UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Efficient RISC-V SIMD extension for AI inference acceleration

Author:
Pavlo HILEI

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Science

in the

Department of Computer Sciences and Information Technologies
Faculty of Applied Sciences

Lviv 2023

Declaration of Authorship

"Okay, Houston... we've had a problem here."

Jack Swigert

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Efficient RISC-V SIMD extension for AI inference acceleration

by Pavlo HILEI

Abstract

Machine learning has become a popular approach for solving complex problems; however, deploying deep learning models in real-life applications poses challenges due to their computational requirements. Edge computing offers a promising solution, but running large models on resource-constrained embedded devices often necessitates software and hardware optimizations. In this thesis, we focus on hardware acceleration using a Custom Function Unit (CFU) extension for a RISC-V processor. Through an iterative development process, we design an accelerator for the convolutional layer and evaluate it in simulation and hardware implementation. We address the automatic modulation classification problem and develop deep neural networks to solve this task. Furthermore, we apply the developed CFU to accelerate the inference of a convolutional neural network (CNN) used for wireless signal modulation classification. We achieve acceleration of x148, compared to the original model, and x18.5, compared to the original model with applied quantization. By combining software and hardware optimizations, our research contributes to enabling the efficient deployment of deep learning models on edge devices.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor Oleg Farenyuk for his invaluable guidance, unwavering support, and dedication throughout the entire duration of this project and my years of study.

A special word of thanks goes to Ievgen Korotkyi, Associate Professor at the Department of Electronic Computational Equipment Design, Igor Sikorsky Kyiv Polytechnic Institute. His mentorship, insightful discussions, and assistance in overcoming various challenges have been extremely valuable to the development and execution of this project.

I would also like to extend my appreciation to the Ukrainian Catholic University and all the academic staff who have been instrumental in my education, providing me with a nurturing environment and the necessary resources to enhance my learning and professional growth.

Lastly, I am deeply grateful to my family and friends I met at UCU. Their support and encouragement have contributed to my personal and academic journey.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AM	**A**mplitude **M**odulation
AMR	**A**utomatic **M**odulation **R**ecognition
ASIC	**A**pplication **S**pecific **I**ntagrated **C**ircuit
CFU	**C**ustom **F**unction **U**nit
CLB	**C**onfigurable **L**ogic **B**lock
CLB	**C**onfigurable **L**ogic **B**lock
CLDNN	**C**onvolutional **L**ong **S**hort-term **D**eep **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
CPU	**C**entral **P**rocessing **U**nit
DNN	**D**eep **N**eural **N**etwork
FM	**F**requency **M**odulation
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSK	**F**requency **S**hift **K**eying
FSK	**F**requency **S**hift **K**eying
GPU	**G**raphics **P**rocessing **U**nit
HDL	**H**ardware **D**escription **L**anguage
I/O	**I**nput/**O**utput
IQ	**I**n-phase and **Q**uadrature
ISA	**I**nstruction **S**et **A**rchitecture
IoT	**I**nternet **o**f **T**hings
LDA	**L**inear **D**iscriminant **A**nalysis
LSTM	**L**ong **S**hort-**t**erm **M**emory
LUT	**L**ook**U**p **T**able
LUT	**L**ookup **T**able
ML	**M**achine **L**earning
NLP	**N**atural **L**anguage **P**rocessing
NPU	**N**eural **P**rocessing **U**nit **A**rray
PE	**P**rocessor **E**ngine
PM	**P**hase **M**odulation
PSK	**P**hase **S**hift **K**eying
QAM	**Q**uadrature **A**mplitude **M**odulation
RNN	**R**ecurrent **N**eural **N**etwork
SNR	**S**ignal-to-**n**oise **r**atio
SVD	**S**ingular **V**alue **D**ecomposition
SVM	**S**upport **V**ector **M**achines
SoC	**S**ystem **o**n **C**hip
TF	**T**ensorflow
TFLM	**T**ensorflow **L**ite **M**icro
TPU	**T**ensor **P**rocessing **U**nit **A**rray

Chapter 1

Introduction

Machine learning (ML) has become increasingly popular and useful in recent years due to its ability to handle complex tasks and make accurate predictions. It is used for computer vision, natural language processing, speech recognition, robotics, and so on [69]. One of the critical areas where ML models have shown great promise is signal modulation classification, an important problem in wireless communication systems [18]. Accurately classifying the modulated signals in a communication channel is crucial for some wireless communication, military, and IoT systems.

Unfortunately, most ML models today are computationally intensive [70], and the traditional processing units may not be efficient enough to perform these computations effectively. Cloud computing is a widespread approach to using complex models that require much computation power on simple and cheap systems. But it has many problems related to latency, reliability, communication bottleneck, privacy, etc. That is why there is a growing need for edge computing. Edge computing allows for processing and storing data closer to the source, reducing the need to transmit data back and forth from a centralized server [8]. Custom Function Units (CFUs) have emerged as a promising solution [5]. CFUs are specialized processing units that can be integrated into a general-purpose processor to accelerate the execution of specific tasks. They are designed to perform specific operations and can be programmed to suit the application's specific needs.

The rise of open-source hardware and processors like RISC-V [80] has made developing custom function units for edge computing much easier and gives more flexibility to the design since the CPU can also be configured for a concrete application. Field Programmable Gate Arrays (FPGA) today are powerful enough to fit such processors, significantly speeding up, simplifying, and making cheaper experimenting with such hardware designs. Frameworks provided by FPGA manufacturers and the open-source community help to optimize custom hardware for latency and energy efficiency.

Modern ML frameworks, like Tensorflow Lite Micro (TFLM) and PyTorch Mobile [45] are highly optimized for limited hardware. They can deploy models on embedded systems fairly simply, and the open-source nature of these frameworks enables developers to seek for computation bottleneck of the deployed models. CFU-Playground [62] allows to unite listed above technologies – it allows CFU development for VexRiscV [56] soft processor that implements RISC-V ISA and CFU extension and runs TFLM model(s) on it. The resulting model can be executed on both the Renode simulator [65] and synthesized on the FPGA board.

Therefore, in this bachelor thesis, we propose using a CFU in a RISC-V processor to accelerate the execution of the Convolutional Neural Network (CNN) used for signal modulation classification. Different architectures and datasets are considered to train such a model. We investigate deployed model to find the execution bottlenecks that CFU can accelerate. We investigate the design and implementation of the CFU and its integration with the RISC-V processor. We also evaluate the performance of the proposed system in both simulation and FPGA boards. The results show that the proposed system can significantly reduce execution time and power consumption while maintaining high accuracy in signal modulation classification. This research could contribute to developing more efficient and accurate wireless communication systems and enable deploying edge computing systems in resource-constrained environments.

1.1 Structure

- Chapter 2 describes the signal modulation classification problem and reviews current classical and ML approaches to the problem. Different acceleration approaches are described. RISC-V ISA and FPGA are overviewed.
- Chapter 3 introduces technologies for developing accelerators and the development pipeline in CFU-Playground.
- Chapter 4 describes models we developed for the modulation classification problem.
- Chapter 5 explains developed accelerators, presents results of model acceleration, comparison of embedded and simulated systems, measurements of inference execution time, and utilization of FPGA resources.
- Chapter 6 concludes the thesis with a summary of the findings and discusses further work.
- The work includes the following appendixes: appendices A to D

Chapter 2

Background

2.1 Signal modulation

Signal modulation is a process of modifying a signal to enable it to carry information from one point to another. In wireless communication systems, modulation encodes information onto a carrier wave transmitted over electromagnetic waves. Signal modulation includes manipulating a carrier signal by a modulating signal to produce a modulated signal that contains the desired information. The carrier signal is typically a high-frequency sine wave. The modulating signal is the signal that contains the information that needs to be transmitted. The modulated signal is then transmitted to the receiver, which is demodulated to extract the original modulating signal. The main reasons for modulation are bandwidth utilization, noise, interference mitigation, and efficient signal propagation [60].

Generally, frequency modulation can be divided into analog and digital. The name describes the nature of the data transmitted. The most common types of analog modulation used in wireless communication systems are Amplitude Modulation (AM), Frequency Modulation (FM), and Phase Modulation (PM) [17]. Some of the popular digital modulation techniques include Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), Phase Shift Keying (PSK), and Quadrature Amplitude Modulation (QAM) [60, 4]. These modulation techniques vary in terms of the number of bits they can transmit per symbol, spectral efficiency, and susceptibility to noise and interference. They are widely used in various communication systems, including radio and television broadcasting, cellular networks, and satellite communication systems.

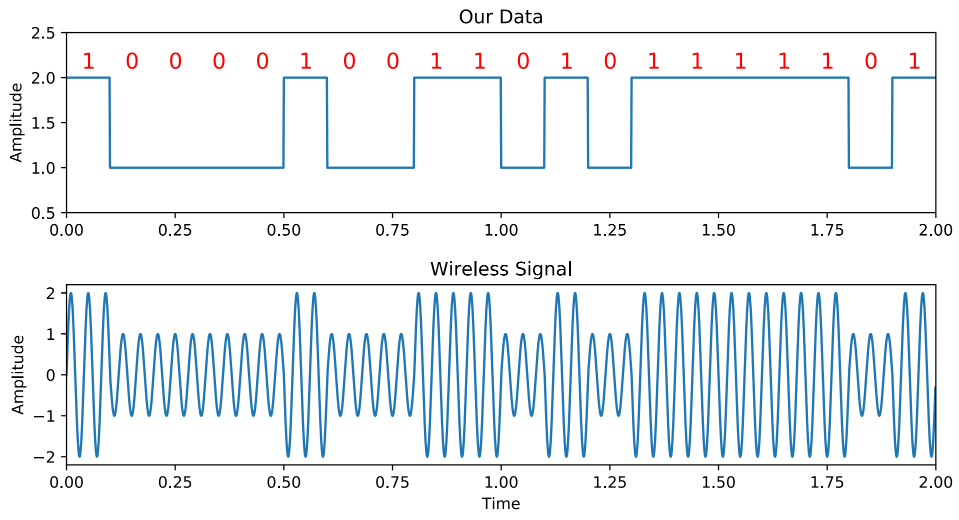
Example of digital modulation Amplitude Shift Keying – when carrier signal amplitude encodes transmitted digital signal is depicted in Figure 2.1.

2.2 Signal modulation classification

Signal modulation classification or Automatic Modulation Recognition (AMR) is the process of identifying the modulation technique used to transmit a signal. Accurately identifying the modulation scheme used in a communication signal is essential for designing effective communication systems, as different modulation techniques require different receiver configurations and signal processing algorithms. Some of AMR applications are:

- Cognitive Radio [10]: Cognitive radio systems aim to dynamically allocate available spectrum to different users based on real-time conditions. AMR plays a crucial role in cognitive radio by enabling the radios to identify and adapt to the modulation schemes used by primary users. This allows users to

FIGURE 2.1: Amplitude Shift Keying (ASK) visualization [21]



utilize the spectrum efficiently for better avoidance of interference with other primary users.

- Wireless Network Planning: When planning and deploying wireless networks, AMR assists in identifying the modulation schemes employed by neighboring networks. This information is valuable for selecting appropriate frequency channels, optimizing signal parameters, and minimizing interference between neighboring networks.

- Electronic Warfare [1]: In military applications, AMR is utilized in electronic warfare to automatically identify and classify enemy communication signals. By recognizing the modulation schemes, military personnel can assess the capabilities, intentions, and threat levels of adversaries and develop appropriate countermeasures.

2.2.1 Model input

There are different approaches to data preparation as the input to the model. Some of them are presented in [81]. For example, constellation maps can be used as 2-dimensional input to convolutional neural network [84, 83, 19]. Time-frequency diagram, Eye-diagram, and Amplitude Histogram are also used as preprocessed input to the model [81]. A constellation map refers to a graphical representation (xy plane) that depicts the relationship between the different symbols or signal points used in a modulation scheme, like real and complex parts in in-phase and quadrature (IQ) representation of signal, or phase in PSK, as shown in Fig. 2.2.

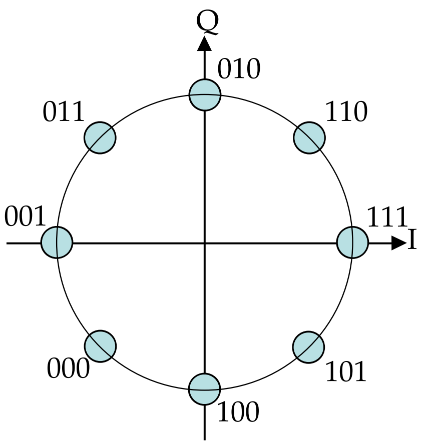
2.2.2 Classical approaches

Over the years, various methods have been proposed for signal modulation classification, ranging from traditional statistical methods to modern machine learning-based approaches. Some of the popular classical classification methods include Linear Discriminant Analysis (LDA) [27] , Singular Value Decomposition (SVD) [29], Maximum Likelihood [72], Support Vector Machines (SVMs) [85] and XGBoost [55].

2.2.3 Deep Learning solutions

Usually, transmission data in real life is subject to various sources of distortion, like fading, interference, timing drift due to clock offset, and general noise of the environment. Deep learning models can perform well with much noise if trained properly with high-quality data.

FIGURE 2.2: Constellation map of 8-PSK: each symbol is encoded as a different phase shift of the carrier sine wave: 0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°, Credit: Wikipedia [15].



In recent years, Convolutional Neural Networks (CNNs) have emerged as a powerful tool for signal modulation classification. CNNs have shown remarkable performance in various machine-learning tasks, including image recognition, natural language processing, and speech recognition [2]. CNNs have also been applied to signal modulation classification and have achieved good performance, even in noisy and complex environments. As shown in [55, 83], they can outperform classical approaches.

As shown in [83], popular CNN models, like VGG-16 [73] and AlexNet [38], perform well on modulation classification. They convert the received data to a constellation image. Another approach is to use an IQ representation of the received data as input for the regular CNN model, which combines convolutional layers, pooling layers, and fully connected layers to extract features from the input signal and classify it into one of the modulation classes. Timothy J O'Shea et al. [54] show that such models can perform well and outperform classical approaches. Also, they show that the CNN has noticeably higher accuracy than Deep Neural Network (DNN) model. Xiaoyu Liu et al. [44] demonstrated that increasing the depth of the model and using ResNet [30] doesn't improve accuracy much, and Convolutional Long Short-term Deep Neural Network (CLDNN) has a better classification accuracy score.

Transformer models are a deep learning architecture primarily used for natural language processing (NLP) tasks, such as language translation and text generation [59]. Unlike traditional recurrent neural networks (RNNs), which process input sequences sequentially, transformers use a self-attention mechanism to process input sequences in parallel, allowing them to capture long-term dependencies more effectively. Recent studies show that transformers perform well on other tasks, like image generation [57], object detection [9], and image recognition [20].

Transformer models also show high potential in this problem. Zheng, Yujun, et al. [86] achieved better accuracy using a Transformer-based automatic classification recognition network improved with Gate Linear Unit (TMRN-GLU) than ResNet, DenseNet [33], CLDNN [68], and LSTM [31] with more parameters. As shown in [42], the Transformer-based model noticeably outperforms CNN, LSTM, and CLDNN.

2.3 Hardware accelerators

There are several approaches to hardware acceleration of AI workload.

Graphics Processing Units (GPUs) have become increasingly popular for accelerating machine learning workloads [49]. GPUs are highly parallel and can have thousands of processing cores, which allows them to perform large matrix operations much faster than traditional CPUs. In addition, many machine learning frameworks, such as TensorFlow [47] and PyTorch [58], have GPU support built-in, making it easy to run machine learning workloads on GPUs. There are some challenges associated with using GPUs for machine learning. One of the primary challenges is memory limitations. GPUs typically have less memory than CPUs, which can limit the size of the networks that can be trained. In addition, GPUs require specialized programming models and tools, which can be difficult for some developers to learn and use effectively.

Tensor Processing Unit, shown in Fig 2.3 (TPU), is a hardware accelerator designed by Google [37] specifically for machine learning workloads. TPUs are highly optimized for processing large matrix calculations and are particularly effective for deep-learning models. TPUs are built on a custom chip design and consist of multiple cores optimized for matrix multiplication and other tensor operations. These cores are also connected by a high-bandwidth interconnect, which enables efficient data transfer between them.

Field Programmable Gate Arrays (FPGAs) are another type of hardware accelerator that can accelerate AI workloads. FPGAs are integrated circuits that can be programmed and reprogrammed to implement custom logic functions. It is possible to synthesize the whole model on the FPGA board using frameworks like hls4ml [23] or Matlab Deep Learning HDL Toolbox [64].

There are many other types of accelerators [11]. Neural Processing Unit (NPU) [22] uses eight processor engines (PE) to calculate multiply-accumulate-sigmoid to accelerate neural network's computations. Matrix-vector multiplications are carried out by RENO, a reconfigurable neuromorphic computing accelerator, which uses the ReRAM crossbar as its fundamental computing unit. RENO inputs and outputs are digital, while intermediate results are analog [43].

2.4 RISC-V

RISC-V [80] is an open-source instruction set architecture (ISA) that has recently gained popularity due to its modularity, scalability, and flexibility. Unlike traditional proprietary ISAs, RISC-V is freely available for academic and commercial use and allows users to design custom instruction sets tailored to their specific application requirements. This makes RISC-V attractive for embedded systems, Internet of Things (IoT) devices, and other resource-constrained applications.

The RISC-V ISA is based on a simple load-store architecture that employs a small set of basic instructions that can be combined to perform complex operations. This simplicity allows for a smaller, more power-efficient implementation and easier verification and testing. Moreover, RISC-V supports variable-length instructions, which means it can support different instruction sizes depending on the application requirements.

One of the key advantages of RISC-V is its modularity. The ISA is divided into several standard extensions, such as the Integer, Floating-Point, and Vector extensions, that can be selectively included or excluded depending on the application requirements [80]. This allows for a highly customizable and scalable implementation optimized for specific use cases. Another advantage of RISC-V is its open-source nature. The ISA is developed and maintained by the RISC-V Foundation, a non-profit organization that promotes the use and adoption of the RISC-V ISA. This

FIGURE 2.3: TPU architecture overview [37].

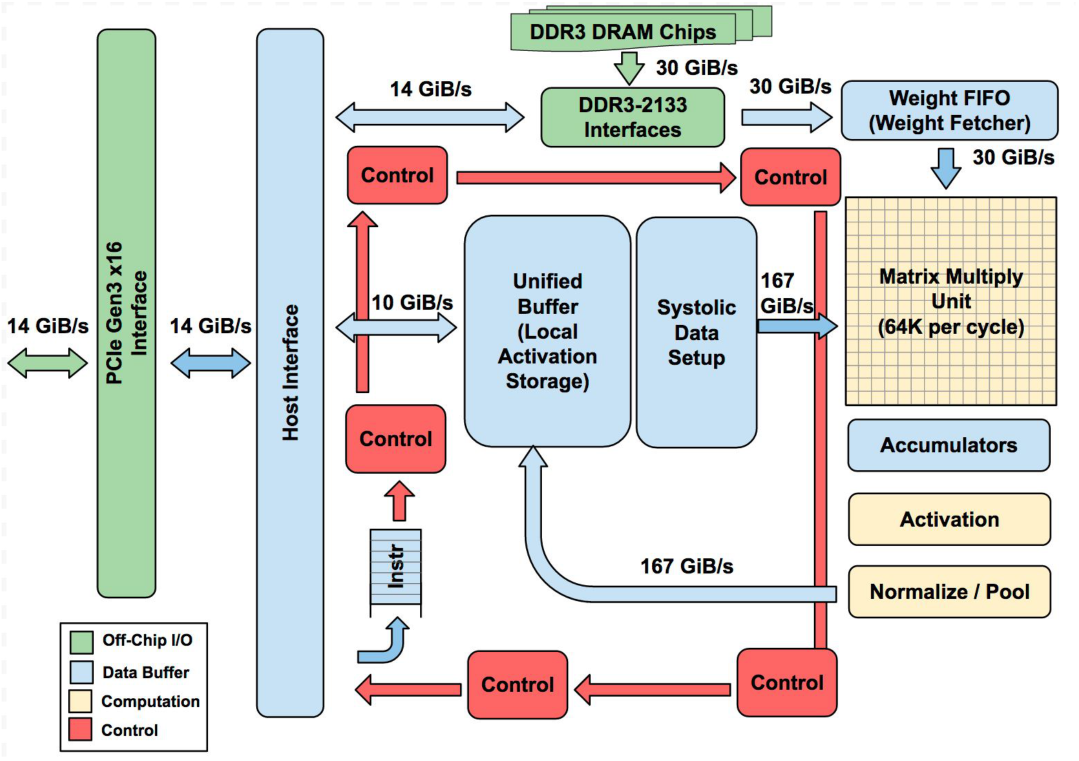


FIGURE 2.4: RISC-V R instruction format [80]

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

open-source model encourages collaboration and innovation, allowing for a more transparent and inclusive development process.

2.4.1 Custom Function Unit

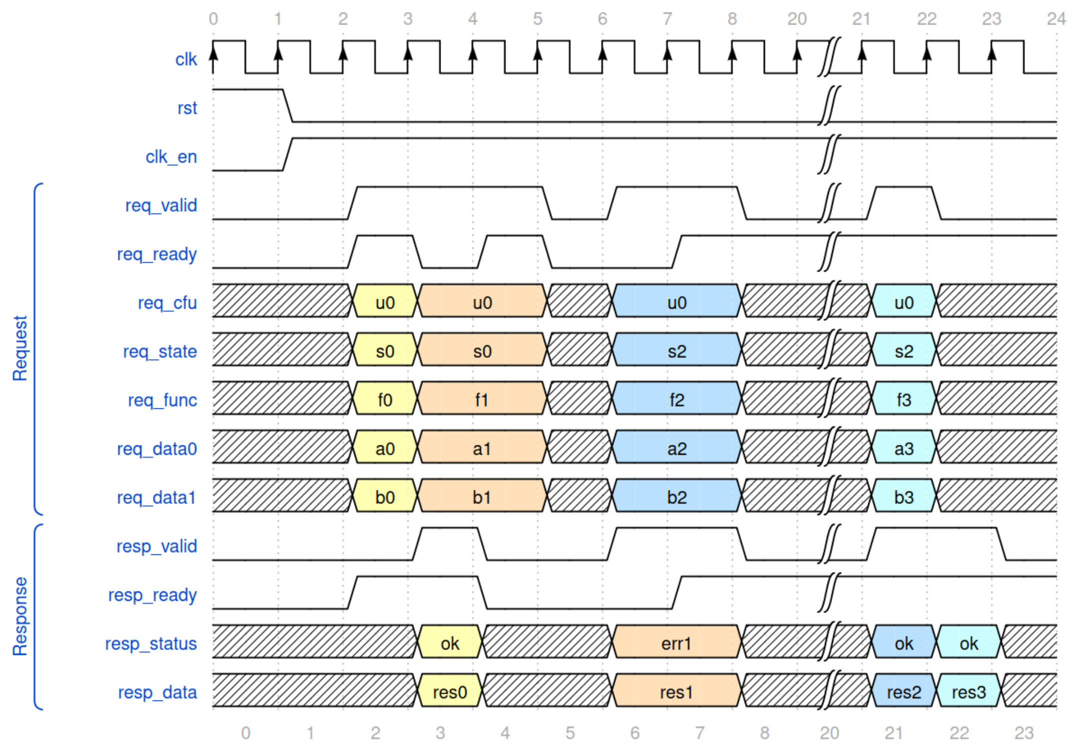
Custom Function Units (CFUs) are specialized hardware that can be integrated into a general-purpose processor to accelerate specific operations or tasks. CFUs are designed to work in tandem with the processor, offloading computation from the main pipeline and reducing the system's overall energy consumption and latency.

Authors in [5] propose the implementation of CFU for RISC-V ISA. CFU is invoked by a custom R-format ISA instruction (Fig. 2.4) added to the CPU. R-format instruction has two input registers with the size of 32 bits ($rs1$, $rs2$), a destination register (rd), and 10 bits to encode what operation to perform ($funct7$ and $funct3$).

A two-way handshake is used for CPU-to-CFU communication. When the CPU calls CFU, it puts $funct7$, $funct3$, and both operands on the bus. After that, signal *cmd_valid* is set to 1 to ask CFU to start the computation. CFU confirms that it received a request by setting *cmd_ready* to 1. CPU is stuck until a response is returned. When the job is done, CFU informs CPU by setting *rsp_ready* to 1, and CPU acknowledges the response with *rsp_valid*. The timing diagram is shown in Fig. 2.5.

CFU doesn't impose any restrictions on the hardware design – the only limiting factor is FPGA resources. But there are two important limitations:

FIGURE 2.5: CFU timing diagram (CFU-L2 signaling protocol) [5]



- Interface is strictly R-format 2.4, and no changes are permitted without direct interference with ISA implementation,
- CFU doesn't have access to the RAM of the CPU.

Because of the second restriction, the current CFU specification limits the acceleration potential since the CPU has to copy data to the on-chip memory buffer accessible by the accelerator.

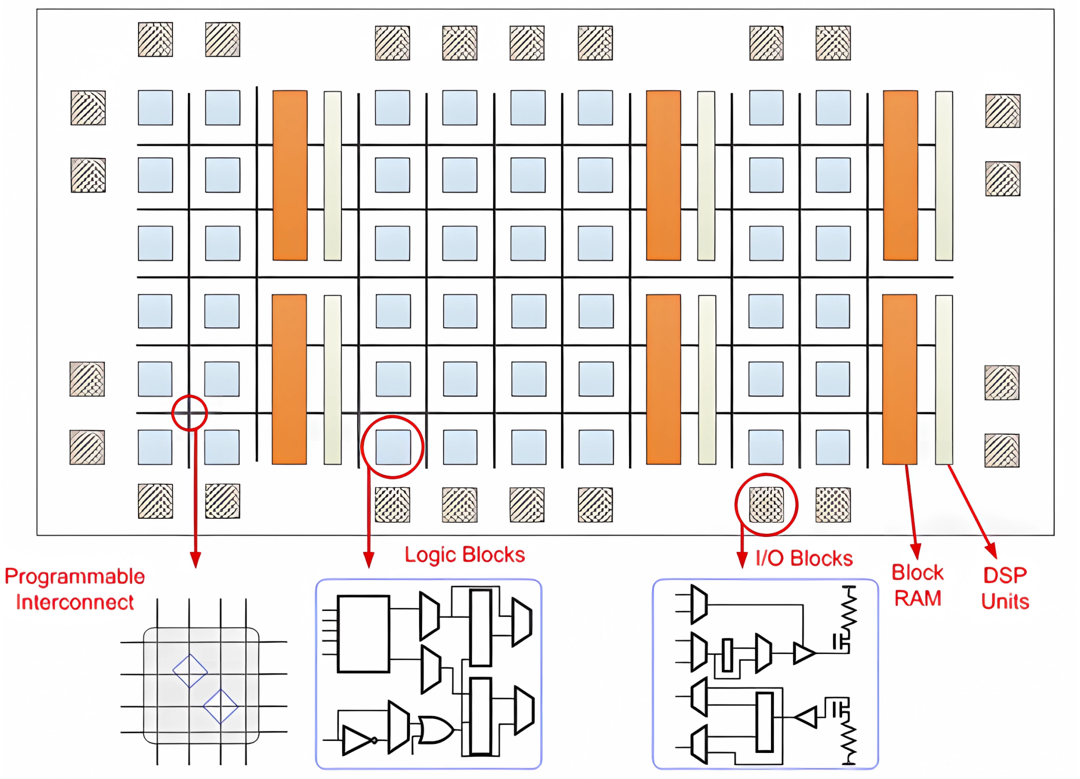
2.5 Field Programmable Gate Array

FPGAs are programmable logic devices that allow digital circuits and systems to be implemented through configurable logic blocks and interconnections. FPGAs offer high flexibility and customization, as they can be reprogrammed to implement different circuits or functions.

FPGAs are typically programmed using a Hardware Description Language (HDL), such as Verilog or VHDL, which describes the functionality and behavior of the circuit to be implemented. The HDL code is then synthesized into a configuration file that can be loaded onto the FPGA, configuring it to implement the desired circuit.

FPGAs consist of many configurable logic blocks (CLBs), input/output (I/O) blocks, and resources interconnected through a programmable routing matrix. Each CLB typically contains a lookup table (LUT), which can implement any Boolean function, and a flip-flop or latch for storing state. The interconnect resources consist of programmable switches and wires that can be configured to connect the CLBs and I/O blocks as needed. The routing matrix provides a flexible and customizable way to connect the logic blocks and I/O blocks, allowing various circuit topologies and functions to be implemented [6].

FIGURE 2.6: High-level Xilinx FPGA architecture overview. Credit: [14]



One of the key benefits of FPGAs is their ability to provide hardware acceleration for computationally intensive tasks. FPGAs can be programmed to implement custom hardware accelerators for specific applications, providing significant speedup and energy efficiency compared to software-based implementations on general-purpose processors. [63]

FPGAs also offer other benefits, such as high throughput [74] and integrating multiple functions or peripherals into a single device. Finally, FPGAs are also highly adaptable, allowing for quick iterations and modifications to the design. Experiments are often conducted on the FPGA, and later design is deployed as Application-specific integrated circuit (ASIC) [24]. After design finalization, ASIC is synthesized to reduce power consumption and design area and further improve the accelerator's throughput, latency, price, etc [39].

Diagram showing high-level architecture of Xilinx FPGA – Fig.2.6.

Chapter 3

Methodology

3.1 Tensorflow Lite Micro

TensorFlow Lite Micro (TFLM) [16] is a version of TensorFlow Lite designed for microcontrollers and other embedded systems with limited memory and processing power. TFLM provides a lightweight inference engine optimized for running deep learning models on devices with constrained resources. It supports a wide range of hardware platforms and allows for easy deployment of models on microcontrollers. For example, it doesn't require any support for dynamic memory and doesn't expect to run under the operating system. It has a simple yet effective way of managing allocations in the static memory of the program, using a two-stack allocation strategy and bin packing.

TFLM has several useful features that make it a popular choice for AI model deployment on microcontrollers. For example, it supports a variety of neural network architectures, including convolutional, recurrent, and fully connected networks. It also includes a range of pre-trained models that can be used for various tasks, such as image classification and voice recognition. Additionally, TFLM provides tools for quantization, which is a process that allows models to be compressed and run more efficiently on resource-constrained devices.

However, TFLM also has some limitations that should be considered when using it for AI model deployment. One limitation is that it only supports a limited subset of TensorFlow operations, which means that some models may not be fully compatible with TFLM. Additionally, TFLM doesn't support training, only inference.

3.2 CFU-Playground

To develop a custom Function Unit (CFU) for accelerating AI workloads, we will use the CFU-Playground tool described in [62]. CFU-Playground is an open-source framework that provides a platform for developing and testing custom CFUs for RISC-V processors. The framework includes tools and scripts to create Custom Function Unit for a RISC-V soft-core. It can test performance and profile design, run a simulation of the processor, and synthesize it to deploy on the FPGA board. CFU-Playground supports different tools to conduct place and route and generate FPGA bitstream. It is integrated with TensorFlow lite micro (TFLM) for deployed ML model interference.

The main idea is to find hotspots – places in the model code where most CPU time is spent. Usually, such hotspots include repeated simple operations, such as multiply-add accumulate and matrix multiplications. Some code may require logical operations, such as quantizing calculated results. These kinds of operations can be implemented very efficiently in the hardware and map perfectly to the parallel nature of FPGA.

3.2.1 Soft core

VexRiscV [56] is a soft processor implementation of the RISC-V ISA that is open-source and fully customizable. It implements RV32I base integer instruction set and with a subset of standard extensions[1] ISA and pipeline from 2 to 5+ stages ([Fetch*X], Decode, Execute, [Memory], [WriteBack]). Its small and efficient design makes it ideal for embedded systems with limited resources. Additionally, VexRiscV has built-in support for various extensions, including custom instructions and support for plugins.

VexRiscV was chosen as the platform for developing the custom CFU in CFU-Playground. VexRiscV is one of the few soft processors that support the CFU extension. Also, it is synthesizable on the FPGA and doesn't require many resources, which leaves more FPGA resources to be allocated for the custom CFU. It is very flexible. A developer can change a number of caches, cache sizes, and pipeline stages and add custom instructions.

3.2.2 HDL

Hardware Description Language (HDL) is a specialized computer language that describes the behavior of digital circuits and systems. HDLs can be used to create high-level behavioral models of digital circuits or low-level designs that specify individual logic gates and connections. Verilog and its successor SystemVerilog are a popular HDL for designing digital systems. They are commonly used to design digital circuits, such as CPUs, FPGAs, and ASICs. Verilog has a C-like syntax, providing features such as modules, functions, and procedural constructs to describe digital circuits. Verilog and SystemVerilog are standardized by the Institute of Electrical and Electronics Engineers under the standards of IEEE 1364-2005 [35] and IEEE 1800-2017 [34], respectively.

3.2.3 Simulation

Simulating hardware design is an important step in developing any hardware system. CFU-Playground uses Renode [65] to simulate the processor and CFU together. Renode is a framework developed by Antmicro for simulating IoT devices. It can simulate heterogeneous multicore SoCs and various peripherals. The processor is simulated on the ISA level. Renode supports the most popular ISAs, like ARMv7 and ARMv8, X86, RISC-V, SPARC, POWER, and Xtensa. CFU is simulated in a cycle-accurate Verilator simulation.

3.2.4 Synthesis. Place and Route

Synthesis translates the hardware design, described in a Hardware Description Language (HDL) such as Verilog, into a netlist that describes the logical functions and their interconnections. The synthesis tool considers the constraints of the target device and the timing requirements of the design. The output of the synthesis tool is a gate-level netlist, which represents the hardware design in terms of primitive

[1][M][A][F[D]][C]: "M" – Integer Multiplication and Division, "A" – Atomic Instructions, "F/D" – Single/Double Point Precision Point, "C" – Compressed instructions [80].

logic gates. CFU-Playground supports open source Yosys [82] and Vivado synthesis tool [76] for Xilinx FPGA boards for the design synthesis.

Place and Route is the process of determining the physical placement of the logic elements in the FPGA and the routing of the interconnects between them. The place and route tool takes into account the physical constraints of the target device, such as the number of available logic elements, the number of I/O pins, and the routing resources. The output of the place and route tool is a bitstream file that can be loaded onto the FPGA to program the device. Vivado place and route tool [77] does this for Xilinx FPGA boards. An open-source alternative is Verilog to Routing's [51] tool called "Versatile Place and Route" (vpr) combined with an f4pga [25] bitstream generator.

Chapter 4

Classification model

4.1 Modulation Recognition Method Based on Deep Learning

As mentioned in Chapter 2, Automatic Modulation Recognition (AMR) refers to identifying the modulation scheme used to transmit a signal. In other words, given the received signal, the model should classify the modulation scheme used to encode information on a carrier signal. In terms of deep learning, this is a typical supervised classification model. It is very challenging to extract informative features for modulation classification due to noise, interference, fading, and other factors that disrupt the signal. Therefore, deep learning has emerged as a powerful tool in AMR, enabling reliable modulation classification even under challenging noise conditions.

Numerous model architectures were applied to this problem. The most common architectures were mentioned in Chapter 2. In this thesis, we develop, fine-tune, and evaluate the convolutional neural network and transformer model. They are compared in terms of accuracy, sensitivity to noise, speed, and number of parameters.

4.2 Dataset

As mentioned in Chapter 2, there are different ways to prepare data for classification, like constellation map [83] as 2-dimensional image input to the model, eye diagram as a 2-dimensional image [78], Time-frequency diagram [46], etc. In this thesis, we will use the IQ sequence to present input for the model. Models are trained on two datasets - RadioML [54] and Matlab [50] dataset. Models are trained and selected on RadioML 2016.10a. Then two best CNNs and two best transformers are trained and compared on RadioML 2016.10a, RadioML 2016.10b, and Matlab dataset.

4.2.1 RadioML

The dataset proposed by [54] is widely used by different authors for AMR. RadioML 2016.10 data is synthetically generated with GNU Radio. It comprises 11 modulations: Digital: BPSK, QPSK, 8PSK, 16QAM, 64QAM, BFSK, CPFSK, PAM4, and analog: WB-FM, AM-SSB, and AM-DSB. Approximately eight samples per symbol are used, and the sampling rate is 1×10^6 samples/s. Each frame consists of 128 samples. Randomly generated bits are modulated using one of the 11 modulations mentioned before. After that, a pulse-shaping filter (root-raised cosine) is applied. It is crucial to simulate noise factors to get realistic data for model training. The following impairments are applied:

- Flat white Gaussian noise caused by thermal noise.

- Timing offset, sample rate offset, carrier frequency offset, and phase difference.
- Multi-path fading - simulation of the environment where the signal can reflect from buildings, vehicles, trees, and other objects. This causes random amplitude, delay, Doppler effect, and interference of signal with itself

There are two variations of the RadioML dataset - RadioML 2016.10a and RadioML 2016.10b. They differ in size, and RadioML 2016.10b doesn't have an "AM-SSB" class.

4.2.2 Matlab dataset

Also, we decided to generate a dataset based on the code provided in the Matlab documentation example of Deep Learning Toolbox [50]. Similarly, a square-root raised cosine pulse shaping filter is applied to the modulated signal. Then Rician multi-path channel simulation is used to simulate multi-path fading. Frequency offset, sampling time drift, and Gaussian noise are also added to the signal. The sampling rate is 2×10^5. Two variations of the Matlab dataset were generated: with SNR in the range [-30:29] and [0:29], to understand the impact of different noise levels in the dataset on model performance. A comparison of RadioML and Matlab datasets can be found in Tab. 4.1.

Parameter Name	Parameter Value (RadioML)	Parameter Value (Matlab)
Modulations	BPSK, QPSK, 8PSK, 16QAM, 64QAM, BFSK, CPFSK, PAM4, WB-FM, AM-SSB, AM-DSB	BPSK, QPSK, 8PSK, 16QAM, 64QAM, PAM4, GFSK, CPFSK, B-FM, AM-DSB, AM-SSB
SNR Range	[-20dB : 18dB : 2dB]	[0/-30dB : 29dB : 1dB]
Samples per frame	128	1024
Sampling frequency	1×10^6	2×10^5
Signal format	In-phase and quadrature(IQ)	In-phase and quadrature(IQ)
Number of frames	220K (a), 1200K (b)	330K

TABLE 4.1: Datasets comparison

4.3 CNN

As mentioned in Chapter 2, Convolutional Neural Networks (CNNs) are deep neural networks that have shown remarkable performance in many problems. One of the main features of CNNs is their ability to learn hierarchical representations of data. This is achieved by applying convolutional filters to the input data, each capturing increasingly complex patterns [53]. In signal processing, CNNs can extract features from signals, such as the time-varying waveform of a radio signal. The hierarchical nature of CNNs is particularly useful in this context, as it allows the network to learn features at multiple scales, capturing fine and coarse signal details. Filters can detect patterns regardless of their location in the input (shift invariance) [66].

Input data has shape (128/1024, 2) – 128 samples per frame for the RadioML dataset, and 1024 for the Matlab dataset, respectively, each consisting of two numbers – In-phase and quadrature components. IQ components are interpreted as channels, and the input dimension for the first convolution layer is 128. The model architecture consists of `N` consecutive Convolution layers and a Dense layer at the end for classification. Some of the convolutional layers are followed by Max pooling to reduce the dimensionality of the input representation, making the network more efficient by reducing the number of parameters and computations required [28]. Batch Normalization is used after each convolutional layer. It provides several benefits, including increased training speed, reduced sensitivity to the initial weights, and improved generalization performance. Also, it can act as a regularizer, reducing overfitting by adding noise to the activations of each mini-batch [36]. ReLU is used as an activation function between layers. As mentioned in the section 4.5, some hyperparameters were tuned. Initial CNN (`CNN_1`) from which other CNNs are derived is depicted in Figure 4.3.

4.4 Transformer based model

The transformer neural network architecture has revolutionized the field of NLP [59], but it has also been successfully applied in other domains, including signal processing. The transformer neural network architecture consists of an encoder and a decoder, where each layer of the encoder and decoder consists of multi-head attention and fully connected layers. Multi-head attention allows the model to simultaneously attend to different parts of the input signal. This is useful in signal processing since it enables the model to capture complex temporal relationships between signal parts. The transformer architecture also has the benefit of being highly parallelizable [75], which makes it well-suited for hardware acceleration.

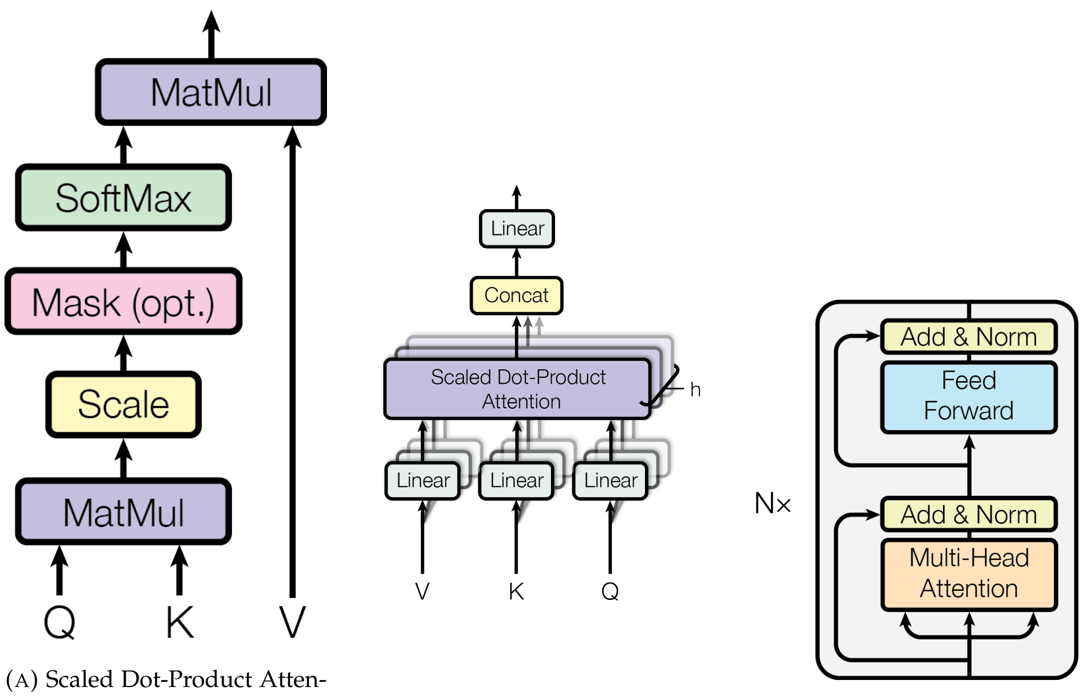
In this thesis, we use only the encoder part of the transformer, so the transformer model is also referenced as "encoder" in this thesis. The reason is that purpose of the encoder is to extract important features from the signal, just like convolutional layers. The encoder layer consists of multi-head attention and a fully connected layer. There are two residual connections connecting multi-head attention output with layer input and output, depicted in Figure 4.1c. Multiple encoder layers are stacked to extract more features.

Multi-head attention was proposed in "Attention is all you need" [75]. The input sequence is transformed into three smaller vectors: query, key, and value, see Fig. 4.1b, by projecting them to lower-dimensional spaces by multiplying with weight matrices. Each query vector is compared to every key vector to obtain a set of attention scores that are scaled and normalized 4.1a. These attention scores are then used to weigh the value vectors. Weighted vectors are concatenated and transformed back to the original dimensionality. In the original paper [75], positional embedding is used to encode positional relationships between samples. But, as explained in [86], there is no need for positional embeddings when working with wireless signals since it already contains location information.

The following neural network architecture was used: one convolutional layer extracts the feature matrix and increases the channel dimension to `W`. The output of convolutional layers is fed to `N` encoder layers, and their output is fed to a small dense layer to make the classification. Dropout is used as a regularization method. Encoder is depicted in Fig. 4.2.

4.5 Training

Models were implemented and trained with the Keras library [12] based on the TensorFlow framework. Nvidia GPU (RTX 3070Ti Mobile, Ampere architecture) was used to accelerate training. CNN networks were trained for 16 epochs with a batch



(A) Scaled Dot-Product Attention (B) Multi-Head Attention (C) Transformer encoder

FIGURE 4.1: Core parts of transformer model [75]

size of 256 on RadioML 2016.10a. Transformers were trained for 40 epochs with a batch size of 128 on the same dataset. A smaller dataset subset (RadioML 2016.10a) was used to select the best hyperparameters and RadioML 2016.10b and Matlab were used to train the best models. Adam optimizer with a base learning rate of 0.001 was used as the optimizer algorithm. For models with more parameters, the learning rate has to be lowered.

For convolutional networks, it was discovered that decreasing the learning rate by a factor of 10 after the 8th epoch showed better convergence of the models. Also, the transformer models tend to converge much slower, and learning rate decay or different optimizer algorithms didn't help to fix that problem. That's why they require much more epochs. The cross-entropy loss function was used for all models in this thesis.

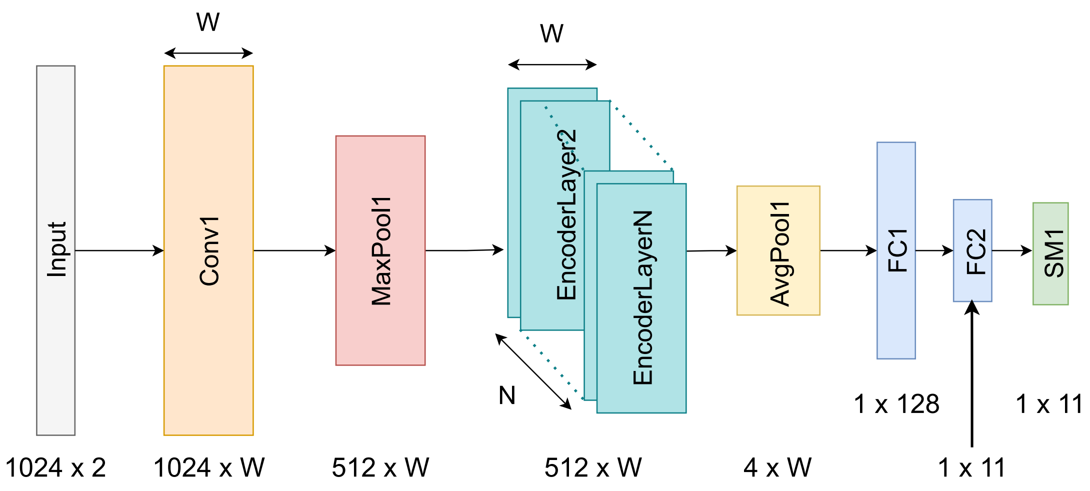
Tables depicting hyper-parameters tuning tables B.1 to B.6 have columns ”Avg Acc”, ”Max Acc”, ”Avg Acc SNR>0”, ”n_parameters”, meaning Average accuracy, Maximum accuracy, Average accuracy for SNR > 0, and the number of parameters respectfully.

4.5.1 CNN tuning

Different hyper-parameters were tested to get the best performance and speed of the network:

- Filter size – Table B.1.
- Model depth – number of CNN layers – Table B.3. The ”Output Channels” column means the number of output channels on each convolutional layer.
- Model width – number of channels in CNN layers – Table B.2.

FIGURE 4.2: Transformer Model: Matlab Dataset
W – encoder width, N – encoder depth



4.5.2 Transformer tuning

The main hyper-parameters that were tried with transformer models are:

- Number of encoder layers – Table B.5.
- Size of the filter of first convolutional layer – Table B.4.
- Size of encoder layer (size of one sample vector representation) – Table B.6.

4.6 Evaluation

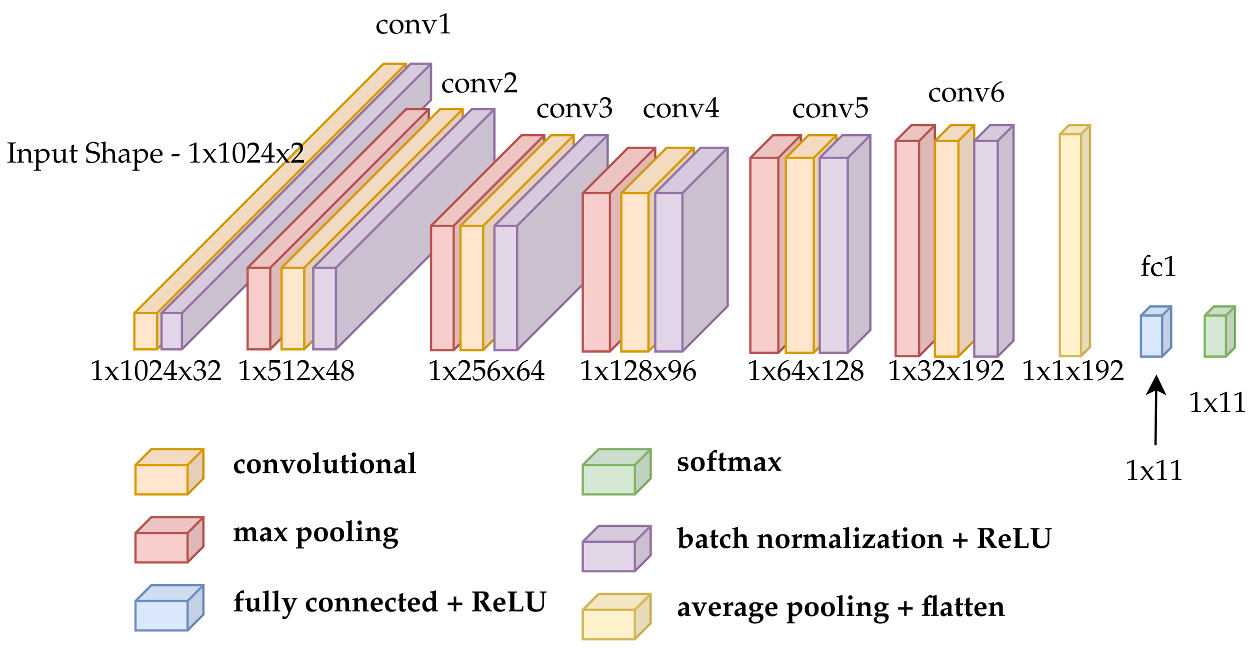
The main metric usually used in AMR is average accuracy. Also, it is very important to understand how the model performs depending on the noise level. Usually, this is shown via signal-to-noise ratio (SNR) to accuracy plot. Another essential factor for model evaluation is accuracy per class. The best CNN and encoder performance can be seen on the confusion matrices in figs. A.1 to A.4.

As shown in Tab. B.1, filter size doesn't influence performance much. But very small or big kernel sizes decrease the model's average accuracy. For CNNs, as expected, the bigger model, the better performance, unlike the encoder, which performs worse when it has depth > 6 and width > 128. From our experiments, we see that CNNs tend to overfit more than encoders.

The model size and inference time are other essential metrics in this thesis's scope. It turns out that the transformer model can achieve noticeably better accuracy than CNN with a much smaller number of parameters. For example, `ENC_3` has $\approx$ 5.5 times fewer parameters but has better (3.7%) average accuracy than `CNN_1`. But, as shown in Tab. 4.2 that compares CNN and transformer models latency and size, even though the number of parameters of the encoder is much smaller, it is still considerably slower than a corresponding CNN model on both GPU and CPU. Also, the actual size of the models is not proportional to the number of parameters.

Tab. 4.3 shows metrics of two best CNNs and two best encoders on different datasets. SNR to the accuracy of models trained on RadioML2010.10a and RadioML2010.10b is shown in figs. 4.4 and 4.5. Most models improve average accuracy by 2-6% when trained on bigger RadioML2010.10b, compared to RadioML2010.10a. An exception

FIGURE 4.3: CNN_1: Matlab Dataset



Model	Dataset	Latency, CPU	Size	n_parameters
CNN_1	RadioML2016.10a	1.39ms	1.54Mb	386K
ENC_3	RadioML2016.10a	12.56ms	0.41Mb	70K

TABLE 4.2: Models latency and size.
Latency calculated on Intel Core I7-12700H.
Models are converted to TF lite

.

is a `ENC_3`, which actually performed worse on a bigger dataset. CNNs improved more than encoders on bigger datasets. Both facts can be explained by the fact that encoders have a much smaller amount of parameters, so when trained on a bigger dataset that has more variable data, they struggle to learn to extract features from data. This hypothesis is supported by the fact that `ENC_9` has more parameters (270K vs 70K) and improved performance on a bigger dataset, unlike `ENC_3`.

Another observation on the Matlab dataset is that models performed much differently when trained on a dataset with ”good” data – SNR > 0, and noisy data – SNR in the range [-30:29]. A model trained on noisy data (Matlab[-30:29]) doesn’t perform as well on ”good” SNRs as a model trained on Matlab[0:30], which means that if it is known that the application, where the model is deployed usually features good signal-to-noise ratio, it’s worth to train a model on the dataset with better SNR. figs. C.1 and C.2 depict SNR to the accuracy of models trained on different datasets respectfully.

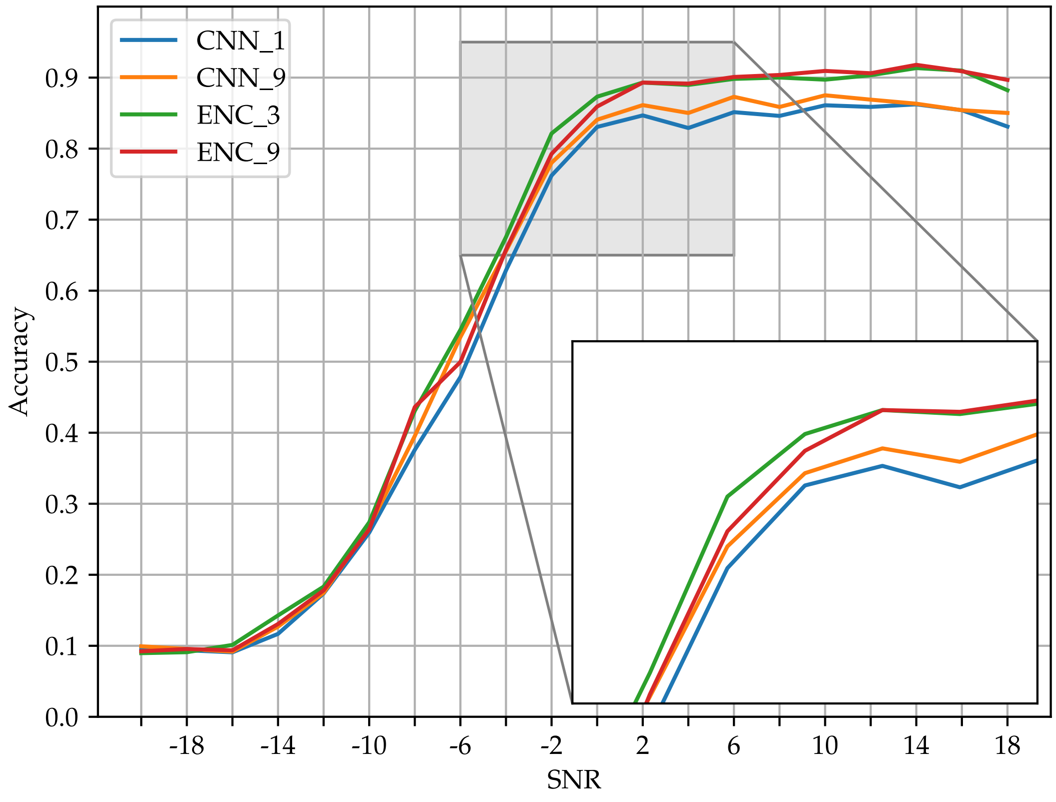


FIGURE 4.4: SNR to accuracy plot for two best CNNs and encoders on RadioML2016.10a dataset

.

4.7 Quantization

AI model quantization is a technique that aims to reduce the model's size while maintaining its performance. The process involves converting high-precision floating-point parameters to low-precision fixed-point integers or other data types. The reduced precision helps reduce the model's memory requirements, enabling it to be deployed on embedded devices with limited resources, such as smartphones, IoT devices, and microcontrollers.

One of the main benefits of quantization is improved model speed. Using lower precision data types makes the model inference time much faster. Another benefit is reduced power consumption, which is important for battery-powered devices. However, model quantization may lead to decreased accuracy due to the loss of precision.

TensorFlow Lite (TF lite) is a framework that supports quantization for neural network models. It provides tools for post-training quantization to improve model efficiency and speed. It provides three quantization approaches: dynamic range quantization, full integer quantization, and float16 quantization. Dynamic range quantization converts model weights from usual single precision floating points to 8-bit integers, but some calculations and layer outputs are still floating points. Full integer quantization removes completely floating point operations from the model.

Model	Dataset	Avg Acc	Max Acc	Avg Acc SNR>0
CNN_1	RadioML2016.10a	57.6%	86.2%	84.9%
CNN_9	RadioML2016.10a	59.0%	87.5%	86.2%
ENC_3	RadioML2016.10a	61.3%	91.3%	89.9%
ENC_9	RadioML2016.10a	61.1%	91.8%	90.3%
CNN_1	RadioML2016.10b	63.6%	93.5%	92.8%
CNN_9	RadioML2016.10b	63.7%	93.4%	92.7%
ENC_3	RadioML2016.10b	60.1%	91.6%	90.8%
ENC_9	RadioML2016.10b	63.3%	93.3%	92.2%
CNN_1	Matlab:SNR[0:29]	91.0%	93.1%	91.0%
CNN_9	Matlab:SNR[0:29]	90.5%	92.1%	90.5%
ENC_3	Matlab:SNR[0:29]	92.1%	93.7%	92.2%
ENC_9	Matlab:SNR[0:29]	90.7%	92.2%	90.7%
CNN_1	Matlab:SNR[-30:29]	47.1%	86.3%	74.9%
CNN_9	Matlab:SNR[-30:29]	48.0%	89.5%	76.7%
ENC_3	Matlab:SNR[-30:29]	49.2%	89.9%	78.2%
ENC_9	Matlab:SNR[-30:29]	48.3%	93.2%	77.4%

TABLE 4.3: Models performance when trained and evaluated on different datasets.

According to TensorFlow Lite documentation [61], the model quantized in such a way is 4x smaller and 3x+ faster with little loss in accuracy. One of the key features of TF lite is that it can apply quantization to the network trained with a regular TensorFlow, which makes model development and subsequent deployment quicker. TF lite uses a quantization algorithm implemented in [26].

Full integer quantization was chosen in this project. A comparison of the speed and accuracy of original and quantized models can be seen in Tab. 4.4.

Model	Avg Acc	Max Acc	Avg Acc SNR>0
CNN	90.63%	92.17%	90.61%
CNN + quantization	90.58%	92.22%	90.55%

TABLE 4.4: Impact of full integer quantization on model performance.

4.8 Conclusion

CNN and transformer models were developed, trained, and evaluated in this thesis project. Models were trained on two datasets – RadioML2016.10 [54] and Matlab [50]. The impact of some parameters of datasets on model performance is analyzed. Different hyperparameters were optimized in both CNNs and Transformers

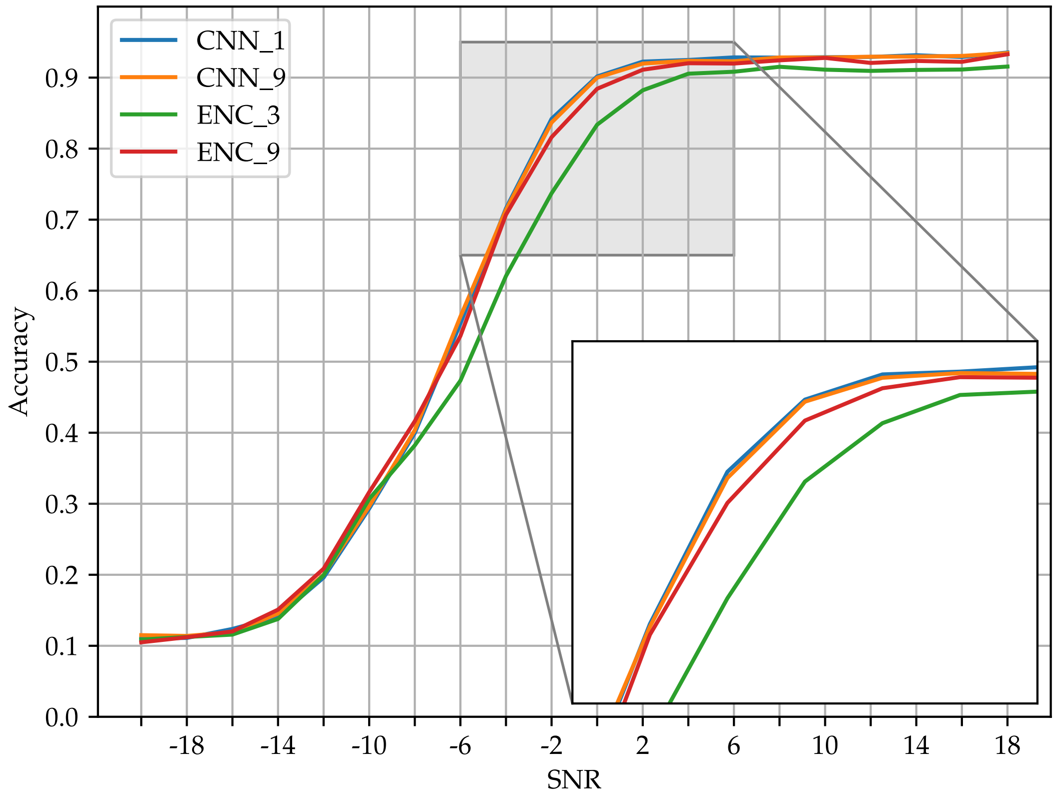


FIGURE 4.5: SNR to accuracy plot for two best CNNs and encoders on RadioML2016.10b dataset.

to get the best performances. Models are compared in terms of average accuracy, accuracy per class, and accuracy per SNR. We have achieved up to 63.6% average accuracy over the SNR range of [-20:18] and 92.8% average accuracy over the SNR range of [0:18] with a CNN trained on RadioML 2016.10b. Also, inference times and the number of parameters are compared.

It appears that transformer models, on average, perform slightly better for this task than CNN, but it is also much longer to train and evaluate it. Also, generally, CNNs are easy to scale – more parameters – better accuracy if overfit is prevented with regularization, but the configuration of the encoder should be selected more carefully.

The long inference time and the complexity of the transformers are the main reasons why we decided to focus on CNN in our work and leave the transformers for future work. `CNN_1:Matlab[0:29]` is chosen as a CNN network to accelerate in the next Chapter.

Chapter 5

Accelerator design

5.1 Accelerator development tools

5.1.1 CFU-Playground

As mentioned in Chapter 3, CFU-Playground is used to experiment with the hardware acceleration of the model deployed in TFLM. After training and evaluating the model, it should be converted to TF lite model and optionally quantized. Since TFLM doesn't assume the availability of dynamic memory must be exported as a statically sized C array.

The model is exported with some test data and expected output so that it can be tested that model with the accelerator produces correct results. CFU-Playground implements a simple Terminal User Interface (TUI) menu where users can choose what they want to run on the processor. So the developer has to add a menu entry where model inference on test data is started, and model output is compared with the expected output.

CFU extension [5] implemented for VexRiscV softcore is used to implement custom hardware logic that can be used as a regular R-format instruction. CFU can be implemented using Verilog or its extension SystemVerilog HDL and Amaranth – "Toolchain for developing hardware based on synchronous digital logic using the Python programming language" [3]. SystemVerilog was chosen since it gives more control, and it's easier to understand how Verilog code is synthesized to search for design bottlenecks.

Simulation is conducted in Renode [65] simulator. Renode itself is not a hardware simulator, it simulates ISA. This makes it much quicker, but less accurate at the same time. Also, it allows to co-simulated Verilog design in Verilator, which allows to completely simulate VexRiscV CPU in tandem with custom CFU.

5.1.2 FPGA board specification

The FPGA board used in this project is Xilinx Arty A7-100T. It is officially supported by CFU-Playground, as mentioned in https://github.com/google/CFU-Playground/wiki/Supported-Boards project Wiki on GitHub. Some of the board specifications are depicted in Table 5.1.

5.2 Model profiling

One of the advantages of TFLM is its flexibility. Since it is open-source, the developer can modify the source code to optimize the framework for their specific use case, such as adding support for custom hardware accelerators. CFU-Playground provides a simple function that gives an interface to the clock counter in soft-core.

	Arty A7-100T
Look-up Tables (LUTs)	15,850
Flip-Flops	126,800
Block RAM	4,860 KBits
DSP slices	240
DDR3L RAM	256MB, 16-bit bus @ 667MHz
Ethernet	10/100 Mbps
Internal clock	450MHz

TABLE 5.1: Arty A7-100T specification according to official distributor

Since clock frequency is constant in this processor, clock counters can be directly mapped to execution time. By default, CFU-Playgtound measures clock counts for each layer of the model. But, if needed, the user can measure how much clock counts are spent on some particular piece of code to find hotspots. We decided to profile the CNN model with 6 convolutional layers (`CNN_1`), depicted in 4.3. For example, Table 5.2 depicts clock cycles spent in each layer. Obviously that the absolute majority of time is spent in a 2-dimensional convolution.

Layer name	Cycles share (Simulation)	Cycles share (Hardware)
CONV_2D	99.310%	99.416%
MAX_POOL_2D	0.672%	0.563%
SOFTMAX	0.002%	0.004%
AVERAGE_POOL_2D	0.010%	0.011%
RESHAPE	$2.5 \times 10^{-4}\%$	$9.4 \times 10^{-4}\%$
FULLY_CONNECTED	0.005%	0.005%

TABLE 5.2: Cycles spent in different layers of CNN_1

Listing 5.1 depicts the pseudo-code of Conv2D in TFLM that doesn't consider implementation details. Most of the time is spent in the inner loop that calculates the accumulator – this is a hotspot of this model.

LISTING 5.1: 2-dimensional convolution pseudo code

```
Conv2D() {
  for (int batch in 0..batches) {
    for (int out_y in 0..output_height) {
      for (int out_x in 0..output_width) {
        for (int out_channel in 0..output_depth) {
          int32_t acc = 0;
          for (int filter_y in 0..filter_height) {
            for (int filter_x in 0..filter_width) {
              for (int in_channel in 0..filter_input_depth) {
                int8_t input_val  = input_data[Offset(...)];
                int8_t filter_val = filter_data[Offset(...)];
                acc += filter_val * (input_val + input_offset);
              }
            }
          }
          acc += bias_data[Offset(...)]
          acc = postprocess(acc);
          output_data[Offset(...)] = cast<int8_t>(acc);
        }
      }
    }
  }
}
```

5.3 Experiments

5.3.1 Software optimizations

A model can be optionally quantized. As shown in Tab. 4.4, quantization doesn't degrade CNN_1 performance much, so there is no actual need for using an original model with floating point weights in this case. Nonetheless, some models may be worsened more by quantization [67], so it worth understand the speed gain from quantization. As shown in Fig. 5.1, the speedup is ∼8 times in simulation and ∼10.3 times in hardware.

We start with software optimizations. Since we plan to optimize only this deployed model – CNN_1, we can remove some degree of generalization in Conv2D implementations. For example, we know that even though a 2-dimensional convolution layer is used for the model, data is actually 1-dimensional. That's why we can remove two cycles – over filter's y coordinate and input's y coordinate. Also, some convolution parameters are constant in this model, such as filter_width, min/max_activation, etc. CFU-Playground provides convenient functions to detect if some variable changes from call to call of the layer. Other optimization techniques, like loop unrolling, can produce considerable speed up, as shown in Figure 5.1. Overall we managed to speed up code by 26.4% in simulation and 34.7% in hardware with just code simplification and optimization. Note that most optimizations consider specifics of this concrete model and can't be applied in general cases. As shown in Figure 5.8, software acceleration accelerates inference time from 24.9s to 16.2s on the FPGA board.

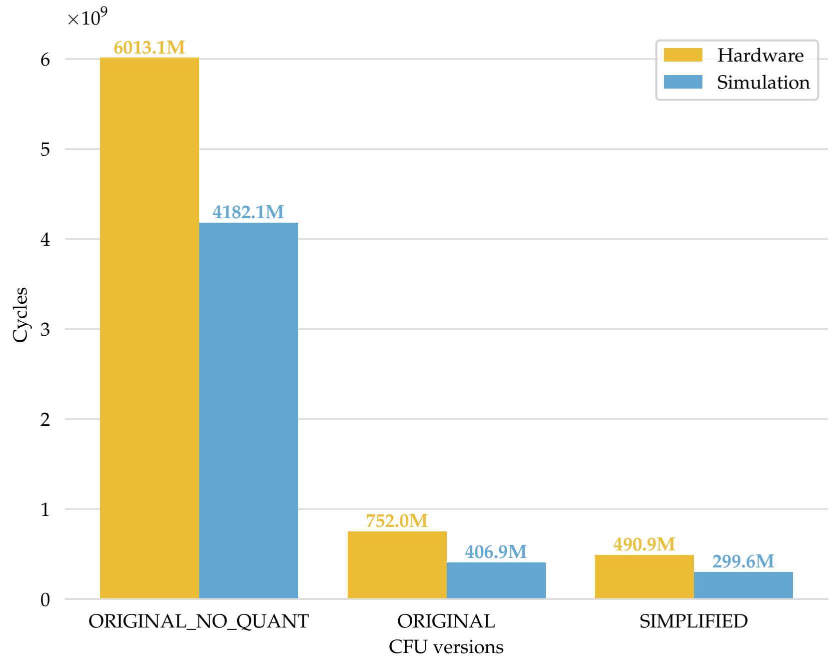


FIGURE 5.1: Software optimizations of CNN_1.

5.3.2 Software CFU

Designing and developing an efficient custom accelerator for neural networks requires careful consideration of many factors. It is almost impossible to design an optimal accelerator immediately without testing, debugging, and fine-tuning. That's why iterative development can be a more effective approach. By starting with a basic design and testing it, we can identify bottlenecks and inefficiencies and improve the design. This approach allows for faster development, and the resulting accelerator can better match the application's needs.

That's why we start from the naïve implementation that is easy to implement and is very similar to software implementation. Such an accelerator is quick in the simulator but impossible to synthesize for obvious reasons. Then we can iteratively add new features to the design to make it more effective regarding resource utilization and timing constraints. Timing constraints are limitations imposed by signals that must be propagated between flip-flops through some combinational logic.

CFU access interface in CFU-Playground is provided by 8 `cfu_op0-7` functions for each `funct3` on Figure 2.4. Each such function has 3 inputs – 7-bit number to encode command (`funct7`), and two 32-bit inputs: `rs1` and `rs2` on Figure 2.4. Such API allows one to first implement CFU logic in software, which is considerably quicker and easier than designing it immediately with HDL. Converting such software to CFU is easier since SystemVerilog provides mathematical primitives similar to those in the C++ programming language used to write programs in CFU-Playground.

5.3.3 CPU-CFU interface

CPU-CFU interface and communication protocol were mentioned in Chapter 2. There are different ways to design CFU behavior based on this protocol. For example, CFU can be completely combinatoric; it can stall the CPU while executing; It can be asynchronous, with the possibility of making the CPU wait for the result, etc. More cases with timing diagrams are described in the official CFU-Playground documentation: https://cfu-playground.readthedocs.io/en/latest/interface.html. It was decided to make CFU completely asynchronous, e.g., always returning something to the CPU. This enables better potential CPU utilization, and if the CPU wants to get the CFU computation result, it can check the computation status and, if it is done, get a result. Verilog code that implements such protocol and abstracts CFU logic from the interface depicted below, Listing 5.2.

LISTING 5.2: CFU interface

```verilog
module Cfu (
    input                   cmd_valid,
    output                  cmd_ready,
    input          [ 9:0]   cmd_payload_function_id,
    input          [31:0]   cmd_payload_inputs_0,
    input          [31:0]   cmd_payload_inputs_1,
    output reg              rsp_valid,
    input                   rsp_ready,
    output wire    [31:0]   rsp_payload_outputs_0,
    input                   reset,
    input                   clk
);
  wire output_valid;

  wire [6:0] funct7 = cmd_payload_function_id[9:3];
  wire [31:0] ret;

  conv1d CONV_1D (
      .clk(clk),
      .en(cmd_valid),
      .cmd(funct7),
      .inp0(cmd_payload_inputs_0),
      .inp1(cmd_payload_inputs_1),
      .ret(rsp_payload_outputs_0),
      .output_valid(output_valid)
  );

  assign cmd_ready = ~rsp_valid;
  always @(posedge clk) begin
    if (reset) begin
      rsp_valid <= 1'b0;
    end else if (rsp_valid) begin
      // Zero out rsp_valid, if CPU confirmed it has read resutls
      rsp_valid <= ~rsp_ready;
    end else if (cmd_valid) begin
      rsp_valid <= output_valid;
    end
  end
endmodule
```

5.3.4 Accelerator building blocks

The main idea of the accelerator developed in this thesis is to intervene in a convolution code, and ”outsource” computations to the CFU. The majority of computations in convolution is multiply-accumulate. All further developed accelerators consist of 3 main parts: input/filter buffers, parameters registers, and computation logic.

Memory buffers

Buffers are required since, as mentioned in Chapter 2, CFU doesn't have permission to access the CPU's RAM. So processing data must be copied to some buffers accessible by CFU. This is a huge performance limiter, but the relative simplicity of CFU development may justify it in some applications. For Xilinx FPGA boards, it is worth utilizing block RAM (`https://docs.xilinx.com/r/en-US/am007-versal-memory/Block-RAM-Summary`). This memory is quick but very limited in size. One of the main features of block RAM is that it has 1 clock edge delay, making latency extremely small. As shown in Tab. 5.1, Arty A7-100T has 4860Kbits of block RAM.

Parameter registers

Even though many parameters are constant due to the architecture of the accelerated network, many parameters need to be saved to registers inside CFU so that they can be effectively used in calculations. So different `funct7` opcode should encode what parameter to save. Such parameters are `input_offset` – is added to input value when calculating convolution, `input_depth` – number of input channels, used to calculate effective buffer size of calculations, `output_activation_max`, `output_activation_min` – used for quantization, etc.

Computation logic

This is a core part of the accelerator. This is where most of the changes will occur. Basically, convolution requires only 2 operators – multiplication and addition. As mentioned in 4.7, full integer quantization is used, so the accumulation result must be squeezed back to an 8-bit integer using a particular algorithm is used in TFLM. It requires different logical operations, such as right/left bit shift, logical `and`/`or`, division. Different approaches to accelerating will be described in the next section.

5.3.5 Iterative development

Naive accelerator CFU_V1

The first accelerator `CFU_V1` works in the following way. Input is completely copied to CFU's input buffer beforehand. In convolution code, there is an iteration over output channels. For each such channel, input is convolved with a filter of size (8 x `input_channel`). So, a filter is copied to CFU's filter buffer for each output channel. After that, computation is started – CFU iterates over `output_x`, over `filter_x`, and saves all accumulators to the output buffer. CPU reads these results, quantizes them, and saves them to the output array. As shown in Figure 5.7, this design is the quickest and speeds up code by ∼40.5 times (relative to the original convolution code) in the simulator. Unfortunately, this design is completely unsynthesizable for a few reasons:

- Input buffer is of size 1024 x 128, and has width of 8-bits. 1024 – number of samples in the input frame, 128 – maximum number of input channels in the model. Filter buffer is of size 8 x 128 – 8 width of the filter. The output buffer size is 1024. Summing up, this design requires (1024 * (128 + 8 + 1)) * 8 bits

= 1.12Mbits, which is more than 23% of all theoretically available block RAM on the FPGA 5.1 we use in this project. This design is non-synthesizable in practice since some memory is already used in the soft-core.

- All computations are done in 1 clock cycle. According to Figure 4.3, the maximum effective size of the input is (1024 x 32). So that is 1024 * 32 * 8 = 262144 multiplications and 262144 * 2 additions of 32-bit integers per cycle, which is impossible even with modern ASICs due to the timing restrictions.
- Since data for computations is read from block RAM, that's also a 262144 * 2 accesses to memory per cycle, making this design impossible not only because of timing constraints but also resource constraints.

CFU_V2

One way to fix the problems mentioned in the previous subsection is to decrease the input buffer size. Since the input buffer can't be copied fully to the CFU's buffer, it has to be copied in a cycle in a CPU code. This way, iteration over output x is moved to the code, and CFU computes only one convolution of filter and slice of input of size up to $8 \cdot 128$. Also, since iteration over output x is moved to the code, this eliminates the output buffer. Only one 32-bit register is required to accumulate convolution.

CFU_V3

The next upgrade is to make the input buffer a ring buffer. This decreases copied data by a factor of 8 since only one row is copied instead of 8 for each iteration over output x. This decreases the number of clock cycles for inference. Also, `CFU_V3` divides computation into multiple clock cycles. After the CPU starts computation, CFU moves to the "computational state", and at each clock cycle, accumulates, multiplies, and adds 8 values from the input and filter buffer to the accumulator. This fixes timing constraints but decreases inference speed by a lot. That is why `CFU_V3` is noticeably slower. Eight multiply-accumulate was chosen arbitrarily, and later, the number of computations per cycle will be examined in greater detail.

Unfortunately, `CFU_V3` is not synthesizable. The reasons are:

- Extensive usage of blocking assignments. Blocking assignments in Verilog guarantees the order execution of lines of code. This requires much more hardware to be implemented. The main reason for blocking assignments usage is that in the `CFU_V3` implementation address is updated in the same cycle as an accumulator. This requires a guarantee that address is finished updating before computation begins.
- Extensive usage of division remainder. This was convenient for the ring buffer, but the remainder division is a relatively slow operation, which makes timing constraints much worse.

CFU_V4

Blocking assignments can be removed by adding a new CFU state to update the address. So, while CFU is in a "computational state", the multiply-accumulate (computation) is done during the first clock edge. During the second clock edge, addresses for current input and filter values in buffers are updated. Division remainder can be deleted by conditional updating – if the input buffer address is bigger than the effective input buffer size, it is set to zero. Such logic requires much less hardware for synthesizing since if-statement is synthesized into multiplexers.

`CFU_V4` is, in fact, synthesizable, and it correctly works on the FPGA. Acceleration can be seen in Figure 5.7 – ∼7.83x (relative to the "original" quantized convolution code).

CFU_V5

LISTING 5.3: Quantization pseudocode

```
1  int32_t
2  rounding_divide_by_POT(int32_t x, int32_t exp) {
3      int32_t mask = (1ll << exp) - 1
4      int32_t remainder = x & mask
5      int32_t thresh = (mask >> 1) + (((x < 0) ? ~0 : 0) & 1)
6      return (x >> exp) + (((remainder > thresh) ? ~0 : 0) & 1)
7  }
8
9  int32_t
10 saturating_rounding_doubling_high_mul(int32_t a, int32_t b) {
11     bool overflow = a == b && a == -2 ^ 31
12     int64_t ab_64 = int64_t(a) * int64_t(b)
13     int32_t nudge = ab_64 >= 0 ? (1 << 30) : (1 - (1 << 30))
14     int32_t ab_x2_high32 =
15         cast<int32_t>((ab_64 + nudge) / (1ll << 31))
16     if overflow:
17         return 2 ^ 31 - 1
18     else:
19         return ab_x2_high32
20 }
21
22 int32_t
23 quant(int32_t acc, int32_t mult, int32_t shift) {
24     int32_t left_shift  = max(shift, 0)
25     int32_t right_shift = min(-shift, 0)
26     return rounding_divide_by_POT(
27         saturating_rounding_doubling_high_mul(
28             acc * (1 << left_shift), mult),
29         right_shift
30     )
31 }
```

Next upgrade – move post-processing to the CFU. Post-processing consists of adding bias to the accumulated value and fitting it in an 8-bit integer (quantization). The pseudocode for how the accumulator is quantized is depicted in Listing 5.3. `CFU_V5` implements bias and quantization post-processing as combinatorial logic. Unfortunately, such design doesn't meet timing constraints. Profiling has shown that the reason is the multiplication of two 64-bit numbers (Listing 5.3, line: 10).

Timings problems usually are solved via "Pipelining.", e.g., an operation is split into smaller parts executed during a few clock cycles. It was already applied in `CFU_V3` when computation was split into many clock edges. `CFU_V5.1` implements multiplication pipelining in post-processing. As expected, such CFU is a bit slower compared to `CFU_V5.0`.

CFU_V6

The next upgrade involves utilizing CFU input size. There is a lot of data copying to CFU's buffers. Since the model is quantized, input and weights are quantized, e.g., have a size of 8 bits. All previous CFUs were copying one 8-bit value at a time, but CFU inputs have a size of 32 bits. That means that theoretically, data copying can be accelerated four times. The same data layout inside CFU's buffers and input/filter data simplifies the implementation of this feature.

Unfortunately, not all data in this network is aligned on 32 bits. Firstly, this was fixed by adding a special register that configures how many bytes (1-4) should be written to the buffer simultaneously. This solution requires little accelerator and convolution code changes, but it is not synthesizable because it requires too many LUT tables.

Conditional write size was removed in `CFU_V6.1`, but there was still a possibility to write to any address (not multiple of 4). In other words, when writing 32bit input to address `addr`, 4 consecutive bytes are written to memory addresses `addr`, `addr + 1`, `addr + 2`, `addr + 3`. This approach requires some convolution code changes and still requires a lot of FPGA resources, as shown in Figure 5.5, but it is synthesizable. This is one of the most effective accelerating features giving 1.55x speed up relative to the previous version and 13.9x from the original quantized model.

CFU_V7

A feature introduced in `CFU_V7` is asynchronous memory copying. It is possible to copy data to the CFU buffer while it is still doing computation. This can be achieved by increasing the input buffer from 8x128 to 9x128. While CFU convolves up to 8x128 values from input with filter, CPU can write next row. This eliminates the need for synchronization since such implementation has no critical sections, and the input buffer size increase is not noticeable. Unfortunately, this improvement gives a minor speedup – 5.5% in simulation.

CFU_V8

Resources utilization for `CFU_V6` and `CFU_V7` increased dramatically. A probable reason is that since 4 values are written simultaneously, 4-port memory is synthesized, which requires considerably more FPGA resources. In `CFU_V8`, 4 port memory with a width of 8 bits is replaced with 1 port memory with a width of 32 bits. This decreased FPGA resource utilization dramatically but made the accelerator a bit slower compared to `CFU_V7` – 4% in simulation. The final design's functional diagram is depicted in Figure 5.2, and the state machine in Figure 5.3.

5.4 Results

5.4.1 Computations per cycle

As mentioned above, the computation stage is pipelined. It can be configured how many multiply-accumulate operations are conducted during one cycle. Unfortunately, speeding up is not linear, as shown in Figure 5.4. The reason is probably data copying, which makes the problem more I/O bound as the speed of accelerator computation grows.

FIGURE 5.2: CFU_V8.0 Functional diagram

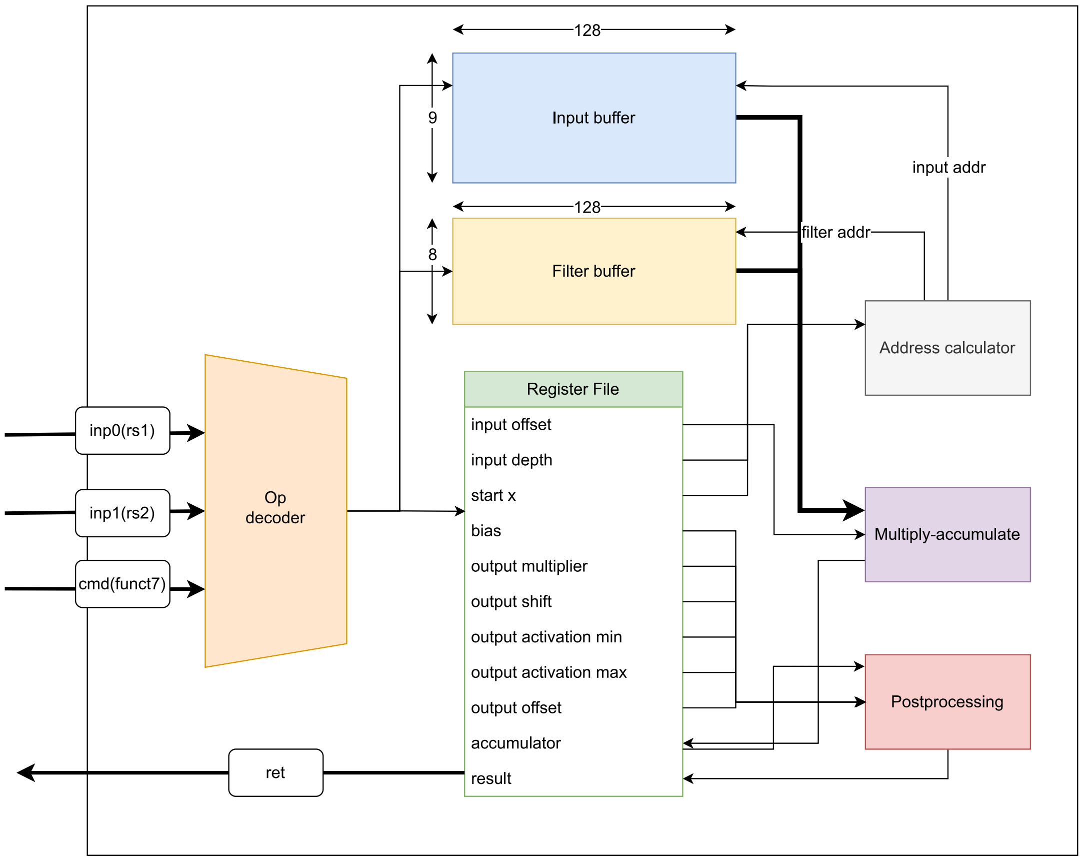
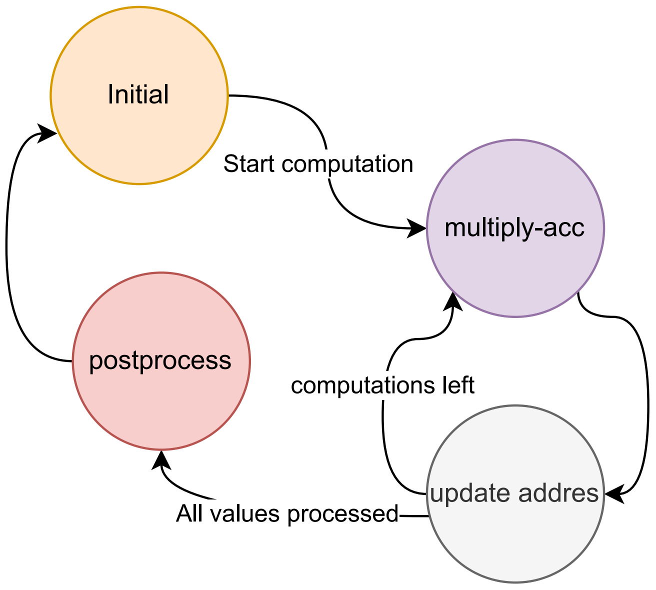


FIGURE 5.3: CFU_V8.0 State machine



5.4.2 Resource utilization

The main resources of Arty A7-100T are Lookup Tables (LUTs), Flip-Flops (FFs), BRAM – block RAM, LUT RAM, and DSP blocks – Digital Signal Processing block, as shown in Fig. 2.6. Lookup Table is a core part of any FPGA and is basically SRAM

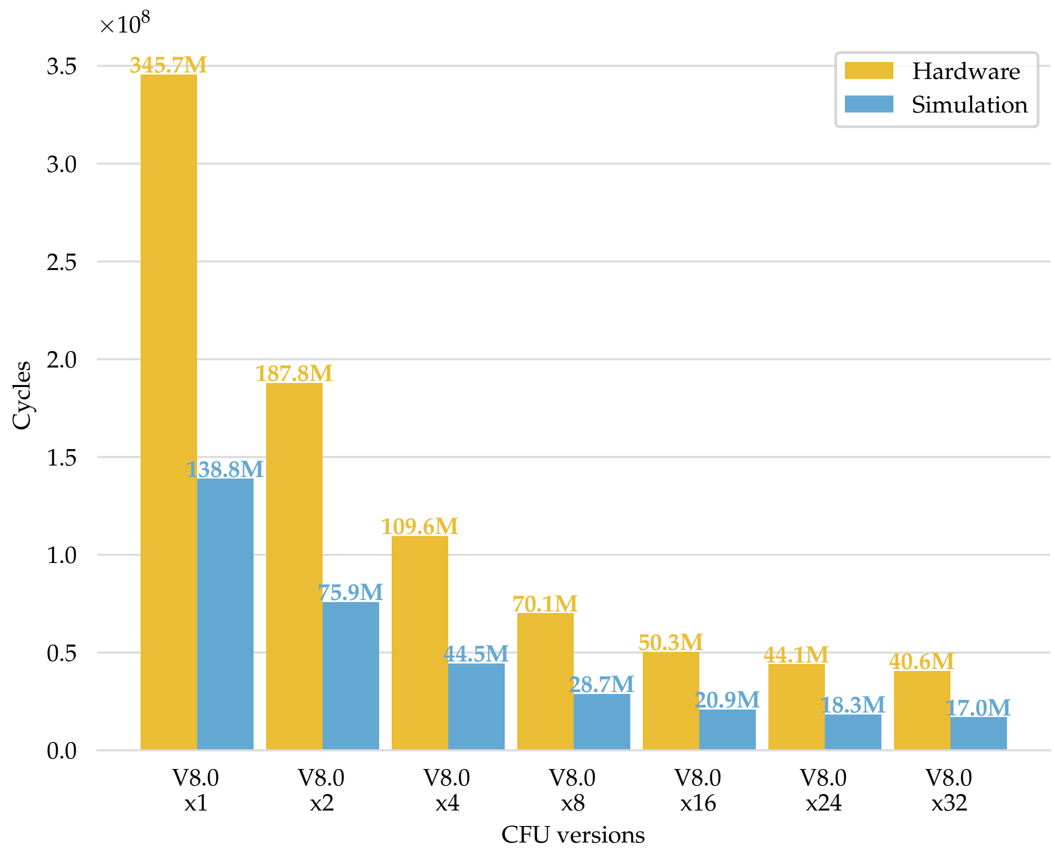


FIGURE 5.4: Scaling of CFU V8.0 – multiply-accelerate per clock cycle

combined with a multiplexer that can be programmed to implement any boolean function. Flip-Flops are building blocks of any asynchronous logic since they allow the design to store state. LUT RAM – a way to add more memory to FPGA design by utilizing SRAM that is used for LUTs programming. Finally, DSP – is a block that can implement multiply, multiply-add, and multiply-accumulate functions.

As expected, LUT usage and DSP usage grow with the scaling of `CFU_V8.0`. Since most operations in convolution are multiply-accumulate, DSP utilization is growing quicker than other resources. Clear outlier are `CFU_V6.1` and `CFU_V7.0`. As mentioned in 5.3.5, the reason is writing 4 values simultaneously to the input and filter buffers. This requires 4-port memory, which is resource intensive design. Plenty of available resources are left, leaving much room for future improvements.

5.4.3 Power consumption

Figure 5.6 depicts different CFU versions of power consumption in Watts. Different designs consume different amounts of power. Clearly, there is a correlation between the complexity of CFU design and power consumption. Just like resource utilization subsection 5.4.2, `CFU_V6.1` and `CFU_V7.0` are outliers due to non-optimal block RAM utilization.

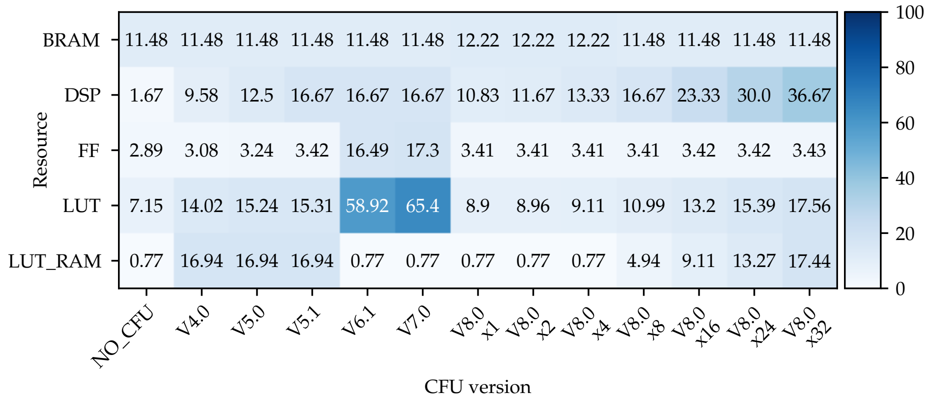


FIGURE 5.5: Resource utilization, %.

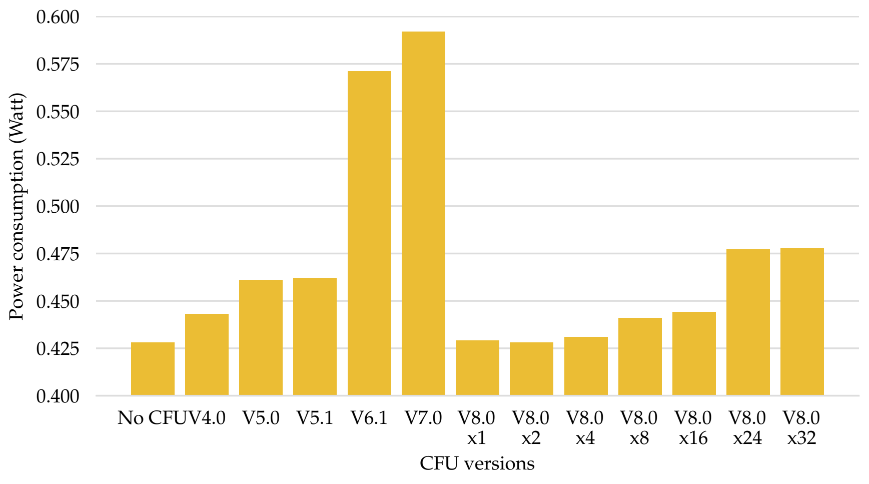


FIGURE 5.6: Power consumption of different CFU versions.

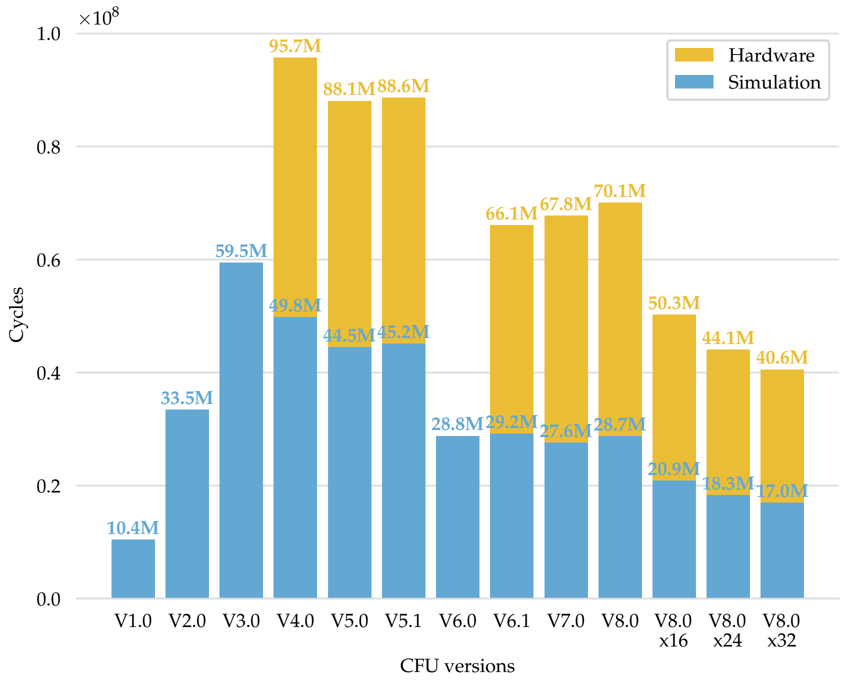


FIGURE 5.7: Evolution of CFU accelerator – Cycles per inference of CNN_1.

5.4.4 Speedup

Software optimization can give a noticeable inference time speedup. Figure 5.1 shows that quantization gives a huge performance boost (x8 in hardware and x10.3 in simulation). Other software optimizations of the quantized model gave an additional x1.53 speed up for hardware inference.

The iterative development approach was chosen to implement the CFU accelerator for the CNN model. Figure 5.7 depicts progress in terms of clock cycles, and Figure 5.8 depicts progress in inference time. The system clock was 30MHz when calculating inference time in seconds. Best CFU accelerates model inference 24 times in the simulator and 18.5 times in hardware in terms of clock cycles, compared to the original quantized model. If compared with an original model without quantization, then the speed up is x148 times. Inference time was reduced from 200.66s / 24.94s (without/with quantization) to 1.35s.

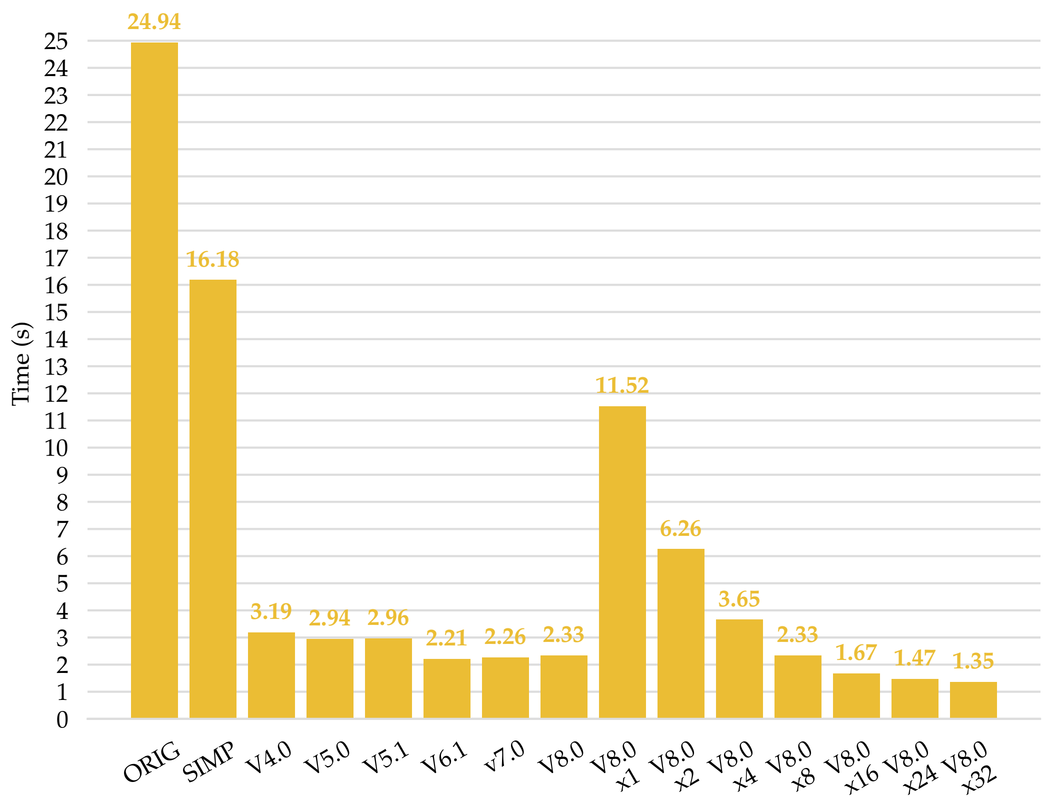


FIGURE 5.8: CNN_1 Inference time with different
"ORIG" stands for original, "SIMP" for simplified convolution code.

Chapter 6

Conclusions and Future work

6.1 Conclusions

In this work, we explore the Automatic Modulation Recognition problem. The problem is researched, and modern classical and deep learning solutions are explored. We implemented a convolutional neural and transformed-based neural network to solve this problem. Models are trained, fine-tuned, and evaluated on two different datasets [54, 50]. The impact of different characteristics of datasets and model hyper-parameters and quantization on performance is analyzed. After training and evaluating, one configuration of the CNN model was chosen as the target model for acceleration.

This work gave an overview of existing hardware accelerators for AI applications. One such approach – Custom Function Unit extension for RISC-V soft-core processor implemented in FPGA was chosen to accelerate Neural Networks for inference on resource-constrained embedded devices. The pros and cons of such an approach are shown, and all parts, such as RISC-V architecture, FPGA, HDLs, Hardware simulation, and synthesis tools, are reviewed.

CNN was quantized and exported as Tensorflow Lite micro model and integrated into the CFU-Playground framework. After that, we profile the model and find the bottleneck – convolutional layer. Implementation of a convolutional layer in TFLM is reviewed and changed to integrate with the accelerator. Accelerator was developed iteratively; all iterations are described. The final design is depicted in Fig. figs. 5.2 and 5.3. Experiments were conducted in simulation and hardware with the Xilinx Arty A7-100T development board. Resulting speed up, inference time, FPGA resource utilization, and power consumption are described.

6.2 Future work

There are many points worth exploring in future work. First of all, it's worth deploying and profiling the encoder model. It is more complex, but there can be more bottlenecks worth accelerating. Also, more experiments on the models should be conducted. We used the "vanilla" encoder, proposed in [75], but there are many different variations of transformer [52] architecture that can potentially work well for this problem. It is also worth trying depth-wise CNN [71, 32, 13], or other optimization methods such as pruning [79, 41], fusion [40], and other [7, 48].

The current accelerator design can be improved in terms of both resource utilization and fitting timing constraints. The latter is very important since, for experiments, the clock frequency was lowered to 30MHz, while VexRiscV works on 75MHz on board used for experiments, and the board itself supported frequency up to 450MHz. Architectural changes should also be explored. For example, the input buffer is utilized only on the biggest model layer. Theoretically, more computations per cycle can be achieved with pipelining and a more clever utilization of block RAM. Also, it is worth exploring ways to generalize the accelerator, and generalization impact on performance, since the current design makes some assumptions about model configuration. Finally, frameworks that directly synthesize neural networks to the FPGA or SOC, like hls4ml [23] and Deep Learning HDL Toolbox [64] should be compared to the method used in this work.

Appendix A

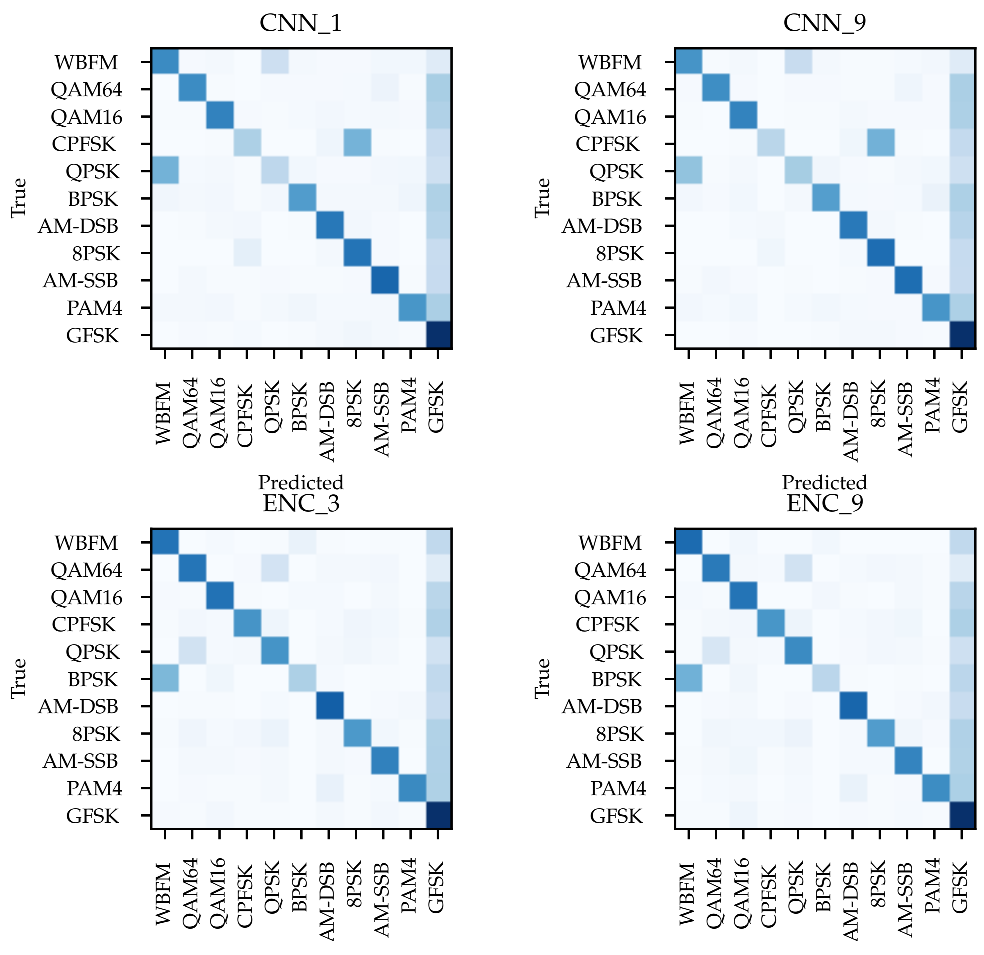


FIGURE A.1: CNN and Encoder – radioML_2016.10a confusion matrix

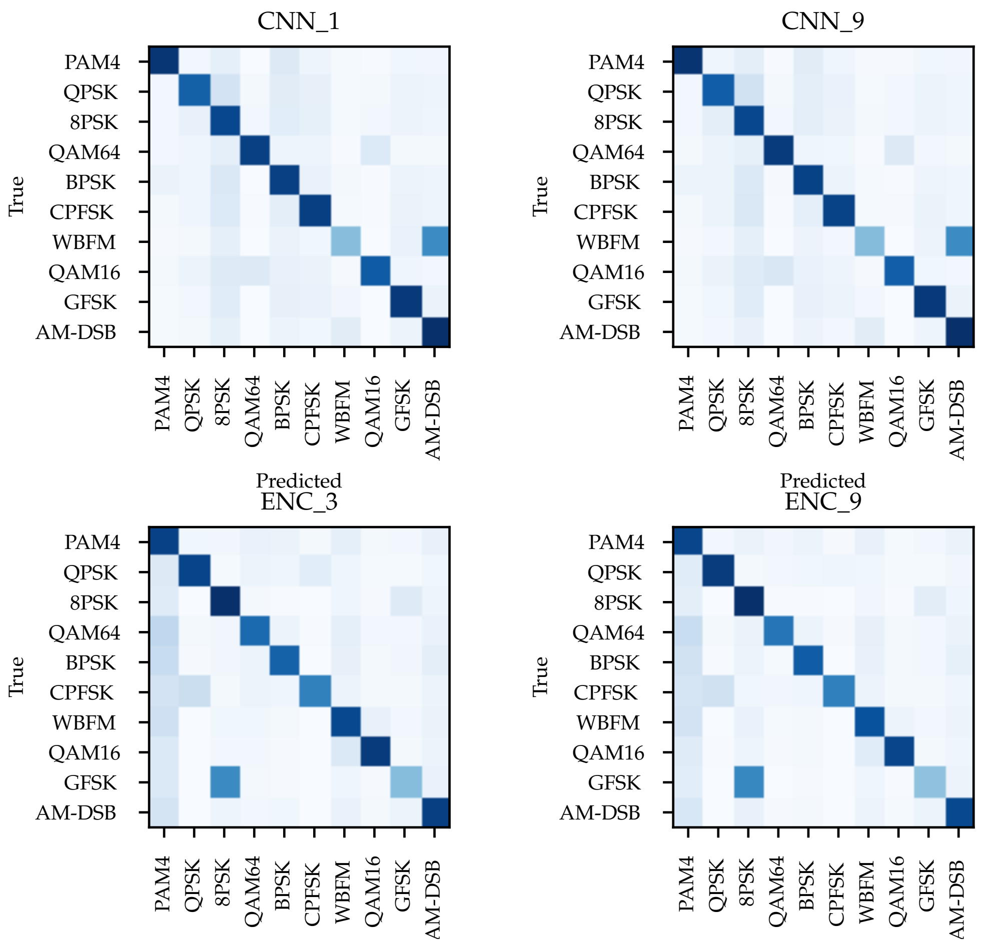


FIGURE A.2: CNN and Encoder – radioML_2016.10b confusion matrix

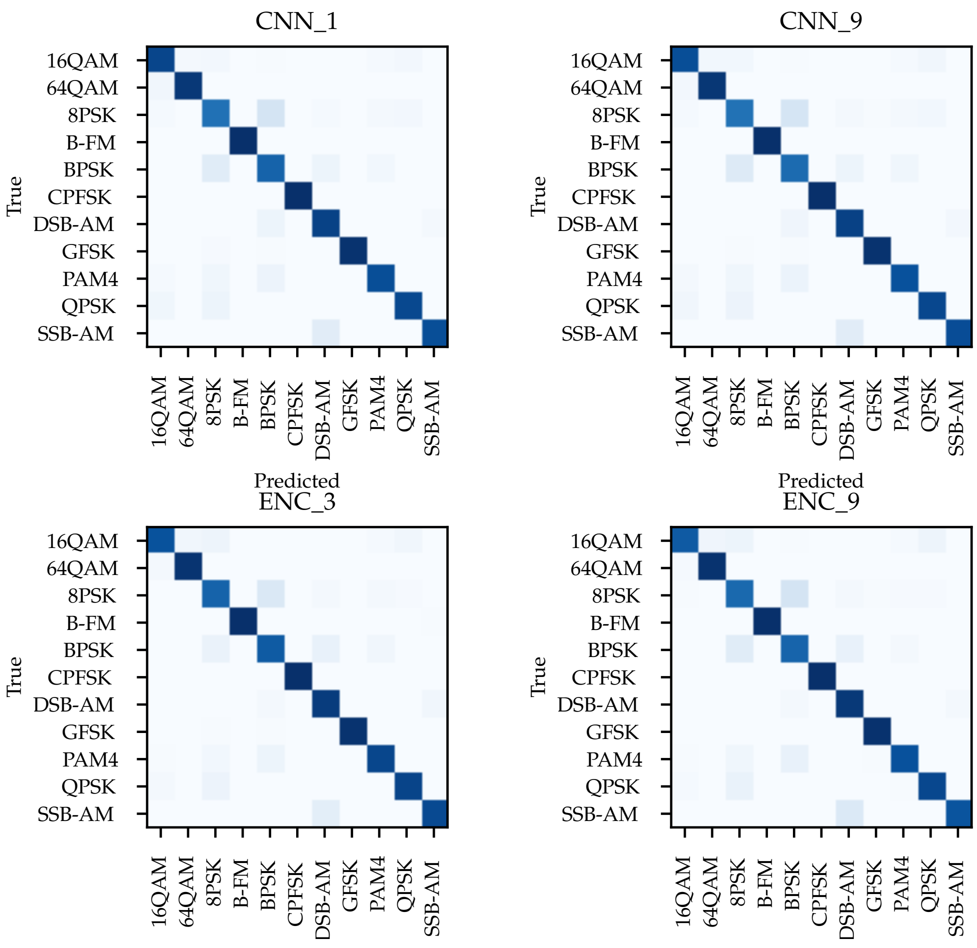


FIGURE A.3: CNN and Encoder – Matlab[0:29] confusion matrix

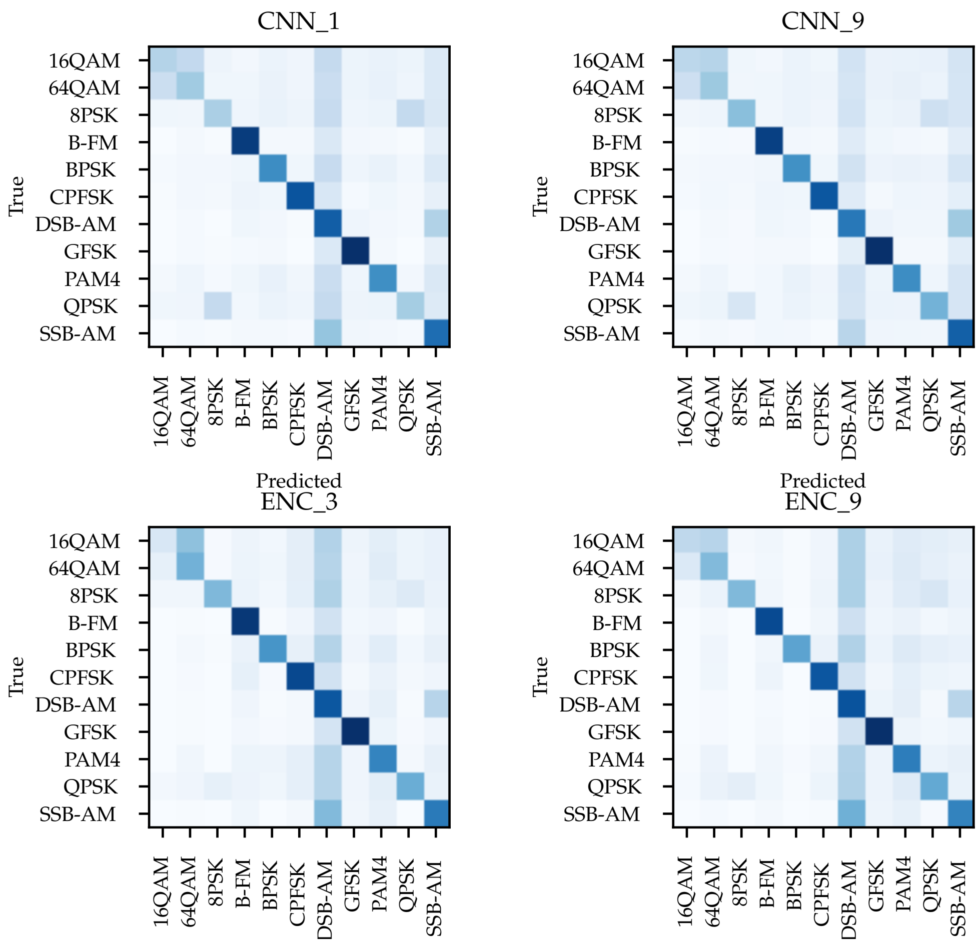


FIGURE A.4: CNN and Encoder – Matlab[-30:29] confusion matrix

Appendix B

Label	Filter Size	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
CNN_0	3	55.6%	85.0%	83.5%	147K
CNN_1	8	57.6%	86.2%	84.9%	386K
CNN_2	17	57.8%	86.2%	84.9%	815K
CNN_3	33	57.5%	85.8%	84.2%	1578K
CNN_4	65	53.7%	79.9%	77.7%	3104K

TABLE B.1: CNN: Filter size

Label	Output Channels	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
CNN_5	[16, 16, 32, 32, 64, 64]	55.6%	82.6%	80.8%	65K
CNN_6	[32, 32, 48, 64, 64, 96]	56.4%	84.9%	83.3%	130K
CNN_1	[32, 48, 64, 96, 128, 192]	57.6%	86.2%	84.9%	386K
CNN_7	[64, 64, 128, 192, 192, 256]	58.2%	86.2%	85.4%	991K

TABLE B.2: CNN: Model width

Label	Output Channels	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
CNN_8	[32, 48, 64]	51.9%	79.6%	76.5%	38K
CNN_1	[32, 48, 64, 96, 128, 192]	57.6%	86.2%	84.9%	386K
CNN_9	[32, 48, 64, 96, 128, 192, 256, 512]	59.0%	87.5%	86.2%	1835K

TABLE B.3: CNN: Model depth

Label	Filter Size	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
ENC_0	3	57.6%	88.1%	86.9%	68.0K
ENC_1	9	60.6%	90.2%	89.7%	69K
ENC_2	17	60.7%	90.6%	89.3%	69K
ENC_3	33	61.3%	91.3%	89.9%	70K
ENC_4	65	61.2%	90.5%	89.9%	72K

TABLE B.4: Encoder: Filter size

Label	Encoder Depth	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
ENC_5	2	60.0%	89.1%	88.0%	43K
ENC_1	4	60.6%	90.2%	89.7%	69K
ENC_6	6	60.6%	91.2%	89.7%	94K
ENC_7	8	60.4%	90.9%	89.3%	120K

TABLE B.5: Encoder: Encoder depth

Label	Encoder width	Avg Acc	Max Acc	Avg Acc SNR>0	n_parameters
ENC_1	32	60.6%	90.2%	89.7%	69K
ENC_8	64	60.9%	91.0%	89.9%	136K
ENC_9	128	61.1%	91.8%	90.3%	270K
ENC_10	256	60.7%	91.7%	90.1%	537K

TABLE B.6: Encoder: Encoder width

Appendix C

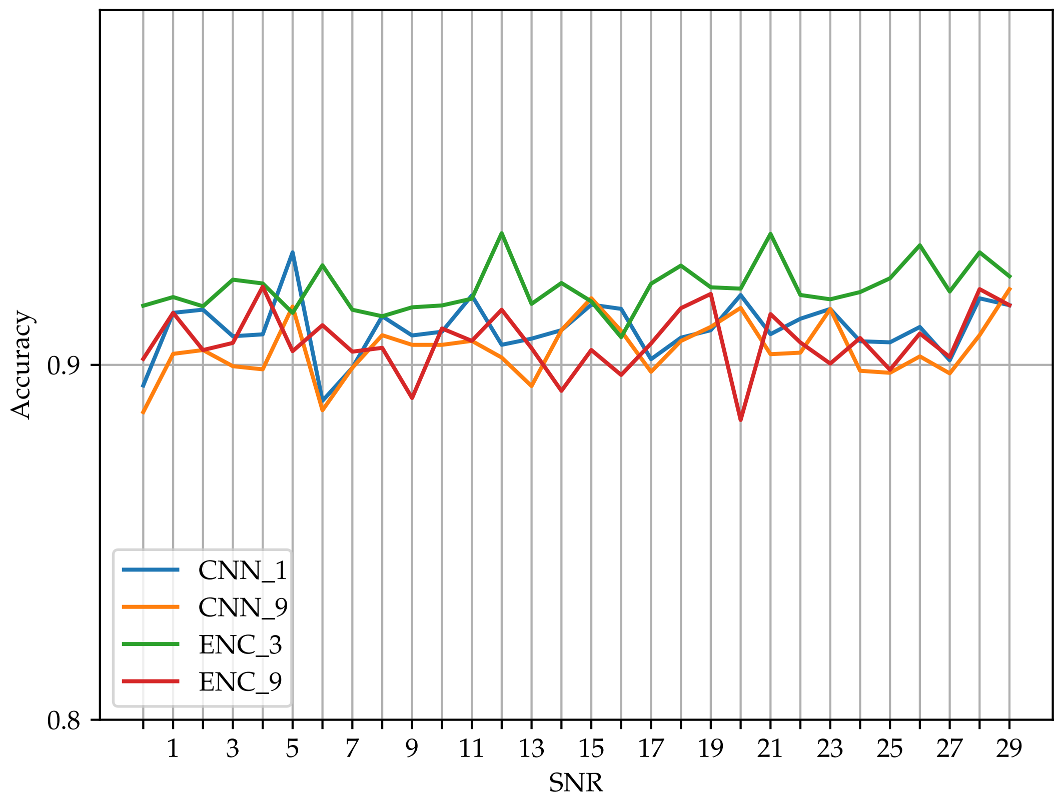


FIGURE C.1: SNR to accuracy plot for 2 best CNNs and Encoders on Matlab[0:29] dataset

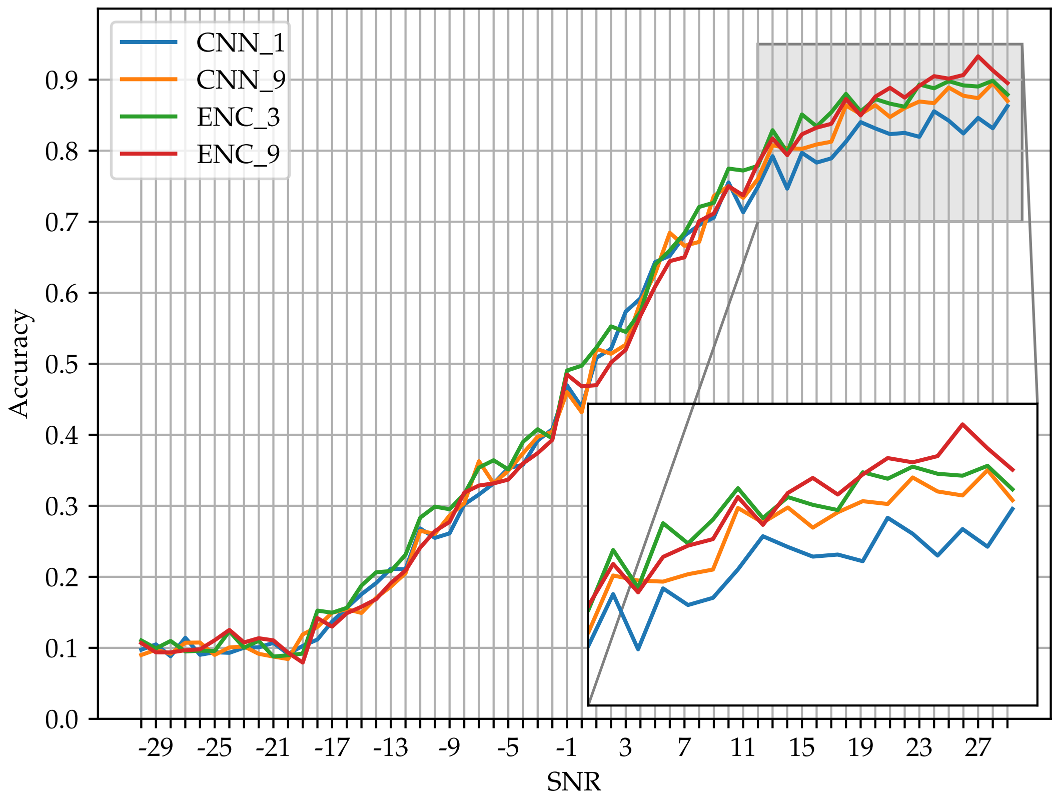


FIGURE C.2: SNR to accuracy plot for 2 best CNNs and Encoders on Matlab[-30:29] dataset

Appendix D

GitHub repository link: https://github.com/Pavlik1400/RISC-V-SIMD-extension-for-the-AI-workload

Bibliography

[1] Fatih Çağatay Akyön et al. "Deep learning in electronic warfare systems: Automatic intra-pulse modulation recognition". In: *2018 26th Signal Processing and Communications Applications Conference (SIU)*. 2018, pp. 1–4. DOI: 10.1109/SIU.2018.8404294.

[2] Laith Alzubaidi et al. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions". In: *Journal of Big Data* 8.1 (Mar. 2021), p. 53. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00444-8. URL: https://doi.org/10.1186/s40537-021-00444-8.

[3] *Amaranth HDL*. https://github.com/amaranth-lang/amaranth. 2020.

[4] Kaur Amritpal. "Analog & Digital Modulation Techniques: An Overview". In: *IJESRT* (2014).

[5] Tim Ansell, Callahan Tim, and Gray Jan. *Draft Proposed RISC-V Composable Custom Extensions Specification*. 2022. URL: https://github.com/grayresearch/CFU/blob/main/spec/spec.pdf.

[6] Andrew Boutros and Vaughn Betz. "FPGA Architecture: Principles and Progression". In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[7] Han Cai et al. *Once-for-All: Train One Network and Specialize it for Efficient Deployment*. 2019. eprint: arXiv:1908.09791.

[8] Keyan Cao et al. "An Overview on Edge Computing Research". In: *IEEE Access* PP (Jan. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2991734.

[9] Nicolas Carion et al. *End-to-End Object Detection with Transformers*. 2020. eprint: arXiv:2005.12872.

[10] Mei CHEN and Qi ZHU. "Cooperative automatic modulation recognition in cognitive radio". In: *The Journal of China Universities of Posts and Telecommunications* 2010 (Apr. 2010), pp. 46–71. DOI: 10.1016/S1005-8885(09)60445-3.

[11] Yiran Chen et al. "A Survey of Accelerator Architectures for Deep Neural Networks". In: *Engineering* 6.3 (2020), pp. 264–274. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2020.01.007. URL: https://www.sciencedirect.com/science/article/pii/S2095809919306356.

[12] François Chollet et al. *Keras*. https://keras.io. 2015.

[13] François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2016. eprint: arXiv:1610.02357.

[14] Kyrkou Christos. *What are FPGAs?* 2017. URL: https://ckyrkou.medium.com/what-are-fpgas-c9121ac2a7ae.

[15] Wikimedia Commons. *File:8PSK Gray Coded.svg — Wikimedia Commons, the free media repository*. [Online; accessed 14-May-2023]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:8PSK_Gray_Coded.svg&oldid=454679020.

[16] Robert David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2020. eprint: arXiv:2010.08678.

[17] Tse David and Viswanath Pramod. *Fundamentals of Wireless Communication*. Cambridge University Press, 2004.

[18] Mr. John Davies. *Radio Frequency Machine Learning Systems (RFMLS)*. https://www.darpa.mil/program/radio-frequency-machine-learning-systems.

[19] Van-Sang Doan et al. *Learning Constellation Map with Deep CNN for Accurate Modulation Recognition*. 2020. eprint: arXiv:2009.02026.

[20] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2020. eprint: arXiv:2010.11929.

[21] Dr. Marc Lichtman. *PySDR: A Guide to SDR and DSP using Python*. 2023. URL: https://pysdr.org/index.html.

[22] Hadi Esmaeilzadeh et al. "Neural Acceleration for General-Purpose Approximate Programs". In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 2012, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

[23] Farah Fahim et al. *hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices*. 2021. eprint: arXiv:2103.05579.

[24] WenHai Fang and Lambert Spaanenburg. "Power-driven FPGA to ASIC conversion - art. no. 659005". In: *Proceedings of SPIE - The International Society for Optical Engineering* (June 2007). DOI: 10.1117/12.722901.

[25] *FOSS Flows For FPGA*. https://github.com/chipsalliance/f4pga. 2018.

[26] *gemmlowp: a small self-contained low-precision GEMM library*. https://github.com/google/gemmlowp. 2015.

[27] Sajjad Ahmed Ghauri et al. "Classification of Digital Modulated Signals Using Linear Discriminant Analysis on Faded Channel". In: 2014.

[28] Hossein Gholamalinezhad and Hossein Khosravi. *Pooling Methods in Deep Neural Networks, a Review*. 2020. eprint: arXiv:2009.07485.

[29] Qiang Guo et al. "Recognition of radar emitter signals based on SVD and AF main ridge slice". In: *Journal of Communications and Networks* 17.5 (2015), pp. 491–498. DOI: 10.1109/JCN.2015.000087.

[30] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. eprint: arXiv:1512.03385.

[31] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[32] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. eprint: arXiv:1704.04861.

[33] Gao Huang et al. *Densely Connected Convolutional Networks*. 2016. eprint: arXiv:1608.06993.

[34] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[35] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[36] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. eprint: arXiv:1502.03167.

[37] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. eprint: arXiv:1704.04760.

[38] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[39] Ian Kuon and Jonathan Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[40] Yukhe Lavinia, Holly H. Vo, and Abhishek Verma. "Fusion Based Deep CNN for Improved Large-Scale Image Action Recognition". In: *2016 IEEE International Symposium on Multimedia (ISM)*. 2016, pp. 609–614. DOI: 10.1109/ISM.2016.0131.

[41] Hao Li et al. *Pruning Filters for Efficient ConvNets*. 2016. eprint: arXiv:1608.08710.

[42] Lingyun Li et al. "Transformer-based radio modulation mode recognition". In: *Journal of Physics: Conference Series* 2384.1 (Dec. 2022), p. 012017. DOI: 10.1088/1742-6596/2384/1/012017. URL: https://dx.doi.org/10.1088/1742-6596/2384/1/012017.

[43] Xiaoxiao Liu et al. "RENO: A High-Efficient Reconfigurable Neuromorphic Computing Accelerator Design". In: *Proceedings of the 52nd Annual Design Automation Conference*. DAC '15. San Francisco, California: Association for Computing Machinery, 2015. ISBN: 9781450335201. DOI: 10.1145/2744769.2744900. URL: https://doi.org/10.1145/2744769.2744900.

[44] Xiaoyu Liu, Diyu Yang, and Aly El Gamal. *Deep Neural Network Architectures for Modulation Classification*. 2017. eprint: arXiv:1712.00443.

[45] Chunjie Luo et al. *Comparison and Benchmarking of AI Models and Frameworks on Mobile Devices*. 2020. eprint: arXiv:2005.05085.

[46] Yongjiang Mao, Wenjuan Ren, and Zhanpeng Yang. "Radar Signal Modulation Recognition Based on Sep-ResNet". en. In: *Sensors (Basel)* 21.22 (Nov. 2021).

[47] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[48] Gaurav Menghani. *Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better*. 2021. eprint: arXiv:2106.08962.

[49] Sparsh Mittal and Shraiysh Vaishay. "A survey of techniques for optimizing deep learning on GPUs". In: *Journal of Systems Architecture* 99 (2019), p. 101635. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.101635. URL: https://www.sciencedirect.com/science/article/pii/S1383762119302656.

[50] *Modulation Classification with Deep Learning*. https://www.mathworks.com/help/deeplearning/ug/modulation-classification-with-deep-learning.html.

[51] Kevin E. Murray et al. “VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling”. In: *ACM Trans. Reconfigurable Technol. Syst.* (2020).

[52] Sharan Narang et al. *Do Transformer Modifications Transfer Across Implementations and Applications?* 2021. eprint: arXiv:2102.11972.

[53] Keiron O’Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. eprint: arXiv:1511.08458.

[54] Timothy J O’Shea, Johnathan Corgan, and T. Charles Clancy. *Convolutional Radio Modulation Recognition Networks*. 2016. eprint: arXiv:1602.04105.

[55] Timothy James O’Shea, Tamoghna Roy, and T. Charles Clancy. “Over-the-Air Deep Learning Based Radio Signal Classification”. In: *IEEE Journal of Selected Topics in Signal Processing* 12.1 (2018), pp. 168–179. DOI: 10.1109/JSTSP.2018.2797022.

[56] Charles Papon. *VexRiscv*. https://github.com/SpinalHDL/VexRiscv. 2017.

[57] Niki Parmar et al. *Image Transformer*. 2018. eprint: arXiv:1802.05751.

[58] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[59] Narendra Patwardhan, Stefano Marrone, and Carlo Sansone. “Transformers in the Real World: A Survey on NLP Applications”. In: *Information* 14.4 (2023). ISSN: 2078-2489. DOI: 10.3390/info14040242. URL: https://www.mdpi.com/2078-2489/14/4/242.

[60] Education Pearson and Salehi Masoud. *Communication systems engineering*. 2nd. Tom Robbins, 2008, pp. 70–143.

[61] *Post-training quantization*. Google. 1600 Amphitheater Parkway Mountain View, CA 94043 USA, 2018. URL: https://www.tensorflow.org/lite/performance/post_training_quantization.

[62] Shvetank Prakash et al. *CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (tinyML) Acceleration on FPGAs*. 2022. eprint: arXiv:2201.01863.

[63] Murad Qasaimeh et al. *Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels*. 2019. eprint: arXiv:1906.11879.

[64] *R: A Language and Environment for Statistical Computing*. MathWorks, Inc. Three Apple Hill Drive Natick, MA 01760 USA. URL: https://www.mathworks.com/help/deep-learning-hdl/.

[65] *Renode*. https://github.com/renode/renode. 2017.

[66] Shamnaz Riyaz et al. “Deep Learning Convolutional Neural Networks for Radio Identification”. In: *IEEE Communications Magazine* 56 (Sept. 2018), pp. 146–152. DOI: 10.1109/MCOM.2018.1800153.

[67] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. *A Comprehensive Survey on Model Quantization for Deep Neural Networks*. 2022. eprint: arXiv:2205.07877.

[68] Tara N. Sainath et al. "Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, pp. 4580–4584. DOI: 10.1109/ICASSP.2015.7178838.

[69] Iqbal H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2.3 (Mar. 2021), p. 160. ISSN: 2661-8907. DOI: 10.1007/s42979-021-00592-x. URL: https://doi.org/10.1007/s42979-021-00592-x.

[70] Jaime Sevilla et al. *Compute Trends Across Three Eras of Machine Learning*. 2022. eprint: arXiv:2202.05924.

[71] Laurent SIfre and Stéphane Mallat. *Rigid-Motion Scattering for Texture Classification*. 2014. eprint: arXiv:1403.1687.

[72] J.A. Sills. "Maximum-likelihood modulation classification for PSK/QAM". In: *MILCOM 1999. IEEE Military Communications. Conference Proceedings (Cat. No.99CH36341)*. Vol. 1. 1999, 217–220 vol.1. DOI: 10.1109/MILCOM.1999.822675.

[73] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. eprint: arXiv:1409.1556.

[74] Wim Vanderbauwhede, S. R. Chalamalasetti, and M. Margala. "Throughput Analysis for a High-Performance FPGA-Accelerated Real-Time Search Application". In: *International Journal of Reconfigurable Computing* 2012 (Feb. 2012), p. 507173. ISSN: 1687-7195. DOI: 10.1155/2012/507173. URL: https://doi.org/10.1155/2012/507173.

[75] Ashish Vaswani et al. *Attention Is All You Need*. 2017. eprint: arXiv:1706.03762.

[76] *Vivado Design Suite User Guide: High-Level Synthesis*. Xilinx. San Jose, 2100 Logic Dr, United States, 2012. URL: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[77] *Vivado Design Suite User Guide: Implementation*. Xilinx. San Jose, 2100 Logic Dr, United States, 2012. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug904-vivado-implementation.pdf.

[78] Danshi Wang et al. "Modulation Format Recognition and OSNR Estimation Using CNN-Based Deep Learning". In: *IEEE Photonics Technology Letters* 29.19 (2017), pp. 1667–1670. DOI: 10.1109/LPT.2017.2742553.

[79] Zi Wang, Chengcheng Li, and Xiangyang Wang. *Convolutional Neural Network Pruning with Structural Redundancy Reduction*. 2021. eprint: arXiv:2104.03438.

[80] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[81] Wenshi Xiao, Zhongqiang Luo, and Qian Hu. "A Review of Research on Signal Modulation Recognition Based on Deep Learning". In: *Electronics* 11.17 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11172764. URL: https://www.mdpi.com/2079-9292/11/17/2764.

[82] *Yosys Open SYnthesis Suite*. https://github.com/YosysHQ/yosys. 2013.

[83] Sun Yulin et al. "Automatic Signal Modulation Recognition based on Deep Convolutional Neural Network". In: 2019.

[84] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. "Deep Learning in Mobile and Wireless Networking: A Survey". In: *IEEE Communications Surveys and Tutorials* 21.3 (2019), pp. 2224–2287. DOI: 10.1109/COMST.2019.2904897.

[85] Zhijin Zhao et al. "Automatic Modulation Classification by Support Vector Machines". In: *Advances in Neural Networks – ISNN 2004*. Ed. by Fu-Liang Yin, Jun Wang, and Chengan Guo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 654–659. ISBN: 978-3-540-28647-9.

[86] Yujun Zheng, Yongtao Ma, and Chenglong Tian. "TMRN-GLU: A Transformer-Based Automatic Classification Recognition Network Improved by Gate Linear Unit". In: *Electronics* 11.10 (2022), p. 1554.