Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2025

Automatic Modulation Classification: Dataflow Hardware Acceleration on FPGA

Oscar Wahllöf

LINKÖPING UNIVERSITY

Master of Science Thesis in Electrical Engineering

Automatic Modulation Classification: Dataflow Hardware Acceleration on FPGA

Oscar Wahllöf

LiTH-ISY-EX–25/5808–SE

Supervisors: **Olle Hansson**
ISY, Linköping University
Carl Ingemarsson
Swedish Defense Research Agency - FOI

Examiner: **Oscar Gustafsson**
ISY, Linköping University

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

This thesis details the full process of producing a system in the form of a dataflow accelerator on an FPGA that performs modulation classification. Some background on the four areas: dataset generation, quantization in the context of machine learning, and hardware acceleration of algorithms, along with some introductory information about the framework FINN, is provided. The implementation process is described, and the findings show that, with synthetic data, machine learning models can be trained with quantization-aware training to produce a classification of the modulation type with acceptable accuracy. Hardware can be constructed to run such models on an FPGA, with low power consumption on consumer-grade devices, with sufficient performance to be usable for practical applications. There are limitations and considerations that need to be made when constructing such a system, and the design of both hardware and model jointly is critical for success.

Acknowledgments

Firstly, I would like to thank the Swedish Defense Research Agency, FOI, for the opportunity to do my thesis with them and for their support. I have received lots of help from many people and have learned a lot.

Secondly, I would like to thank my supervisors, Carl Ingemarsson and Olle Hansson, for their help, guidance, and interesting conversations on topics I struggled with. My examiner, Oscar Gustafsson, who held many of the courses I chose to take during my education that agreed to be the examiner and has taught me so much.

Thirdly, I would like to thank all my friends, peers, and lab partners who have kept me somewhat sane and for making my studies fun.

Finally, I would like to thank my family, my father especially, for always being there to support me and for long conversations on anything I struggled with for as long as I have lived, and hopefully for many years to come.

We choose to do things, not because they are hard, not because they are easy, but because we thought they were going to be easy.

Linköping, August 2025
Oscar Wahllöf

Contents

List of Figures

List of Tables

Terms & Abbreviations

Abbreviations

Abbreviation	Description
ADC	Analog-to-Digital Converter
AMC	Automatic Modulation Classification
ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DSP	Digital Signal Processing
DWC	Data Width Converter
FIFO	First in First out
FPGA	Field-Programmable Gate Array
GLB	Global On-Chip Buffer
HDL	Hardware Description Language
IP	Intellectual Property
LUT	Lookup Table
MAC	Multiply-Accumulate
MPSoC	Multiprocessor System on Chip
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
QNN	Quantized Neural Network
RF	Radio Frequency
RFSoC	Radio Frequency System on Chip
RTL	Register Transfer Logic
SDR	Software Defined Radio
SIGINT	Signal Intelligence
SoC	System on Chip
SoM	System on Module

1 Introduction

Neural networks have been shown to be effective for performing Automatic Modulation Classification (AMC) [1]. However, the large amounts of computation that can be parallelized and power constraints in a number of applications make them very suitable for acceleration on dedicated hardware. This thesis tries to investigate accelerators for neural network acceleration in FPGAs, while evaluating what performance trade-offs can be made. A selected model is implemented in an embedded system and evaluated to determine the performance in practice.

1.1 Motivation

Due to congestion of available transmission channels, it is of interest for a system such as cognitive radio networks [2] to be able to change its transmission and reception configuration. This information can be used to ensure correct signal decoding and demodulation, but also allows different modulation schemes for adapting to channel conditions. Real-time communication systems depend on the channel quality, which can change due to noise, interference, and fading. Automatic modulation classification [3] can therefore be used in many applications to help optimize spectral efficiency [2].

For this thesis, AMC is to be used in a *signal intelligence* (SIGINT) context, where it is of high interest to identify the modulation type of signals sent by an adversary in order for signal interception and analysis to be performed. Additionally, modulation schemes could be constructed to deliberately confuse traditional surveillance systems or constructed dynamically depending on the channel conditions and interference [4].

Field-Programmable Gate Arrays (FPGA) have performance benefits for certain applications compared to CPU or GPU implementations in terms of latency, throughput, and power efficiency [1, 5, 6]. It is therefore of interest to investigate how these benefits can be leveraged in order to perform AMC.

1.2 Aim

The aim of this thesis is to investigate how a Convolutional Neural Network (CNN) model for AMC can be accelerated in dedicated hardware. A main issue to consider is how existing frameworks and tools can be used to aid in the design process. An implementation of the network is to be created and evaluated, particularly with focus on the parameters that can be adjusted and how they affect the throughput, latency, energy consumption, and design size. How much of the model weights and computation can be quantized, and to what degree they affect the performance, is also a key factor to be considered.

1.3 Research Questions

1. How can a machine learning model inference be implemented on an FPGA platform?
2. What restrictions exist for a model that would make it suitable for inference on an FPGA?
3. How does the number of input samples affect model and hardware results?
4. What are the performance results of an accelerator for CNN for Automatic Modulation Classification running on a Zynq-UltraScale+ MPSoC? What parameters can be configured?

1.4 Delimitations

This project is limited to convolutional neural network models for hardware implementation, with novel model creation being outside the scope. The models can be modified and optimized to improve their suitability for FPGA-based acceleration. A selection will be made of frameworks, tools, and models to be evaluated, and performance evaluations will be conducted specifically in terms of classification accuracy, throughput, latency, energy consumption, and hardware utilization.

2

Related Work

In this section, a number of related works are discussed relating to several frameworks and tools for neural network accelerator design, design methodologies for dataflow accelerators, but also some specific work on modulation classification of radio frequency signals.

Hamanaka *et al.* [7] explore the differences between Vitis AI and FINN in terms of the performance of accelerators for Deep Neural Networks (DNN). The paper demonstrates that FINN-based accelerators hold a significant advantage in terms of latency, throughput, and power efficiency over models running on a consumer-grade desktop PC, but also a Jetson Nano platform with similar power envelope as the FPGA. The paper concludes that the use of dataflow accelerator automation frameworks yields DNN accelerators with significant performance advantages compared to overlay types, but that there are limitations and restrictions on what types of models are suitable.

The paper by Jentzsch *et al.* [5] explores the performance of DNN accelerators generated by FINN on RFSoC SoC ZCU111 and PYNQ RFSoC 2x2 boards. The findings show very promising performance with flexible configuration options due to the framework used. The objectives of the first section of this thesis overlap with the work in this paper. However, the datasets used, namely the RadioML, have since been shown to contain errors [8], and there is further value in evaluating how the performance of synthetic data used in this thesis compares.

Research by Tridgell *et al.* [9] shows that real-time RF AMC for very high throughput applications is a very viable approach. Very high performance is demonstrated on an RFSoC ZCU111 platform, but they also use the RadioML dataset used by [5] above. The models investigated are VGG10 and ResNet33. The paper focuses on investigating low precision, ternary weights, and how the model can be trained by incrementally increasing the precision of activations to maximize classification accuracy for full utilization. The design framework used is a purpose-built Python generator and not any of the aforementioned frameworks; hence, this does not overlap much with the objectives of this thesis, but provides some interesting performance comparison opportunities. It is also noteworthy that the built-in RFSoC ADCs were used, specifically with a custom-made transmission IP in loop-back for only 4 modulation types with no added noise.

The work by Maclellan *et al.* [10] on modulation classification demonstrates a viable design alternative with MathWorks HDL Coder [11]. The design implements a streaming-based architecture, and evaluates three configurations: a 16-bit weight and activation, an 8-bit weight and 16-bit activation, and a 4-bit weight and 16-bit activation quantized model. The models are trained with Brevitas and tested live on the PYNQ RFSoC 2x2 board in a loopback configuration. While the data used was synthetically generated, and a detailed description of how this was done is provided, only eight modulation types were considered. Further, as noted in the paper, the HDL Coder processes data in byte increments. The 4-bit model was shown to save no BRAM resources compared to the 8-bit model, which is a very poor restriction for implementations of models for this application. Fairly unimpressive results compared to other works [9], and not very efficient resource use.

Natale *et al.* [12] present a design methodology for dataflow accelerators based on Streaming Stencil Time-step. Separating the design into two parts: a memory system and a computing system. These are interconnected with FIFO channels, with the benefit that data can be managed between channels such that it is only read once from off-chip memory. A computational core performs the convolution and fully connected layers, with data flowing through input to output channels. Some extrapolation on the relationship between the parallelization and performance of the design is presented. However, the implemented designs are very small in size and different from other works in the area, making a clear comparison difficult. While the presented methodology is functional, it does not seem to provide a clear benefit over other frameworks or tools.

2.1 Network Architecture and Signal Synthesis

The paper by Svensson and Becirovic [13] on one-shot modulation recognition using machine learning was the foundation for this work. Their model inspired the network topology that this thesis aims to explore in hardware. While the model used in this work is a more standard CNN network, the results can be seen as a step towards implementing theirs in hardware. The signal theory used to create the raw signals in this thesis is also taken from their work.

3

Signal Theory

This chapter gives some introductory information on the topic of digital communication systems, signal modulation, and noise that should be sufficient to understand the rest of this thesis.

3.1 Signal Modulation

Signal modulation is the process of encoding information onto a carrier wave for transmission. In digital communication systems, bitstreams are divided into segments, each corresponding to a symbol in the modulation scheme. The symbols are mapped onto analog waveforms by varying one or more properties of the wave, typically its amplitude, frequency, or phase.

3.2 I/Q-Data

In communication systems, modulated signals can be represented using in-phase (I) and quadrature (Q) components. These signals are generated by applying a chosen modulation scheme, which encodes digital information into variations of amplitude, phase, or frequency.

The resulting I/Q data consists of pairs of samples corresponding to orthogonal projections of the modulated waveform. On the transmit side, I/Q data is converted to Radio Frequency (RF) by upconversion. On the receive side, RF signals are downconverted to I/Q for demodulation.

3.3 Channel

Signal Distortion is modeled with the following effects in addition to additive white Gaussian noise (AWGN).

- Carrier-frequency offset
- Timing offset
- Phase offset
- Flat Rayleigh fading due to multipath propagation

3.4 Signal-to-Noise Ratio

Signal-to-Noise Ratio (SNR) is a metric in communication systems that quantifies the relationship between the power of a desired signal and the power of background noise. SNR is defined as:

$$\mathrm{SNR}_{\mathrm{dB}} = 10\log_{10}\left(\frac{P_{\mathrm{signal}}}{P_{\mathrm{noise}}}\right) \tag{3.1}$$

Where P_{signal} is the average power of the received signal. P_{noise} is the average power of the noise.

Higher SNR indicates a signal with less interference, which generally leads to lower bit error rates and improved demodulation accuracy. In digital communication systems, SNR directly impacts the performance and reliability of modulation schemes. More complex modulation formats require higher SNR values to maintain acceptable error rates compared to simpler schemes.

3.5 Signal-to-Noise and Distortion Ratio

Signal-to-Noise and Distortion Ratio (SINAD or SNDR) is a metric used to evaluate the performance of analog-to-digital converters (ADCs) and other communication system components. SINAD quantifies the ratio between the power of the signal and the combined power of all noise and distortion components. SINAD is defined as:

$$\mathrm{SINAD}_{\mathrm{dB}} = 10\log_{10}\left(\frac{P_{\mathrm{signal}}}{P_{\mathrm{noise}} + P_{\mathrm{distortion}}}\right) \tag{3.2}$$

Where P_{signal} is the power of the input signal. P_{noise} is the power of the noise components. $P_{\mathrm{distortion}}$ is the power of distortion effects.

3.6 Effective Number of Bits

The Effective Number of Bits (ENOB) is a performance metric that quantifies the dynamic resolution of an analog-to-digital converter (ADC). While an ADC may have a nominal resolution of N bits, non-idealities such as thermal noise, quantization error, and nonlinearities reduce its effective resolution. ENOB reflects how many bits of a converter are actually useful in the presence of noise and distortion.

ENOB is derived from the Signal-to-Noise and Distortion Ratio (SINAD), and is calculated using the following formula:

$$\mathrm{ENOB} = \frac{\mathrm{SINAD}_{\mathrm{dB}} - 1.76}{6.02} \tag{3.3}$$

Where SINAD_{dB} is the signal-to-noise and distortion ratio measured in decibels. The value 1.76 dB is the quantization noise floor for an ideal ADC. The value 6.02 dB corresponds to the increase in SNR per bit of resolution in an ideal converter.

4 Quantized Machine Learning

Machine learning is typically done with floating-point arithmetic in software. Models need to be adapted to be implemented in hardware using fixed-point; this chapter addresses some of the ways this may be performed.

4.1 Brevitas

Brevitas [14] is a PyTorch library for neural network quantization, with support for quantization-aware training and post-training quantization. Brevitas creates abstracted sections to model reduced-precision hardware data paths at training time. Primarily, it can be used for affine quantization, with a focus on uniform quantization.

4.2 Post Training Quantization

Post-training quantization is a technique where a trained neural network's weights and activations are converted from high-precision, typically floats, to lower-precision formats such as low bit-width fixed point after training by simply removing precision.

4.3 Quantization-Aware Training

Quantization-aware training is a technique where the effects of low-precision quantization and arithmetic are simulated during the training process. This allows the model to learn and adapt to the reduced precision, resulting in better accuracy.

4.3.1 QuantTensor

A QuantTensor is a tensor that includes both data and quantization metadata, enabling handling of quantized values during training and inference in quantization-aware machine learning.

4.4 Quantization

Quantization is a much larger topic than what is covered in this work; the following two schemes are sufficient to understand how quantization is performed in Brevitas.

Uniform Affine Quantization

Uniform Affine Quantization is a mapping of continuous real values to discrete integer values using a fixed scale and zero-point.

$$q = \mathrm{round}\left(\frac{x}{s}\right) + z \tag{4.1}$$

Uniform Symmetric Quantization

Uniform Symmetric Quantization is a mapping of continuous real values to discrete integer values symmetrically around zero, using a fixed scale but no zero-point offset.

$$q = \mathrm{round}\left(\frac{x}{s}\right) \tag{4.2}$$

4.5 Determining Quantization Parameters

Quantization parameters can be configured statically prior to training, or allowed or be set as a learnable parameter. This can allow the model to automatically learn where there is increased sensitivity and a need for more precision, and conversely, where more aggressive quantization can be performed. In this work, all quantization was set statically and uniformly between all layers. Quantization and normalization can be configured to be set per layer or per tensor.

4.6 Quantized Layers

To enable quantization-aware training, a model is constructed through the use of quantized layers instead of typical PyTorch layers. The following lists the quantized layers used in this work.

QuantLinear: Replaces the Linear layer in a quantized model.

QuantConv2d: Performs the convolution operation with quantization. One can observe that a 1D convolution can be thought of as a $1 \times X$ 2D convolution.

QuantReLU: The quantized version of the rectified linear unit function.

4.7 Input Data Quantization

Typically, the input data to a quantized model does not have the specific precision and quantization as well as range required by the model. This entails a requirement to pre-process the input data. This can be done either through the addition of a layer that performs this task as part of the model, given that the data has known characteristics. The issue can also be addressed by ensuring that the data is generated in the correct format or by processing it in some other way before it is passed to the model.

4.8 Binarized Neural Network

A specific benefit for some models implemented on FPGAs is that, with high enough resilience, they can be implemented as Binarized Neural Networks. In essence, the weights are absorbed into the logic fabric, and activations and weights are simply one bit that make up the logic. This leads to small implementations, which means massive parallelization and performance.

4.9 Network Surgery

Network surgery is the process of manually adjusting and changing a model network by introducing new operations or manipulating existing ones.

4.10 ONNX

ONNX [15] provides a framework and definitions for extensible computation graph models, support for standard data types, and built-in operators for inference. A graph is represented in an acyclic form, where each node has inputs & outputs and represents a call to an operator. The focus of such a dataflow graph is to illustrate how the data moves and the order of operations applied to it, but also to act as a machine-understandable format for further analysis and processing.

QONNX

Quantized ONNX or QONNX is an extension of the ONNX format designed to support quantized neural network models. It adds operators and metadata for representing quantization details in addition to normal ONNX functionality. In this work, all (Q)ONNX models are acyclic feed-forward graphs, as opposed to models such as recurrent neural networks.

4.11 PyTorch

PyTorch [16] is a machine learning library for Python, based initially on the Torch library. PyTorch can be used for computer vision and numerous other applications and was developed by Meta AI, and is now part of the Linux Foundation. PyTorch is one of the most popular deep learning frameworks and is open-source.

4.12 Netron

Netron [17] is a popular open-source tool for visualizing neural network models as a graph, where nodes represent layers or computation units, and the edges as activations or intermediate values.

5

Hardware Acceleration

The purpose of this chapter is to give a brief overview of hardware acceleration concepts and structures typically employed, as a basis for explaining the design choices made.

5.1 Acceleration of Machine Learning in Hardware

Machine learning workloads are computationally intensive and involve large volumes of data movement and computation. To meet the performance and efficiency requirements, hardware acceleration techniques are employed in many forms. These techniques include exploiting parallelism, reducing numerical precision, optimizing memory access patterns, and much more. Custom hardware platforms such as FPGAs, GPUs, and ASICs can be used, providing a trade-off between flexibility, performance, power efficiency, and design implementation time. This section outlines fundamental strategies used in hardware design to accelerate machine learning computations.

5.1.1 Compute-Bound and Memory-Bound

Fundamentally, a design can be either compute-bound or memory-bound. A completely efficient hardware design is close to both compute and memory-bound, utilizing both at near full capacity. As this work is centered on the creation of accelerators for an FPGA platform, most of the following argumentation is with that in focus.

FPGAs are suitable for High Compute per Data, where pipelining can be utilized and operations can be heavily parallelized. Machine learning is typically done with floating-point arithmetic, and while this certainly can be implemented on an FPGA, it is not efficient.

5.1.2 Reduced & Custom Precision

Reduced and custom numerical precision is a crucial optimization in FPGA-based hardware acceleration. By lowering the bit width of data representations, from standard 32-bit floating point to formats like fixed-point, or even binary or polar representations, both computation and memory requirements are significantly reduced. FPGAs are particularly well-suited for this, as they allow custom datapaths and arithmetic units tailored to the chosen precision.

5.1.3 Convolutional Neural Networks

Convolutional Neural Networks are a type of neural network used for processing structured data, typically images and video. The core of how CNNs operate is through the convolution operation, a mathematical operation that can be thought of as sliding a function over another and adding up the overlap. The result produces a representation of how the two original functions interact. This can be used to extract feature maps like edges, corners, or textures for use in later layers where the neural network can recognize more complex shapes or patterns.

5.1.4 Convolution Lowering

Convolutions can be lowered to vector-matrix multiplication, this is illustrated below in Figure 5.1 and Figure 5.2 for the case of 2D and in Figure 5.3 and Figure 5.4 for 1D convolution. By doing this, efficient algorithms for performing such multiplications can be leveraged. There is, however, a penalty in terms of data duplication. As can be observed in the case of the 1D convolution, for a kernel size of N, there is a data duplication factor approaching N for large inputs.

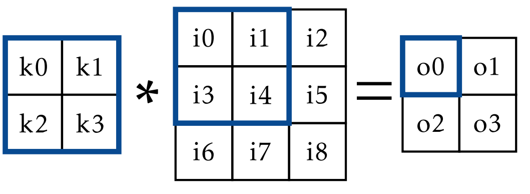

Figure 5.1: *Illustration of 2D convolution with kernel, input feature map, and resulting output with highlights.*

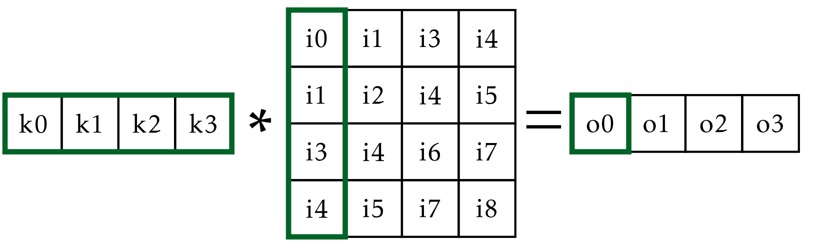

Figure 5.2: *Illustration of the lowered 2D convolution with highlights.*

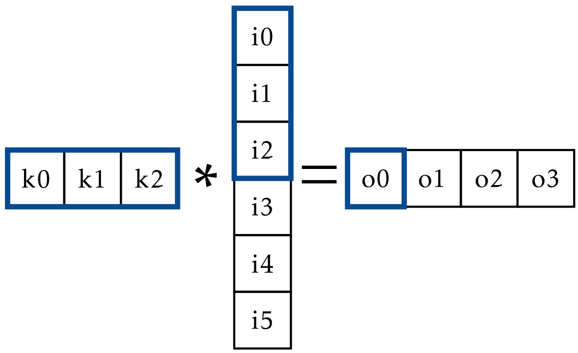

Figure 5.3: *Illustration of 1D convolution with kernel, input feature map, and resulting output with highlights.*

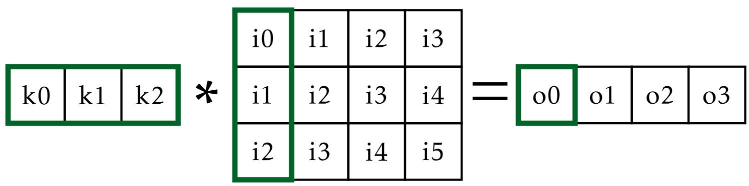

Figure 5.4: *Illustration of the lowered 1D convolution with highlights.*

5.1.5 Process Element

A process element is the functional unit performing some arithmetic, typically tailored to be task specific through implementing a specific, non-standard instruction, custom data formats, or fused operations.

Fused Operations

A process element typically performs some type of fused operation, such as Multiply-Accumulate (MAC).

5.1.6 Weight Stationary

As machine learning can be heavily data transfer dependent in terms of moving both large volumes of intermediate data and weights, it is efficient to store weights integrated into computational units instead of in a centralized memory. This can lessen the issue of being memory-bound and lower the utilization of the network on chip.

5.1.7 Output Stationary

With similar benefits as weight stationary configurations, the intermediate data can be fed directly to a consuming computational unit, and therefore not have to be stored in the GLB.

5.2 Synchronous Dataflow

A Synchronous Dataflow accelerator is a hardware design that executes computations modeled as a static dataflow graph, where each operator consumes and produces a fixed number of data. The spatial and temporal mapping of computation allows for minimal use of interconnect and no need for a GLB, as data is forwarded between nodes in the graph either directly or with the use of FIFOs.

5.3 High-Level Synthesis

High-Level Synthesis is a design methodology that converts high-level programming languages like C/C++ or some other construct into hardware descriptions such as Verilog or VHDL. It attempts to enable faster development of digital circuits by abstracting low-level RTL design.

5.4 PYNQ

PYNQ [18] is an open-source framework that aids in the process of hardware design for Xilinx Zynq-based FPGAs by providing a Python framework. Through integration with Jupyter Notebooks, PYNQ supports high-level abstractions for running pre-built FPGA configurations called *overlays* on the device. Allowing for ease of use for development, testing, benchmarking, and educational purposes in a number of fields such as machine learning, image processing, and embedded systems.

5.4.1 Overlay

In PYNQ, an overlay is a FPGA hardware design and metadata that can be dynamically loaded onto the FPGA fabric from Python. Overlays abstract hardware interfacing, enabling users to run hardware accelerators or peripherals by controlling them through high-level Python APIs.

5.5 Vivado and Zynq Platform

Vivado is Xilinx's design suite for developing FPGA and SoC designs, providing tools for synthesis, implementation, and IP integration. The Zynq platform combines ARM processors with FPGA fabric on a single chip.

5.5.1 PS

The Processing System in Zynq devices includes ARM processor cores and standard peripherals. It handles control and software tasks, working alongside the

Programmable Logic via high-speed Advanced eXtensible Interface (AXI) interfaces for efficient hardware–software integration.

5.5.2 PL

The Programmable Logic in Zynq devices is the FPGA fabric used to implement custom hardware. It enables parallel and optimized processing offloading, tailored to specific tasks or custom protocol implementation for peripheral interfacing, for instance.

5.5.3 IP

An Intellectual Property core is a reusable, hardware module used within the Programmable Logic of an FPGA. IPs can perform specific functions such as arithmetic operations, memory access, or communication, and can be custom-built or sourced from libraries like Xilinx's IP catalog.

Direct Memory Access (DMA)

The Direct Memory Access enables high-speed data transfer between the Processing System and Programmable Logic or memory without direct CPU management. This improves performance by offloading data movement tasks, making it essential for efficient hardware acceleration of data-intensive applications.

Data Width Converter (DWC)

The Data Width Converter IP adjusts data bus widths between components with differing interface sizes.

5.5.4 IP Integrator

The IP Integrator is a graphical design tool within the Xilinx Vivado environment. It enables a user to connect and configure multiple IP cores, manage data paths, and create custom hardware designs through a drag-and-drop interface.

5.5.5 AXI

The Advanced eXtensible Interface [19] (AXI) is a high-performance, flexible bus protocol commonly used in FPGA and SoC designs for connecting components like processors, memory, and peripherals. It supports high-bandwidth data transfers with features such as burst transactions, out-of-order completion, and separate read/write channels.

5.6 KR260

The Kria KR260 [20] is a development board for the K26 System on Module, which contains a Zynq UltraScale+ MPSoC built for robotics and industrial applications. For this thesis, the Kria will be used to test and benchmark any hardware implementations.

5.6.1 Zynq Ultrascale+

The Zynq UltraScale+ [21] family of System on a Chip devices is a combination of FPGA fabric and conventional Central Processing Cores on a singular chip. This delivers both software programmability alongside hardware configurability. The device also features connectivity options such as Gigabit Ethernet, USB, PCIe, memory interfacing, and much more for peripheral connectivity. The device exists in multiple configurations featuring application-specific devices such as video codecs, but mainly consists of four ARM Cortex-A53 alongside two ARM Cortex-R5F suitable for real-time operating systems.

6
FINN

FINN [22, 23] is a machine learning FPGA framework by the Integrated Communications and AI Lab of AMD Research & Advanced Development. FINN can be used for quantization-aware training of neural network models using Brevitas, and a complete toolchain for exploration and implementation of Quantized Neural Networks (QNN) on FPGAs. The tool is limited to generating dataflow architectures. FINN can produce IP blocks for any Xilinx/AMD FPGA, but only generate automatic system integration for PYNQ-supported boards.

6.1 RTL and HLS Primitives

Computational units can be implemented either in RTL or HLS. A set of hardware primitives is made available as part of these two libraries; a user needs to ensure all components of the design have a matching entry in either library.

HLSlib

FINN was initially developed with the intention of using mainly HLS to implement all components; as such, most modules have HLS support.

RTLlib

Due to performance and flexibility reasons, some modules have been implemented in RTL; however, not all modules have both HLS and RTL versions.

6.2 FINN-ONNX

An extension to the ONNX format is used as an Intermediate Representation (IR) that the tool operates on. In addition to the graph representation, FINN-ONNX stores information about the data types, custom node support for implementing non-standard operations, parallelization metadata to be used for hardware generation with folding and streaming configurations, and support for Brevitas quantization schemes.

6.3 Analysis Pass

A cornerstone of how FINN operates is the analysis pass. Functionally, the graph is traversed, and some specified data, relationship, or other information and metadata about it is collected and reported as a directory.

6.4 Transformations

Another cornerstone is the transformation, typically guided by analysis passes, the graph structure is traversed and modified, returning a changed model. Custom transformations and analysis passes are how FINN is used to go from a machine learning model to hardware, incrementally transforming the model to a graph consisting of only components that can be implemented as an RTL or HLS library entry.

6.5 Stitched IP Design

Each node in the graph structure maps to a hardware unit, implemented as an IP. All IPs are then stitched together with datawidth converters and FIFOs to create the entire pipeline from input to output.

6.6 Tensor Dimension

A consideration when implementing models in hardware is that the tensor dimensions need to be handled according to how hardware primitives are designed. Typically, the following issue needs to be addressed.

6.6.1 NHWC & NCHW Format

Brevitas operated on tensors in the same way as PyTorch, in NCHW format. However, for hardware implementation reasons, FINN uses NHWC format. Where N-Batch, H-Height, W-Width, C-Channels. Easiest visualized through an image, where N corresponds to each image, H to the rows, W to the columns, and C to each RGB value.

6.7 Thresholding

Thresholding in FINN is a function that compares the input value to a set of fixed thresholds, where the output is the binary encoding of the index of the largest threshold still smaller than the input. Thresholding is defined as follows:

$$T(x,t) = \begin{cases} 0 & \text{for } x < t_0 \\ 1 & \text{for } t_0 < x \leq t_1 \\ \vdots & \vdots \\ n-1 & \text{for } t_{n-2} < x \leq t_{n-1} \\ n & \text{for } t_{n-1} < x \end{cases} \tag{6.1}$$

6.8 Streamlining

Streamlining is a transformation that simplifies the graph by combining nodes and operations together, removing redundancy and reducing complexity. The main purpose is to remove all floating-point operations by embedding them into the weights or thresholding functions.

6.9 Reordering of Operations

Some operations can benefit from being reordered, mainly to collapse redundant transposes, but also some commutative properties can be exploited.

6.10 Modeling Nodes

The following section serves to give some brief information about some nodes used in this work that make up the Brevitas model, but are not suitable for hardware implementation.

Global_In and Global_Out: The specified input and output of the model.

Quant: Quantization node performing a specific quantization.

Conv: A node representing a convolution operation.

ReLU: A node performing the Rectified Linear Unit function.

MaxPool: A pooling node performing maxpool.

MatMul: Matrix multiplication node.

BatchNormalization: A normalization operation on the batch.

Mul: Standard multiplication with a scalar.

Add: Standard addition with a scalar.

Div: Standard division with a scalar.

Reshape: A matrix or activation reshape function, typically used to format the data appropriately for the following node.

6.11 Intermediate Transformation Nodes

The following nodes are used as the model is successively transformed towards being able to be mapped to hardware units. Mostly, these transformations do not change the intermediate outputs, but some introduce some quantization.

Im2Col: A custom operator performing input reformatting to matrix form in order to perform lowering of convolutions to matrix multiplication.

MultiThreshold: The node representing the threshold function acting on each channel.

MaxPoolNHWC: Maxpool, but a custom implementation for inputs in NHWC format.

TopK: A classification node inserted at the end of the model to output a classification vector instead of probabilities.

6.12 Primitives or Computational Units

This section covers the hardware building blocks that are the intended result of the transformations and what is synthesized on the FPGA as RTL.

FMPadding: A feature map padding unit that applies padding to the channels as they flow through the architecture.

ConvolutionInputGenerator: A streaming component that reformats and prepares streams for the way convolutional layers are implemented in hardware by the following node.

MatrixVectorActivationUnit: The central building block is the MVAU, performing the lowered convolution by streamed matrix vector multiplication, followed by an activation unit that implements the threshold function.

StreamingMaxPool: The maxpool operation in hardware.

LabelSelect: Implements the TopK node from above in hardware, extracting the index of the maximum value as the predicted class label.

6.13 Specialization

Finally, once the graph consists of only library nodes, the specified version to be used is selected through a specialization transformation. Note that some nodes only have an HLS or RTL variant, while some have both.

6.14 Folding

Folding of each computational unit can be set by configuration of the following parameters for each computational unit, where PE is the parallelization over outputs and SIMD is the parallelization over inputs.

$$\text{PE Folding Factor} = \frac{\text{Total Number of Output Channels}}{\text{Number of PEs}}$$

$$\text{SIMD Folding Factor} = \frac{\text{Total Number of Input Channels}}{\text{Number of SIMD Lanes}}$$

$$\text{Total Folding} = \text{PE Folding Factor} \times \text{SIMD Folding Factor}$$

There are constraints on the values of the folding parameters for each computational unit. Below, MH is the number of rows, and MW is the width of the input matrix of the lowered convolution.

Table 6.1: *Folding parameter constraints*

Computational Unit	**Parameter**	**Constraint**
FMPadding	SIMD	in_channels % SIMD == 0
ConvolutionInputGenerator	SIMD	in_channels % SIMD == 0
MatrixVectorActivation	PE	MH % PE == 0
MatrixVectorActivation	SIMD	MW % SIMD == 0
StreamingMaxpool	PE	out_channels % PE == 0
Labelselect	PE	num_labels % PE == 0

6.15 Heterogeneous Structure

The benefit of implementing a full pipeline is that hardware resources can be allocated where they are most needed. Folding can be set such that the pipeline is balanced, and utilization is maximized.

6.16 Driver

If used as an accelerator paired with the PS, a driver in Python for PYNQ can be created that packs the data into memory in the appropriate format, given the generated hardware, which is read by the DMA. Similarly, unpacking of the output needs to be performed.

7

System Design and Implementation

7.1 Signal Generation

The initial stage of the thesis consisted of the dataset generation to be used for machine learning. The following sections describe that process.

7.1.1 Signal Modulation Classes

Signals were generated for the following 12 modulation schemes.

Table 7.1: *Modulation schemes*

Modulation Type	**Schemes**
PAM	4PAM, 8PAM
QAM	8QAM, 16QAM, 64QAM
APSK	16APSK, 32APSK, 64APSK
PSK	BPSK, QPSK, 8PSK
Differential PSK	$\pi/4$-DQPSK

7.1.2 Oversampling

In this application, the symbol transmission rate is not known, which necessitates varying the number of samples per symbol (SPS) during signal generation. To account for this uncertainty, an oversampling factor is selected randomly between a lower bound of 3 and an upper bound of 9 for each signal.

To ensure balanced representation across different sampling rates, the total number of signals is chosen as a multiple of the number of SPS values. This guarantees an equal number of signals for each oversampling factor, allowing for consistent dataset composition.

7.1.3 Dataset Generation

The signal generation process was performed in MATLAB, with the output as 32-bit floating-point values for both the I and Q components. Signals were generated with an SNR range between 10 and 30 dB. The maximum ENOB is therefore slightly less than 5 bits. With this as motivation, an input bitwidth of 8 was thought to be sufficient to not lose much accuracy. For training, 2100 signals per SNR, per modulation type were generated. For validation and test, 420 signals per SNR, per modulation type.

QDQ Dataset

The machine learning process operates on normalized [-1, 1] floating-point input. As such, the dataset needs to either be provided in this format or, optionally, it may be performed on the fly as the data is used in the training, which is quite a large overhead. The following normalization, rounding, and scaling were performed to generate the QDQ dataset.

```
round(127*signal/max(abs(signal)))/128
```

Integer Dataset

In addition to the QDQ dataset, scaling was performed to the range [-127, 127], and then saved as an 8-bit signed int. The finished hardware accelerator should input 8 bits, which this is for.

```
int8(round(127*signal/max(abs(signal))))
```

7.2 Brevitas Model

Two separate models were constructed, one that takes 1024 I/Q samples as input, and one that takes 512. Both models are quite similar, and were trained with the configurations specified in the following sections. The Adaptive Moment Estimation optimizer and cross-entropy loss function were used with Brevitas layer primitives covered in the theory chapters. Note that the 512 sample model was trained for longer, as it seemed to be converging at a slower rate compared to the 1024 version.

7.2.1 1024 Model Training

The parameters used for training, the input, and the weight and activation bitwidths are listed below in Table 7.2. All layers that make up the 1024 sample model are shown in Table 7.8, with the specific configurations of the convolution layers, pooling, and fully connected operations shown in Table 7.4, Table 7.5, and Table 7.6. respectively.

Table 7.3: *The 1024 sample model*

Layer	Operation
0	QuantConv1d
0	BatchNorm1d
0	QuantReLU
0	MaxPool1d
1	QuantConv1d
1	BatchNorm1d
1	QuantReLU
1	MaxPool1d
2	QuantConv1d
2	BatchNorm1d
2	QuantReLU
2	MaxPool1d
3	QuantConv1d
3	BatchNorm1d
3	QuantReLU
3	MaxPool1d
4	QuantConv1d
4	BatchNorm1d
4	QuantReLU
4	MaxPool1d
-	Reshape
5	QuantLinear
5	QuantReLU
6	QuantLinear
6	QuantReLU
7	QuantLinear
7	QuantReLU
8	QuantLinear

Table 7.2: *Training configuration parameters*

Parameter	Value
Iterations	4000
Learning rate	0.0005
Batch size	64
Input bitwidth	8
Weight bitwidth	8
Activation bitwidth	4

Table 7.4: *Convolutional layer configuration*

Layer	In Channels	Out Channels	Kernel	Padding
0	2	32	13	6
1	32	64	13	6
2	64	64	13	6
3	64	64	13	6
4	64	64	13	6

Table 7.5: *Pooling configuration for the convolution layers*

Layer	Type	Pooling Kernel	Pooling stride
0	MaxPool	4	4
1	MaxPool	2	2
2	MaxPool	2	2
3	MaxPool	2	2
4	MaxPool	2	2

Table 7.6: *Fully connected layer configuration*

Layer	Input Neurons	Output Neurons
5	1024	256
6	256	128
7	128	64
8	64	12

7.2.2 512 Model Training

The parameters used for training, the input, and the weight and activation bitwidths are listed below in Table 7.7. All layers that make up the 512 sample model are shown in Table 7.8, with the specific configurations of the convolution layers, pooling, and fully connected operations shown in Table 7.9, Table 7.10, and Table 7.11. respectively.

Table 7.8: *The 512 sample model*

Layer	Operation
0	QuantConv1d
0	BatchNorm1d
0	QuantReLU
0	MaxPool1d
1	QuantConv1d
1	BatchNorm1d
1	QuantReLU
1	MaxPool1d
2	QuantConv1d
2	BatchNorm1d
2	QuantReLU
2	MaxPool1d
3	QuantConv1d
3	BatchNorm1d
3	QuantReLU
3	MaxPool1d
-	Reshape
4	QuantLinear
4	QuantReLU
5	QuantLinear
5	QuantReLU
6	QuantLinear
6	QuantReLU
7	QuantLinear

Table 7.7: *Training configuration parameters*

Parameter	Value
Iterations	8000
Learning rate	0.0005
Batch size	64
Input bitwidth	8
Weight bitwidth	8
Activation bitwidth	4

Table 7.9: *Convolutional layer configuration*

Layer	In Channels	Out Channels	Kernel	Padding
0	2	32	13	6
1	32	64	13	6
2	64	64	13	6
3	64	64	13	6

Table 7.10: *Pooling configuration for the convolution layers*

Layer	Type	Pooling Kernel	Pooling stride
0	MaxPool	4	4
1	MaxPool	2	2
2	MaxPool	2	2
3	MaxPool	2	2

Table 7.11: *Fully connected layer configuration*

Layer	Input Neurons	Output Neurons
4	1024	256
5	256	128
6	128	64
7	64	12

7.3 FINN Model

The following section will describe the implementation of the hardware.

7.3.1 Model Import

The Brevitas model was imported and converted to the FINN-ONNX format as the first part of the implementation.

7.3.2 Division Node Insertion and Labelselect

A separate model performing a division by 128 to rescale the integer input to the range [-1, 1] was created and inserted at the start of the machine learning model through network surgery. This is needed to match the expected input range from the training. In addition, a TopK node was inserted at the end of the model to handle the label output. The Netron visualization can be viewed below in Figure 7.1.

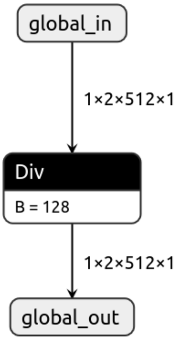

Figure 7.1: *Preprocessing model.*

7.3.3 Model Transformations

A series of transformations was imposed on the model to incrementally transform it to be comprised of hardware building blocks. It should be noted that the division node is absorbed into the weights by the streamlining transformation.

Specialize

The RTL or HLS variants for each computational unit were selected. The FMPadding and ConvolutionInputGenerator were implemented as RTL, while the MVAU, LabelSelect, and StreamingMaxPool were implemented with HLS.

Partition

At this stage, the model is comprised of a single transpose to modify the input from NCHW to NHWC, and the rest are hardware synthesizable compute units. Everything except the transpose is moved to a dataflow partition subgraph for further configuration.

Folding

The folding parameters of the dataflow graph were configured based on the desired parallelization. Analysis passes may be performed on the model to get estimates on performance and hardware required to help guide the design process; these aspects are covered in more detail in Section 7.4. The SIMD of FMPadding and ConvolutionInputGenerator is set to the number of input channels.

7.3.4 Hardware Generation

The hardware is generated through templating of each computational unit with the set configuration, followed by insertion of datawidth converters and FIFOs between each unit to introduce some clock cycle flexibility for data shuffling. In addition, two DMAs were inserted to handle the input and output of the system.

7.3.5 Drivers

Python drivers were created as part of the build process to package the input buffers and to unpack the outputs.

7.4 Accelerator Skews

To investigate how performance and hardware utilization scale with parallelization parameters, three skew configurations of the 512-model accelerator were implemented and deployed on the development board. The first one, that will be referred to as Max Folded, has all folding configurations set to the maximum, to serve as a minimal implementation. The second implementation, Some Folding, has a medium amount of parallelization. While the third, referred to as Tuned, has close to the maximum possible parallelization possible on the selected hardware. The following section presents the estimate reports for the implementation of each version.

7.4.1 Estimates for Maximally Folded Accelerator

For the maximally folded accelerator, the estimated clock cycles per computational unit are shown in Figure 7.2 and in Table 7.12. The estimated LUT usage, BRAM, and DSP are shown in Figure 7.3 and Table 7.13.

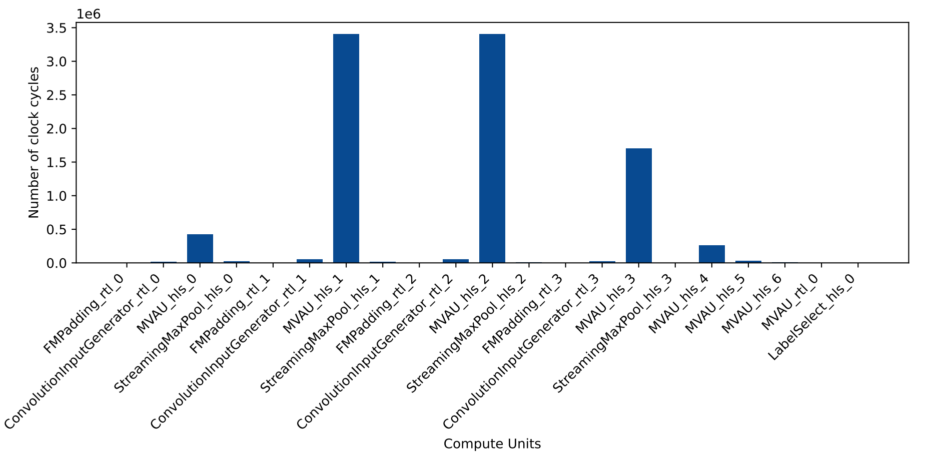

Figure 7.2: *Estimated required cycles for computation per computational unit.*

Table 7.12: *Estimated number of cycles per computational unit*

Compute Unit	Estimated Number of Cycles	% of Total Cycles
FMPadding_rtl_0	524	0.01%
ConvolutionInputGenerator_rtl_0	13314	0.14%
MVAU_hls_0	425984	4.51%
StreamingMaxPool_hls_0	20480	0.22%
FMPadding_rtl_1	140	0.00%
ConvolutionInputGenerator_rtl_1	53250	0.56%
MVAU_hls_1	3407872	36.11%
StreamingMaxPool_hls_1	12288	0.13%
FMPadding_rtl_2	76	0.00%
ConvolutionInputGenerator_rtl_2	53250	0.56%
MVAU_hls_2	3407872	36.11%
StreamingMaxPool_hls_2	6144	0.07%
FMPadding_rtl_3	44	0.00%
ConvolutionInputGenerator_rtl_3	26626	0.28%
MVAU_hls_3	1703936	18.05%
StreamingMaxPool_hls_3	3072	0.03%
MVAU_hls_4	262144	2.78%
MVAU_hls_5	32768	0.35%
MVAU_hls_6	8192	0.09%
MVAU_rtl_0	768	0.01%
LabelSelect_hls_0	12	0.00%
Total	**9 439 946**	

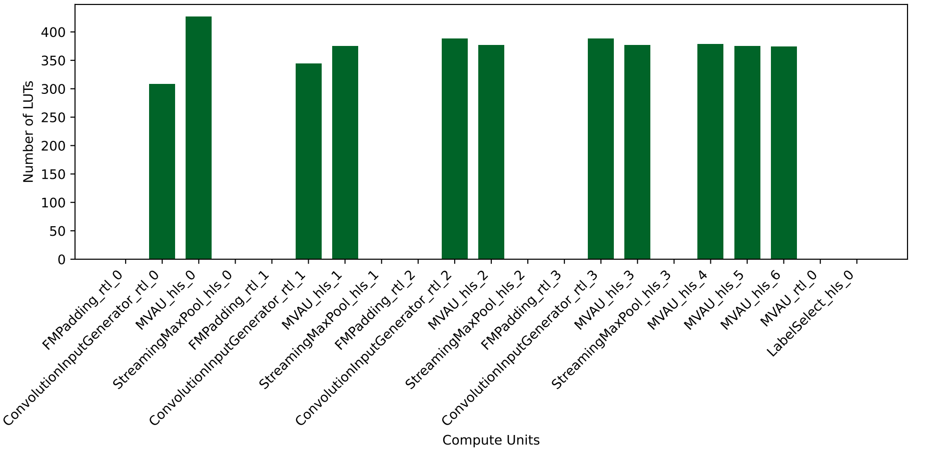

Figure 7.3: *Estimated LUT usage per computational unit.*

Table 7.13: *Resource estimate per computational unit*

Compute Unit	BRAM_18K	BRAM_efficiency	LUT	DSP
FMPadding_rtl_0	0	-	0	0
ConvolutionInputGenerator_rtl_0	0	-	308	0
MVAU_hls_0	1	0.36	427	0
StreamingMaxPool_hls_0	0	-	0	0
FMPadding_rtl_1	0	-	0	0
ConvolutionInputGenerator_rtl_1	0	-	344	0
MVAU_hls_1	13	0.89	375	0
StreamingMaxPool_hls_1	0	-	0	0
FMPadding_rtl_2	0	-	0	0
ConvolutionInputGenerator_rtl_2	0	-	388	0
MVAU_hls_2	26	0.89	377	0
StreamingMaxPool_hls_2	0	-	0	0
FMPadding_rtl_3	0	-	0	0
ConvolutionInputGenerator_rtl_3	0	-	388	0
MVAU_hls_3	26	0.89	377	0
StreamingMaxPool_hls_3	0	-	0	0
MVAU_hls_4	128	0.89	378	0
MVAU_hls_5	16	0.89	375	0
MVAU_hls_6	4	0.89	374	0
MVAU_rtl_0	1	0.33	0	1
LabelSelect_hls_0	0	-	0	0
Total	215	-	4111	1

7.4.2 Estimates for Somewhat Folded Accelerator

For the somewhat folded accelerator, the estimated clock cycles per computational unit are shown in Figure 7.4 and in Table 7.14. The estimated LUT usage, BRAM, and DSP are shown in Figure 7.5 and Table 7.15.

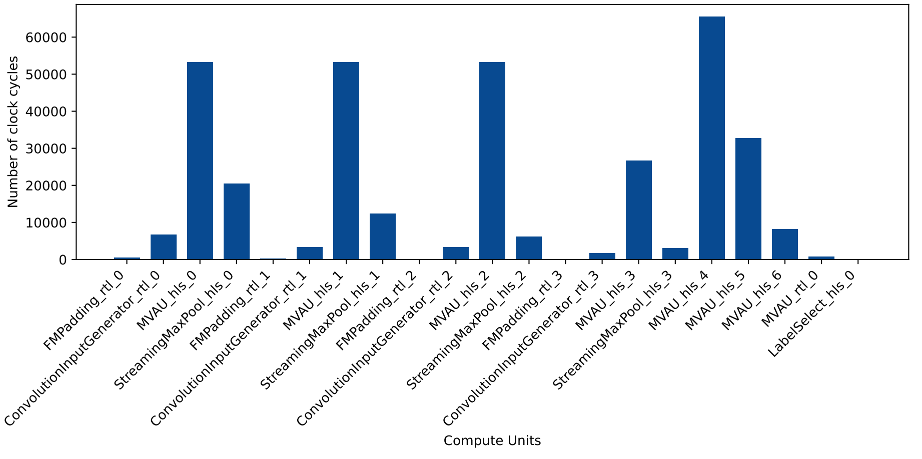

Figure 7.4: *Estimated required cycles for computation per computational unit.*

Table 7.14: *Estimated number of cycles per computational unit*

Compute Unit	Estimated Number of Cycles	% of Total Cycles
FMPadding_rtl_0	524	0.15%
ConvolutionInputGenerator_rtl_0	6658	1.89%
MVAU_hls_0	53248	15.15%
StreamingMaxPool_hls_0	20480	5.83%
FMPadding_rtl_1	140	0.04%
ConvolutionInputGenerator_rtl_1	3330	0.95%
MVAU_hls_1	53248	15.15%
StreamingMaxPool_hls_1	12288	3.50%
FMPadding_rtl_2	76	0.02%
ConvolutionInputGenerator_rtl_2	3330	0.95%
MVAU_hls_2	53248	15.15%
StreamingMaxPool_hls_2	6144	1.75%
FMPadding_rtl_3	44	0.01%
ConvolutionInputGenerator_rtl_3	1666	0.47%
MVAU_hls_3	26624	7.58%
StreamingMaxPool_hls_3	3072	0.87%
MVAU_hls_4	65536	18.65%
MVAU_hls_5	32768	9.33%
MVAU_hls_6	8192	2.33%
MVAU_rtl_0	768	0.22%
LabelSelect_hls_0	12	0.00%
Total	**351 712**	

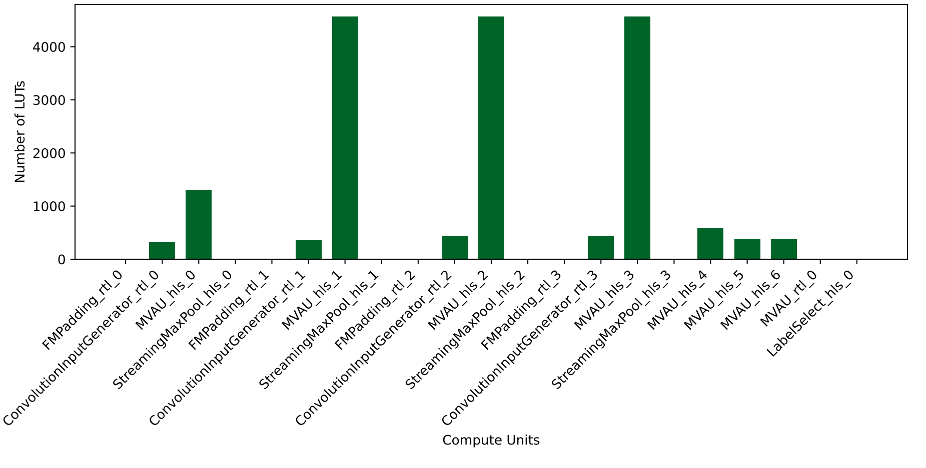

Figure 7.5: *Estimated LUT usage per computational unit.*

Table 7.15: *Resource estimate per computational unit*

Compute Unit	BRAM_18K	BRAM_efficiency	LUT	DSP
FMPadding_rtl_0	0	-	0	0
ConvolutionInputGenerator_rtl_0	0	-	316	0
MVAU_hls_0	2	0.18	1303	0
StreamingMaxPool_hls_0	0	-	0	0
FMPadding_rtl_1	0	-	0	0
ConvolutionInputGenerator_rtl_1	0	-	364	0
MVAU_hls_1	15	0.77	4563	0
StreamingMaxPool_hls_1	0	-	0	0
FMPadding_rtl_2	0	-	0	0
ConvolutionInputGenerator_rtl_2	0	-	428	0
MVAU_hls_2	29	0.80	4568	0
StreamingMaxPool_hls_2	0	-	0	0
FMPadding_rtl_3	0	-	0	0
ConvolutionInputGenerator_rtl_3	0	-	428	0
MVAU_hls_3	29	0.80	4568	0
StreamingMaxPool_hls_3	0	-	0	0
MVAU_hls_4	128	0.89	576	0
MVAU_hls_5	16	0.89	375	0
MVAU_hls_6	4	0.89	374	0
MVAU_rtl_0	1	0.33	0	1
LabelSelect_hls_0	0	-	0	0
Total	224	-	17863	1

7.4.3 Estimates for Tuned Accelerator

For the tuned accelerator, the estimated clock cycles per computational unit are shown in a Figure 7.6 and in Table 7.16. The estimated LUT usage, BRAM, and DSP are shown in Figure 7.7 and Table 7.17.

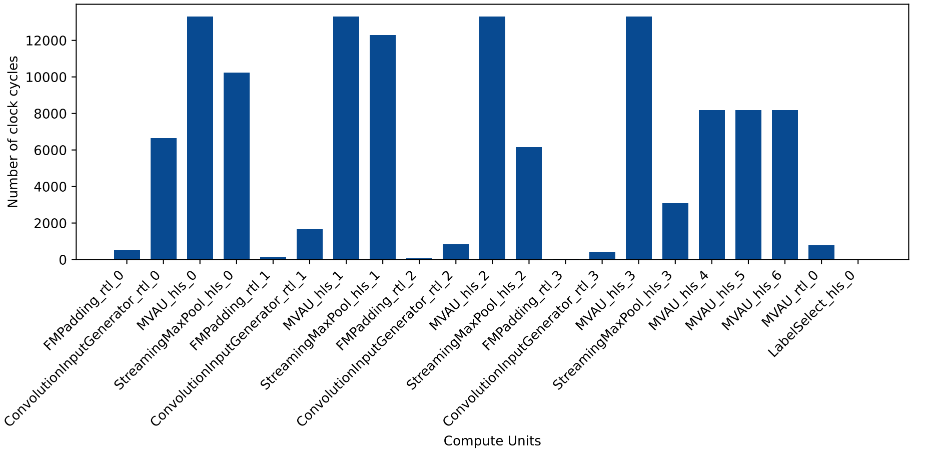

Figure 7.6: *Estimated required cycles for computation per computational unit.*

Table 7.16: *Estimated number of cycles per computational unit*

Compute Unit	Estimated Number of Cycles	% of Total Cycles
FMPadding_rtl_0	524	0.43%
ConvolutionInputGenerator_rtl_0	6658	5.52%
MVAU_hls_0	13312	11.03%
StreamingMaxPool_hls_0	10240	8.48%
FMPadding_rtl_1	140	0.12%
ConvolutionInputGenerator_rtl_1	1666	1.38%
MVAU_hls_1	13312	11.03%
StreamingMaxPool_hls_1	12288	10.18%
FMPadding_rtl_2	76	0.06%
ConvolutionInputGenerator_rtl_2	834	0.69%
MVAU_hls_2	13312	11.03%
StreamingMaxPool_hls_2	6144	5.09%
FMPadding_rtl_3	44	0.04%
ConvolutionInputGenerator_rtl_3	418	0.35%
MVAU_hls_3	13312	11.03%
StreamingMaxPool_hls_3	3072	2.54%
MVAU_hls_4	8192	6.79%
MVAU_hls_5	8192	6.79%
MVAU_hls_6	8192	6.79%
MVAU_rtl_0	768	0.64%
LabelSelect_hls_0	12	0.01%
Total	**121 738**	

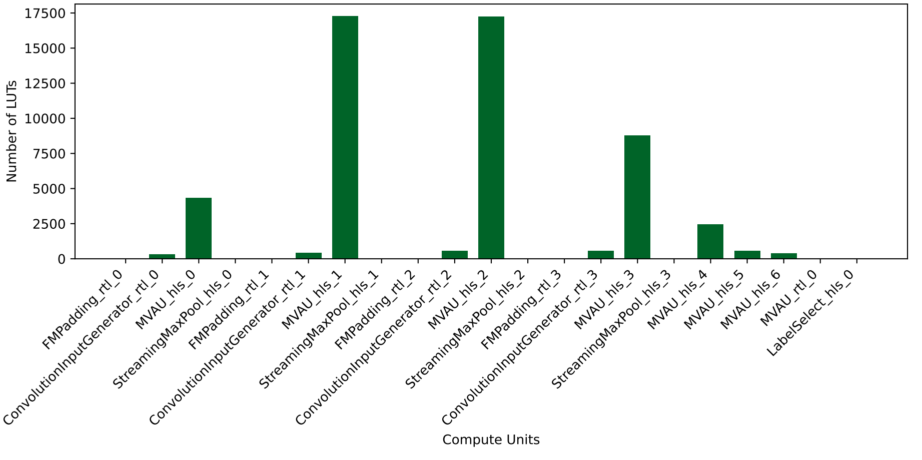

Figure 7.7: *Estimated LUT usage per computational unit.*

Table 7.17: Resource estimate per computational unit

Compute Unit	BRAM_18K	BRAM_efficiency	LUT	DSP
FMPadding_rtl_0	0	-	0	0
ConvolutionInputGenerator_rtl_0	0	-	316	0
MVAU_hls_0	8	0.05	4312	0
StreamingMaxPool_hls_0	0	-	0	0
FMPadding_rtl_1	0	-	0	0
ConvolutionInputGenerator_rtl_1	0	-	428	0
MVAU_hls_1	57	0.20	17275	0
StreamingMaxPool_hls_1	0	-	0	0
FMPadding_rtl_2	0	-	0	0
ConvolutionInputGenerator_rtl_2	0	-	556	0
MVAU_hls_2	57	0.41	17240	0
StreamingMaxPool_hls_2	0	-	0	0
FMPadding_rtl_3	0	-	0	0
ConvolutionInputGenerator_rtl_3	0	-	556	0
MVAU_hls_3	29	0.80	8770	0
StreamingMaxPool_hls_3	0	-	0	0
MVAU_hls_4	120	0.95	2436	0
MVAU_hls_5	16	0.89	573	0
MVAU_hls_6	4	0.89	374	0
MVAU_rtl_0	1	0.33	0	1
LabelSelect_hls_0	0	-	0	0
Total	292	-	52836	1

7.5 FINN Hardware Simulation Model Classification Results

The following figures show the simulated classification accuracy at 30 SNR from the models once all transformations have been performed, and the models consist of only primitives to be synthesized. The classification results of the simulated 1024 model are shown in Figure 7.8. The classification results of the simulated 512 model are shown in Figure 7.9.

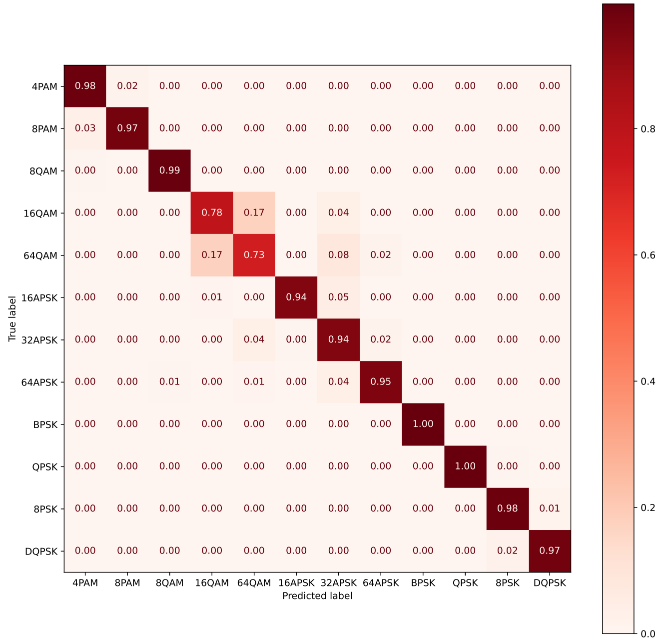

Figure 7.8: *Confusion matrix of 1024 I/Q sample model in C++ simulation.*

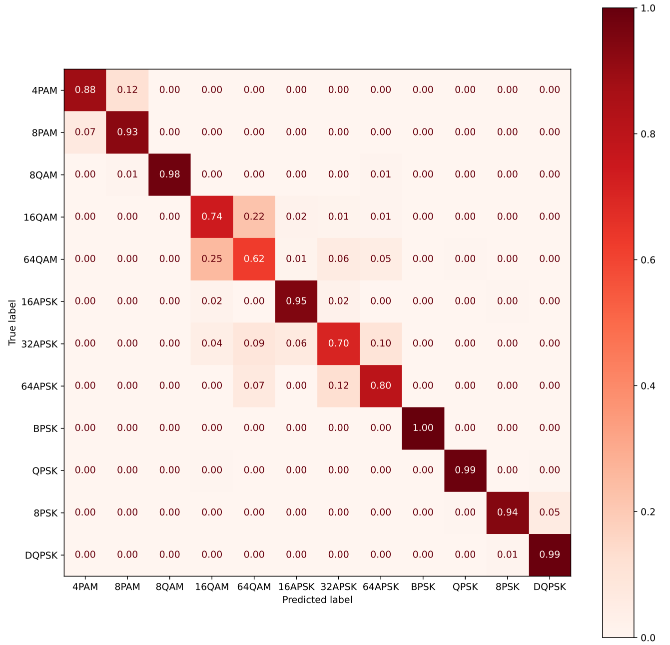

Figure 7.9: *Confusion matrix of 512 I/Q sample model in C++ simulation.*

8

Results

This chapter presents the results for both the 1024 and the 512 I/Q sample models, their hardware-adapted results, and the accelerator performance in terms of classification accuracy across the selected range of SNR. The hardware performance in terms of latency, throughput, as well as the utilization of hardware resources on the chip, along with power consumption, is presented only for the 512-sample accelerator.

8.1 QAT Brevitas Model Results

The following section presents the model classification results for the Brevitas QAT models.

8.1.1 Classification Accuracy 1024 I/Q Model

Below in Figure 8.2 are the confusion matrix results for the 1024 I/Q sample model for both 30 dB SNR and the classification accuracy for the range 10-30 dB SNR in Figure 8.2.

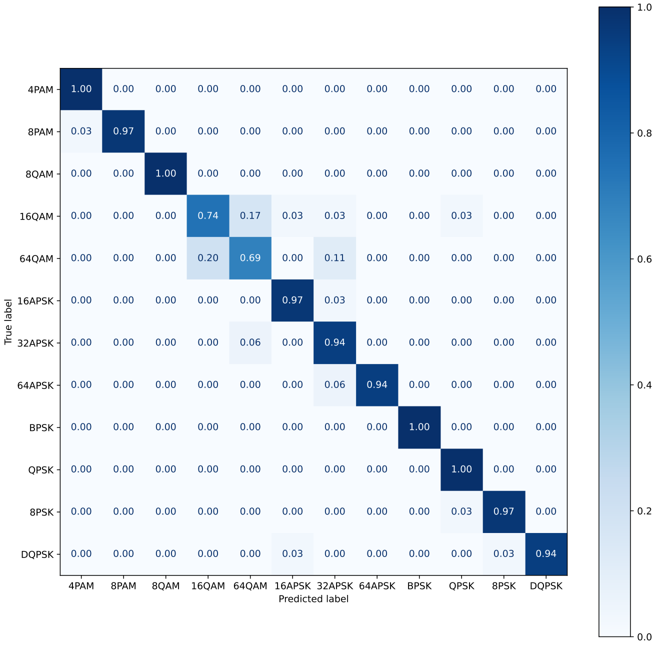

Figure 8.1: *Confusion matrix of 8-bit weight, 4-bit activation, 1024 I/Q sample model for 30 dB SNR.*

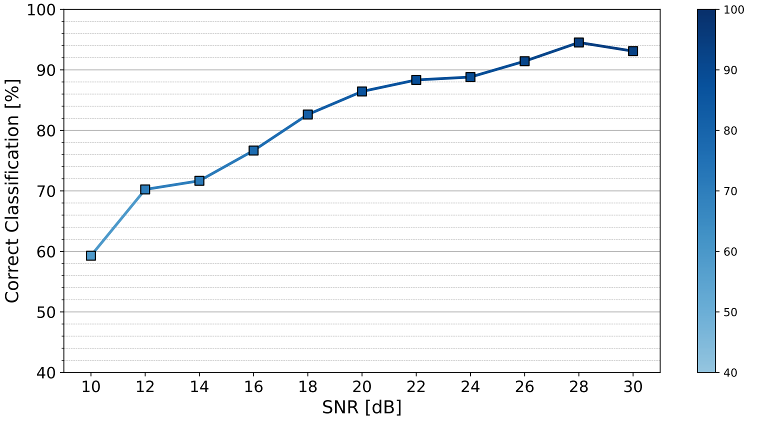

Figure 8.2: *Confusion matrix of 8-bit weight, 4-bit activation, 1024 I/Q sample model, for SNR levels between 10 and 30 dB.*

8.1.2 Classification Accuracy 512-I/Q Model

Below in Figure 8.3 are the confusion matrix results for the 512 I/Q sample model for both 30 dB SNR and the classification accuracy for the range 10-30 dB SNR in

Figure 8.4.

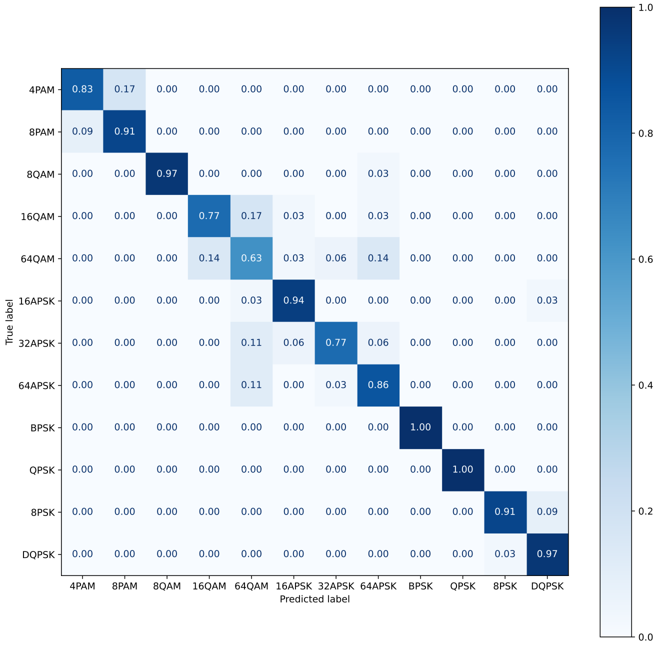

Figure 8.3: *Confusion matrix of 8-bit weight, 4-bit activation, 512 I/Q sample model for 30 dB SNR.*

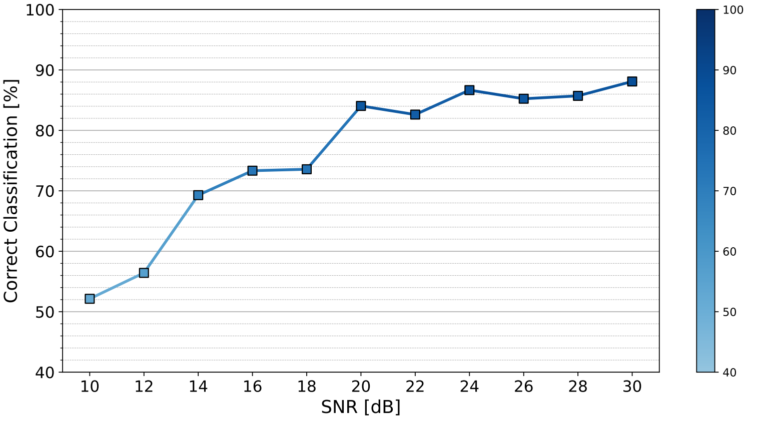

Figure 8.4: *Confusion matrix of 8-bit weight, 4-bit activation, 512 I/Q sample model, for SNR levels between 10 and 30 dB.*

8.2 Hardware Accelerator Classification Results

These are the results for the classification accuracy of the accelerators running on the hardware.

8.2.1 Classification Accuracy 1024-I/Q Model

The hardware for the 1024 model was generated, but observed classification results on the physical hardware were not consistent with simulation results that can be observed in Section 7.5. The cause of the problem was unfortunately not able to be found.

8.2.2 Classification Accuracy 512 I/Q Model

The confusion matrices for the SNR levels of 30 dB in Figure 8.5, 20 dB in Figure 8.6, and 10 dB in Figure 8.7, along with the classification accuracy as a function of SNR for the 512 I/Q hardware accelerator in Figure 8.8, are presented below.

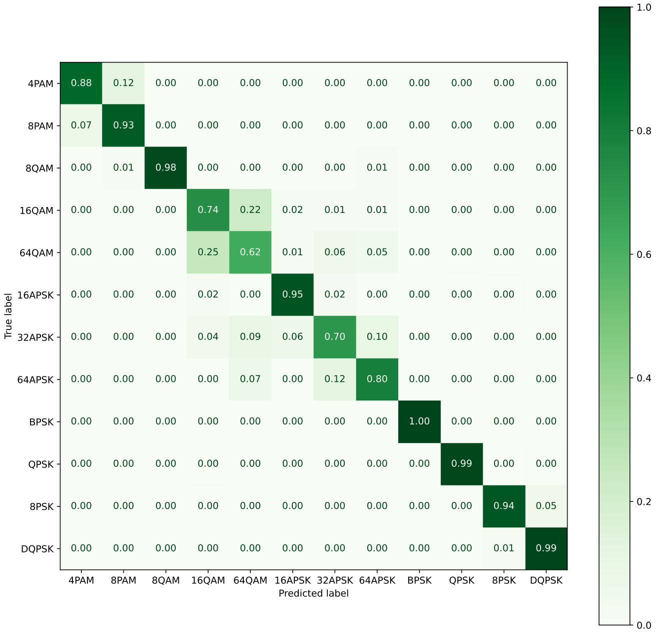

Figure 8.5: *Confusion matrix of 512 I/Q accelerator for 30 dB SNR.*

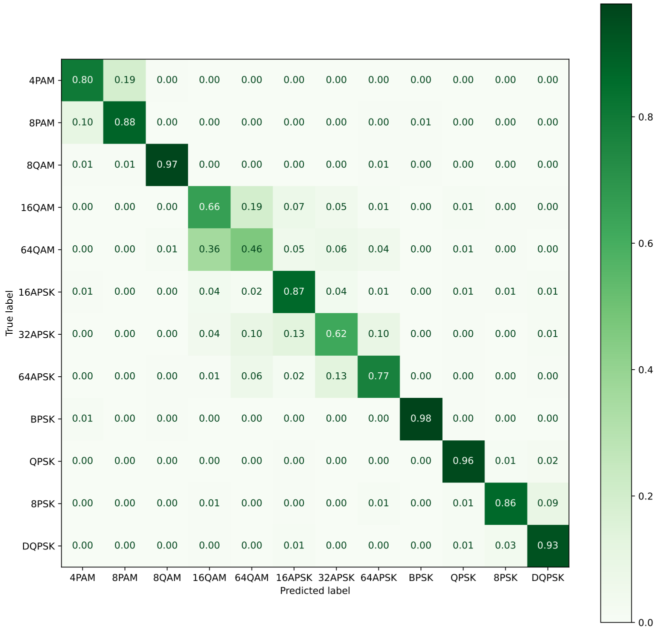

__Figure 8.6:__ Confusion matrix of 512 I/Q accelerator for 20 dB SNR.

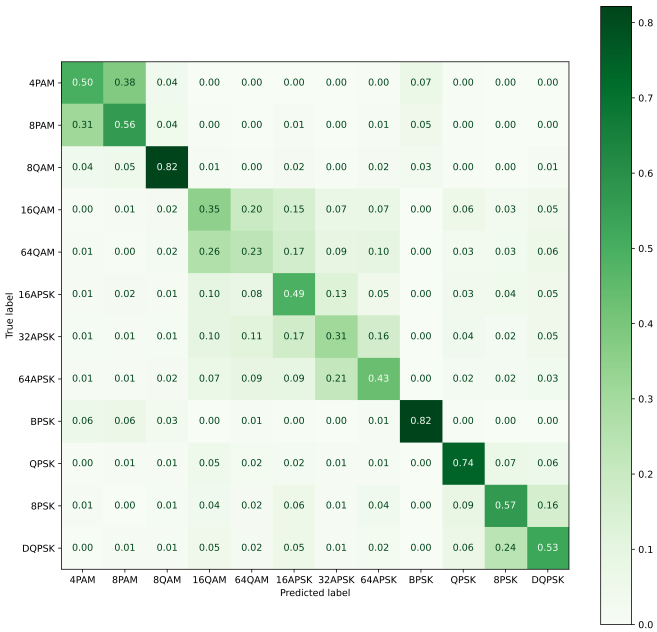

__Figure 8.7:__ Confusion matrix of 512 I/Q accelerator for 10 dB SNR.

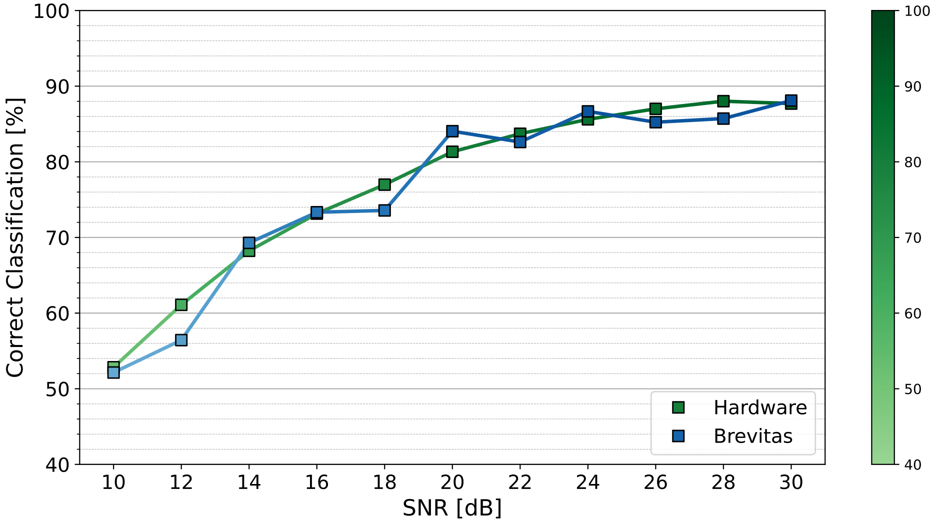

Figure 8.8: *Classification accuracy over all SNR levels for 512 I/Q sample model.*

8.3 512-I/Q Accelerator Performance

The following sections cover the hardware design and performance metrics for three skews of the implemented 512 I/Q sample accelerators. An overview of the performance metrics is shown below in Table 8.1.

Table 8.1: *Overview of performance metrics for the accelerator skews*

Metric	Max Folded	Some Folding	Tuned
Latency (ms)	44.81	1.77	0.485
Throughput (signals/s)	28.04	1298.82	6305.60
Initiation Interval (µs)	35669.1	769.9	158.6
LUTs	22,438	26,412	38,272
BRAM	110.5	107.7	141
DSP	8	231	747
PL Power (W)	0.531	0.725	1.317
PL Power per Signal (µW)	18 937	558.2	208.9

8.4 Accelerator Max Folded

Folding parameters of FMPadding and ConvolutionInputGenerators are set based on the folding configuration of the compute units the data is fed into. For the max folded accelerator, all PE and SIMD that can be configured are set to 1.

Utilization

The utilization of the max folded accelerator is listed below in Table 8.2.

Table 8.2: *Utilization summary*

Resource	Used	Utilization (%)
CLB Logic		
CLB LUTs (total)	22,438	19.16
LUT as Logic	20,607	17.59
LUT as Memory	1,831	3.18
CLB Registers (total)	31,908	13.62
Flip-Flops	31,908	13.62
Latches	0	0.00
Block RAM		
Block RAM Tiles	110.5	76.74
RAMB36/FIFO	109	75.69
RAMB18	3	1.04
URAM	0	0.00
Arithmetic		
DSPs	8	0.64

Power Consumption

The power consumption of the max folded accelerator is listed below in Table 8.3.

Table 8.3: *Power summary for max folded 512 I/Q accelerator*

Parameter	Value	Unit
Total On-Chip Power	2.948	W
Dynamic Power	2.645	W
Static Power	0.303	W
PL Power Breakdown		
Clocks	0.047	W
CLB Logic	0.089	W
LUT as Logic	0.061	W
LUT as Dist. RAM	0.022	W
LUT as Shift Reg	0.003	W
Registers	0.003	W
CARRY8	<0.001	W
Signals	0.068	W
Block RAM	0.020	W
DSPs	0.004	W
PL Static Power	0.303	W
PL Total Power	**0.531**	**W**
Processing System (PS)		
PS8 (Dynamic)	2.417	W

Floorplan

The floor plan of the implemented max folded accelerator is listed below in Figure 8.9.

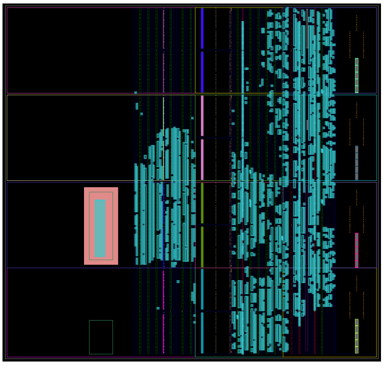

Figure 8.9: *FPGA floorplan of implemented max folded accelerator.*

8.5 Accelerator Some Folding

Folding parameters of FMPadding and ConvolutionInputGenerators are set based on the folding configuration of the compute units the data is fed into. Unless listed in Table 8.4 below, all unconstrained PE and SIMD are set to 1.

Table 8.4: *Folding configuration*

Compute Unit	PE	SIMD
MVAU_0	4	2
MVAU_1	4	16
MVAU_2	4	16
MVAU_3	4	16
MVAU_4	1	4
MVAU_5	1	1
MVAU_6	1	1
MVAU_7	1	1

Utilization

The utilization of the some folding accelerator is listed below in Table 8.2.

Table 8.5: *Utilization summary*

Resource	Used	Utilization (%)
CLB Logic		
CLB LUTs (total)	26,412	22.55
LUT as Logic	23,727	20.26
LUT as Memory	2,685	4.66
CLB Registers (total)	39,958	17.06
Flip-Flops	39,958	17.06
Latches	0	0.00
CARRY8	912	6.23
Block RAM		
Block RAM Tiles	107.5	74.65
RAMB36/FIFO	105	72.92
RAMB18	5	1.74
URAM	0	0.00
Arithmetic		
DSPs	231	18.51

Power Consumption

The power consumption of the some folding accelerator is listed below in Table 8.6.

Table 8.6: *Power summary for some folding 512 I/Q accelerator*

Parameter	Value	Unit
Total On-Chip Power	3.139	W
Dynamic Power	2.836	W
Static Power	0.304	W
PL Power Breakdown		
Clocks	0.061	W
CLB Logic	0.117	W
LUT as Logic	0.080	W
LUT as Dist. RAM	0.024	W
LUT as Shift Reg	0.005	W
Registers	0.005	W
CARRY8	0.003	W
Signals	0.111	W
Block RAM	0.039	W
DSPs	0.090	W
PL Static Power	0.304	W
PL Total Power	**0.725**	**W**
Processing System (PS)		
PS8 (Dynamic)	2.417	W

Floorplan

The floor plan of the implemented some folded accelerator is listed below in Figure 8.10.

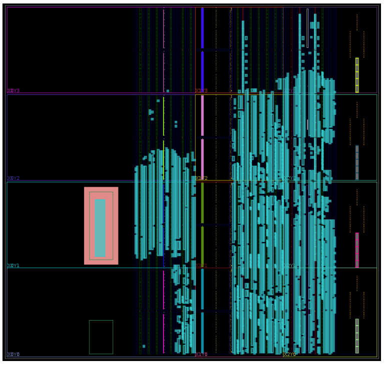

Figure 8.10: *FPGA floorplan of some folding accelerator.*

8.6 Accelerator Tuned

Folding parameters of FMPadding and ConvolutionInputGenerators are set based on the folding configuration of the compute units the data is fed into. Unless listed in Table 8.7 below, all unconstrained PE and SIMD are set to 1.

Table 8.7: *Folding configuration*

Compute Unit	PE	SIMD
MVAU_0	16	2
StreamingMaxPool_0	2	-
MVAU_1	8	32
MVAU_2	4	64
MVAU_3	2	64
MVAU_4	2	16
MVAU_5	1	4
MVAU_6	1	1
MVAU_7	1	1

Utilization

The utilization of the tuned accelerator is listed below in Table 8.8.

Table 8.8: *Utilization summary*

Resource	Used	Utilization (%)
CLB Logic		
CLB LUTs (total)	38,272	32.68
LUT as Logic	33,243	28.38
LUT as Memory	5,029	8.73
CLB Registers (total)	52,121	22.25
Flip-Flops	52,121	22.25
Latches	0	0.00
Block RAM		
Block RAM Tiles	141	97.92
RAMB36/FIFO	138	95.83
RAMB18	6	2.08
URAM	0	0.00
Arithmetic		
DSPs	747	59.86

Power Consumption

The power consumption of the tuned accelerator is listed below in Table 8.9.

Table 8.9: *Power summary for tuned 512 I/Q accelerator*

Parameter	**Value**	**Unit**
Total On-Chip Power	3.668	W
Dynamic Power	3.361	W
Static Power	0.307	W
PL Power Breakdown		
Clocks	0.076	W
CLB Logic	0.166	W
LUT as Logic	0.114	W
LUT as Dist. RAM	0.026	W
LUT as Shift Reg	0.012	W
Registers	0.007	W
CARRY8	0.006	W
Signals	0.242	W
Block RAM	0.097	W
DSPs	0.363	W
PL Static Power	0.307	W
PL Total Power	**1.317**	**W**
Processing System (PS)		
PS8 (Dynamic)	2.417	W

Floorplan

The floor plan of the implemented tuned accelerator is listed below in Figure 8.11.

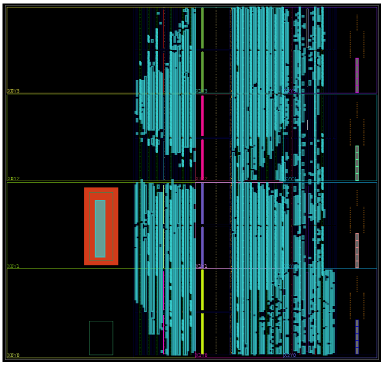

Figure 8.11: *FPGA floorplan of tuned accelerator.*

9

Discussion

This chapter contains some analysis on the work, some discussion of the results, and some ideas for improvement and areas that may be of interest for future work.

9.1 Implementation Issues

There were significant difficulties in setting up the development environment, running FINN, and configuring the development board. Some, if not most, of these issues arose due to the lab environment and networking having a restricted configuration. The documentation, and especially the explanation about how and why certain things work and interact, is somewhat lacking. At the initial stage of the thesis, the hope was to reuse existing models and data and simply implement hardware. However, it became clear that it was not an option, which increased the scope of work significantly.

9.2 Signal Generation

The raw modulation signal generation code was provided for the thesis and worked well. What could be improved is to increase the range of SNR levels generated, the number of modulation classes, as well as the number of signals to give an even more representative picture of how such a system might perform in practice. Reducing the oversampling factor would also likely be of interest.

9.3 Machine Learning

The machine learning aspects of the thesis have a lot of room for improvement, many of which are discussed in Subsection 9.7.1. Overall, the number of input neurons to the first layer of the fully connected layers should be excessively large, and could likely be reduced significantly. The hyperparameters for the training stage could be explored much further, along with many more optimizations. Around 30 iterations were spent revising the models as a part of this work, many of simply learning the basics.

9.4 Hardware

For the selected hardware platform, the performance in terms of throughput is likely somewhat close to the maximum achievable, as the generated pipeline is fairly balanced, and close to full memory port utilization is achieved. However, there are certainly some efficiency gains to be had in terms of power and utilization of other hardware resources.

The one thing that could significantly improve parallelization is the utilization of URAM. From what could be discerned, at least in the version of FINN that was used, it did not seem possible, but more effort could be spent on this area to mitigate the memory port bottleneck issue.

Estimates Reports

The reports that can be generated throughout the design phase on the expected hardware utilization and clock cycle requirements per computational unit have issues. The report for the LUT usage seems to be overestimated, but roughly near the actual value. The number of DSPs required is clearly not working at all, but the memory requirement and the relative clock cycle estimates have proven to work quite well, and can be used to balance the pipeline effectively. Overall, the process does seem to require some trial and error to arrive at a final design.

Folding Scaling Issues

As part of the model-hardware co-design process, the number of channels and the dimensions of matrices in the model impact and constrain the values that the folding parameters PE and SIMD can take. With how the computational units are implemented, according to the documentation, increasing parallelization through an increase in SIMD is more efficient, which should be a consideration to avoid folding scaling issues.

4-Bit Thresholding

The number of thresholding intervals scales exponentially with the number of activation bits. The choice of 4 was made early on as roughly the maximum acceptable value and is likely to be improved between certain layers.

8-Bit Weights

In a similar manner to the thresholding, the weight bitwidth was also not explored enough. While the first layer likely would suffer from much heavier quantization, the following layers could likely use much more aggressive quantization and could be a topic of its own.

9.5 Framework

Overall, the choice of exploring FINN as a way to implement hardware accelerators for modulation classification on an FPGA was a success and produced a working hardware for the 512, but not the 1024 model. It should be noted that FINN is perhaps not a particularly good fit for this specific application due to the fact that higher precision is needed for an acceptable classification probability, while computational units in FINN seem to be designed and optimized with BNN and lower precision networks in mind. Likely, some low-precision GPU is perhaps a more suitable option, but investing more resources into a larger FPGA or ASIC to achieve better bandwidth is of course possible.

9.6 Results

Some discussion of the results is presented in this section

9.6.1 1024 Sample Model

While a direct comparison cannot be made due to the changes in the model architecture, the results for the initial model with standard floating-point weights and other parameters perform very close to the 8-bit weight, 4-bit activation version in terms of classification accuracy. Only a very small penalty is incurred by this precision loss, which is very promising for efficient hardware implementations.

9.6.2 512 Sample Model

Similar to the 1024 version, the 512 models show very promising classification accuracy. There is an observable increase in misclassifications between similar classes as the number of samples, and therefore the number of symbols, is decreased from 1024, which was expected. In hindsight, this is probably not a great idea; the convolutional layers dominate in terms of execution time too much to make the trade-off from the quadratic scaling of weights required by the fully connected layers worthwhile, and were not even compared in this work for this reason. There is also the consideration that while the latency is doubled for the 1024 sample model compared to the 512 version, the time to compute a classification when attempting to run such a system in real-time is also doubled. Meaning that effectively the trade-off is only between latency and classification accuracy, as the hardware requirements and throughput should likely be similar between the two.

9.6.3 1024 Sample Accelerator

A working accelerator for the 1024 model could not be completed within the remaining time of the thesis. It should be noted that the simulation of the hardware in FINN by means of a C++ simulation provides the correct results shown in

Section 7.5, and the synthesis completes without errors. However, with hardware that produces nonsense when tested, troubleshooting proved to be quite difficult.

9.6.4 512 Sample Accelerator

The three options of accelerators implemented for the 512 sample model work very well, and show that the folding parameters can be set effectively to give excellent increases in performance of the system with very little power penalty.

9.7 Future Work

Some additional topics and interesting areas to perform a deeper exploration on are presented below.

9.7.1 Machine Learning Model Optimizations

It is clear that the model-hardware co-design is fundamental in achieving success. While the focus of this work was centered mainly on the hardware design, many aspects of the initial model were unchanged and unexplored. Changing the kernel size of the convolutional layers, increasing or decreasing the number of channels, utilizing the tendency of models to be more sensitive to quantization at the input and output layer to perform more aggressive weight bitwidth reduction as well as letting some of these parameters be learned through training instead of manually configured are all areas that have room for improvements.

The overall model certainly can be improved by pruning additional layers, especially the fully connected classification part. The training and hyperparameter tuning leave room for more sophisticated methods and improvements as well.

9.7.2 Input Bitwidth Reduction

The choice to quantize the input to 8-bits, based on an ENOB of around 5-bits, is quite an interesting topic that I think warrants a deeper investigation into what effects can be observed.

9.7.3 Mixed Precision

The overall model weights were limited to 8-bits, activations to 4-bits. Dynamic parameter configuration in the training stage could likely improve the model performance and warrants a deeper look.

9.7.4 One-Shot Modulation Recognition

A natural next step is extending this work to implement an accelerator for one-shot modulation recognition.

9.7.5 RFSoC

The results of the thesis are a system running on a physical device, but performing classifications on synthetic data. It would be interesting to implement a design on, for example, an RFSoC FPGA and investigate if the classification accuracy of non-synthetic signals match, and what difficulties the entire design from sampling to classification brings.

9.7.6 FPGA Limitations

With the model, hardware, and selected framework, there are certain constraints on the performance. The number of hardware resources is certainly one aspect, but specifically, some constraints on the SIMD and PE values may bottleneck the entire design. A more in-depth look at this area may prove fruitful for more intelligent model-hardware co-design.

9.7.7 Alternative Hardware

A comparison of performance to Nvidia Jetson or DPU FPGA implementations would be of interest to determine what hardware could be suitable for a certain application could be an interesting area to explore.

9.7.8 Alternative Frameworks

This thesis focused on the FINN framework; there certainly exist other alternatives that would be prudent to compare with.

9.7.9 Adversarial Attacks & Mitigation

The use case and context of this thesis is in signal intelligence, to classify signal modulation types in critical applications comes with the additional aspect to consider adversarial actions and adaptation. An investigation into how the classifier may be attacked, in terms of intended missclassification, for example, and how that can be mitigated or detected, may also be an interesting area to explore.

10 Conclusion

In conclusion, this thesis investigates the implementation of dataflow accelerators on an FPGA platform with promising results. The primary findings are that, with synthetic data, machine learning models can be trained with quantization-aware training to produce a classification of modulation type with acceptable accuracy. Hardware can be constructed to run such models on an FPGA, with low power consumption on consumer-grade devices, with sufficient performance to be usable for practical applications. There are limitations and considerations that need to be made when constructing such a system, and the design of both hardware and model jointly is critical for success.

10.1 Research Questions

The research questions are revisited below.

How can a machine learning model inference be implemented on an FPGA platform?

Machine learning inference can be implemented in a number of ways, through custom-tailored implementations, systolic arrays, dataflow pipelines, or shared memory architectures with compute units. However, it is clear that when the complexity and size of the model increase, some trade-off needs to be made to automate the implementation.

What restrictions exist for a model that would make it suitable for inference on an FPGA?

From the models considered in this thesis, it is clear that arithmetic operations, bitwidth, and the number of ports required were the limiting factors in the complexity and performance of models that could be implemented. In addition, hardware implementation makes operations such as matrix transposition, padding, and certain concatenation or flattening operations non-trivial, and while used in software models, they need to be worked around. The model-hardware co-design is fundamental in both achieving a working design and the intended performance.

How does the number of input samples affect model and hardware results?

As the results of the modeling show, there is a penalty in classification accuracy when reducing the number of I/Q input samples from 1024 to 512. And while the fully connected layers scale quadratically with the number of nodes, the dominant convolutional layers only scale linearly with input samples. As the full pipeline is bottlenecked by the slowest stage, this trade-off was less favorable than expected, and a larger number of input samples is better than initially thought. However, the fully connected layers require a large amount of weights, which could be allocated elsewhere in the design to improve performance.

What are the performance results of an accelerator for CNN for Automatic Modulation Classification running on a Zynq-UltraScale+ MPSoC? What parameters can be configured?

Three separate skews of accelerators were implemented, where the folding parameters SIMD and PE were adjusted for each compute unit. The highest performing accelerator for the 512 sample model had a latency of 485 μs and a throughput of 6305.6 classifications per second. The maximum bandwidth for real-time classification is approximately 3.2 MHz. The power consumption of the PL was 1.317 W, the utilization of LUTs around 33%, DSP blocks 60%, and block RAM close to 98%.

Bibliography

[1] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[2] B. Ramkumar, "Automatic modulation classification for cognitive radios using cyclic feature detection," *IEEE Circuits Syst. Mag.*, vol. 9, no. 2, pp. 27–45, 2009.

[3] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," *IEEE J. Sel. Topics Signal Process.*, vol. 12, no. 1, pp. 168–179, 2018.

[4] F. Alberge, "Deep learning constellation design for the AWGN channel with additive radar interference," *IEEE Trans. Commun.*, vol. 67, no. 2, pp. 1413–1423, 2019.

[5] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott, and M. Platzner, "RadioML meets FINN: Enabling future RF applications with FPGA streaming architectures," *IEEE Micro*, vol. 42, no. 6, pp. 125–133, 2022.

[6] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *Proc. IEEE 26th Annu. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2018, pp. 93–96.

[7] F. Hamanaka, T. Odan, K. Kise, and T. V. Chu, "An exploration of state-of-the-art automation frameworks for FPGA-based DNN acceleration," *IEEE Access*, vol. 11, pp. 5701–5713, 2023.

[8] "DeepSig RadioML dataset," https://www.deepsig.ai/datasets/, Accessed: Feb. 12, 2025.

[9] S. Tridgell, D. Boland, P. H. W. Leong, R. Kastner, A. Khodamoradi, and Siddhartha, "Real-time automatic modulation classification using RFSoC," in *Proc. 2020 IEEE Int. Parallel and Distrib. Process. Symp. Workshops*, 2020, pp. 82–89.

[10] A. Maclellan, L. H. Crockett, and R. W. Stewart, "RFSoC modulation classification with streaming CNN: Data set generation & quantized-aware training," *IEEE Open J. Circuits Syst.*, vol. 6, pp. 38–49, 2025.

[11] "MathWorks HDL coder," https://se.mathworks.com/products/hdl-coder.html, Accessed: Feb. 12, 2025.

[12] G. Natale, M. Bacis, and M. D. Santambrogio, "On how to design dataflow FPGA-based accelerators for convolutional neural networks," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2017, pp. 639–644.

[13] T. Svensson and E. Becirovic, "One-shot modulation recognition using artificial modulation types for enhanced meta-learning," in *Proc. 2024 IEEE Military Commun. Conf.*, 2024, pp. 688–694.

[14] G. Franco, A. Pappalardo, and N. J. Fraser, "Xilinx/brevitas," https://doi.org/10.5281/zenodo.3333552.

[15] "ONNX: Open neural network exchange," https://onnx.ai, Accessed: Jun. 24, 2025.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst. 32 (NeurIPS)*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[17] "Netron: Visualizer for neural network, deep learning, and machine learning models," https://netron.app, Accessed: Jun. 24, 2025.

[18] "PYNQ," http://www.pynq.io/, Accessed: Feb. 12, 2025.

[19] ARM Ltd., "AMBA 4 AXI and ACE protocol specification," https://developer.arm.com/architectures/system-architectures/amba, 2010, Accessed: Feb. 12, 2025.

[20] "Kria KR260 robotics starter kit," https://www.amd.com/en/products/system-on-modules/kria/k26/kr260-robotics-starter-kit.html, Accessed: Feb. 12, 2025.

[21] "Xilinx Zynq UltraScale+," https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-ultrascale-plus-mpsoc.html, Accessed: Feb. 12, 2025.

[22] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfig. Technol. Syst.*, vol. 11, no. 3, pp. 1–23, 2018.

[23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 65–74.