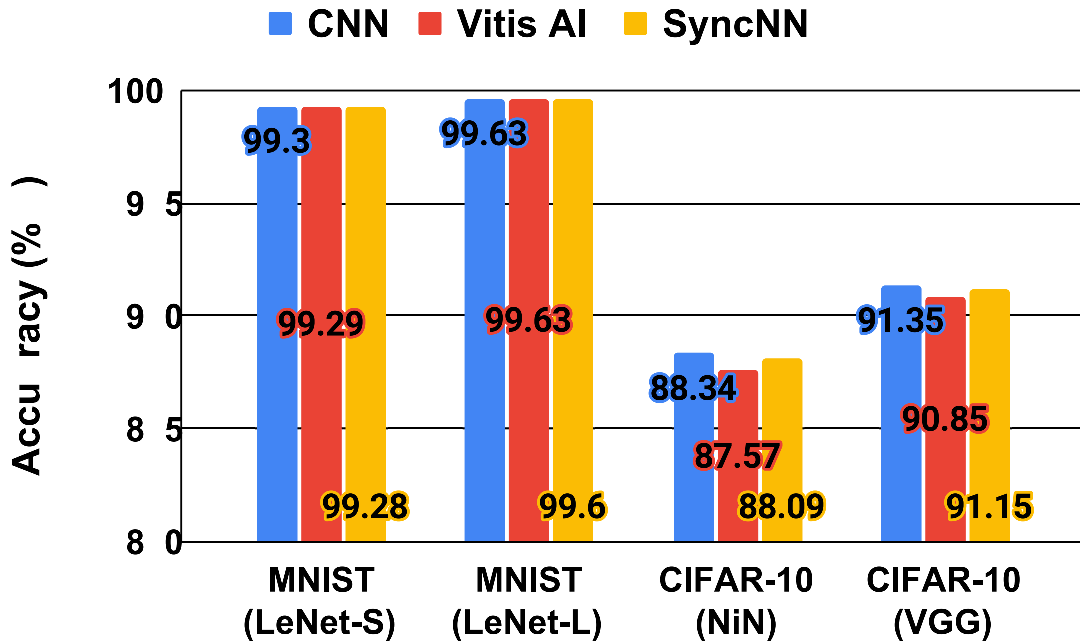
Our SyncNN framework stands out in terms of all four metrics—accuracy, FPS, FPS per DSP, and FPS per 1K LUTs usage—for the MNIST dataset, compared to the prior neuromorphic hardware implementations, FPGA and GPU implementations. SyncNN achieves a maximum of 13,086 FPS at 99.3% accuracy for the same *Lenet-S* network and the same FPGA board used in [11]. Also, 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.

9.6 Comparison with CNN Acceleration in Vitis AI

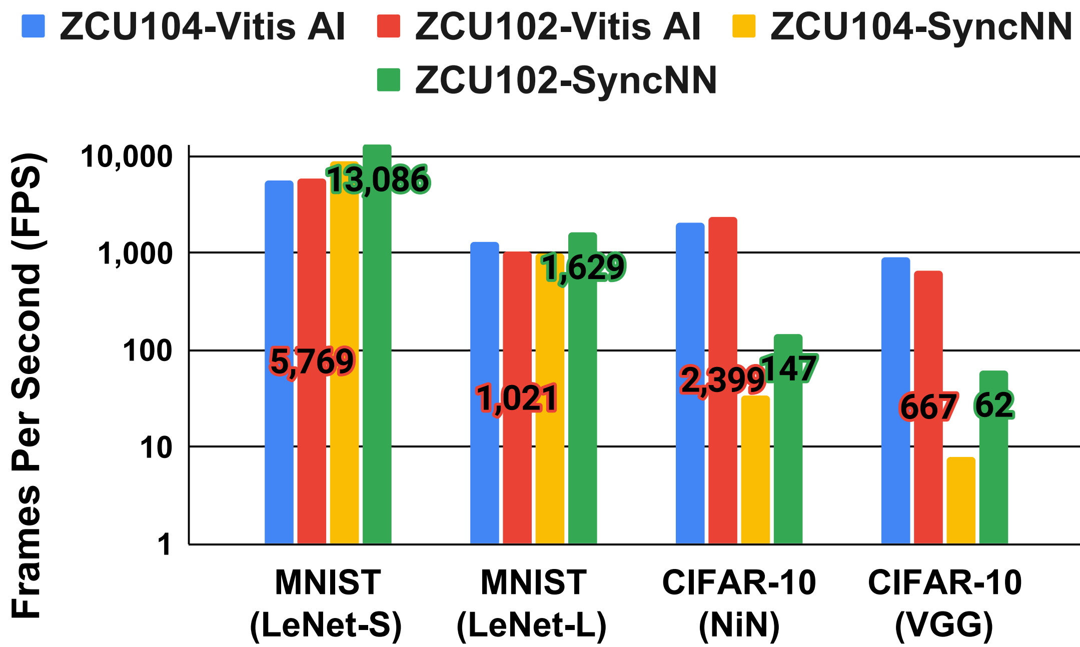
In Section 6.2, we observed that, SyncNN has about 4.67x to 14.47x savings than the conventional CNNs in terms of number of computing operations. Although there are great algorithmic savings, the event-driven nature of SNNs makes it challenging when accelerated on hardware. On the other hand, the memory 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).

We implement the same networks discussed in Section 9.1 and evaluate the CNN performance. First, as shown in Figure 9.6a, 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 dataset, we see that the converted SyncNNs achieves slightly better accuracy than that of Vitis AI.

Next, as shown in Figure 9.6b, we compare the hardware performance in terms of Frames Per Second (FPS) between Vitis AI and SyncNN on ZCU104 and ZCU102 Xilinx 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 *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 8.3. But for deeper networks like *NiN* and *VGG*, Vitis AI has



(a) Accuracy



(b) Performance

Figure 9.6: Accuracy and performance comparison of CNN in Vitis AI and SyncNN

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.

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. 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.

Chapter 10

Conclusion

In this thesis, 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.

10.1 Future Work

At this moment, SyncNN is scalable to run any rate-encoding based deeper SNN network on FPGA. However, temporal encoding based SNNs is another alternative and interesting research area. In future, investigation of temporal encoding based SNNs using the SyncNN framework would lead a pathway for various applications in the area of Machine Learning. Also, further improvement of accuracy for deeper networks by investigation of several CNN optimization techniques within our SyncNN framework would be an engaging research area.

Chapter 11

Publications

This chapter summarizes the list of the publications that are a part of this thesis.

11.1 Journal (Under Submission)

Sathish Panchapakesan, Zhenman Fang, and Jian Li. "SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs". Journal submission to TRETS under review [30].

11.2 Full Conference Paper (Published)

Sathish Panchapakesan, Zhenman Fang, and Jian Li. "SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs". International Conference on Field-Programmable Logic and Applications (FPL), 2021 [31].

11.3 Conference Poster (Published)

Sathish Panchapakesan, Zhenman Fang, and Nitin Chandrachoodan. "EASpiNN: Effective Automated Spiking Neural Network Evaluation on FPGA". International Symposium onField-Programmable Custom Computing Machines (FCCM), 2020 [29].

11.4 Open Source Software

The project is open-sourced at: **https://github.com/SFU-HiAccel/SyncNN**.

Bibliography

[1] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, 18(24):10464–10472, 1998.

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

[3] Olaf Booij and Hieu tat Nguyen. A gradient descent rule for spiking neurons emitting multiple spikes. *Inf. Process. Lett.*, 95(6):552–558, September 2005.

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

[5] Kit Cheung, Simon R. Schultz, and Wayne Luk. Neuroflow: A general purpose spiking neural network simulation platform using customizable processors. *Frontiers in Neuroscience*, 9:516, 2016.

[6] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. Best-effort FPGA programming: A few steps can go a long way. *CoRR*, abs/1807.01340, 2018.

[7] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2018.

[8] M.A. De, Shen JunCheng, G.U. ZongHua, Zhang Ming, Zhu XiaoLei, X.U. XiaoQiang, Qi Xu, Shen YangJing, and Gang Pan. Darwin: a neuromorphic hardware co-processor based on spiking neural networks. *Journal of Systems Architecture*, 77:43–51, 01 2017.

[9] 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)*, pages 1–8, 2015.

[10] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 1117–1125. Curran Associates, Inc., 2015.

[11] H. Fang, Z. Mei, A. Shrestha, Z. Zhao, Y. Li, and Q. Qiu. Encoding, model, and architecture: Systematic optimization for spiking neural network in fpgas. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[12] Haowen Fang, Amar Shrestha, Ziyi Zhao, and Qinru Qiu. 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)*, pages 2771–2778, 07 2020.

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

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

[15] Eric Hunsberger and Chris Eliasmith. Training spiking deep networks for neuromorphic hardware. *arXiv:1611.05141*, 2016.

[16] Xiping Ju, Biao Fang, Rui Yan, Xiaoliang Xu, and Huajin Tang. An fpga implementation of deep spiking neural networks for low-power and fast classification. *Neural Computation*, 32(1):182–204, 2020.

[17] Taro Kawao, Masato Neishi, Tomohiro Okamoto, Amir Masoud Gharehbaghi, Takashi Kohno, and Masahiro Fujita. Spiking neural network simulation on fpgas with automatic and intensive pipelining. In *2016 International Symposium on Nonlinear Theory and Its Applications, NOLTA2016*, pages 202–205, 11 2016.

[18] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined fpga acceleration: Status and future prospects. *ACM Trans. Reconfigurable Technol. Syst.*, 14(4), September 2021.

[19] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, 05 2015.

[20] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. 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*, FPGA '21, page 105–115. Association for Computing Machinery, 2021.

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

[22] Christian Mayr, Sebastian Höppner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv:1911.02385*, 2019.

[23] S. McKennoch, Dingding Liu, and L. G. Bushnell. Fast modifications of the spikeprop algorithm. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 3970–3977, 2006.

[24] 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. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[25] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeff Krichmar, Alex Nicolau, and Alexander Veidenbaum. 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*, 22:791–800, 08 2009.

[26] Morcos, Benjamin. Nengofpga: an fpga backend for the nengo neural simulator, 2019.

[27] D. Neil and S. Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, 2014.

[28] D. Neil and S. Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, 2014.

[29] Sathish Panchapakesan, Zhenman Fang, and Nitin Chandrachoodan. Easpinn: Effective automated spiking neural network evaluation on fpga. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 242–242, 2020.

[30] Sathish Panchapakesan, Zhenman Fang, and Jian Li. Syncnn: Evaluating and accelerating spiking neural networks on fpgas. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*.

[31] Sathish Panchapakesan, Zhenman Fang, and Jian Li. Syncnn: Evaluating and accelerating spiking neural networks on fpgas. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 286–293, 2021.

[32] Danilo Pani, Paolo Meloni, Giuseppe Tuveri, Francesca Palumbo, Paolo Massobrio, and Luigi Raffo. An fpga platform for real-time simulation of spiking neuronal networks. *Frontiers in Neuroscience*, 11:90, 2017.

[33] Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: Opportunities and challenges. *Frontiers in Neuroscience*, 12:774, 2018.

[34] Filip Ponulak and Andrzej Kasiundefinedski. Supervised learning in spiking neural networks with resume: Sequence learning, classification, and spike shifting. *Neural Comput.*, 22(2):467–510, February 2010.

[35] A. Rosado-Muñoz, M. Bataller-Mompeán, and J. Guerrero-Martínez. Fpga implementation of spiking neural networks. *IFAC Proceedings Volumes*, 45(4):139 – 144, 2012. 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control.

[36] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, and Michael Pfeiffer. Theory and tools for the conversion of analog to spiking convolutional neural networks. *arXiv:1612.04052*, 2016.

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

[38] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience*, 13:95, 2019.

[39] Amar Shrestha, Haowen Fang, Qing Wu, and Qinru Qiu. Approximating back-propagation for a biologically plausible local learning rule in spiking neural networks. In *Proceedings of the International Conference on Neuromorphic Systems*, ICONS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[40] Sumit Bam Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 1412–1421. Curran Associates, Inc., 2018.

[41] Evangelos Stromatias, Dan Neil, Francesco Galluppi, Michael Pfeiffer, Shih-Chii Liu, and Steve Furber. Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on spinnaker. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 07 2015.

[42] A. Taherkhani, A. Belatreche, Y. Li, and L. P. Maguire. Dl-resume: A delay learning-based remote supervised method for spiking neurons. *IEEE Transactions on Neural Networks and Learning Systems*, 26(12):3137–3149, 2015.

[43] Amirhossein Tavanaei and Anthony Maida. Bp-stdp: Approximating backpropagation using spike timing dependent plasticity. *Neurocomputing*, 330:39 – 47, 2019.

[44] Runchun M. Wang, Chetan S. Thakur, and André van Schaik. An fpga-based massively parallel neuromorphic cortex simulator. *Frontiers in Neuroscience*, 12:213, 2018.

[45] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in Neuroscience*, 12:331, 2018.

[46] Xilinx. Ultrascale architecture and product data sheet: Overview (ds890), 2021.

[47] Xilinx. Vitis ai: Adaptable and real-time ai inference acceleration, 2021.

[48] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2019.

[49] Zhenman, Fang. Ensc 462/894: Programming for heterogenous computing systems, 2020.