Algorithm Hardware Co-Design of Neural Networks for Always-On Devices

Pavan Kumar Chundi

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy under the Executive Committee of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021
Abstract

Algorithm Hardware Co-Design of Neural Networks for Always-On Devices

Pavan Kumar Chundi

Deep learning has become the algorithm of choice in many applications like face recognition, object detection, speech recognition, etc. because of superior accuracy. Large models with several parameters were developed to obtain higher accuracy, which eventually gave diminishing returns at very large training and deployment cost. Consequently, greater attention is now being paid to the efficiency of neural networks.

Low power consumption is particularly important in the case of always-on applications. Some examples of these applications are the datacenters, cellular base stations, battery-powered devices like implantable devices, wearables, cell phones and UAVs. Improvement in the efficiency of these devices by reducing the power consumed will bring down the energy cost or extend the battery life or decrease the form factor of these devices, thereby improving the acceptability and adoption of the device.

Neural networks are a significant component of the total workload in the case of IoT devices with smart functions and datacenters. Base stations can also employ neural networks to improve the rate of convergence in channel estimation. Efficient execution of the neural networks on always-on devices, therefore, helps in lowering the overall power dissipation.

Algorithm only solutions target CPU or GPU as a platform and tend to focus on the number of computing operations. Hardware only solutions tend to focus on programmability, low voltage operation, standby power reduction and on-chip data movement. Such solutions fail to take
advantage of the joint optimization of both algorithm and hardware for the target application.

This thesis contributes to improving the efficiency of neural networks on always-on devices through both algorithmic and hardware interventions. It presents works of algorithm-hardware co-design which can obtain better power reduction in the case of a smart IoT device, a datacenter and a small cell base station. It achieves power reduction through a combination of appropriate neural network algorithm and architecture, simpler operations and a reduction in the number of off-chip memory accesses.
Table of Contents

List of Figures ................................................................. v

List of Tables ................................................................. x

Acknowledgments ............................................................. xi

Chapter 1: Introduction ..................................................... 1

1.1 Neural Networks in always-on computing systems ................. 1

1.2 Challenges and Contributions of the thesis ......................... 4

   1.2.1 Ultralow Power Wake-Up Circuit for an IoT device ............ 4

   1.2.2 FPGA based DNN Training Acceleration in a Datacenter ..... 5

   1.2.3 Channel Estimation in a 5G Small Cell Base Station .......... 6

1.3 Previous co-design techniques ........................................ 6

   1.3.1 Binarization and In-Memory-Computing ....................... 6

   1.3.2 Pruning, Quantization and supporting Hardware ............... 7

1.4 Organisation of the thesis ............................................ 8

Chapter 2: Always-On Spiking Neural Network Classifier based on Spike-Driven Clock- and Power-Gating for an Ultra-Low-Power Intelligent Device .......... 9

2.1 Introduction ................................................................ 9

2.2 SNN Hardware Architecture ......................................... 12
Chapter 3: FPGA-based Acceleration of Binary Neural Network Training with Minimized
Off-Chip Memory Access

3.1 Introduction ...................................................... 36
3.2 Data Footprints in CNN Training ........................... 38
3.3 Algorithm ........................................................ 39
  3.3.1 Convolutional Layers during Training ................. 39
  3.3.2 Batch Renormalization Layer during training .......... 41
3.4 Processor Architecture ........................................ 44
  3.4.1 System Architecture .................................... 44
  3.4.2 Conv Microarchitecture Design ......................... 45
  3.4.3 Batch renormalization after convolutional layer (BRN_C) ........ 47
  3.4.4 Batch renormalization after fully connected layer (BRN_F) .... 49
  3.4.5 Max-Pooling Layers (MP) Design ....................... 51
  3.4.6 Fully-Connecting Layer (FC) and Loss Layer Design .... 52
List of Figures

1.1 Wearables like smart watch, body sensor and a smart phone are always-on and require low-power. They share some of the data through the internet to the cloud where neural networks are trained. .............................................. 2

1.2 The thesis attempts to tackle power consumption in always-on systems through algorithm hardware co-design for neural networks. .............................................. 4

2.1 The proposed SNN classifier architecture (bottom) with the maximum supported network size (top) for always-on functions like keyword spotting. ................. 12

2.2 (A) Neuron Block Architecture with wake-up circuits on the left, FSM with zigzag power gating on the right and Shared Clock generator on the bottom (B) Hidden Neuron FSM (C) Input Neuron FSM (D) Output Neuron FSM ................. 13

2.3 Waveforms for a hidden neuron FSM when (A) potential is less than the threshold and shared clock was disabled, (B) potential is less than the threshold and shared clock is running (assume other neurons in the same neuron block are active), (C) potential is greater than the threshold and shared clock is running. ................. 16

2.4 Impact of spike-driven clock gating and a combination of clock and power-gating on the standby power consumption of a hidden neuron ...................... 17

2.5 Energy consumption of the hidden neuron as a function of sleep time between two spikes, when the clock is free running and when clock gating and power gating are used. ................................................................. 17
2.6 (A) Proposed synapse block architecture (B) Arbiter FSM showing the fixed priority and Spkgen sub-FSM ................................................................. 19
2.7 Waveforms showing the operation of the synapse block when neuron 1 and 2 generate a request. Acknowledgement is given to neuron 1 because of higher priority. Spkgen sub-FSM executes while the acknowledgement is high. ....................... 20
2.8 Area Comparison between round-robin arbiter and fixed priority arbiter for a different number of inputs. The normalized area is obtained by dividing the cell area with the area of the fixed priority arbiter with 32 inputs. ......................... 21
2.9 Threshold and clock frequency optimization for no starvation during operation. Neurons are not starved when the arbiter clock frequency and threshold for the layer are high enough. ................................................. 22
2.10 SNN accuracy on 300 samples of the MNIST dataset and the total number of neuron requests dropped (Request Drop Count) per sample when the threshold value for the (A) first hidden layer, (B) second hidden layer and (C) third hidden layer are varied. ................................................................. 23
2.11 Ratio of the leakage of the peripheral circuits to that of the bitcells (Leakage Ratio) for different SRAM sizes where the number of rows is same as the number of columns. ......................................................... 24
2.12 Read Delay for different boosted supply voltages (VDDH). ......................... 24
2.13 (A) Die photo of the SNN Classifier with the Neurosynaptic Cores along with the Test Circuits. (B) Area break-down of the SNN Classifier. ......................... 26
2.14 Binary Coding in a BNN and spike-rate coding in an SNN. ......................... 27
2.15 Test chip with its peripheral circuits for communication with the FPGA for providing the inputs and capturing the outputs. ................................. 28
2.16 Clock frequency measurement and min frame length for 8-bit activations as a function of the supply voltage for the MNIST dataset. Frame size is the latency of the SNN classifier and it reduces as the supply voltage increases. ......................... 29
2.17 Power consumption of the chip as a chip of the input rate at two supply voltages
for KWS datasets. Power consumption increases linearly with the input rate. . . . . . 29
2.18 Accuracies across multiple benchmarks. . . . . . . . . . . . . . . . . . . . . . . . 30
2.19 ROC curves from KWS benchmarks. . . . . . . . . . . . . . . . . . . . . . . . . . 31
2.20 Accuracy across 0 to 40dB SNR levels. . . . . . . . . . . . . . . . . . . . . . . . 32
2.21 Power consumption of chip and error for the HeySnips dataset as a function of
activation precision. Stars denote the operating point used for comparison with
other works. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.22 Leakage power of the chip as a function of temperature at different supply voltages.
(B) Variation of Neuron Clock Frequency with temperature and supply voltage. . . 33
2.23 Measurement of the neuron clock frequency in 50 cores, i.e., 10 chips x 5 (cores/chip)
at a supply of 0.52 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.1 (a). Data size of each layer in training a CNN for a batch size of 16. (b) Data sizes
of training recent CNNs for a batch size of 32. The ratios of intermediate data size
to weight are indicated on top . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.2 Convolution-layer operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.3 System architecture of the accelerator and the peripherals . . . . . . . . . . . . . . 45
3.4 The microarchitecture of the Conv block in a Compute Lane. It performs most of
the computation for training except batch renormalization . . . . . . . . . . . . . . 47
3.5 Graph showing the dependencies between the variables in the backpropagation
through the renormalization layer that follows a convolutional layer. Red boxes
contain the data computed during the first pass. Green boxes contain data com-
puted during the second pass. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.6 Microarchitecture of the BRN block for forward propagation . . . . . . . . . . . 49
3.7 Graph showing the dependencies between the variables in the backpropagation through the renormalization layer that follows a fully connected layer. Red boxes contain the data computed during the first pass. Green boxes contain data computed during the second pass. Blue boxes contain the data computed during the third pass.

3.8 Blocks for computing \( \frac{\partial L}{\partial x} \) in BRN. (a) Datapath for P1. (b) Datapath for P2

3.9 Microarchitecture of the FC in the Compute Lane

3.10 The impact of batch size on accuracy in training the \( 1 \times N_0 \) network with batch normalization and batch renormalization.

3.11 Training time comparison of the FPGA and the GPU. (a) The impact of batch size on the time taken to train \( N_0 \). (b) The impact of the network size on training time for the FPGA and the GPU. The performance-optimal batch sizes are used.

3.12 Energy consumption comparison. (a) The FPGA based accelerator consumes 9.33X less energy than the GPU in training \( 1 \times N_0 \) CNN. (b) The FPGA achieves 5.77-9.33X smaller energy consumption across CNNs of different sizes. Energy-optimal batch sizes are used for the GPU.

4.1 mmWave communication systems where a base station and user equipment inside the car use beamforming and multipath propagation for communication.

4.2 Block Diagram indicating exemplary communication systems with the transmitter (TX) on the left, the obstacles in the center, and the receiver (RX) on the right with three beams from the transmitter to the receiver.

4.3 Model-based neural networks based on ISTA. (a) NET1: \( \eta_\theta \) non-linearity for all iterations ([48]) (b) NET2: \( \eta_\theta \) non-linearity for all iterations ([46]) (c) NET3: \( \eta_\theta \) non-linearity for all iterations (Variant of [48])(d) NET4 and NET4ss: Networks have a scaling factor \( \gamma \) for each iteration. NET4 uses \( \eta_\theta \) for all iterations while NET4ss uses \( \eta_\theta \gamma \) for 1st iteration and \( \eta_\theta \) for the rest (Proposed, Variant of [49]).
4.4 Comparison of the rate of convergence between ISTA and NET1 for on-grid communication data. ...................................................... 68
4.5 Impact of grid size on loss for NET1 and NET2. ................................. 68
4.6 Comparison of loss of different networks at the end of six iterations plotted against the number of trainable parameters. ............................ 70
4.7 Comparison of the cumulative number of non-zero values for two different networks across iterations. ...................................................... 71
4.8 Impact of using only topk values of \( \hat{\mathcal{G}} \) on the loss at the end of six iterations. ...................................................... 71
4.9 Image of \( \Sigma_D \) matrix with colored squares being non-zero values indicating a path from TX to RX for a certain AoA and AoD. The AoA and AoD of the channel change slowly as the car goes around the base station. ...................................................... 73
4.10 Loss as a function of the number of complex MAC operations without the use of topk optimization in \( \hat{\mathcal{G}} \) for different rates of change. The number of clusters corresponding to the operations is labeled below the green line. ...................................................... 75
4.11 Reduction in the number of complex MAC operations by the proposed techniques. Initial refers to initial computation that uses none of the proposed optimizations. TOPK indicates the use of only the top 22 values of \( \hat{\mathcal{G}} \). CT refers to cluster tracking. ...................................................... 76
4.12 Impact of quantization of the weights on the loss. ................................. 77
4.13 (a) Block diagram of the hardware system indicating the major components of the accelerator. (b) Flow chart depicting the sequence of operations followed in channel estimation in both initial computation and cluster tracking computation. ...................................................... 78
4.14 Reduction in DRAM accesses due to cache hits. ...................................................... 80
4.15 Impact of topk value on latency and the corresponding loss for (a) initial computation (b) cluster tracking computation. ...................................................... 83
List of Tables

2.1 Comparisons with recent KWS hardware ........................................ 35
2.2 Comparisons with recent SNN hardware ........................................ 35

3.1 Summary of resource usage in training CNNs of different sizes using BRN. ALM
here means Adaptive Logic Module and MLAB means Memory Logic Array Block. 55

4.1 Resource Utilization Table ......................................................... 82
4.2 Comparison Table ................................................................. 82
Acknowledgements

The PhD program has been an intense learning experience, one I could not have managed without the support of my family, teachers, friends and the nature itself.

I am forever grateful for the support I received and will continue to receive from those who matter the most, my parents, Murali Ramesh Chundi and Uma Maheswari Chundi and my brother, Dev Chundi.

I thank my advisor Prof. Mingoo Seok for providing me with the support and academic training. I am also thankful to Prof. Martha Kim, Prof. Xiaodong Wang, Prof. Xiaofan (Fred) Jiang and Prof. Aurel Lazar for agreeing to be a part of the thesis defense committee and sharing their valuable time.

I am glad for having had Seongjong Kim, Doyun Kim, Jiangyi Li, Teng Yang, Joao Cerqueira, Zhewei Jiang, Minhao Yang, Sung Kim, Dongkwun Kim, Ashish Shukla, Ayushparth Sharma, Daniel Jang, Dewei Wang, Bo Zhang and Chuan Tung Lin as colleagues and friends, all of whom were a source of support and learning at various points in time.
Chapter 1: Introduction

1.1 Neural Networks in always-on computing systems

Advancements in VLSI, storage and networking has made several always-on sensors and systems possible. Today there are several devices pertaining to personal health like pacemakers, smart watches (fig. 1.1), sensor systems for monitoring the safety of critical processes in industries, critical structures like dams, water and waste treatment plants and camera systems for surveillance. These devices need to have very low standby power to extend battery life and reduce the cost of maintenance.

The use of neural networks to implement intelligent functions on-chip increased the functionality of these devices and their adoption. Functions like keyword recognition, face recognition and anomaly detection which act as wake-up circuits in a smart phone, smart watch or surveillance and monitoring system are possible because of the accuracy of neural networks and are the key enablers in the reduction of the standby power of these devices.

The penetration of smart phones and devices for personal health monitoring, Internet Of Things (IoT) devices in smart homes and industries has generated a lot of data, providing opportunities to extract actionable insights for improving safety, efficiency and convenience. For example, the analysis and interpretation of the data collected by the sensors on devices like pacemakers and wearables provide useful insights into the health and fitness of an individual. Data from voice commands, appointments and past purchases can help in organising the schedule and making future purchases. Data from vibration, temperature and pressure sensors in critical industrial systems can inform the choices for better safety and reduction in downtime.

The analysis of large amounts of data requires the use of a powerful computer for training a neural network model. This is being performed in the cloud in addition to several other soft-
Figure 1.1: Wearables like smart watch, body sensor and a smart phone are always-on and require low-power. They share some of the data through the internet to the cloud where neural networks are trained.
ware services on account of improved network bandwidth and reliability, increasing the need for always-on computer systems. In addition, services like grammar and spell check in email, social media, translation, online shopping and entertainment all make use of Natural Language Processing (NLP), recommendation systems, face and object detection which rely heavily on neural networks. Improving the infrastructure in data centers can reduce standby power consumption thereby reducing the energy cost.

The advent of 5G communication is expected to increase the bandwidth and reduce the latency of the network. It will accelerate the adoption of IoT devices. Applications like augmented and virtual reality (AR/VR) and self-driving cars will make IoT devices an inalienable part of modern human society.

Small cell base stations is one of the key interventions needed to increase the bandwidth available to users. These base stations need to have a small form factor so that they can be made pervasive. These can be then located such that they can cater to very few users. They, therefore, need to be operated under smaller power budgets to reduce the surface area needed for cooling and also reduce the size of the battery backup in case of a power outage.

These base stations are always-on and can benefit from model-based neural networks to improve their efficiency. Model-based neural networks are gaining traction on account of their superior performance to conventional algorithms in terms of the convergence rate. The computational complexity of the optimization algorithms used for channel estimation, detection and precoding can likely be reduced with the use of model-based neural networks.

The thesis focuses on reducing the power consumption of the always-on components in the above-mentioned scenarios, i.e., a battery-powered IoT device, a server in a datacenter and a small cell base station (Fig. 1.2). We focus on neural network based wake-up circuit in the case of the IoT device, study FPGA as a platform for convolutional neural network (CNN) training in a datacenter and channel estimation in millimeter-wave based 5G small cell base station.
1.2 Challenges and Contributions of the thesis

1.2.1 Ultralow Power Wake-Up Circuit for an IoT device

A smart watch, a smart phone or a low power IoT device tends to have a low input activity rate. For example, a person may check his smart watch once an hour during the day and not at all during the night. It is therefore very critical to lowering the standby power of the device. The standby power is typically reduced by turning off the power-hungry portions of the device. These circuits are turned on by wake-up circuits only when needed. The always-on wake-up circuits, implement functions like keyword recognition. The wake-up circuits, therefore need to consume extremely low leakage power.

The thesis tries to tackle this challenge by using the concept of spiking neural networks (SNN) and spike-based event-driven operation on hardware. It shows that the combination of SNN and accompanying hardware can achieve state-of-the-art power consumption at real-time operation for a wake-up circuit. The SNN we employ uses integrate and fire neuron model which uses spikes for computation and communication. Spike based operation is used to implement fine-grained clock
and power-gating. This work was published in *IEEE Asian Solid State Circuits Conference* [1] and a journal version of this work is under review as of April 2021.

1.2.2 FPGA based DNN Training Acceleration in a Datacenter

The energy cost of a datacenter is the most significant portion of the total cost. According to Song et al., 2015 [2] 40% of the total energy is consumed for cooling. Reducing the power consumption of the servers in a datacenter therefore has multiple benefits. It reduces both the energy consumed by the servers and also the cooling required for the server systems.

The increase in the use of services like voice commands, face recognition, etc. led to a sharp increase in the deep neural network (DNN) computation [3]. So, datacenter operators have shown a great deal of interest in platforms that improve the performance and energy efficiency of DNNs. For example, Google uses Tensor Processing Units (TPU)[3], [4] for the training and inference of its DNNs, Facebook uses GPUs and CPUs for training [5] and CPUs for inference while Microsoft uses FPGAs for inference [6] and perhaps GPU for training.

The power consumed by an FPGA tends to be lower than that of a GPU, making it an alternative one would like to explore to lower energy costs. Further, the FPGA can be reprogrammed to cater to new DNN architectures as they evolve. While the case for the use of FPGAs for inference is clear, the benefits in the case of training are not obvious. This thesis examines the feasibility of FPGAs as a platform for training binary weight convolutional neural networks (CNN).

The thesis presents an attempt to study the benefit of using an FPGA for training CNNs in comparison to a GPU. It presents an accelerator that stores all the weights on-chip and makes use of batch renormalization to reduce the amount of memory needed to store intermediate results. The assumption in the design of the accelerator gives it the advantage of reduced off-chip memory accesses and being able to use FPGA’s programmable fabric for binary weight-multibit activation multiply and accumulate (MAC) computation. We observe that FPGA may offer limited benefit in the case of small neural networks but no benefit in the case of large neural networks. This work was published in *ACM/IEEE International Symposium on Low Power Electronics and Design* [7].
1.2.3 Channel Estimation in a 5G Small Cell Base Station

Higher carrier frequencies are used to avail greater bandwidth in 5G cellular networks. But, the large path loss at millimeter-wave (mmWave) frequency makes it necessary to use a large array of antennas to obtain gain using beamforming. Hybrid beamforming and massive MIMO is used to localize the signals in space and make use of multipath propagation to improve spectral efficiency. Small cell base stations are used to cater to a small set of users to make more bandwidth available per user. These base stations need to be efficient and small to allow easy deployment. But, the strict latency constraints and the high computational demands due to large antenna array sizes makes it challenging.

This thesis focuses on the problem of channel estimation at mmWave frequencies. Channel estimation when there are a lot of transmit and receive antennas either becomes computationally expensive or spectrally inefficient based on the approach used. In this thesis, we focus on compressed sensing based approach as it has higher spectral efficiency and make it computationally tractable. We use model-based neural networks to get a good convergence rate. We exploit the sparsity inherent in mmWave communication channels to reduce the computational complexity by two orders of magnitude and design a hardware subsystem on FPGA to reduce the off-chip memory accesses by about two orders of magnitude. This work is under review as of April 2021.

1.3 Previous co-design techniques

1.3.1 Binarization and In-Memory-Computing

Neural networks with binary weights [8] and later neural networks with binary weights and activation [9] were introduced to improve the efficiency during inference. These typically involve training networks with a larger number of neurons and use binary weights during forward and backpropagation and higher precision during weights update. Only the most significant bit of the weights are needed for inference, reducing the overall memory footprint pf the neural network parameters while making each MAC operation simpler. Binary neural networks allow for more
efficient use of memory bandwidth and can provide much better performance than a CPU or GPU when used with custom hardware.

In-Memory-Computing (IMC) is a recent hardware technique [10] where the storage of weights and MAC computation is achieved within a special SRAM macro. It achieves very high performance per watt. It gives the most benefit for inference in small binary neural networks where all the weights can be stored in the on-chip SRAM. Additional techniques are now underway to make the technique fully digital and hence more robust while trying to meet variable precision requirements.

1.3.2 Pruning, Quantization and supporting Hardware

Weight pruning [11] was introduced to reduce the number of parameters that neural networks use. Pruning during training involves the identification of weights of small magnitude typically in the fully connected layers of the network and forcing them to be zero. Pruning can sometimes induce sparsity of over 80% depending on the number of excess parameters in the original network.

Significant sparsity in the weights can help in achieving a high compression ratio (or small memory footprint) when techniques like Huffman coding are used. Sparsity can also be used to reduce the precision needed for an accurate representation of the weights. Low memory footprint during inference can help in better use of memory bandwidth. It does not avoid fetching weights that are zero from off-chip to on-chip because of irregularity in pruning, which makes anticipating the address of non-zero values not possible in the general case.

Hardware needs to support compression and decompression when techniques like Huffman coding are used to improve the utilization of memory bandwidth and reduce the cost of communication[12]. Further, zero skipping reduces the latency of computation by avoiding MAC operations involving weights whose values are zero. Hardware needs to optimized to compute based on the precision of the weights obtained after the pruning and quantization to translate the benefits of low precision into a small area and low latency.
1.4 Organisation of the thesis

Chapter 1 provides an introduction to the use of neural networks in always-on systems. It briefly describes the problems that the thesis tries to solve in the context of a low power IoT device, a datacenter and a small cell base station in 5G millimeter-wave communications. It also introduces the popular existing optimization techniques for neural network computation.

Chapter 2 presents a combination of spiking neural networks and spike-driven clock- and power-gating techniques. It presents an ultralow-power wake-up circuit on a chip that operates by consuming sub-300 nW power which is relevant for a low power IoT device.

Chapter 3 presents an attempt to study the feasibility of an FPGA as a platform for accelerating binary weight CNNs. It presents an accelerator that stores all the weights on-chip and makes use of batch renormalization to reduce the amount of memory needed to store intermediate results. The results are compared with a GPU to understand if an off-the-shelf FPGA board is a promising platform for training.

Chapter 4 presents a set of techniques to reduce the computational complexity of model-based neural networks used for channel estimation in a small cell base station in 5G millimeter-wave communications. It demonstrates the techniques using a subsystem on an FPGA and compares it to CPU and GPU.

Lastly, we discuss the conclusions that can be drawn from the research work presented in the thesis.
Chapter 2: Always-On Spiking Neural Network Classifier based on Spike-Driven Clock- and Power-Gating for an Ultra-Low-Power Intelligent Device

This chapter presents a novel SNN classifier architecture for enabling always-on artificial intelligent (AI) functions, such as keyword spotting (KWS) and visual wake-up, in ultra-low-power Internet-of-things (IoT) devices. Such always-on hardware tends to dominate the power efficiency of an IoT device and therefore it is important to minimize its power dissipation. A key observation is that the input signal to always-on hardware is typically sparse in time. This is a great opportunity that a spiking neural network (SNN) classifier can leverage because the switching activity and the power consumption of SNN hardware can scale with spike rate. To leverage this scalability, the proposed SNN classifier architecture employs event-driven architecture, especially fine-grained clock generation and gating and fine-grained power gating, to obtain very low static power dissipation. The prototype is fabricated in 65 nm CMOS and occupies an area of 1.99 mm². At 0.52 V supply voltage, it consumes 75 nW at no input activity and less than 300 nW at 100% input activity. It still maintains competitive inference accuracy for KWS and other always-on classification workloads. The prototype achieved power consumption reduction of over three orders of magnitude over the state-of-the-art for SNN hardware and of about 2.3X the state-of-the-art KWS hardware.

2.1 Introduction

An SNN classifier is an attractive option for ultra-low-power intelligent IoT devices. It is promising especially for always-on functions due to their spike-based operation for computation and communication, allowing their switching activity and power to scale smoothly with the input
activity rate. An SNN, therefore, can be suited for applications like keyword spotting or face recognition in surveillance, thanks to their event-driven operation.

SNN based hardware works so far, however, focused on either the acceleration of neural simulations or the improvement of both performance and energy efficiency. In other words, they are not designed for always-on function. For example, Neurogrid [13] targets large-scale neural simulations. It employs analog neurons and address event representation (AER) for communication, the latter using a multi-bit bus. SpiNNaker [14] also targets neural simulation and employs an array of embedded digital processors communicating asynchronously. On the other hand, TrueNorth [15] was designed to be a scalable low power neurosynaptic inference engine for SNNs. The architecture was event-driven and employed synchronous circuits for computation blocks and asynchronous circuits for communication. Also, Tianjic chip was designed to support the inference with both neuromorphic and deep-learning models [16]. Some works proposed architectures for both training and inference of SNNs. Chen et al. (2018)[17] presented an SNN accelerator with on-chip spike-timing-dependent plasticity (STDP) based learning. This chip has 64 cores that communicate using a network-on-chip (NoC) with each core supporting 64 leaky integrate and fire (LIF) neurons. Also, Loihi [18] was designed to support a variation of the current based dynamics LIF neuron model and a wide range of synaptic learning rules for both supervised and unsupervised learning. It is built for performance. It has 128 cores, three x86 cores, off-chip interfaces and an asynchronous NoC for communication between cores. Also, Seo et al. (2011)[19], implemented a scalable architecture with a set of 256 neurons and transposable memory for synapses in near-threshold voltage (NTV) circuits. It mapped an auto-associative memory model. Some other works implemented different learning rules for on-chip training. Knag et al. (2015)[20], implemented a feature extractor based on a sparse coding algorithm using LIF neurons. Park et al. (2019)[21], developed a new neuromorphic training algorithm and hardware which supports low overhead on-chip learning. Some of these chips e.g., ([15], [18]) employ asynchronous logic such as quasi-delay-insensitive (QDI) dual-rail dynamic logic or bundled data communication. Dual-rail dynamic logic circuits are, however, significantly bulkier and power-hungrier than the single-rail
static counterpart and also not very voltage-scalable ([22], [23]) and bundled data communication incurs significant overhead because of the handshake. Some other chips employ power-efficient static logic ([18], [17], [16], [21]), but they target high throughput, not always-on function. As a result, they exhibit the power consumption of more than tens of mW, which makes it difficult to use them for always-on function.

In this chapter, we focus on ultra-low-power always-on inference hardware and thereby propose an SNN classifier consuming less than 300 nW. Our architecture uses fully spike-based event-driven operation and only static logic operating at a near-threshold voltage (NTV) to achieve such low power. Specifically, our design is centered around the neurosynaptic core. It is implemented using static gates and spike-driven i) spatiotemporally fine-grained clock-generation, ii) clock-gating and iii) power-gating. Also, the communication between neurosynaptic cores is free from information loss due to the collision of spikes, despite using only wires to connect the cores. The architecture exhibits active power consumption that is proportional to the input rate due to its event-driven nature.

We also employ the technique in [24] to train a neural network with binary weights and use the weights for the SNN we intend to deploy. Binary weights, which is a recent development in deep learning for making inference efficient [8], are of special interest because of their reduced memory footprint and simple computations, making them well suited for low power hardware. On the other hand, we keep the activations as spike-rate-coded multi-bit values, which improves the model’s inference accuracy.

We prototyped an SNN classifier in 65-nm CMOS technology. It has 5-layers and a total of 650 neurons and 67,000 synapses. It consumes 2.3X to 7X lower power at state-of-the-art accuracies on two well-known KWS benchmarks, i.e., Google Speech Command Dataset (GSCD) for multi-keyword recognition [25] and HeySnips for single-keyword spotting [26].

In the remaining portion of this chapter, we will present our SNN hardware architecture and the experimental results. In Sec. 2.2, we discuss the high-level SNN classifier architecture, elaborate on each of the components of the neurosynaptic core and introduce the experiment setup. In Sec.
2.2 SNN Hardware Architecture

The SNN classifier in our proposed design is depicted in Figure 2.1. It can support a fully connected network as large as 256-128-128-128-10 with binary weights onto five neurosynaptic cores. We map each layer of the network to a different neurosynaptic core. The neuron block in the neurosynaptic core for the input layer has 256 neurons while the ones for the hidden layers, each contains 128 neurons. The size of each layer can be altered to make it smaller by configuring the neurosynaptic core. The input and hidden neurosynaptic cores have a neuron block and a synapse block while the output neurosynaptic core has only a neuron block. A neuron block contains all the IF neurons in that layer, a synapse block has (i) an arbiter, (ii) an SRAM storing up to 256-by-128 binary weights for the input neurosynaptic core and up to 128-by-128 binary weights for the
hidden neurosynaptic cores, and (iii) a spike generator that simultaneously generates 128 spikes.

2.2.1 Neuron Block

We propose spike-event-driven architecture. Figure 2.2 shows the neuron block based on that architecture. Each neuron has i) asynchronous wake-up circuits and ii) a synchronous finite state machine (FSM). Also, all the neurons in a neuron block share a clock generator based on a ring oscillator. The architecture contains fine-grained clock-generation and clock-gating circuits based on spike input as an event. In the absence of input spikes, each neuron gates its clock and also power-gates the non-retentive parts of the neuron using zigzag power-gating switches (PGSs) [27], to reduce static power dissipation.

The wake-up circuit of each neuron (Figure 2.2 left) has the static flip-flops, $\text{FF}_{+1}$ and $\text{FF}_{-1}$, which detect the rising edge of the incoming spikes from two inputs, $\text{Spk}_{+1}$ and $\text{Spk}_{-1}$. Positive
spikes which increase the potential of the neuron are directed to $Spk_{+1}$ and negative spikes which
decrease the potential to $Spk_{-1}$. As shown in Figure 2.3(A), the detection of a spike makes the
output of the clock-enable flip-flop ($FF_{clk-en}$) high. It also un-gates the PGS of the neuron. Thanks
to the zigzag PGS, the ungating (i.e., wake-up) is done in a single clock cycle.

This process starts up the shared clock generator in the neuron block if it was not already started
by another neuron. The first falling edge of the clock generator’s output after an active $FF_{clk-en}$
sets the un-gate flip-flop ($FF_{clk-ug}$) to high, ungating the clock signal that goes into the FSM. The
use of $FF_{clk-ug}$ ensures that there is a complete low phase of the clock signal before the rising edge
at the clock input of the FSM, giving sufficient setup time to the flip-flops in the FSM.

Once waken up, the neuron FSM gets executed. Figure 3 shows the waveforms. The FSMS
are slightly different for the input core, hidden cores, and the output core (Figure 2.2 (B, C, D).
In the case of hidden neurons, the FSM, as shown in Figure 2.2(B), enters the Potential Update
state on receiving the positive edge of the clock. The neuron’s potential is increased or decreased
by one based on the input spike’s type. Then, the neuron’s potential is compared with the preset
threshold (TH) in Check Pot. State. If the potential is less than the threshold, the FSM goes back to
the Start/Standby state while resetting all the flip flops in the wake-up circuit ($FF_{+1}$, $FF_{-1}$, $FF_{clk-en}$
and $FF_{clk-ug}$; find them in Figure 2.3(A, B)). Otherwise, it resets the neuron’s potential to zero
and also $FF_{+1}$ and $FF_{-1}$ in the Potential Reset state, allowing for receiving the next spike (Figure
2.3(C)). The FSM then enters the Spk Req state, asserts the firing request ($Req_i$) and waits for the
acknowledgement ($Ack_i$) from the arbiter in the synapse block. While waiting for $Ack_i$, if the FSM
receives a new spike it enters another state, Potential update 2, where the neuron’s potential is cal-
culated. Once $Ack_i$ from the arbiter is received, the neuron’s FSM goes back to the Start/Standby
state after resetting the flip-flops ($FF_{+1}$, $FF_{-1}$, $FF_{clk-en}$, $FF_{clk-ug}$) in the asynchronous wake-up
circuits. This cuts off the clock and power to the neuron.

The operation of input and output neurons are slightly different. The input neuron’s FSM is
depicted in Figure 2.2(C). On receiving a spike, the FSM directly enters the Spk Req state, asserts
a firing request ($Req_i$) and waits for an acknowledgement ($Ack_i$) from the arbiter in the synapse
On receiving Ack_i from the arbiter, the FSM resets FF_{+1}, FF_{-1}, FF_{clk-en}, FF_{clk-ug} and goes back to the Start/Standby state. Again, in this state, the clock and power to the neuron are gated. The output neuron’s FSM is depicted in Figure 2.2(D). Upon receiving a spike, the FSM enters the Potential Update state, then in the next state, it resets FF_{+1}, FF_{-1}, FF_{clk-en}, FF_{clk-ug} and then goes back to the Start/Standby state. The neuron with the highest potential determines the classification result.

This spike-based event-driven operation enables large power reduction and energy savings. First of all, if the input has no activity, which is common for always-on applications, the proposed neuron architecture can enjoy a very long sleep time. The hidden neuron without spike-event-driven power management would consume 1.16 nW (Figure 2.4). The proposed clock-generation/gating enables 74.6% power savings and the zigzag power gating provides an additional 17.68%, resulting in an overall power reduction of 4.8X when the circuit is not processing any spikes.

If the input has non-zero activity, the proposed neuron would experience shorter sleep time but it still saves a considerable amount of energy. For the targeted benchmarks, the shortest idle time between two spikes per neuron is estimated to be around 4 ms at the maximum input rate. Figure 2.5 shows the energy consumption of the hidden neuron as a function of sleep time. We consider the hidden neurons with no low power technique used, with only clock gating used, and with both clock and power gating used. We can observe that the neuron with clock and power gating can save energy consumption by 4.35X for 4 ms sleep time. Also, if the sleep time of the neuron is greater than ~1.3 ms, we stand to gain due to the proposed fine-grained clock and power gating.

2.2.2 Synapse Block

The synapse block was also designed based on the event-driven architecture. Figure 2.6(A) shows its microarchitecture. The synapse block has an arbiter FSM, an SRAM array, spike generators, and its own clock generator. A request signal (Req_i) from the neurons within the same neurosynaptic core starts the local clock generator of the synapse block, which makes the arbiter FSM get executed. In case multiple neurons assert Req_i, the arbiter handles the requests, i.e.,
Figure 2.3: Waveforms for a hidden neuron FSM when (A) potential is less than the threshold and shared clock was disabled, (B) potential is less than the threshold and shared clock is running (assume other neurons in the same neuron block are active), (C) potential is greater than the threshold and shared clock is running.
Figure 2.4: Impact of spike-driven clock gating and a combination of clock and power-gating on the standby power consumption of a hidden neuron.

Figure 2.5: Energy consumption of the hidden neuron as a function of sleep time between two spikes, when the clock is free running and when clock gating and power gating are used.
grants access to the single-port weight SRAM based on a fixed priority. To serve n-th neuron’s request, the arbiter asserts the n-th wordline (WL$_n$) and loads the binary weights on the read-bitlines (RBLs) whose values are captured by the flip flops. Each row of the SRAM contains 128 binary weights which are equal to the number of neurons in the neurosynaptic core. This means all the weights needed to serve a neuron’s request are obtained in a single access. The spike generator uses these weight values to generate 128 positive or negative spikes to the neuron in the next layer. It is to be noted that the spike generator is connected to the neurons in the next neurosynaptic core by wires only. The arbitration among the neurons also has the effect of managing access to these wires by allowing only one spike per wire at once. Therefore, we avoid the loss of information due to the collision between two (post-synaptic) spikes travelling to a single neuron at the same time.

When the local clock generator is enabled, the arbiter FSM gets executed (Figure 2.6(B)). The FSM starts in the **Start/Standby** state and when the positive edge of the clock arrives the FSM moves to one of the Ack[i] states. The exact Ack[i] state is determined based on the indices of the neurons making the request. The neurons with a smaller index have a higher priority.

The waveforms in Figure 2.7 show an exemplary operation of the circuit when neuron 1 and neuron 2 generate a request at the same time. We can see from the figure that once the requests are generated, the FF$_{clk-en}$ flip-flop is set. This turns on the local clock generator and disables power gating. Acknowledgement (Ack$_1$) is provided to neuron 1 because it has a higher priority determined in design time. The same acknowledgement signal acts as the read wordline for the SRAM.

The arbiter then starts executing the spike generation sub-FSM (Spkgen). The Spkgen waveform in Figure 2.7 shows the state of the sub-FSM. When Spkgen is in the state St1, weight values are captured in flip-flops and when Spkgen enters state St2, 128 positive or negative spikes (spk$_{+/-1}$) are generated for all the neurons in the next layer based on the weight values. The arbiter acknowledges back to neuron 1 by asserting Ack$_1$ while the spike generator goes through the states St0, St1 and St2. Ack$_1$ stays high until the request from the neuron is high or the spike generation completes, whichever is later. If there are any outstanding Req$_i$, the arbiter FSM continues to serve,
Figure 2.6: (A) Proposed synapse block architecture (B) Arbiter FSM showing the fixed priority and Spkgen sub-FSM
Figure 2.7: Waveforms showing the operation of the synapse block when neuron 1 and 2 generate a request. Acknowledgement is given to neuron 1 because of higher priority. Spkgen sub-FSM executes while the acknowledgement is high.

otherwise, the clock and power are disabled.

We chose the fixed priority arbiter instead of a round-robin one as the area saving is about 17X for 128 inputs. Figure 2.8 shows the area of the round-robin arbiter and fixed priority relative to a fixed priority arbiter with 32 inputs. We can see from Figure 2.8 that the area required for a round-robin arbiter is superlinear as a function of the input size while the area for a fixed priority arbiter increases approximately linearly with the number of inputs.

The fixed priority scheme, however, could cause the neuron with the lowest priority to starve, i.e., its requests may not be served if the arbiter is busy serving the requests of the neurons with higher priority. In our design process, therefore, we ensure the fixed priority arbiter starves no neuron. To do so, we have chosen the thresholds of the neurons and the clock frequencies of
the neuron and synapse blocks so that spikes are not missed while the neurons are waiting for acknowledgement from the arbiter.

The process to determine those key design parameters is as follows. We can start by formulating the number of requests in the i-th layer ($N_{req,i}$), which is:

$$N_{req,i} = \frac{N_{spk,i} \times N_{nrrn,i}}{TH_i},$$  \hspace{1cm} (2.1)$$

where $N_{spk,i}$ is the number of incoming spikes per frame (a time duration in which a feature vector is generated) and per neuron, $N_{nrrn,i}$ is the number of neuron, and $TH_i$ is the threshold of neurons in the i-th neurosynaptic core. On the other hand, the number of requests that the arbiter can serve ($N_{serve,i}$) can be formulated to be:

$$N_{serve,i} = \frac{f_{clk,a} \times T_{frame}}{N_{cyc,a}}$$  \hspace{1cm} (2.2)$$
Figure 2.9: Threshold and clock frequency optimization for no starvation during operation. Neurons are not starved when the arbiter clock frequency and threshold for the layer are high enough.

where $N_{\text{cyc},a}$ is the number of cycles that the arbiter consumes to serve one request, $T_{\text{frame}}$ is the frame length, $f_{\text{clk},a}$ is the arbiter’s clock frequency.

If $N_{\text{req},d}$ (Eq. 2.1) exceeds $N_{\text{serve},d}$ (Eq. 2.2), starving happens. When starving happens, incoming spikes can get dropped as the arbiter is not fast enough to serve all the requests. We can ensure $N_{\text{req},d}$ does not exceed $N_{\text{serve},d}$ by increasing $T_{i}$ or increasing $f_{\text{clk},a}$. The former, however, can incur a degradation in the accuracy. This is because the increase of $T_{i}$ would reduce the number of output spikes generated in the i-th layer. On the other hand, increasing $f_{\text{clk},a}$ increases the power consumption of the synapse block. Therefore, we swept $T_{i}$ and $f_{\text{clk},a}$ values and as shown in Figure 2.9 we found a few optimal operating points which are used in this chip.

Indeed, the threshold value affects the number of spikes generated in a layer and this affects the inference accuracy. Recall that the activations are spike-rate-coded multi-bit values and the reduction of the number of spikes leads to smaller bits. We can observe the impact of the choice of threshold values for different hidden layers on the accuracy of the SNN classifier in Figure 2.10. It shows through simulation the accuracy obtained on 300 test samples of the MNIST dataset [28].
Figure 2.10: SNN accuracy on 300 samples of the MNIST dataset and the total number of neuron requests dropped (Request Drop Count) per sample when the threshold value for the (A) first hidden layer, (B) second hidden layer and (C) third hidden layer are varied.

In Figure 2.10(A), we varied the threshold for the first hidden layer while the thresholds of the second and third hidden layer are chosen to be 16 and 8. We can observe that the accuracy of the classifier is worse for the small (roughly < 5) and the large threshold values (roughly > 40). Figure 2.10(A) also shows the total number of neuron requests dropped across layers as a function of the threshold of the first hidden layer. It indicates that if the threshold is too small, too many spikes are produced, causing arbiter starvation, which leads to too many neuron requests being dropped, resulting in a deterioration in the accuracy. Figure 2.10(B) and Figure 2.10(C) show a similar trend when the threshold for the second hidden layer and the third hidden layer is varied. But, we can also observe that the impact of the threshold of the second and third hidden layer on accuracy is relatively small if a proper threshold is chosen for the first hidden layer. This is because the number of spikes and hence the number of neuron requests are large in the first hidden layer. The threshold of the first hidden layer determines the number of requests dropped in the first hidden layer which is also a large portion of the total number of requests dropped.
Figure 2.11: Ratio of the leakage of the peripheral circuits to that of the bitcells (Leakage Ratio) for different SRAM sizes where the number of rows is same as the number of columns.

Figure 2.12: Read Delay for different boosted supply voltages (VDDH).
The chip has 65.25 Kb of SRAM and so it was important to minimize SRAM leakage power dissipation. We designed the SRAM based on the circuit described by Cerqueira et al., 2019[29] for ultra-low-power operation. High threshold voltage (HVT) transistors with three times minimum length were used for the bitcell to reduce leakage. The buffer in the peripheral circuits employed zig-zag power gating with cut-off transistors separate for each row, ensuring fast wake-up.

We chose to have all the weights for a layer in a single SRAM macro size instead of smaller banks. Figure 2.11 shows the ratio of the leakage of the peripheral circuits to the leakage of the bit cells as a function of the number of rows. From Figure 2.11 we can see that because of this the leakage of peripheral circuits would get amortized among more bitcells, helping us in the overall objective of reducing the leakage.

We use supply voltage boosting during a read operation to speed up the charge or discharge of the read bitline. The delay in the read operation arises mostly from charging the read wordline and charging or discharging the read bit line. Supply voltage boosting was needed to improve the speed of read operation which took a hit due to the use of a single SRAM macro for storing all the weights in a layer. In Figure 2.12 we can observe that on increasing the boosted supply voltage (VDDH) we will eventually be limited by the time taken to charge the wordline. We operate our circuit so that read delay is not the critical path in the design by choosing a high enough VDDH, which is roughly 0.8 V if the regular VDD is set to 0.52 V.

2.3 Experiment Setup

2.3.1 Chip Prototype

We prototyped the test chip in a 65 nm CMOS process. Figure 2.13(A) shows the die photo with the boundaries of different cores marked. The input and the hidden cores have the dimensions 0.7 mm x 0.7 mm. The output neurons take an area of 0.0276 mm$^2$. Each of the hidden cores is logically equivalent but have a different layout to simplify the routing. The total core area is around 1.99 mm$^2$. The area breakdown of the chip can be seen from the pie chart in Figure 2.13(B). The chip also contains the input decoder and the output encoder for reducing the number of I/O the
chip would require and also a scan chain to configure the thresholds of the neurons in different 
neurosynaptic cores and write the weights into the SRAMs.

2.3.2 Input Preparation

We envisioned the SNN to interface directly with a spike-generating feature-extraction front 
end such as the ones discussed in ([30]; [31]). For our experiments, we used the software model 
([31]) to generate features of size 16. Each dimension of the feature captures the energy at a central 
frequency in the form of the number of spikes that are generated by the analog front end in a certain 
time frame. The central frequencies of the 16 channels are geometrically scaled from about 100 
to 5 kHz. We configure the front end so that the number of spikes can be represented by 6-bits, 
i.e., each element in the feature has 6-bit precision. We set a frame length $T_{frame}$ of 80 ms with 
no overlap between successive frames, based on the length of the audio clip of the datasets and the 
dimension of the input layer that the chip supports. In GSCD and HeySnips datasets, each keyword 
audio sample is roughly 1s. We send the current frame together with the past 15 frames to SNN 
classifier so that the input dimension is 256.
2.3.3 Training

We train a binary neural network (BNN) model that uses binary weights (+1, -1), has no bias and 6-bit ReLU activation (Figure 2.14) [24]. The network structure is equivalent to the SNN model we deploy. The BNN provides the weights for the SNN model. The 6-bit activations in the BNN are encoded for the SNN using spike-rate, e.g., 010000 \(_{(2)}\) is mapped to 16 spikes/frame. We set the threshold of the neurons in each layer such that each neuron generates at most 63 spikes/frame, which matches the 6-bit activation of the BNN model. This is possible because in the SNN model, as spikes pass through the neurons in a layer, the number of spikes scales roughly by the ratio of the threshold. We can easily change the activation precision after deployment for different models by configuring the thresholds. For example, we use 8-bit activation with the Tframe of 0.5 s for the MNIST grayscale.

2.3.4 Inference Testing

Altera DE1 board containing a Cyclone II field programmable gate array (FPGA) chip is used to interface with the input decoder and the output encoder in the SNN chip as shown in Figure
2.15. The clock for the input decoder and the output encoder is obtained from the FPGA. LabView is used to configure the scan-chain and write weights into the SRAMs in the neurosynaptic cores. FPGA board reads out the input data from its memory. It sends an 8-bit signal to the input decoder identifying the neuron which is supposed to receive a spike and another signal identifying whether the spike is an incrementing spike or decrementing spike. The input decoder then sends a pulse to the appropriate neuron. After an interval equal to frame size the FPGA enables the output encoder to read out the potential of the output neurons in a serial fashion to the FPGA. Then the FPGA resets the potential in the neurons and sends in the next set of spikes to the input decoder.

2.4 Results

Most of the results are based on a supply voltage of 0.52 V and the clock frequency of the neuron block of 70 kHz and that of the synapse block of 17 kHz, while the chip can operate at other supply voltages and achieve different frequencies (Figure 2.16). The SNN chip output is observed after a time interval equal to the frame size in Figure 2.16. The minimum frame size we can use to operate the chip with 8-bit activations reduces with a supply voltage as the speed of the circuit increases.

As shown in Figure 2.17, the SNN chip power consumption scales with the input rate and dissipates a power of 75 nW when there is no input and power of 220 nW when running a KWS dataset like GSCD or HeySnips.
Figure 2.16: Clock frequency measurement and min frame length for 8-bit activations as a function of the supply voltage for the MNIST dataset. Frame size is the latency of the SNN classifier and it reduces as the supply voltage increases.

Figure 2.17: Power consumption of the chip as a chip of the input rate at two supply voltages for KWS datasets. Power consumption increases linearly with the input rate.
We can see the accuracy of the chip across the different classification tasks in Figure 2.18. In GSCD, the SNN can recognize four keywords ("yes", "stop", "right", and "off", arbitrarily chosen) and fillers with an accuracy of 91.8%. The SNN architecture we use is 256-128-128-128-5 and configure the thresholds to be (1,28,18,10) where 1 is the threshold for the input layer (fixed) and the rest are for the hidden layers. For the HeySnips dataset, the chip can recognize one keyword ("Hey Snips") and fillers with an accuracy of 95.8%. And in the MNIST grayscale dataset, we downsample the image size to 16x16 by utilizing 2x2 max-pooling so that we can match the image with the size of the input layer of the chip. The trained SNN structure is 256-128-128-128-10 with the thresholds of (1,24,12,8) and it gives an accuracy of 97.6%.

Figure 2.19 shows the receiver operating characteristic (ROC) curve for GSCD and HeySnips. It shows the false reject rate (FRR) as a function of the false alarm rate (FAR) for 1-hour-long audio obtained by concatenating the test set samples. In addition, we measured the accuracy of the chip in the presence of noise by mixing the speech audio with white noise at various SNRs. We adopted noise-dependent training for this experiment [31]. The SNN classifier chip achieves reasonably high accuracy across 0 to 40dB SNR levels as shown in Figure 2.20. The configurability
of the thresholds of different layers in the SNN classifier architecture allows us to change the data precision after deployment. This can be used to trade-off accuracy for power savings as shown in Figure 2.21. We choose a precision of 6-bit which is a good compromise between the power consumption and the accuracy.

We also measured the impact of temperature on the leakage power dissipation and the speed of our circuits. Figure 2.22(A) shows the leakage power of the circuit while Figure 2.22(B) shows the clock frequency of the circuit at different supplies as the temperature is varied. We can notice that it is beneficial to operate the circuit at a lower supply when the temperature is high and at a higher supply when the temperature is low, to obtain the performance needed while keeping the power consumption low.

On the other hand, Figure 2.23 shows the variation in the neuron clock frequency in roughly 50 cores across 10 chips at a supply of 0.52 V. The chip used for comparison performs one standard deviation better than an average chip.
Figure 2.20: Accuracy across 0 to 40dB SNR levels.

Figure 2.21: Power consumption of chip and error for the HeySnips dataset as a function of activation precision. Stars denote the operating point used for comparison with other works.
Figure 2.22: Leakage power of the chip as a function of temperature at different supply voltages. (B) Variation of Neuron Clock Frequency with temperature and supply voltage.

Figure 2.23: Measurement of the neuron clock frequency in 50 cores, i.e., 10 chips x 5 (cores/chip) at a supply of 0.52
2.5 Summary

Prior works on SNN hardware focused on non-always-on application ([15], [18], [17]), support for on-chip training ([17], [18], [21]) and support for both deep learning and neuromorphic workloads [16]. The absence of any prior work on SNNs for targeting always-on hardware motivated us to explore a new architecture for SNNs.

We presented a fully spike-event-driven SNN classifier for an always-on intelligent function. We employed a fine-grained clock and power-gating to take advantage of the input signal sparsity, low leakage SRAM and a fixed priority arbiter to achieve a very low standby power of 75 nW. We trained the SNN for multiple always-on functions, notably multi- and single-keyword spotting benchmarks, achieving competitive accuracies. The average power consumption of the SNN chip scales with input activity rate and lies somewhere between 75 nW and 220 nW for the KWS benchmarks. Table 2.1 summarizes the comparison of our work with other recent KWS accelerators while Table 2.2 summarizes the comparison of our design with other SNN hardware works. Our design achieves 2.3X - 7X power savings compared to state-of-the-art KWS hardware and over 1000X power savings compared to SNN hardware at competitive accuracies. It contributes to a growing body of literature that supports SNNs as an attractive low-power alternative to deep learning based hardware architectures.
### Table 2.1: Comparisons with recent KWS hardware

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>Shan 2020</th>
<th>Guo 2019</th>
<th>Giraldo 2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology (nm)</td>
<td>65</td>
<td>28</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>Algorithm</td>
<td>SNN</td>
<td>DSCNN</td>
<td>RNN</td>
<td>LSTM</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>1.99</td>
<td>0.23</td>
<td>6.2</td>
<td>1.035</td>
</tr>
<tr>
<td>VDD (V)</td>
<td>0.52-1</td>
<td>0.41</td>
<td>0.9-1.1</td>
<td>0.575</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td><a href="mailto:70KHz@0.52V">70KHz@0.52V</a></td>
<td>40KHz</td>
<td>75MHz</td>
<td>250KHz</td>
</tr>
<tr>
<td>Benchmark 1</td>
<td>GSCD (4 Keywords)</td>
<td>GSCD (2 Keywords)</td>
<td>GSCD (10 Keywords)</td>
<td>TIMIT (4 Keywords)</td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>91.8</td>
<td>94.6</td>
<td>90.2</td>
<td>92.0</td>
</tr>
<tr>
<td>Benchmark 2</td>
<td>HeySnips (1 Keyword)</td>
<td>GSCD (1 Keyword)</td>
<td>HeySnips (1 Keyword)</td>
<td>N/A</td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>95.8</td>
<td>98.0</td>
<td>91.9</td>
<td>N/A</td>
</tr>
<tr>
<td>Power</td>
<td>75-220 nW*</td>
<td>510 nW**</td>
<td>134 µW</td>
<td>5 µW</td>
</tr>
</tbody>
</table>

* Power consumption scales with input rate; ** Feature extraction circuits included

### Table 2.2: Comparisons with recent SNN hardware

<table>
<thead>
<tr>
<th></th>
<th>This Work</th>
<th>Park 2019</th>
<th>Chen 2018</th>
<th>TrueNorth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology (nm)</td>
<td>65</td>
<td>65</td>
<td>10</td>
<td>28</td>
</tr>
<tr>
<td>Neuron Count</td>
<td>650</td>
<td>410*</td>
<td>4096</td>
<td>1M</td>
</tr>
<tr>
<td>Synapse Count</td>
<td>67K</td>
<td>N/A</td>
<td>1M</td>
<td>256M</td>
</tr>
<tr>
<td>Area (mm²)</td>
<td>1.99</td>
<td>10.08</td>
<td>1.72</td>
<td>430</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td><a href="mailto:70KHz@0.52V">70KHz@0.52V</a></td>
<td>20MHz</td>
<td><a href="mailto:105MHz@0.5V">105MHz@0.5V</a></td>
<td>N/A</td>
</tr>
<tr>
<td>MNIST Classification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>305nW</td>
<td>23.6mW</td>
<td>9.42mW**</td>
<td>63mW</td>
</tr>
<tr>
<td>Accuracy (%)</td>
<td>97.6</td>
<td>97.8</td>
<td>97.9</td>
<td>97.6***</td>
</tr>
<tr>
<td>Throughput (inf/s)</td>
<td>2</td>
<td>100K</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Energy per inference (nJ)</td>
<td>152.5</td>
<td>236</td>
<td>1700</td>
<td>N/A</td>
</tr>
<tr>
<td>Energy per SOP (pJ)</td>
<td>1.5</td>
<td>N/A</td>
<td>3.8</td>
<td>26</td>
</tr>
</tbody>
</table>

* Input layer not included; ** Estimated from neuron’s power dissipation; *** Estimated from Hsin-Pai Cheng et al, IEEE DATE 2017 [32]
Chapter 3: FPGA-based Acceleration of Binary Neural Network Training with Minimized Off-Chip Memory Access

Training a neural network requires more computation and data movement compared to inference. Acceleration of training is, therefore, a challenge and is typically done in a datacenter on a GPU. FPGA tends to have smaller standby power consumption compared to the GPU and would be a platform of choice if it consumes lower energy even when employed for training neural networks. In this chapter, we examine the feasibility of FPGA as a platform for training a convolutional binary-weight neural network using an off-the-shelf FPGA board and GPU. For the case where all data except the input images can be stored on an FPGA chip, we design an accelerator for training CNNs to classify the CIFAR-10 dataset. We adopt batch renormalization which reduces the size of temporary intermediate data in training, allowing for the training of a larger network. Using the accelerator, we study the impact of network size on the performance and energy of FPGA and compare it to the GPU. The accelerator mapped in the Arria 10 FPGA chip obtains up to 9.33X higher energy efficiency compared to the Nvidia Geforce GTX 1080 Ti GPU for the same number of training iterations. The performance of the GPU improves relative to the performance of the FPGA as the size of the network increases resulting in a decrease in energy savings.

3.1 Introduction

Deep neural networks have achieved the state of the art performance in many tasks using large networks and large amounts of data. It results in very inefficient use of memory and compute resources. This has caught researchers’ attention and therefore efforts are being made to make neural networks efficient, more so for inference. A few algorithm-level works were proposed such as pruning neural networks [11], neural networks with limited precision [33], and binarized neural
networks [9].

GPUs are the typical choice to experiment with new algorithms because of the availability of a range of software libraries, which allow application developers to save design time. GPUs, however, are optimized for floating-point operations and may not be as efficient as hardware that is tailor-made for these algorithms using fixed-point operations. Since the algorithms are relatively new and not mature, it is reasonable to accelerate them on an FPGA instead of a custom chip.

The acceleration of inference on FPGAs especially CNNs has received a lot of attention. This is probably because inference is considered as the common case and it has a smaller memory requirement compared to training. But, a recent paper [34] suggests that even training of neural networks is done fairly frequently. Therefore, addressing the challenge of quick and energy-efficient training is an important matter. The acceleration of FPGA-based training is largely untouched. The works that we are aware of [33],[35] discussed this as an aside.

In this chapter, we try to answer if training of neural network can be energy efficient on an FPGA. We approach this problem by minimizing the number of off-chip accesses. This is effective as DRAM access is very costly in terms of energy and speed [36]. We choose a smaller neural network and use batch renormalization to accommodate all the weights and intermediate on the chip. This assumption can be relaxed if we find that FPGAs are indeed beneficial. Our contributions through this work are the following:

- Analysis of the data footprints in training CNNs and the recent trends of such data footprints
- Accelerator architecture for training a binary-weight, fixed-point activation CNN using shift based batch renormalization.
- Study of the impact of network size on energy consumption and performance. Higher energy efficiency (up to 9.33X) for smaller networks on Arria 10 FPGA compared to Geforce GTX 1080 Ti GPU for the same number of training epochs.

The remaining chapter is organized as follows: Sec. 3.2 discusses the impact of recent trends in CNNs on data footprint, Sec. 3.3 briefly describes the algorithms, Sec. 3.4 elaborates on the
3.2 Data Footprints in CNN Training

A recent trend in the design of CNNs is to have several convolutional layers and very few fully connected layers. For example, relatively old AlexNet [37] and Overfeat-Fast [38] have 5-6 convolutional layers and 3 fully connected layers, whereas more recent GoogLeNet [39] and Resnet-18 [40] have 11-17 convolutional layers with just one fully connected layer. In such deep networks, we need normalization techniques like batch layer normalization to address issues like exploding and vanishing gradients and weight initialization. But batch normalization requires a decent batch size to obtain good statistics.

The two main kinds of data in CNN training are the intermediate activation & gradient data (called intermediate data hereafter) and the weight data. Intermediate data increases with batch size but weight data does not. Fig. 3.1(a) shows the data size for different layers of a CNN for a batch size of 16, where CX, FX, and MX respectively represent a convolutional layer, fully-connected
layer and max-pooling layer. From Figs. 3.1(a), it is clear that the intermediate data dominates the overall data footprint. Also, most of the weights, in this case, are from the fully connected layers. The dominance of intermediate data further increases as one reduces the number and size of fully-connected layers and increases the batch size. This trend can indeed be observed in CNNs [37][38][39][40][41]. It is depicted in Fig. 3.1(b). It can be seen that the ratio of intermediate data to weight data during training has increased over the years. This can be critical in the design of accelerators for CNNs, as the kind of data that dominates the entire training algorithm can determine the architecture. Algorithms that target reduction in intermediate data become more important than the ones that reduce weight data.

3.3 Algorithm

3.3.1 Convolutional Layers during Training

![Figure 3.2: Convolution-layer operation](image)

The convolutional layer in a neural network realizes a set of filters and its operation is depicted in Fig. 3.2. Each output feature map has associated with it, Z kernels of size KxK where Z is the
number of input feature maps. There are MxZ kernels for the convolutional layer where M is the number of output feature maps. Algorithm 2.1 and 2.2 give the pseudo-code for convolution in forward propagation and backpropagation respectively. ‘I’ in the pseudo-code refers to the zero-padded input feature map, ‘$w^b$’ to the weights after they are binarized and ‘O’ to the output feature map. The variables ‘M’, ‘Z’, ‘R’, ‘C’ and ‘K’ are as shown in Fig. 3.2. The backpropagation of gradient through the convolutional layer again involves convolution. It is noteworthy that the weights in the case of backpropagation are transposed. Also, $\frac{\partial L}{\partial O}$ is zero-padded for convolution in backpropagation where $L$ is the loss. Weight update involves the multiplication of multi-bit terms.

**Algorithm 2.1: Convolutional layer during forward propagation.** Involves multiplications of binary weights with activations followed by accumulations

**Require:** Binarized weights $w^b$, zero-padded input $I$ with $z$ feature maps

**Ensure:** Output $O$ with $m$ feature maps

```
for(m=0; m<M; m=m+1) Loop6 — Outer most loop
  for (z=0; z<Z; z=z+1) Loop5
    for (r=0; r<R; r=r+1) Loop4 — Tile size of 8
      for (c=0; c<C; c=c+1) Loop3 — Tile size of 2
        for (y=0; y<K; y=y+1) Loop2 — Unrolled Fully
          for (x=0; x<K; x=x+1) Loop1 — Unrolled Fully
            $O[m][r][c] \leftarrow O[m][r][c] + w^b[m][y][x] \times I[z][r+y][c+x]$ 
```

**Algorithm 2.2: Convolutional layer during backpropagation.** Compute gradients with respect to input activations and new weights.

**Require:** Old weights updates $\Delta w_{old}$, binarized weights $w^b$, zero-padded activation $I$, zero-padded gradients $\frac{\partial L}{\partial O}$, $\text{Log2 of learning rate } l$

**Ensure:** Weight updates $\Delta w_{new}$, gradients with respect to input activation $\frac{\partial L}{\partial I}$

```
for(m=0; m<M; m=m+1) Loop6 — Outer most loop
  for(z=0; z<Z; z=z+1) Loop5
```

40
for(r=0; r<R; r=r+1) Loop4 — Tile size of 8
for(c=0; c<C; c=c+1) Loop3 — Tile size of 2
for(y=0; y<K; y=y+1) Loop2 — Unrolled Fully
for(x=0; x<K; x=x+1) Loop1 — Unrolled Fully

\[ \frac{\partial L}{\partial f}[m][r][c] = w^b[z][m][2 - y][2 - x] \times \frac{\partial L}{\partial f}[z][r][c] \times I[m][r + y][c + x] \]

\[ \frac{\partial L}{\partial f}[z][m][y][x] = \frac{\partial L}{\partial f}[z][r][c] \times I[m][r + y][c + x] \]

\[ \Delta w_{new}[z][m][y][x] \leftarrow \Delta w_{old}[z][m][y][x] - \frac{\partial L}{\partial w}[z][m][y][x] \gg l \]

3.3.2 Batch Renormalization Layer during training

The batch normalization layer normalizes the distribution of the feature map before it enters the subsequent layer as input [42]. It acts as a regularizer, increases the accuracy and makes the network robust to hyperparameter variations. But, it needs a decent batch size to get proper mini-batch statistics, i.e., mean and standard deviation. The size of the intermediate data generated restricts the use of large batch sizes when all of it is stored on-chip. On the other hand, the batch renormalization layer is proposed to replace the batch normalization layer [43]. It uses clipped-moving averages of mean and standard deviation to normalize, instead of mini-batch mean and variance. The advantage is that it works even for very small batch sizes (~4 to 7), and therefore substantially reduces the size of intermediate data.

We want to combine binary neural network training and batch renormalization. To do so, we modify shift-based batch normalization given by Courbariaux et al. [9] and use it for batch renormalization. Algorithm 2.3 and 2.4 give the algorithm for batch renormalization (BRN) in forward propagation and backpropagation respectively. The values \( x_i \) denote the values of a particular feature across a batch of inputs. \( \hat{x}_i \) denotes the normalized values of \( x_i \). \( \mu \) and \( \sigma \) denote the moving averages of mean and standard deviation, \( \mu_b \) and \( \sigma_b \) denote the mean and standard deviation of the mini-batch, while \( \beta \) and \( \gamma \) denote the shift and scale parameters used to get the output of batch renormalization layer denoted by \( y_i \). \( \alpha \) is the update factor for the statistics. \( AP2 \) stands for approximate power of 2.
In forward propagation, as in Algorithm 2.3, we first find $\mu_b$ and $\sigma_b$ of the minibatch data (steps 1-3). In the calculation of $\sigma_b$, the centered data is shifted by $AP2$ to avoid multiplication. $\mu_b$, $\sigma_b$ are used in the calculation of values $r$ and $d$ which are the standard deviation adjust factor and the mean adjust factor respectively. They together modify the normalization equation to use the clipped values of moving averages $\mu$ and $\sigma$ instead of $\mu_b$, $\sigma_b$ (step 4). The normalized values are then scaled and shifted to get the final output of the layer (steps 5-6). The values of moving averages are updated and the hyperparameter $\alpha$ determines the size of the update (step 7).

Algorithm 2.4 depicts the backpropagation process. It treats the values $r$ and $d$ as constants. The outputs of backpropagation are the updates to the values of $\beta$, $\gamma$, and $\frac{\partial L}{\partial \alpha}$, i.e., the gradient with respect to the input of the renormalization layer. We recalculate the normalized values ($\hat{x}$) from the $y$ values (step 1) as we store only the $y$ values to reduce memory usage. The calculation of update to $\gamma$ (step 3) involves the sum of products of multi-bit values while the update to $\beta$ (step 2) involves only the summation of the gradient values. The calculation of $\frac{\partial L}{\partial \alpha_\gamma}$ is complicated. It is broken down into parts for calculating the final value. It requires the computation of intermediate result $\frac{\partial L}{\partial \hat{x}_\gamma}$ (step 4) and differs from batch normalization because of the introduction of parameters $r$ and $d$. The equation for $\frac{\partial L}{\partial \hat{x}_\gamma}$ in Algorithm 2.4 uses $AP2$ and shift operation for multiplication or division with $r$, $\sigma_b$, and $m$. It has been omitted for the sake of clarity.

**Algorithm 2.3:** Batch renormalization applied to activation $x$ over mini-batch during forward propagation. The approximate power of 2 is $AP2(x) = sign(x)2^{floor(log_2(x))}$ and $\Theta$ represents either right or left shift.

**Require:** Values of $x$ over a minibatch $x_1, x_2, \ldots, x_m, \mu, \sigma, \beta, \gamma$

**Ensure:** Batch renormalized values $y$, updated moving averages $\mu, \sigma$ and parameters $r, d$

1. Mini-batch mean:
   \[
   \mu_b \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i
   \]
2. Centered Input:
   \[
   C(x_i) \leftarrow (x_i - \mu_b)
   \]
3. Approximate Variance:
   \[
   \sigma_b^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} C(x_i) \Theta AP2(C(x_i))
   \]
4. Adjust factors $r$ and $d$

$$r \leftarrow (\text{clip}_{r_{\text{max}}, r_{\text{max}}} (\sigma_b >> \text{AP2} (\sigma)))$$

$$d \leftarrow (\text{clip}_{-d_{\text{max}}, d_{\text{max}}} (\mu_b - \mu) >> \text{AP2} (\sigma))$$

5. Normalize using adjust factors, which make sure clipped $\mu$ and $\sigma$ are used:

$$\hat{x}_i \leftarrow C(x_i) \Theta(\text{AP2}(r) - \text{AP2}(\sigma_b)) + d$$

6. Scale and shift:

$$y_i \leftarrow \hat{x}_i \Theta \text{AP2}(\gamma) + \beta$$

7. Update moving averages:

$$\mu \leftarrow \mu + (\mu_b - \mu) \Theta \text{AP2}(\alpha)$$

$$\sigma \leftarrow \sigma + (\sigma_b - \sigma) \Theta \text{AP2}(\alpha)$$

---

**Algorithm 2.4:** Backpropagation through batch renormalization layer.

**Require:** Values of $x$ over a minibatch $x_1, x_2, \ldots, x_m, \mu, \sigma, \beta, \gamma$

**Ensure:** Batch renormalized values $y, \mu, \sigma$ and parameters $r, d$

1. Recalculate normalized values:

$$\hat{x}_i \leftarrow (y - \beta) \Theta \text{AP2}(\gamma)$$

2. Gradient with respect to $\beta$:

$$\frac{\partial L}{\partial \beta} \leftarrow \sum_i^m \frac{\partial L}{\partial y_i}$$

3. Gradient with respect to $\gamma$:

$$\frac{\partial L}{\partial \gamma} \leftarrow \sum_i^m \frac{\partial L}{\partial y_i} \times \hat{x}_i$$

4. Gradient with respect to $\hat{x}$:

$$\frac{\partial L}{\partial \hat{x}_i} \leftarrow \frac{\partial L}{\partial y_i} \Theta \text{AP2}(\gamma)$$

5. Gradient with respect to input activations:

$$\frac{\partial L}{\partial x_i} \leftarrow \left[ \frac{r}{\sigma_b} \left( \frac{\partial L}{\partial \hat{x}_i} - \frac{1}{m} \sum_j^m \frac{\partial L}{\partial \hat{x}_j} \right) - \frac{\hat{x}_i - d}{m \sigma_b} \sum_j^m \frac{\partial L}{\partial \hat{x}_j} (\hat{x}_j - d) \right]$$

In the training algorithm, we do not binarize the activations because with renormalization (or normalization) employed, binarizing the activations gives benefit only in the arithmetic complexity but not in the data footprint. Further, binarizing activation tends to reduce the accuracy.
Batch renormalization is typically employed if the batch size needs to be small. It is beneficial to decouple the backpropagation and the weight update step and perform the weight update over multiple batches [44]. It can help offset the hit to accuracy because of reduced batch size. So, the changes to weights need to be stored and accumulated until the weight update step.

3.4 Processor Architecture

3.4.1 System Architecture

We design a training accelerator for CNNs to classify the CIFAR-10 dataset. In the design, we focus on maximizing the speed and energy-efficiency benefits of having most of the data stored on-chip. Toward this goal, we design the hardware to support batch renormalization to reduce the required batch size and hence the intermediate data. The network and the batch size need to be small enough to fit the weights and intermediate data in the on-chip memory of an FPGA. Only the inputs and the labels are fetched from DRAM. The design is modular and makes batch size, the number of layers and the number of features in convolution, FC layer easily configurable. Also, we exploit the finite lifetime of gradient values of a layer. The gradient values are not needed after they are used and hence are overwritten.

Fig. 3.3 shows the top-level block diagram of the system consisting of the accelerator and the interface. The interface is contained in the red rectangle and the main accelerator in the blue rectangle in the figure. The interface logic consists of Altera’s IPs (NIOS processor, SGDMA, DESC-Table, and DRAM Controller) and custom-made components (DATA FSM and SGDMA Control). The accelerator contains compute lanes for each input image of the mini-batch. Each lane contains computing logic for convolution (Conv), Maxpool layer (MP), fully connected layer (FC), and loss calculation (Loss). The intermediate data would be local to each compute lane (MEMLocal). The renormalization logic makes use of data from all the lanes. Multipliers (Mults) are shared by renormalization logic (BRNC and BRNF) and by the logic in the compute lanes. MEMLocal mainly consists of activation and gradient memory. Global parameters like weights (MEMKer and MEMFC) and FSM are shared across all the compute lanes.
The high-level operation sequence of the accelerator is as follows. The NIOS processor loads the configuration data and gives a start signal. The accelerator initializes the weights and fetches the inputs and labels from DRAM using \textit{SGDMA} controller. The inputs are moved to \textit{MEM} of the Compute Lanes, which start forward propagation. When the compute lane starts backpropagation, new inputs are loaded to \textit{MEM}_{Local} in a ping pong fashion to hide the input loading latency.

3.4.2 Conv Microarchitecture Design

Setting unrolling and tiling factors is critical in the design of convolutional layers. As shown in Algorithm 2.1 and 2.2, we fully unroll the 3x3 kernel loops (loops 1 and 2) for both forward and backward propagation. To do so, the processor has 9 banks for storing the kernels. This is shown in Fig. 3.4, which depicts the microarchitecture for the \textit{Conv} block. It employs tiling for loops 3 and 4. The tile size for loop 3 is 2 and that for loop 4 is 8, i.e., 8 rows and 2 columns of pixels are fetched simultaneously from the input feature map. To do so, we design the activation, gradient, and temporary memories each to have 8 banks. Each bank has a port width that is twice the bit
width of the pixel. This gives rise to a tile size of two for loop 3. The loops 5 and 6, which are loops respectively over the number of input and output feature maps, were not unrolled because the number of feature maps is generally needed to be flexible. It also avoids low hardware utilization if the number of feature maps is not a multiple of the unroll factor.

Each iteration of the outermost loop produces an output feature map. Each output feature map is computed by accumulating the results of convolution of kernels with the set of all input feature maps. We store the partial results obtained in the iterations of the inner loops in Temp Banks and fetch them for accumulation when the corresponding pixels are being computed. Temp Banks were needed as the two ports of the activation and gradient banks (Act/Grad Banks) were already being used to read the input feature maps and store the results. Another port was needed to fetch the partial results of the computation. The fetched pixels from the Temp Banks and Act/Grad Banks go to the FIFO through the rearrange block (Rearr In). Zero padding the input feature map on the sides is taken care of by filling the FIFO with zeros at the beginning of a row. The Prepare Input block zero pads the input on the top and the bottom in addition to providing the input to each MAC in the convolution path. MAC in the convolution path computes the output of 3x3 inner product between the kernel and the input feature map. There are 16 MACs for computing the output feature maps. The first 14 MACs compute the full 3x3 inner product for 14 output pixels. The last 2 MACs require data from the next row of the input feature map and hence they compute the inner product partially. The last row needs to be fetched again for computation in the next row of pixels. This is where the rearrange input blocks (Rearr In) are needed. They present the values fetched from the 8th memory bank to the first 2 MACs by rearranging the rows. The Rearr Out block will undo the rearrange and store the values in the correct banks. The partial results needed for accumulation in the current computation are fetched along with the inputs. The completed results are stored in Act/Grad Banks.

To perform backpropagation the processor transposes the weights through proper addressing. The ΔW compute operation makes use of multi-bit multipliers (Mults). The calculated ΔW values are added to ΔW values from previous batches and updated in the memory. For the last batch per
Figure 3.4: The microarchitecture of the Conv block in a Compute Lane. It performs most of the computation for training except batch renormalization weight update, the old weight values are added to the accumulated ΔW values to obtain the new weight values which are then written to the Kernel Banks. During backpropagation, Mults are time shared with batch renormalization layers and the fully connected layers.

3.4.3 Batch renormalization after convolutional layer (BRN\(_C\))

We designed a batch renormalization layer (BRN\(_C\)) that follows a convolutional layer and the one that follows a fully connected layer (BRN\(_F\)) differently. This is because we have one \(\mu_b\) and \(\sigma_b\) for one feature map in the case of convolutional layers which can be available only after fetching and processing the entire feature map. But, in the case of fully connected layers, the statistics for each neuron are independent.

For renormalization after a convolutional layer during forward propagation, the processor iterates over each input feature map thrice. This reduces the register count at the cost of higher latency. In the first pass, the processor calculates \(\mu_b\) for the feature map, by accumulating the activation values using pipelined adder tree and finally shifting. In the second pass, it calculates \(\sigma_b\),
Figure 3.5: Graph showing the dependencies between the variables in the backpropagation through the renormalization layer that follows a convolutional layer. Red boxes contain the data computed during the first pass. Green boxes contain data computed during the second pass.

$r$ and $d$ and updates the moving averages $\mu$ and $\sigma$. The mean centered data which is computed for variance replaces the original data in the Act Banks, as further only centered data is needed. The multiplication and division in the calculation of $r$ and $d$ and update to moving average statistics are done using shift operations with $AP2$. In the third pass, the feature map is normalized using the newly computed $r$, $d$ parameters, scaled (using $AP2(\gamma)$) and shifted (using $\beta$) to give the output of the layer.

In backpropagation, the processor iterates twice over the gradient feature map. $\hat{x}$ is needed for computation of $\frac{\partial L}{\partial \hat{y}}$ and $\frac{\partial L}{\partial \hat{x}}$. It is computed from the stored $y$ values (see Algorithm 2.4). The computation of $\frac{\partial L}{\partial \beta}$, $\hat{x}$, $\frac{\partial L}{\partial \gamma}$, $\frac{\partial L}{\partial \hat{x}}$, $M$ and $N$ all happen in the first pass.

\[
M = \frac{1}{m} \sum_{j}^{m} \frac{\partial L}{\partial \hat{x}_j},
\]

\[
N = \frac{1}{m \sigma_B} \sum_{j}^{m} \frac{\partial L}{\partial \hat{x}_j} (\hat{x}_j - d)
\]

$M$ and $N$ are used in the second pass along with $\hat{x}$ and $\frac{\partial L}{\partial \hat{x}}$, to give $\frac{\partial L}{\partial x}$. Fig. 3.5 shows the dependencies and the colour shows the which variables are computed in the first the second pass.
3.4.4 Batch renormalization after fully connected layer (BRN$_F$)

The microarchitecture for renormalization after the fully connected layer for forward propagation is depicted in fig. 3.6. The result for each feature is obtained in a single iteration because independent statistics for each neuron allow us to reduce the latency while keeping the register use small. The 8 Act Banks give 16 features on each fetch. Batch renormalization is done for these 16 features at once.

The sum of features in a batch is computed and then the average is taken using pipelined adder tree and shifter which are present in Center Data block. The input feature is delayed until $\mu_b$ is available and then subtracted. The centered data is then used to compute the variance by shifting it by its AP2 value to give an approximate value of $(x - \mu_b)^2$. These values are averaged to give the variance. The AP2 of the standard deviation ($\sigma_b$) is then computed and used to get the values of $r$ and $d$ along with updated values of moving averages $\mu$ and $\sigma$. These are then used in normalization. The normalized values are scaled using AP2($\gamma$) and then added to $\beta$ to give the final values ($\gamma$). The final values are stored in the same location as the input data. The AP2($\sigma_b$), AP2($r$) and $d$ are stored in the memory and used in backpropagation.
Backpropagation through the batch renormalization is done in three parts. This can be seen in fig. 3.7 based on the colour of the boxes. In the first part, $\beta_{new}$ and $\hat{x}$ are calculated from $\frac{\partial L}{\partial y}$, learning rate, $\beta$, $\gamma$ and $y$ values. This is done because $\hat{x}$ is needed for other calculations while $\beta$ is no longer needed. The calculation of new $\beta_{new}$ and $\hat{x}$ is done concurrently. The hardware is designed so that both the computations need $\beta$ in the same cycle and there is no need to delay it. In the second part, $\gamma_{new}$ and $\frac{\partial L}{\partial x}$ are calculated. $\frac{\partial L}{\partial x}$ is needed in the third part and hence computed. $\gamma_{new}$ computation requires the use of Mults because it involves multiplying $\frac{\partial L}{\partial y}$ with $\hat{x}$ both of which are multi-bit values. The calculation of $\frac{\partial L}{\partial x}$ requires shifting $\frac{\partial L}{\partial y}$ with the $AP2(\gamma)$. The third part of backpropagation through the batch renormalization layer involves the computation of $\frac{\partial L}{\partial x}$, i.e., the derivative of the loss with respect to the input of the batch renormalization layer. This involves the implementation of the last step in Algorithm 2.4. This equation is computed in two phases to reduce the use of registers as the two phases have very different latency. The portions computed in the first and second phase are given in equations (3.3) and (3.4):

$$P1 = [AP2(r) - AP2(\sigma_b)] \Theta \left( \frac{\partial L}{\partial \hat{x}_i} - \frac{1}{m} \sum_{j}^{m} \frac{\partial L}{\partial \hat{x}_j} \right) \quad (3.3)$$
Datapath for P2 The datapath for P1 and P2 is in Fig. 3.8. Computation of P2 has higher latency because it involves two products for each output value while P1 involves no products. The products make use of Mults which take 4 cycles to give the output. Phase 1 goes first and its results are stored in the Temp Banks. Phase 1 ends when either the Temp Banks are full or when all the phase 1 results are computed for all the values in that layer. The result of phase 2 (P2) is subtracted from the result of phase 1 (P1) to give the final result.

3.4.5 Max-Pooling Layers (MP) Design

The accelerator uses 2x2 maxpooling. The layer computation is implemented using a PE which computes the maximum in 2x2 pixels and gives a 4-bit vector indicating the position of the maximum. The maximum value is stored in Act Banks and the 4bit position vector is stored in a separate memory called MP memory which is a part of MEM\textsubscript{Local}. The position vector is needed for backpropagation through the maxpooling layer. Each 8x2 pixels fetch gives 4x1 output pixels which are put together with 3 other such results and stored in Act Banks. The input pixels are fetched every cycle while the output is written every 4 cycles. In backpropagation, gradient values are read
Figure 3.9: Microarchitecture of the FC in the Compute Lane

every 4 cycles while the output values are written every cycle.

3.4.6 Fully-Connecting Layer (FC) and Loss Layer Design

Fig. 3.9 shows the microarchitecture of the FC block. The weights in the fully connected layers are stored in 8x8 Weight Banks. Each bank has a port size that is twice the size of the weight. 8x16 weights are made available in one fetch. The PE for computation is a MAC, which produces the outputs corresponding to a neuron. The weights are binarized and multiplied with the 16 pixels from the Act Banks to compute 8 values which are accumulated with the results of the previous computation pertaining to those neurons. The results are stored in the Act Banks and the accumulators are cleared. In the case of the output layer, the results are stored in the output layer related registers. We implement SoftMax at the output layer with a cross-entropy loss function. The maximum of the output layer’s activation values is subtracted from them to bring down the dynamic range of the exponential.

In backpropagation again 8x16 weights are fetched. The fetched weight matrix is multiplied with 8 gradient values to produce the values corresponding to the gradient of 16 neurons. The
transpose operation is implemented by proper addressing of the weights. Calculation of weight update is carried out simultaneously. For the $\Delta W$ compute step, activation and gradient values are fetched first and given to the hard multipliers ($\text{Mults}$). The $\Delta W$ values from each input are averaged and added to the previous $\Delta W$ values to produce new $\Delta W$. The activation and gradient are fetched in advance because of the latency of $\text{Mults}$, in the computation of $\Delta W$ values. The gradient values are delayed using FIFO until the previous $\Delta W$ values and binary weights are available for backpropagation through the fully connected layers.

### 3.5 Experiments

#### 3.5.1 Setup

We map the accelerator onto Arria 10 GX1150 chip on DE5a-Net board. The time interval needed for computation is measured using a timer peripheral. We access the power measurement circuitry on DE5a-Net through the NIOS processor using I2C protocol. The GPU board’s power consumption is obtained using nvidia-smi command, which is run multiple times and the values are averaged. We measure the power consumption when the FPGA and GPU boards are both active and inactive. In the case of the FPGA, we measure the idle power of the FPGA by mapping only the NIOS processor. The idle power in the case of FPGA and GPU is around 16.5 W and 37-44 W respectively. We then use the difference between the idle and active power values because this difference measurement reduces the impact of stray components on the boards, thereby giving us a fairer comparison between GPU and FPGA. The power difference along with the time taken to finish the computation of an epoch gives the energy consumption values. For GPU, we measure the accuracy, time, and power while we execute Tensorflow codes that use CUDA and CuDNN.

We designed the accelerator at the RTL level. We used 16-bit fixed-point arithmetic for all data types except for ROM used for loss calculation. The number of compute lanes is configured to make proper utilization of the resources available on an FPGA. The baseline network structure ($N_0$) in our experiments is: Input – 2x(8C3-BRN) – MP2 – 2x(16C3-BRN) – MP2 – 2x(32C3-BRN) – MP2 – FC-BRN (64) – FC-BRN (64) –10, where 2x(mC3-BRN) denotes two convolutional layers.
Figure 3.10: The impact of batch size on accuracy in training the $1 \times N_0$ network with batch normalization and batch renormalization.

each with a kernel size of 3x3 and ‘m’ output feature maps followed by batch renormalization layer. MP2 denotes a max-pooling layer of kernel size 2x2 and FC-BRN (64) denotes a fully-connected layer of size 64 neurons followed by batch renormalization layer. This network has two convolutional layers between pooling layers (called stage), both of which have the same number of output feature maps. We consider a network that has $x$ convolutional layers in each stage as having size $x/2 N_0$. This is approximately correct if we consider the number of operations as the metric for network size. The fully-connected layers are small and we expect that the time spent on layers other than convolutional layers is minimal. We perform our experiments and study performance and energy consumption on networks of different sizes from $N_0$ to $3N_0$ and cover a large range. For example, $3N_0$ has 18 convolutional layers while $N_0$ has just 6 convolutional layers. The modularity in design allows us to generate different network by making small changes to the top FSM like the number of layers and memory offsets for each layer.
Table 3.1: Summary of resource usage in training CNNs of different sizes using BRN. ALM here means Adaptive Logic Module and MLAB means Memory Logic Array Block.

<table>
<thead>
<tr>
<th></th>
<th>$N_0$</th>
<th>2.5$N_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batch Size</strong></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td><strong>Frequency</strong></td>
<td>100MHz</td>
<td>100MHz</td>
</tr>
<tr>
<td><strong>ALM</strong></td>
<td>237K(56%)</td>
<td>239K(56%)</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>274429</td>
<td>274670</td>
</tr>
<tr>
<td><strong>MLAB</strong></td>
<td>84%</td>
<td>86%</td>
</tr>
<tr>
<td><strong>BRAM</strong></td>
<td>15Mb (28%)</td>
<td>25Mb (46%)</td>
</tr>
<tr>
<td><strong>DSP</strong></td>
<td>867(57%)</td>
<td>867(57%)</td>
</tr>
</tbody>
</table>

3.5.2 Impact of Algorithm on Resource Usage and Network Size

From [43], we can see that BN does not perform well for small batch sizes. We observed the impact of the algorithm employed for normalization in our network structure as well. Fig. 3.10 shows the accuracy of the networks as a function of batch size when they employ BN versus BRN. We see that the batch size needed to obtain close-to-peak accuracy with batch normalization is around 41 while a batch size of around 7 is sufficient to match when batch renormalization is employed. Smaller batch size frees up memory that can be used to support a larger network which would require more BRAM. So, the remaining memory can be used to instantiate larger networks. Therefore, the computing systems that use algorithms that help reduce the intermediate data, have a huge impact on the size of the network one can train, largely using on-chip memory.

Next, we map the different networks and find the maximum batch size that the FPGA and CAD tools can support while fitting the entire intermediate and weight data. Table 3.1 gives the resource consumption of different networks and the batch size used. The table shows that larger networks occupy more memory resources (i.e., BRAM) but do not require much more for the rest of the logic circuits, e.g., ALM usage increases by less than 1%. We clock all the designs at 100 MHz to avoid multiple logic synthesis and place&route. However, small networks and small batch sizes could enable the use of higher clock frequencies.
3.5.3 Comparison with GPU

We compare the performance of the FPGA accelerator with the GPU. As shown in Fig. 3.11(a), both GPU and FPGA training speeds increase with batch size. GPU speed saturates but in the case of FPGA, the speed increases till the maximum batch size. For small and similar batch sizes, the FPGA has more than one order speed-up but at the speed-optimal batch sizes, the speedup of the FPGA is about 140%.

In Fig. 3.11(b) we observed the time taken per epoch across different sizes of networks. For the FPGA, we use a batch size of 6, which achieves near-peak accuracy. For the GPU, we use a batch size of 180, which gives the best performance. In this experiment, the FPGA achieves a speedup of 140% over the GPU for the network N₀, roughly the same for 1.5N₀, and greater than 77% for the rest of the networks. Fig. 3.11(b) shows that the training time increases with network size at a greater rate for the FPGA than the GPU. This is because the thread management overhead of the GPU gets amortized when a larger network is trained. Training a larger network requires more convolution-layer computations and the GPU excels in those computations in comparison to our accelerator. This is perhaps due to the data dependencies in backpropagation which result in considerable resource idle time. For instance, inside the same compute lane, when the Conv block is working, MP, FC and Loss blocks are idle because they need to wait for the feature map. If we can increase the clock frequency of the FPGA accelerator, perhaps by using a high-speed FPGA chip or a better CAD tool, this could reduce the gap between the FPGA and the GPU for larger networks but it is unlikely to change the trend. This supported further by the fact that FPGA uses fixed-point computations while GPU uses floating-point computations so, the GPU training is expected to converge in fewer epochs than the FPGA.

We also measure the energy consumption of the FPGA and the GPU. First, we measure the energy consumption per epoch of the GPU and FPGA across different batch sizes. As shown in Fig. 3.12(a), the FPGA accelerator achieves 9.33X better energy consumption. We also measure the energy consumption across multiple network sizes. The batch size used for the FPGA accelerator is 6, the maximum that the CAD tool and the FPGA can support for the network 3•N₀. As shown
in Fig. 3.11(b), the FPGA consumes 5.77-9.33X less energy for training, despite not using clock gating. This energy savings reduce for larger network because the GPU’s performance improves relative to FPGA as the size of the network increases. If the trend continues then the energy improvement of the FPGA vanishes for a network of size 7-10N₀. The GPU is expected to consume lower power for a realistic network as they are much larger. For example, the size of Resnet18 is over 500N₀, based on the number of convolution operations.

3.6 Summary

Servers in a datacenter are always-on and can benefit from the use of accelerators that consume lower power both when they are active and idle. It is in this context that we try to examine FPGA as a potential option for replacing the GPU where it is used largely for training neural networks. Further, training of neural networks is being done frequently in some cases, necessitating the study of accelerator options that improve their efficiency. We try to address this by first designing an accelerator that minimizes off-chip memory accesses to save power by using small neural networks and batch renormalization. Batch renormalization brings down the required batch size from ~41
Figure 3.12: Energy consumption comparison. (a) The FPGA based accelerator consumes 9.33X less energy than the GPU in training 1•N\textsubscript{0} CNN. (b) The FPGA achieves 5.77-9.33X smaller energy consumption across CNNs of different sizes. Energy-optimal batch sizes are used for the GPU.

to \sim 7. It is necessary for the study without which the network that can be accommodated would be tiny and of no significance. The accelerator demonstrated in this chapter reduces energy consumed by a factor of up to 9.33X and obtains a speedup of up to 140\% compared to GPU which reduces as the network size increases. The possible reasons for this could be thread management overhead amortization for larger networks. The GPU is expected to perform better overall for larger networks due to the use of floating-point which reduces the number of epochs needed to complete training.
5G millimeter-wave (mmWave) communication systems enable exciting new applications by significantly reducing the latency and increasing the data rate. However, this comes at a large computational cost, which results in long latency and large energy consumption. In this chapter, we aim to address this challenge in the problem of channel estimation of such systems through a set of algorithm-hardware co-optimizations. First of all, we employ a model-based neural network to improve the rate of convergence. We also optimize the neural network to improve loss while using approximately the same number of operations. Furthermore, we reduce the computational complexity through the use of sparsity inherent in mmWave channels. The optimizations reduce the computational complexity of the neural network for channel estimation by over two orders of magnitude. Based on these innovations, we implemented a channel estimation subsystem on Zynq 7020 FPGA. The subsystem obtains an improvement in latency of up to $\sim 10X$ and an improvement in energy consumption of up to $\sim 300X$ over CPU and GPU based systems.

4.1 Introduction

5G wireless communication networks are expected to enable a significant reduction in communication latency and an increase in the bandwidth. The networks will enable many new applications such as virtual/augmented reality, autonomous vehicle/drone communication [45]. To fully enable 5G wireless communication networks, however, we need significant innovations both at the algorithm and hardware level. One of the key enablers is the usage of higher carrier frequencies to increase the available spectrum and thereby communication data rates. This incurs higher path loss but reduces the form factor of the antenna, facilitating the use of arrays with a large number
Figure 4.1: mmWave communication systems where a base station and user equipment inside the car use beamforming and multipath propagation for communication.

of antennas for beamforming. Fig. 4.1 shows an example application where beamforming is used for communication between a base station and user equipment located inside a moving car. Such multi-input multi-output (MIMO) communication systems need accurate channel state information (CSI) to take advantage of multipath propagation in highly scattering scenarios.

On the other hand, deep learning has recently been shown to improve the performance of conventional iterative optimization algorithms, such as the iterative shrinkage-thresholding algorithm (ISTA), which can be employed in channel estimation. A work in this area [46] shows that the deep learning-based approach can offer increased convergence speed and accuracy over standard implementations of such algorithms.

This kind of neural network is designed so that each layer executes a single iteration of the optimization algorithm. These neural networks are called “model-based” since they roughly implement an iterative optimization algorithm. For this reason, they are devoid of the “black-box” nature of standard neural networks while taking advantage of the extraordinary ability of neural networks to learn from large amounts of training data. Communication systems bear signals that are inherently man-made and the physical behavior of the systems can be captured by well-established models. So, it is feasible to generate arbitrarily large sets of training signals representative of what the de-
ployed device will encounter. These characteristics, therefore, allow such a model-based neural network to improve upon methods of solving an important class of optimization problems arising in wireless communication like channel estimation.

However, we estimate that a conventional model-based neural network would have very high computational complexity, e.g., 56 Giga operations per second (GOPS) of computation per sub-carrier frequency for a 3-ms coherence time (see the derivation in Sec. 4.2.2). This high computation complexity is due to the large number of antenna arrays at both the base station and the user equipment and the multiple frequency bands. Better algorithms and hardware architecture are necessary to meet the computational performance requirement at high algorithmic accuracy yet under power and hardware budget.

In this chapter, based on the model-based neural network, we aim to optimize algorithms and hardware architecture for the channel estimation subsystem in mmWave 5G wireless communication networks. We will show that model-based neural networks, algorithm-hardware co-optimizations, and accompanying FPGA hardware architecture can address the computational requirements under low energy and hardware budget. FPGAs seem to be the hardware platform of choice for the emerging 5G communications because it can adapt to the evolving standards while delivering on the performance. Specifically, we make the following contributions in this work:

- We optimized the architecture of the model-based neural network for channel estimation in 5G mmWave communication systems
- We leveraged the sparsity inherent in mmWave channels and successfully reduced the computational complexity by up to 131X
- We implemented the channel estimation subsystem on Zynq 7020 FPGA, demonstrating the improvement in latency by $\sim10X$ and in energy consumption by $\sim300X$ over CPU and GPU.

The remainder of the chapter is as follows. In Sec. 4.2, we will describe the channel estimation in 5G mmWave communication and the recent model-based neural network works. In Sec. 4.3, we will propose our algorithm-hardware co-optimizations on model-based neural networks. In Sec.
4.4, we will introduce the hardware implementation of the channel estimation subsystem on FPGA and its results in comparison to CPU and GPU based ones. Finally, we summarize in Sec. 4.5.

4.2 Background

4.2.1 Channel Estimation in 5G mmWave Communications

Channel Estimation in communication systems involves the characterization of the channel between a transmitting antenna and a receiving antenna at a regular time interval. Wireless medium behaves as a linear time-varying (LTV) system and is approximated as a linear time-invariant (LTI) system for a short interval, whose gain and phase at a certain frequency is obtained during channel estimation.

When there are multiple antennas at either the transmitter or the receiver a channel matrix is obtained where each entry corresponds to a pair of antennas, one from the transmitter (TX) and another from the receiver (RX). If TX has $N_{TX}$ antenna and RX has $N_{RX}$ antenna, the channel matrix at a certain frequency has the dimensions of $N_{RX} \times N_{TX}$. In this chapter, we assume $N_{TX}$ to be 64 and $N_{RX}$ to be 32 and that the antennas are arranged in a uniform linear array.

Conventionally, a column of the channel matrix can be estimated at once by transmitting a pilot signal only from the antenna corresponding to the column index, i.e., the channel between all RX antennas and a particular TX antenna is estimated. To estimate the entire matrix, we need $N_{TX}$ pilot signals for each frequency. As $N_{TX}$ increases, the latency of the entire channel estimation process linearly increases, and a large number of pilots leads to spectral inefficiency makes this approach non-ideal in the case of mmWave systems.

Another approach is to exploit the sparse scattering nature of mmWave channels and formulate it as a problem of sparse recovery [47], [34]. In this approach, the channel matrix ($H$) can be broken down into a product of three matrices as in:

$$H = R \times \Sigma \times T^H,$$  \hspace{1cm} (4.1)
where $R$ is the array response of the receiver and $T^H$ is the Hermitian transpose of the array response of the TX. $\Sigma$ is a diagonal matrix, defined as in:

$$\Sigma = \text{diag}([\Sigma_{11}, \ldots, \Sigma_{MM}]),$$

where $\Sigma_{ii} \in \mathbb{C}$ is the $i^{th}$ multipath gain and $M$ is the total number of paths. Each row of $R$ is a uniformly sampled complex sinusoid whose number of samples is $N_{RX}$ and frequency is given by the angle of arrival (AoA). Similarly, each row of $T$ is a uniformly sampled complex sinusoid whose number of samples is $N_{TX}$ and frequency is the angle of departure (AoD). For $M$ different paths, there will be $M$ corresponding AoAs, AoDs, and gains.

The relation between transmitted symbols and the received symbols is given by:

$$Y = Q \times H \times P,$$

where $P$ is a set of training beamforming vectors which sends ‘p’ beams in successive time slots [47] and $Q$ denotes the combining matrix. The $P$ and $Q$ matrices are designed based on [47]. The number of beams we used is 17 and the number of channels of data obtained after combining using $Q$ is chosen to be 17. The dimensions of the $P$ matrix, therefore, are $64 \times 17$ and that of $Q$ are $17 \times 32$.

Using (4.1) in (4.3), we get the full equation, formulated as below:

$$Y = Q \times (R \times \Sigma \times T^H) \times P.$$  

Fig. 4.2 shows an exemplary system where each of the matrices in (4.4) is noted as a component of the communication system. Three different paths from the TX to the RX to indicate sparse scattering in the case of mmWave signals. $P$ matrix is implemented by the precoder, the effect of $T$ comes from the array of transmitting antennas, $\Sigma$ from the gains of the three paths, $R$ from the receiving antennas, and $Q$ from the combiner in the receiver.
We can find an approximate $H$ by discretizing the AoA and AoD which are used in $R$ and $T$ in (4.4). $R$ with the dimensions $N_{RX} \times M$ is replaced by $R_D$ of dimensions $N_{RX} \times G$, where $G$ is the angular grid size. Similarly, $T$ is replaced with $T_D$ of dimensions $N_{TX} \times G$ and $\Sigma$ is replaced with $\Sigma_D$ of dimensions $G \times G$. The $n^{th}$ row of $R_D$ ($T_D$) is made of a complex sinusoid whose frequency corresponds to an AoA (AoD) of $\frac{2\pi n}{G}$. Element $\Sigma_{ij}$ of the matrix $\Sigma_D$ is the gain of the path whose AoA is $\frac{2\pi i}{G}$ and AoD is $\frac{2\pi j}{G}$. After the discretization, the relationship between $Y$ and $\Sigma_D$ can be expressed as:

$$Y = Q \times (R_D \times \Sigma_D \times T_D^H) \times P + n_q$$  \hspace{1cm} (4.5)$$

where $n_q$ is quantization noise due to the introduction of the angular grid.

To make (4.5) more suitable for sparse recovery formulation, we vectorize it, which gives us:

$$vec(Y) = (P^T \otimes Q) \times (T_D^* \otimes R_D) \times vec(\Sigma_D) + n_q,$$  \hspace{1cm} (4.6)$$

where $vec(\Sigma_D)$ is the only unknown. It is a sparse vector as there are only a few AoAs and AoDs.
Figure 4.3: Model-based neural networks based on ISTA. (a) NET1: $\eta_\theta$ non-linearity for all iterations ([48]) (b) NET2: $\eta_\theta$ non-linearity for all iterations ([46]) (c) NET3: $\eta_\theta$ non-linearity for all iterations (Variant of [48])(d) NET4 and NET4ss: Networks have a scaling factor $\gamma$ for each iteration. NET4 uses $\eta_\theta$ for all iterations while NET4ss uses $\eta_\theta^s$ for 1st iteration and $\eta_\theta$ for the rest (Proposed, Variant of [49]).

for which the path would have a non-zero gain. Finally, one can get an estimate of the channel matrix ($H_{est}$) after recovering $\Sigma_D$ using:

$$H_{est} = R_D \times \Sigma_D \times T_D^H,$$

where $R_D$ and $T_D$ are used instead of $R$ and $T$ in (4.1).

4.2.2 Model-based Neural Network

ISTA is an optimization algorithm used for sparse vector recovery which attempts to solve the following optimization problem:

$$\arg\min_x ||b - Ax||_2^2 + \lambda ||x||_1,$$
where $x$ is the sparse vector to be estimated, $b$ is the measured vector, $A$ is the measurement matrix, $\lambda$ a hyperparameter. The relationship between these parameters is given by:

$$b = Ax + n,$$

(4.9)

where $n$ is the noise in the measured vector. Each iteration in ISTA involves updating the estimate of $x$ as in:

$$\hat{x}^{k+1} = \eta_\theta(\hat{x}^k + \frac{1}{L}A^H(b - A\hat{x}^k)),$$

(4.10)

where $\hat{x}^k$ is the estimate of $x$ in the $k^{th}$ iteration and $L$ is a hyperparameter, $\eta_\theta$ is defined as in:

$$\eta_\theta(z_{r,i}) = sgn(z_{r,i}) \times \max(|z_{r,i}| - \theta, 0).$$

(4.11)

$\eta_\theta$ is applied element-wise on real and imaginary parts of vector $Z$ and induces sparsity. Equation (4.11) is applied independently for the real and imaginary parts of $Z$, each with its $\theta$. The values of $\theta$ and $L$ determine the rate of convergence and it is usually non-trivial to find their optimum values.

Fig. 4.3(a) shows the model-based neural network based on ISTA [48], which we call NET1 in this work. An iteration of NET1 with $W$ as $\frac{1}{L}A^H$ gives the same result as ISTA. [48] has shown that NET1 outperforms ISTA by a large extent based on on-grid data. The key difference is that $\frac{1}{L}A^H$ (i.e., $W$) and $\theta$ were learned from data instead of using the predetermined values. Similarly, NET2 in Fig. 4.3(b) was introduced in [46] whose structure can be obtained by rewriting (4.10) as:

$$\hat{x}^{k+1} = \eta_\theta(\frac{1}{L}A^H \times b + (I - \frac{1}{L}A^HA) \hat{x}^k),$$

(4.12)

where the matrix used to multiply with $\hat{x}$ is replaced with $W_2$.

ISTA based neural networks can be used to find the optimum $vec(\Sigma_D)$ in (4.6). $b$ in (4.9) would correspond to $vec(Y)$ in (4.6) and $x$ to $vec(\Sigma_D)$. The advantage with communication systems is that the model of the channel as an LTI system is well accepted and allows us to generate large
amounts of data and use it for training the neural network. Input \((Y)\) to the network is generated by using different values for \(\Sigma\) using (4.4) and adding noise. The true value for each input is given by (4.1) and the estimated value comes from (4.7).

We can calculate the computational complexity of channel estimation using NET1. The dimensions of \(W\) and \(A\) matrix are \(4096 \times 289\) and \(289 \times 4096\) for a grid size of 64. The number of real number multiplications or additions that need to be performed for 6 iterations of NET1 is about 170M. The computation of \(H_{ext}\) needs to finish within the coherence time of the channel which is the amount of time for which channel is expected to remain constant and this varies with application. Here, if we assume that the coherence time of the channel is 3ms then the rate of computation needed is around 56 GOPS. The amount of computation that is needed if one uses NET2 instead is 862M and the rate of computation needed is 287 GOPS. NET2 involves significantly more operations when compared to NET1 due to the \(W_2\) matrix \((4096 \times 4096)\) which is much larger than twice the size of the A matrix.

### 4.3 Algorithm-Hardware Co-Optimization

#### 4.3.1 Introduction

We first examined the effectiveness of NET1 and compared it with ISTA using on-grid communication data as done in [48]. The performance of the two techniques can be observed in Fig. 4.4 where the normalized L2 loss is shown as a function of the number of iterations. NET1 outperforms ISTA by over two orders of magnitude. The input data was generated using (4.4) with an SNR of 10dB, where there are three paths from TX to RX with randomly generated AoA, AoD, and path gain. Again, the AoA and AoD used for Fig. 4.4 fall on the angular grid.

In reality, the AoA and AoD of the communication channel between TX and RX are not guaranteed to be on the grid, therefore, we examined the impact of grid size on the loss for NET1 and NET2. We assumed three paths and randomly generated off-grid AoA, AoD, and path gain, and an SNR of 10dB for experiment data. We can observe from Fig. 4.5 that an angular grid size of at least 64 is needed for the TX and RX to obtain a decent loss. We thus use a grid size of 64 for
Figure 4.4: Comparison of the rate of convergence between ISTA and NET1 for on-grid communication data.

Figure 4.5: Impact of grid size on loss for NET1 and NET2.
channel estimation throughout the rest of the work.

We worked out in Sec. 4.2.2 that the number of operations needed for NET1 is $\sim 170M$ while it is $\sim 862M$ operations for NET2. This is, however, very inefficient since we could do much better through the use of sparsity in $\hat{x}$, $fw$, and $fb$. To largely reduce the number of operations, therefore, we mainly pursue three ideas. First, we will look at the modification needed for non-linear function so that there is sparsity in $\hat{x}$ throughout the computation, allowing us to reduce operations. Second, we will look at application-level assumptions to make use of sparsity in $fb$ and $fw$. Finally, we prune the weights of the network and also quantize the weights (4b) and the activations (12b) to reduce the latency and the footprint of the hardware.

4.3.2 Optimization of Network Structure

We optimized the network structures to obtain better loss while using roughly the same number of operations as NET1. We came up with three additional network structures, NET3 in Fig. 4.3(c) and NET4 and NET4ss in Fig. 4.3(d), all of which are variants of the networks discussed in [48], [49]. NET4 and NET4ss have an extra scaling factor $\gamma$ for each iteration which multiplies with the result of the upper feedback branch and $W_1 \times b$ to give $fb$ and $fw$. NET4 and NET4ss differ only in the non-linear function used at the end of the first iteration. NET4ss uses $\eta^\theta_{ss}$ at the end of the first iteration and $\eta^\theta$ for all other iterations, while NET4 uses $\eta^\theta$ for all iterations. $\eta^\theta_{ss}$ is defined as:

$$\eta^ss_{\theta,m}(z_{r,i})=
\begin{cases} 
\text{sgn}(z_{r,i}) \times \max(|z_{r,i}| - \theta, 0), & |z_{r,i}| < m \\
z_{r,i}, & |z_{r,i}| \geq m
\end{cases}
$$

where $z$ is an element of vector $Z$ which is the input to the non-linear function, $m$ is a hyperparameter whose value is chosen to be equal to the magnitude of the element of $Z$ at the 95th percentile [48], and ss stands for support selection. Real and imaginary parts of $Z$ are treated independently. This means that the largest values of $Z$ are kept untouched in the first iteration.

We compared the five networks in Fig. 4.6, where the loss for these models at the end of six iterations is plotted against the number of trainable parameters. NET2 attains the best loss of
around -13 dB followed by NET4, NET4ss, NET3, and NET1. It can be noticed that the better loss of NET2 comes at the cost of many more parameters. For this chapter, we have chosen NET4 and NET4ss for further exploration as they strike a good balance between the number of parameters and the loss obtained.

4.3.3 Optimization using Sparsity in $\hat{\mathbf{G}}$

We aim to exploit the fact that $\hat{\mathbf{x}}$ is a sparse vector at the end of the computation because it would reduce the number of operations in $W_2 \times \hat{\mathbf{x}}$. The percentage of non-zero values at the end of the 6th iteration for both NET4 and NET4ss is less than 5% and a majority of them are expected to be negligible values. Fig. 4.7 compares the cumulative number of non-zero values in $\hat{\mathbf{x}}$ as it varies across iterations between these networks. It can be observed that NET4ss has very few non-zero values throughout the computation. A large number of non-zero values at the end of the first iteration impedes significantly reducing the operations in the case of NET4.

In NET4ss, computation of $W_2 \times \hat{\mathbf{x}}$ can be done using only a few values of $\hat{\mathbf{x}}$ which have a
Figure 4.7: Comparison of the cumulative number of non-zero values for two different networks across iterations.

Figure 4.8: Impact of using only topk values of $\hat{x}$ on the loss at the end of six iterations.
large magnitude. We call the number of values of $\hat{x}$ used as $topk$. We explored the optimal $topk$ value in Fig. 4.8. We can see that for NET4ss, the largest $\sim 20$ values of $\hat{x}$ are sufficient to obtain a loss that does not deteriorate significantly. On the contrary, NET4 exhibits largely degraded loss and reaches a good loss only if it uses almost all the values of $\hat{x}$. The reduction in the number of operations overall due the optimization is around 33.15% for a $topk$ value of 22 over NET4 without any optimizations as $W_2 \times \hat{x}$ computation is about a third of the overall computation.

4.3.4 Optimization using Sparsity in fw and fb

The dynamic change of the channel is usually not abrupt. For example, as shown in Fig. 4.1, as a car with a mobile phone inside it goes around a base station, the path gain, AoA, and AoD of the channel gradually change as the car moves, depending on the speed and distance of the car from the base station.

The movement of the user device to the base station indeed necessitates re-estimation of the channel. The opportunity is that we can make use of history or previous computation to inform the choice of the support for fb and fw for current channel estimation as long as the deviation in the input is within an estimation limit. This can significantly reduce computation complexity and thus energy dissipation since multiplication with matrix $W_1$ forms the bulk of the computation. Indeed, one cannot avoid computing all the elements of fb and fw if one makes an initial estimation as there is no knowledge of the support of these vectors.

Each element in the $\Sigma_D$ matrix (the flattened version is $\hat{x}$) contains the gain of the path for certain AoA and AoD in the grid. Each prominent path from the TX to RX has one or more non-zero elements close together like a cluster in the $\Sigma_D$ matrix. Fig. 4.9 shows an exemplar $\Sigma_D$ matrix where such non-zero clusters are indicated by colored squares. The rest of the elements of $\Sigma_D$ are either zero or negligible values. The AoA changes from $0^0$ to $360^0$ across rows and the AoD changes across the columns in the figure.

Here, we assume/set each cluster in $\Sigma_D$ shifts by one row or column as the car goes around the base station. For a grid size of 64 for TX and RX, it would take 70 ms for a non-zero element in
Figure 4.9: Image of $\Sigma_D$ matrix with colored squares being non-zero values indicating a path from TX to RX for a certain AoA and AoD. The AoA and AoD of the channel change slowly as the car goes around the base station.

$\Sigma_D$ to shift by one position if the car moves at a speed of 100 kmph at a distance of 20 m from the base station. This can be understood by first calculating the rate of change of the AoA ($\omega$):

$$\omega = (100 \times \frac{1000}{3600} m/s^{-1}) \times \frac{1}{20m} = 1.388 (rad/s). \quad (4.14)$$

The time it takes ($\Delta t$) for a cluster to move one position in the $\Sigma_D$ matrix is:

$$\Delta t = \frac{2\pi}{64} \times \frac{1}{\omega} = 70 ms \quad (4.15)$$

One needs to obtain the position of the clusters (support) in $\Sigma_D$ ($\hat{x}$), which can then be used to track the clusters’ position and compute gain as the channel changes.

We will refer to the channel estimation done for obtaining the clusters’ position before tracking as initial computation and the channel estimation done using the position of the clusters thereafter as cluster-tracking computation. Initial computation can make use of sparsity in $\hat{x}$, i.e., use topk value, but not the sparsity in fb and fw. Cluster tracking computation, on the other hand, can
make use of sparsity in fb, fw as well as  \( \hat{x} \). The support of \( \hat{x} \) obtained at the end of either initial or cluster-tracking computation gives the position of the clusters which will be used in the next channel estimation.

For this scheme to work, the channel needs to be stable while initial computation is underway. For the application we discussed, this means that the initial computation needs to finish within 70ms, which will be shown to be fulfilled by the proposed hardware later in Sec. 4.4.2. It is to be noted that sudden changes in path gain are not as severe a problem as sudden changes in AoA and AoD. In this work, we assume clusters are of size 3×3, i.e., all the prominent non-zero values for a path lie in a 3×3 window.

The key part of the algorithm is to identify the clusters. This part of the algorithm is:

Algorithm: Identifying the indices of the elements of clusters

\textbf{Require}: \( \hat{x}_{mag}, \hat{x}_{ind} \) - Magnitude and indices of \( topk \) elements of \( \hat{x} \),

Mask - Array used to mask elements of \( \hat{x}_{mag}, \hat{x}_{ind} \), initialized to all 1

\textbf{Ensure}: \( C_{ind} \) - Array of cluster indices (support for cluster tracking)

\begin{verbatim}
for(n=0; n < NumClusters; n=n+1)
    center = \text{argmax}_i (Mask[i] && \hat{x}_{mag}[i])
    append_cluster (C_{ind}, center)
    for (k=0; k < topk; k+1)
        if(\text{within_cluster}(\hat{x}_{ind}[k]))
            Mask[k] = 0
\end{verbatim}

In the above algorithm, \( \hat{x}_{ind} \) is the support of \( topk \) values of \( \hat{x} \) and \( \hat{x}_{mag} \) is the sum of the absolute values of the real and imaginary part of the corresponding elements of \( \hat{x} \). Mask is a bit array of size \( topk \), whose elements indicate the values of \( \hat{x}_{ind} \) that are not yet included as part of any cluster. \( C_{ind} \) is the list of indices indicating the position of the clusters that are obtained for possible use in the next channel estimation.

The algorithm entails finding the largest value in the list and associating its index with the center of a cluster. All the indices of the cluster, i.e., the center and the periphery of the 3×3 cluster are added to \( C_{ind} \). The values of \( \hat{x}_{ind} \) that made it to the list are masked using Mask array and the
process repeated for the number of clusters that one wants to track.

Fig. 4.9 shows how the algorithm works typically. It has three clusters, C1, C2, and C3. The red clusters indicate the support used for fb and fw based on the previous channel estimation. Cluster C1 is determined to have shifted to the right if the largest non-zero value moved to the right by one position after channel estimation using new input data. We track four clusters despite there being only three clusters in the input data to increase the robustness of the algorithm and not miss any prominent non-zero values which might affect loss.

This cluster tracking approach can reduce the number of operations while minimizing the negative impact on the loss. Fig. 4.10 shows the results of experiments for several different assumptions on the rate of change in AoA of each path in the channel (ΔAoA). The loss is averaged over 20 sets of data each with different AoA, AoD, and path gain combinations. As expected, the number of operations increases as more clusters are tracked. The encircled region corresponds to four clusters and is a sweet spot for loss and the number of operations. The degradation in loss is severe if ΔAoA is large, which is when tracking is lost and initial computation needs to be per-
Figure 4.11: Reduction in the number of complex MAC operations by the proposed techniques. Initial refers to initial computation that uses none of the proposed optimizations. TOPK indicates the use of only the top 22 values of $\hat{x}$. CT refers to cluster tracking.

formed again to find the clusters’ position.

In Fig. 4.11, we summarize the benefit of the optimizations that we proposed in Secs. 4.3.3 and 4.3.4. The initial computation that uses none of the techniques takes about 21M complex-number MAC operations. The use of sparsity in $\hat{x}$ through the use of topk values can reduce the number of operations in initial computation by 33.15%. Also, cluster tracking can reduce the number of operations by 114.5X. Combining those techniques reduces the total number of operations by 131.7X.

4.3.5 Quantization and Pruning

Quantization and pruning are highly popular techniques in training a neural network model for a classification task. Multiple works show aggressive quantization, sometimes even down to 1 bit, can reduce computational complexity by orders of magnitude while still maintaining a decent accuracy [50][51]. However, channel estimation involves regression where the values of the $\hat{x}$ matter, unlike classification where only the ranking among the elements of the final vector matters. The
values of weights and especially activation in this problem therefore would require higher precision than classification.

We experimented across ranges of quantization and pruning degrees. We noticed that the 12-bit quantization is sufficient for activations and 16-bit for partial sums (the results obtained during accumulation in vector-vector dot product). Also, we used pruning to induce 65% sparsity in W1 and 80% in the W2 matrix. This further allowed us to reduce the bit widths of weights down to 4 bits. Fig. 4.12 shows some of the results of our experiment. The loss is the average across 20 different sets of data, each at the end of the sixth iteration. Each set contains two inputs, the first used for initial computation and the second for cluster tracking computation. We generated the second input in each set by changing the AoA by either $8^0$ or $4^0$ or $0.5^0$ for each path. The $topk$ value used is 22. We chose 4 bit as the bit width for the weights due to bit alignment constraints in hardware and negligible loss deterioration in the case of cluster tracking with $\Delta AoA$ as $0.5^0$. The rate of change of AoA for the example we discussed for the car going around the base station is $0.43^0$ which is lower than $0.5^0$. 

![Figure 4.12: Impact of quantization of the weights on the loss.](image-url)
4.4 Hardware Implementation and Results

4.4.1 Hardware Development

To measure the benefit of the proposed techniques, we implemented a channel estimation sub-system using the Zybo Z7-20 FPGA board which contains a Zynq 7020 (xc7z020clg400-1) FPGA-processor system and 1GB DDR3L DRAM with a 32-bit bus running at 1066MHz [52]. Fig. 4.13(a) shows the top-level architecture. It consists of a single ARM core, BRAM, an accelerator, and AXI interconnect. The accelerator has six main components: L1-calc, weight caches (cache-W1 and cache-W2), three multiply-and-accumulator blocks (MAC1, MAC2, and MAC3), a sorter, a cluster finder, and a Sum block. It also has an AXI-Lite slave port and an AXI-4 master port. The master port accesses the BRAM for inputs. It also accesses the DRAM through the ARM processor’s high-performance slave port. The SD card initially stores the inputs, weights, and other parameters.

In the setup phase, the ARM processor runs an embedded C program that moves the weights from the SD card to the DRAM. It also writes the inputs to the BRAM. The parameters $\theta$, $\gamma$, and
batch size are passed onto the accelerator using the AXI-Lite slave port. Finally, a start signal is given to the accelerator, and the processor polls for the done signal from the accelerator to measure its performance.

The detailed accelerator operation is described in Fig. 4.13(b). It starts with the L1 Calc which first obtains the inputs from BRAM and computes the L1 difference between the previous and the current inputs. If the difference is over a threshold, the accelerator performs initial computation, otherwise, cluster tracking computation. For initial computation, it uses MAC1 and MAC3 for matrix-vector products. For cluster tracking, it uses MAC1 and MAC2 for matrix-vector products.

If it is the first iteration among six iterations, it performs $W_1 \times b$ using MAC2 or MAC3 and stores the product in DRAM for initial computation and Temp-Mem for cluster tracking computation. If it is not the first iteration, MAC1 computes the product:

$$d = W_2 \times \hat{x}$$ (4.16)

using topk values of $\hat{x}$. The product $d$ is stored in Temp-Mem and used by MAC2 or MAC3 to compute $W_1 \times d$, which is again stored in DRAM if it does initial computation or Temp-Mem if cluster tracking.

Then, the Sum block calculates $f_w$ from the stored product $W_1 \times b$ and $fb$ from $W_1 \times d$ by scaling them. It then adds $fb$, $fw$ and $\hat{x}$ (from sorter) and applies a non-linearity. The sorter then checks if each entry is among the topk elements. Then, it goes back to MAC1 and the next iteration is commenced.

At the end of all six iterations, the cluster finder block identifies the indices of the clusters from the topk elements using the algorithm discussed in Sec. 4.3.4. These indices are used to identify the blocks in Cache-W1 and Cache-W2 which are no longer needed and are marked dirty. The channel matrix is also computed using (4.7).

To reduce data movement between the accelerator and DRAM, we design two custom caches for weights, one each for $W_1$ and $W_2$. The caches handle only read hits and misses as weight data
is read-only. Initially, \( W_1 \) is stored in a row-major format in DRAM, and \( W_2 \) is stored in a column-major format in DRAM. We set the block size of caches to be the size of one row for cache-W1 or the size of one column for cache-W2 and hence the tags used for determining a hit or a miss are the indices in the support of \( \hat{x} \). A read miss is handled by fetching an entire row or column from DRAM and storing its contents in consecutive locations in the on-chip memory. The cache is made fully associative so that the on-chip memory is utilized completely. The comparisons for determining a hit are done serially to reduce hardware utilization on the FPGA. In case of capacity misses, dirty blocks are replaced first, then, we replace the blocks which were accessed first, i.e., we use the first access first out policy.

As shown in Fig. 4.14, the caches largely reduce the number of words fetched from DRAM, improving energy-efficiency. When we start tracking the clusters in the channel (CT-Start), the caches help to reduce DRAM accesses by 269X where each access from DRAM fetches 64 bits of data. If the accelerator continues to track the cluster movements in \( \Sigma_D \) matrix, the DRAM accesses additionally reduces. For example, if the cluster shifts by one row and column, i.e., the center of the cluster moves to the corner (CS-Corner), the DRAM access reduces further by 22%. If the

Figure 4.14: Reduction in DRAM accesses due to cache hits.
cluster center shifts to edge (CS-Edge), i.e., cluster shifts by either one row or column, the DRAM access reduces by \(\sim 25\%\) over CT Start. If the clusters do not move but only the gain changes (No-Shift), the accelerator does not need to fetch from DRAM and it stores only intermediate results in DRAM and reads back. To maximize the benefit of the caches, we set the cache size sufficiently to store all the blocks of weights needed during cluster tracking computation.

4.4.2 Results

We designed our accelerator in C++ and used Xilinx Vitis HLS v2020.2 to get the RTL. We synthesized the system and programmed the FPGA using Vivado v2020.2. We run our system at 125 MHz and the board draws a power of about 2.08W. The measured latency of initial computation is 64.14ms and that of cluster tracking computation is about 1.57ms. The system is energy efficient and consumes only 102mJ for initial computation and 2.4mJ for cluster tracking. The system consumes around 64\% of the Slice LUTs, ~30\% of the Slice Registers, DSPs, and BRAMs on the Zynq 7020 for the chosen topk value of 22. The resource usage is summarized in Table 4.1.

We compared our system with a platform containing Dual CPU Intel Xeon E5-2620v3 running the baseline (NET4 with no optimizations) using NumPy 1.18.5 and also an Nvidia 2080 Ti GPU using TensorFlow 2.2.0. The operating system used by the host is CentOS 7.6.181. In the case of initial computation, the CPU and GPU perform better in terms of latency by \(\sim 4X\). But our system improves on the energy consumption by 8.26X over the CPU and 7.23X over the GPU. In the case of cluster tracking computation, the FPGA achieved the latency improvement by \(\sim 10X\) and the energy improvement by 337.64X when compared to CPU and 295.3X when compared to GPU. The improvement in latency and energy comes at a cost of \(\sim 3dB\) in the loss, of which close to 2dB occurs due to quantization and pruning. Table 4.2 summarizes the comparison between different platforms.

Also, we analyze the impact of the value of topk on the latency of the entire computation. We varied the value of topk from 10 to 25, synthesized different versions of the accelerator and measured the latency in each case. Fig. 4.15(a) shows the latency for initial computation and Fig.
Table 4.1: Resource Utilization Table

<table>
<thead>
<tr>
<th>Resource</th>
<th>Number</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>33,829</td>
<td>63.59</td>
</tr>
<tr>
<td>Registers</td>
<td>34,735</td>
<td>32.65</td>
</tr>
<tr>
<td>DSP</td>
<td>58</td>
<td>26.36</td>
</tr>
<tr>
<td>BRAM</td>
<td>48</td>
<td>34.29</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison Table

<table>
<thead>
<tr>
<th>Platform</th>
<th>FPGA</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Zynq 7020</td>
<td>Dual Intel Xeon</td>
<td>Nvidia 2080 Ti</td>
</tr>
<tr>
<td>Language</td>
<td>C++ HLS</td>
<td>NumPy, Python</td>
<td>Tensorflow, Python</td>
</tr>
<tr>
<td>Frequency (GHz)</td>
<td>0.125</td>
<td>2.4</td>
<td>1.35</td>
</tr>
<tr>
<td>Network</td>
<td>NET4ss</td>
<td>NET4</td>
<td>NET4</td>
</tr>
<tr>
<td>Weight Precision</td>
<td>4 bits</td>
<td>Float32</td>
<td>Float32</td>
</tr>
<tr>
<td>Activation Precision</td>
<td>12 bits</td>
<td>Float32</td>
<td>Float32</td>
</tr>
<tr>
<td>Loss (dB)</td>
<td>-7.159 (I), -6.849 (CT)</td>
<td>-9.73</td>
<td>-9.73</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>64.14 (I), 1.57 (CT)</td>
<td>16</td>
<td>15.1</td>
</tr>
<tr>
<td>Power (W)</td>
<td>2.08*</td>
<td>69.06†</td>
<td>64</td>
</tr>
<tr>
<td>Energy (J)</td>
<td>0.102 (I), 0.0024 (CT)</td>
<td>1.12</td>
<td>1.204</td>
</tr>
</tbody>
</table>

*I obtained current using the equation in [53] † Obtained typical power from [54]

4.15(b) shows the latency for cluster tracking computation for different values of topk. We chose the topk value of 22 as the final value for comparison with other platforms as it gives the best loss. The change in AoA in a single time step is chosen to be 0.5° which is slightly greater than the value for the example application we discussed.

4.5 Summary

We presented algorithm hardware co-optimization and the matching FPGA subsystem implementation based on the model-based neural network for the channel estimation in 5G mmWave communication systems. We explored multiple such neural network structures and methods that make use of sparsity to reduce the computational complexity. Further, a reduction in the bit precision used for weights and activations and a cache tailored for the application allowed for a reduction
in the number of off-chip memory accesses. The FPGA prototype achieved an improvement in latency of up to $\sim$10X over Intel Xeon processor and Nvidia 2080 Ti GPU at a cost of deterioration in the loss of $\sim$3dB largely due to quantization and pruning. The improved latency combined with low resource usage reduced the energy consumption of the subsystem by $\sim$300X over the CPU and the GPU based systems.

Figure 4.15: Impact of topk value on latency and the corresponding loss for (a) initial computation (b) cluster tracking computation.
Conclusion

Always-on IoT devices are becoming ubiquitous, aided partially by the advancements in deep learning and VLSI technology. Neural networks are being trained in the datacenter to recognize the patterns in the data gathered through these devices. The patterns in the data provide opportunities to introduce new services and improve the efficiency or performance of existing services. 5G communications which itself will employ several always-on small cell base stations, is expected to accelerate the adoption of IoT devices and training of neural networks in the cloud. The thesis focused on addressing the issue of power consumption of always-on systems in these three different scenarios by making a set of context-specific algorithmic and hardware interventions targeting neural network computations.

Chapter 2 of the thesis presented the use of SNNs and supporting hardware to achieve ultra-low power for a classifier. The classifier can be used as a wake-up circuit in an IoT device through functions like keyword spotting. It leveraged the fact that spike-based computation and communication in SNNs can be exploited in hardware through event-driven clock and power-gating. This co-design technique along with low leakage SRAM and efficient arbiter implementation which also simplified on-chip communication helped us achieve state-of-the-art power consumption while meeting real-time constraints.

Chapter 3 of the thesis presents an attempt to study the feasibility of the FPGA as an alternate to the GPU for training neural networks. Some of the challenges to the use of FPGA for training neural networks are the lack of efficient libraries to reduce design time and a smaller number of DSP units when compared to GPUs. But, FPGA tends to consume lower standby and
active power compared to a GPU of the same generation, warranting an investigation into its suitability in training neural networks. The chapter presents an accelerator that is designed for binary-weight convolutional neural networks whose weights and temporary data can fit entirely in the on-chip memory. It uses batch renormalization instead of batch normalization to reduce the batch size needed to obtain decent accuracy, consequently leading to better utilization of on-chip memory and the ability to accommodate a much larger network than what would have been possible otherwise. It shows that the performance of the GPU relative to the FPGA improves as the network size increases despite the advantage of using fixed-point computation. The reason for this could be the amortization of the thread management overhead of the GPU and higher resource idle time in the architecture of the accelerator due to data dependencies in training.

Chapter 4 of the thesis proposes the use of model-based neural networks for the problem of channel estimation at mmWave frequencies. It presents techniques to leverage sparsity inherent in mmWave communication channels to reduce the complexity of the computation by up to two orders of magnitude. It makes use of the fact that the angle of arrival and departure of the communication channel does not change abruptly to reduce the computation and also the weights of the neural network that need to be fetched from off-chip memory. It demonstrates an FPGA based subsystem that showcases the benefits of leveraging the sparsity. Further, the subsystem employs a cache that makes use of the continuity in the angle of arrival and departure to minimize the off-chip memory accesses, contributing to a reduction in power consumption. The subsystem obtained a speedup compared to the CPU and GPU when using cluster tracking computation at a much lower power consumption.

The thesis presented a collection of algorithmic and hardware techniques in three contexts. It targeted the power consumption of neural network workloads on always-on systems which is a significant part of the total power consumption. The techniques presented for SNN are candidates for use in wake-up circuits in battery-powered IoT devices. Chapter 3 tried to reveal the challenges of using FPGA for accelerating training. Chapter 4 demonstrated the advantage of using the knowledge of the application to reduce computation and designing supporting hardware
to reduce off-chip memory accesses thereby reducing power and improving efficiency.
References


