Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masters

Accelerating TensorFlow Machine Learning Inferences on FPGA-Based Edge Platforms

Harshith Naganna

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Prof. Dr. Marco Aiello, Benjamin Wild, M.Sc.
Commenced:	February 11, 2025
Completed:	September 15, 2025

Abstract

As the world becomes more interconnected and data-driven, the demand for real-time data analysis and forecasting is increasing. Time-series forecasting, which predicts future values based on historical data, is widely used in this context. Machine learning and deep learning models are effective for such tasks but are computationally intensive, posing challenges for deployment on edge devices with limited processing power and energy constraints.

This work explores hardware–software co-design using FPGAs (Field Programmable Gate Arrays) to accelerate time-series inference. FPGAs offer parallel computation capabilities, reducing latency and increasing throughput for real-time applications. We implement the accelerator on AMD's Zynq UltraScale+ MPSoC, which combines a dual-core Cortex-A53 processor with programmable logic on a single chip, enabling seamless offloading of complex computations.

The accelerator was developed using Vitis HLS and integrated with the processing system via Vivado. A PetaLinux project enabled communication with the Linux kernel, while custom C++ drivers interfaced the accelerator with the TensorFlow Lite runtime over the AXI protocol. A TensorFlow Lite delegate was developed to offload fully connected layer computations seamlessly onto the FPGA.

The complete system was deployed on the Zynq UltraScale+ MPSoC, and experimental evaluation compared CPU-only and CPU–FPGA setups in terms of latency, power consumption, resource utilization, and accuracy. Results showed that inference on the FPGA accelerator was only 0.5% slower than the CPU-only baseline, as the primary focus was on establishing the hardware–software co-design pipeline rather than optimizing the hardware for maximum performance. Model evaluation achieved a mean absolute error (MAE) of 0.01 and mean squared error (MSE) of 0.20 for the magnitude component, while the phase component obtained an MAE of 4.75 and MSE of 14.99. These findings demonstrate that even with minimal optimization, FPGA acceleration integrated with TensorFlow Lite delegates provides a functional and extensible framework for real-time forecasting on edge devices, paving the way for more efficient and responsive edge computing solutions.

Contents

List of Figures

List of Tables

Acronyms

AMBA Advanced Microcontroller Bus Architecture. 23

API Application Programming Interface. 14

AXI Advanced eXtensible Interface. 23, 24

BRAM Block RAM. 43

BSP Board Support Package. 25

CLB Configurable Logic Block. 22

CPU Central Processing Unit. 15

DSP Digital Signal Processor. 43

FF Flip-Flop. 43

FPGA Field Programmable Gate Array. 13, 14, 15, 22, 24

HLS High-Level Synthesis. 14, 24, 29

I/O Input/Output. 22

IP Intellectual Property. 14, 15, 24

LSTM Long Short-Term Memory. 5, 18, 19

LUT Look-Up Table. 22, 43

MPSoC Multi-Processor System on Chip. 13, 14, 21

PL Programmable Logic. 13, 24

PS Processing System. 13, 14

RTL Register Transfer Level. 14, 24

SoC System on Chip. 13, 24

URAM Ultra RAM. 43

VHDL VHSIC Hardware Description Language. 24

1 Introduction

As humanity is pivoting towards the use of AI (Artificial Intelligence) in everyday workflow, there is an increasing need for machine learning and deep learning tasks to become better and more efficient[SS24]. Some of the deep learning models can forecast the future data based on the learnings taken from past occurrences and training data. Deep learning models are highly efficient in capturing complex patterns and relationships in the data and predicting future outcomes. One such use case can be to use the deep learning models to predict uncertainties in electrical signal data. Predicting those uncertainties can help in taking early actions and preventing future events, which can in turn make the system more reliable and efficient[CHC19]. However, these deep learning models can be computationally intensive and require significant processing power and time to run[ZR20].

One solution to this challenge is to use high-performance servers or cloud computing resources to run machine learning algorithms. However, these options can be expensive and may not be suitable for real-time applications, as they often introduce high latency and require a constant internet connection.

This is where edge computing comes in. Edge computing refers to the practice of processing data closer to the source, rather than relying on a centralized cloud server. This approach can reduce latency, improve data privacy and security, and eliminate the need for continuous internet connectivity. It is especially valuable for real-time applications where quick response times are critical.

However, edge computing also presents challenges, such as limited processing power and energy constraints. Performing real-time data analysis and forecasting under these limitations can be difficult. Edge acceleration addresses this problem by offloading computationally intensive tasks to specialized hardware accelerators.

Edge accelerators can be developed using FPGAs, which are reprogrammable integrated circuits that can be customized to perform specific tasks. FPGA can perform computations in parallel and with low latency and high throughput. Choosing an SoC with both the PS and PL on a single chip can provide superior performance and reliability compared to a multi-chip solution[T23] as data transfer between the PS and PL is faster and more efficient because the data is transferred over a high-speed bus rather than through external connections.

AMD's Zynq UltraScale+ MPSoC is a powerful platform that combines a dual-core ARM Cortex-A53 processor with AMD's programmable logic on the same chip, making it an ideal choice for developing hardware accelerators for machine learning tasks [AMD23]. AMD also provides a suite of tools such as Vitis HLS, Vivado, and Petalinux to facilitate the development and integration of custom hardware accelerators for Zynq UltraScale+ MPSoC.

AMD's Vitis HLS tool is used for converting high-level C/C++ code to RTL code. Vitis HLS makes it simple to write C/C++ code for sophisticated FPGA-based algorithms. Complex data formats and mathematical functions are supported[AMD24a]. Vivado is a tool used for integrating the custom hardware accelerators(IPs) with the PS of Zynq UltraScale+ MPSoC. Petalinux is a collection of high-level commands developed on top of the Yocto Linux distribution. Petalinux is used to design, create, and implement embedded Linux images for Xilinx processing systems. It is tailored to accelerate design productivity and works with the Xilinx hardware design tools (like Vivado) to ease the development of Linux systems for Zynq[aoifem23].

In this thesis, we develop a deep learning-based time-series forecaster to predict frequency characteristics of time domain electrical grid signals. The model is developed in such a way that it predicts 10 periods of future frequency characteristics based on the given 1 period time domain electrical signal. The developed model is then trained and evaluated using a dataset of time-domain electrical signals. The trained model is then converted to TensorFlow Lite format for deployment on the edge platform.

To enable efficient execution of the TensorFlow Lite model on our FPGA-based edge platform, we utilize TensorFlow Lite's custom delegate framework. This framework allows us to offload computationally intensive operations from the default CPU execution to our custom hardware accelerators. By replacing the standard TensorFlow Lite execution flow with our own hardware-accelerated implementation, we can significantly reduce inference latency and improve overall system performance.

To support this delegation approach, specialized hardware accelerators are developed to accelerate the most computationally demanding operations of the trained model. For our project, we have chosen to develop a hardware accelerator specifically for fully connected layers, which supports a maximum input size of 32 floating point values and 32 output values. The fully connected layer accelerator is also integrated with the ReLU activation function to minimize data movement and maximize computational efficiency. The hardware accelerator is developed using Vitis HLS and is integrated with the PS of Zynq UltraScale+ MPSoC using Vivado. The hardware accelerator is then interfaced with the Linux kernel using Petalinux, and custom drivers along with C++ APIs are created to provide seamless control and integration with the TensorFlow Lite delegate framework.

1.1 Motivation

In electrical grids, it is crucial to assess the nature of signal disturbances and forecast their future behaviour and performance. This enables readiness to mitigate the effects using power filtration techniques. Deep learning models provide a promising pathway to achieve these goals. However, running deep learning inference on edge devices poses challenges due to constraints in speed and computational resources. This makes it essential to accelerate computation for time-critical applications.

1.2 Problem Statement

Deploying machine learning models on edge devices is challenging due to constraints like limited computational resources, high latency, and power inefficiency. There is a need for an efficient end-to-end pipeline to optimize and accelerate the inference process, enabling real-time performance while minimizing resource usage on edge platforms. Our goal is to construct custom accelerators on FPGA, interface them with the CPU, and establish a hardware-software pipeline for the custom IP accelerators to achieve lower inference latency.

1.3 Objective

- Create a baseline time-series model.
- Convert the trained TensorFlow model into a TF Lite model.
- Run the inferencing on ZYNQ SoC with CPU only setup.
- Identify the most time-intensive computations.
- Develop custom hardware accelerators on the FPGA.
- Optimise FPGA calculations using FPGA optimisation techniques.
- Interface the accelerator with the Linux kernel.
- Develop TensorFlow custom delegates.
- Modify the TensorFlow Runtime to include the created custom delegates.
- Perform inferencing with the CPU-FPGA accelerated setup.
- Compare implementations in terms of latency, power consumption, resource utilization, and model accuracy.

2 Background Information

In this chapter, we discuss the key concepts and technologies used in this project.

2.1 Deep Learning

Deep learning is a subset of artificial intelligence(AI) and machine learning that mimics the way humans acquire knowledge. It is possible to train deep learning models to carry out classification tasks and identify patterns in a variety of data, including text, audio, images, and more. It is also utilized for the automation of tasks like image description and audio transcription that would typically require human intellect. Deep learning uses neural networks made of numerous layers of cooperating software nodes, similar to the millions of interconnected neurones seen in human brains that collaborate to learn new things. Large sets of labeled data and neural network designs are used to train deep learning models[ASG23].

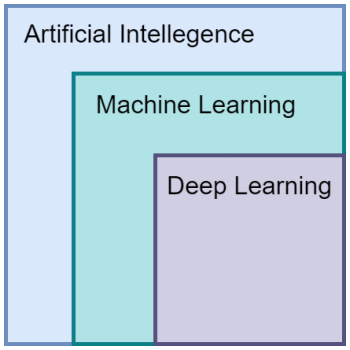


Figure 2.1: Deep Learning as a subset of AI

2.2 1D Convolutional Neural Networks

Convolutional layers are main part of deep learning specially for image processing and signal data. 2D convolutions are used for image processing, while 1D convolution is used for signal data.

1D convolutional layers performs convolution on one dimensional data. In 1D convolution, the kernel slides over single spatial dimension of the input data Figure 2.2.

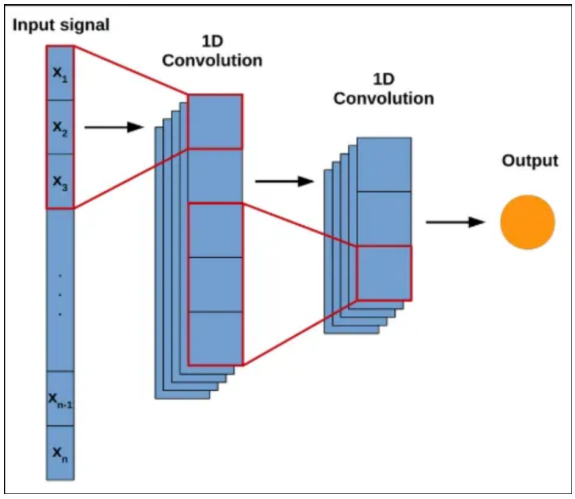


Figure 2.2: 1D Convolutional Layer [AJ20]

The layer uses a kernel (filter), a small matrix of learnable weights, to extract features from the sequence. The kernel is applied to the input data by sliding it over the input sequence and performing a dot product between the kernel and the input data. The obtained result is a feature map of the captured data Figure 2.3. 1D convolutions are particularly useful for extracting features and local patterns from sequential data, such as time series or audio signals, its peaks and valleys [AJ20].

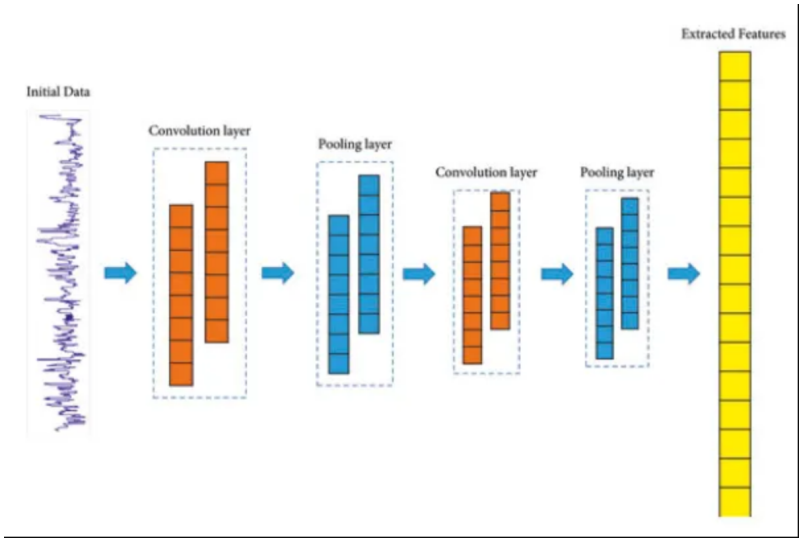


Figure 2.3: Feature Map [AJ20]

2.3 LSTM Networks

LSTM networks are a type of recurrent neural network (RNN) architecture that has gated mechanisms to regulate the flow of information, allowing them to learn long-term dependencies in sequential data.

These gates work together to maintain and update the cell state, enabling the network to capture long-term dependencies effectively

As depicted in Figure 2.4, an LSTM cell comprises of three main components:

- **Input Gate:** Controls the amount of new information to be added to the cell state.
- **Forget Gate:** Determines what information to discard from the cell state.
- **Output Gate:** Regulates the output of the cell state to the next layer.

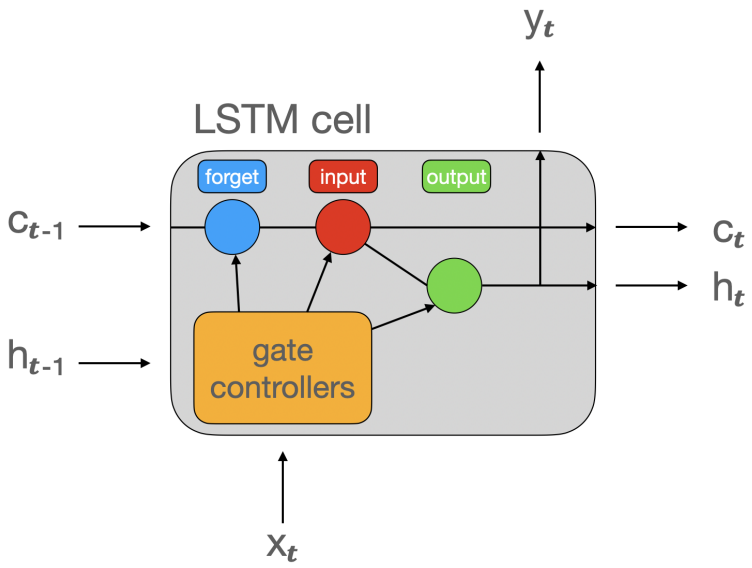


Figure 2.4: LSTM Cell [DC24]

2.4 Fully Connected Networks

A fully connected network, also known as a dense network, is a type of neural network which consists of series of fully connected layers that connect every neuron in one layer to every neuron in the next layer [SS20].

A Fully Connected layer differentiates itself by its numerous interconnections. Here's a breakdown of its key components.

- **Neurons**: They are basic units that receive input from all neurons in the previous layer and send output to all neurons in the next layer.
- **Weights**: Each connection between neurons has a weight that determines the strength and effect of one neuron on another. These weights are learned during the training process.
- **Biases**: Each neuron has a bias term that adjusts the output as well as the weighted sum of inputs.

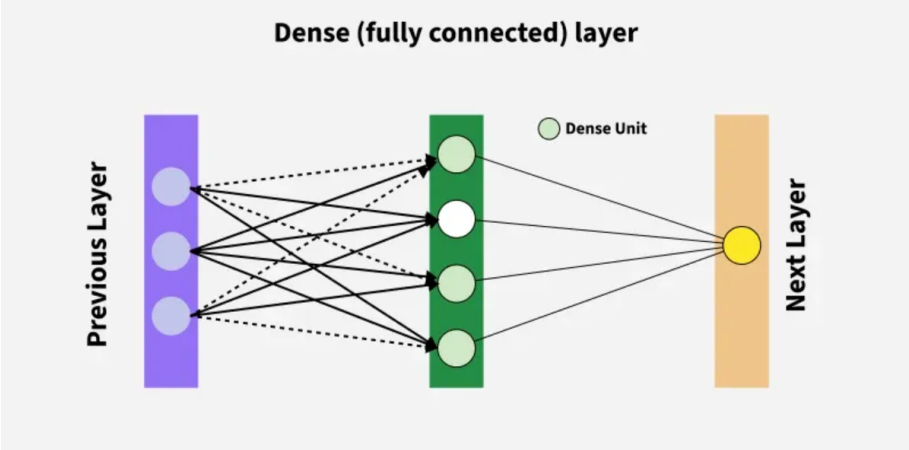


Figure 2.5: Fully Connected Network [GFG22b]

- **Activation Function**: Functions such as ReLU, Sigmoid, and Tanh add nonlinearity to the model, allowing it to learn complicated patterns and behaviors [GFG22b].

2.5 TensorFlow Lite

TensorFlow Lite (TF Lite) is an open-source machine learning framework designed for on-device inference, enabling developers to run trained models on mobile, embedded, and IoT devices. It's a lightweight version of TensorFlow, optimized for resource-constrained environments. TF Lite provides tools and APIs to convert standard TensorFlow models into a TF Lite format, allowing for efficient deployment on various platforms [Google24a].

It enables on-device inference, enabling real-time predictions without relying on online services; lightweight and designed for mobile and embedded devices; cross-platform compatibility; model conversion and optimization [Google24a].

2.6 TensorFlow Lite Delegates

Delegates allow TF Lite models to take advantage of on-device hardware accelerators like GPUs and Digital Signal Processors (DSPs) to boost performance.

While TF Lite runs on CPU kernels by default—these are optimized for the ARM Neon instruction set—the CPU itself is a general-purpose processor and isn't specifically designed for the intensive numerical computations commonly used in machine learning tasks, such as the matrix operations in convolutional and fully connected layers.

Tflite delegates acts as a bridge between the TF Lite runtime and the hardware accelerators, allowing the runtime to offload specific operations to the accelerator as shown in Figure 2.6. This delegation can significantly speed up inference times and reduce power consumption, making it ideal for mobile and embedded devices [Google24b].

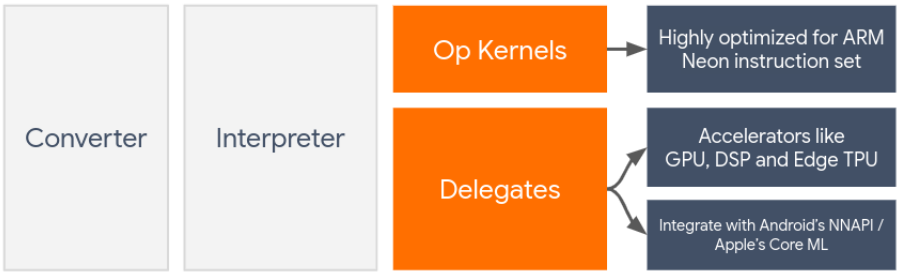


Figure 2.6: TensorFlow Lite Delegate [Google24b]

2.7 Trenz TE0820-05-2AE81MA (Zynq UltraScale+ MPSoC)

The Trenz TE0820-05-2AE81MA is a powerful 4 x 5 cm MPSoC module featuring an AMD Zynq™ UltraScale+™ ZU2CG processor. As in Figure 2.7, it features a dual-core ARM Cortex-A53 processor and an FPGA fabric. The board is designed for high-performance applications, including machine learning, computer vision, and real-time data processing. It provides various interfaces for connectivity and expansion, making it suitable for a wide range of embedded applications [Trenz Electronic24].

Figure 2.7: Trenz TE0820-05-2AE81MA Module [Trenz Electronic24]

2.8 Zynq UltraScale+ MPSoC Architecture

The Zynq UltraScale+ MPSoC architecture is a heterogeneous system-on-chip (SoC) that integrates multiple processing elements, including a dual-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 real-time processor, and an FPGA fabric. This architecture allows for flexible and efficient processing of complex tasks by leveraging the strengths of both the processing system (PS) and the programmable logic (PL) [Xilinx24c].

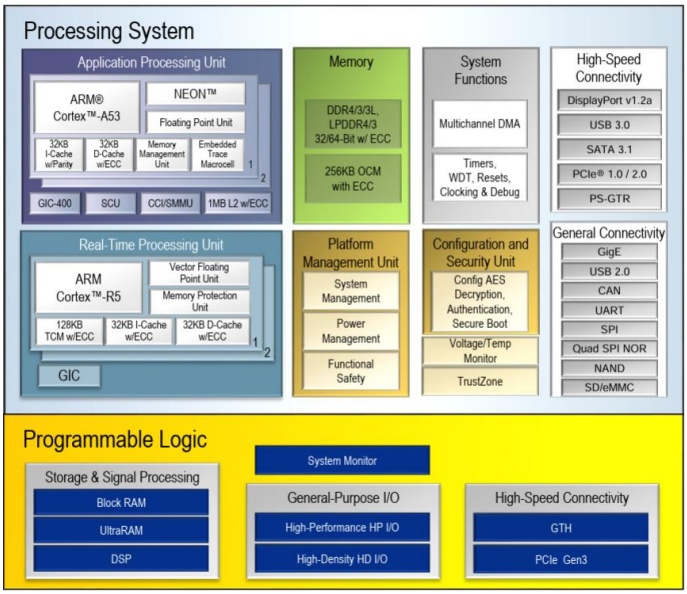


Figure 2.8: Zynq UltraScale+ MPSoC Architecture [Xilinx24c]

2.9 FPGAs

Field-programmable gate arrays (FPGAs) are reprogrammable integrated circuits that contain an array of programmable logic blocks. FPGA chip adoption is driven by their flexibility, hardware-timed speed and reliability, and parallelism[NI24].It is made up of many logic components called "gates"that are connected together to form circuits. By programming the FPGA to change the circuit function, the connections between the gates can be changed[HEINEN24].

Every FPGA chip consists of a finite amount of predetermined resources, programmable interconnects to create a reconfigurable digital circuit, and I/O blocks to connect the circuit to the outside world. FPGA resource parameters frequently include the number of programmable logic blocks, fixed function logic blocks such as multipliers, and memory resource sizes such as integrated block RAM. Configurable logic blocks (CLBs) are the fundamental logic units of FPGAs. CLBs, also known as slices or logic cells, consist of two basic components: flip-flops and lookup tables (LUTs).

FPGAs, unlike processors, are really parallel, which means that distinct processing tasks do not compete for the same resources. Each independent processing task is assigned to a distinct region of the chip and can operate independently without interference from other logic blocks. As a result, increasing the amount of processing has no effect on the performance of one section of the application[NI24].

2.10 AXI Interface

The AXI is an on-chip communication bus protocol and is part of the AMBA specification defined and controlled by Arm. AXI is an interface specification that describes the interfaces between IP blocks rather than the link itself. There are just two AXI interface types: manager and subordinate. These interface types are symmetrical. All AXI connections occur between manager interfaces and subordinate interfaces. AXI interconnect interfaces use the same signals, making the integration of different IP relatively simple. The Figure 2.9 depicts how AXI connections link management and subordinate interfaces. The direct connection provides maximum bandwidth between the manager and subordinate components with no additional logic[ARM24].

Figure 2.9: AXI interface [ARM24]

The AMBA AXI4 interface connections are point-to-point and available in three different types: AXI4, C4-Lite Slave, and AXI4-Stream. AXI4 is a memory-mapped interface that enables burst transactions. AXI4-Lite Slave is a lightweight version of AXI4, with a non-bursting interface. AXI4-Stream is a high-performance streaming interface for unidirectional data transfers (from master to slave) that requires less signaling (than AXI4). AXI4-Stream supports multiple channels of data on the same set of wires[AMD24b].

The AXI protocol has numerous fundamental features meant to enhance bandwidth and latency of data transfers and transactions, such as independent read and write channels, Multiple outstanding addresses, No strict timing relationship between address and data operations, Support for unaligned data transfers, out-of-order transaction completion, and burst transactions based on start address[ARM24].

2.11 Vitis HLS

Vitis HLS (High-Level Synthesis) is a powerful tool developed by Xilinx (now part of AMD) that allows engineers to create complex FPGA-based algorithms using high-level programming languages like C and C++[BS21]. This tool significantly simplifies the process of designing hardware for FPGAs by enabling developers to work at a higher level of abstraction.

Vitis HLS acts as a bridge between software and hardware design. It accepts C/C++ code as input and converts it to RTL code, which may then be implemented in the PL section of FPGAs or adaptive SoCs[AMD24a]. This technique enables developers to explain their algorithms in a more familiar software environment while targeting hardware implementation.

Key features of Vitis HLS include support for C,C++, and OpenCl languages; handle data types including floating point and fixed point arithmetics; built-in support for advanced mathematic functions; facilitate data sharing with other IPs through AXI4-Stream; allow fine-tuning the generated hardware through pragma and directives; and support a wide range of Xilinx devices[BS21].

2.12 Vivado

Vivado is a comprehensive design software suite developed by AMD (formerly Xilinx) for the development of adaptive SoC and FPGAs.It includes various methodologies to help with the full design process, from basic design to final implementation and verification. Vivado is intended to simplify the development process for complicated hardware designs, allowing for faster design cycles and more accurate power and performance calculations[Xilinx24b].

Key features of Vivado include: Vivado supports multiple methods for design entry, which includes support for traditional hardware description language like VHDL, a graphical user interface tool that allows for plug-and-play IP integration, making it easier to design complex systems using pre-built IP blocks. Vivado provides advanced synthesis and implementation capabilities that allow high-level design description into a gate-level netlist and map the netlist onto the physical resources of the FPGA, optimizing for timing and area constraints. The Power Design Manager in Vivado provides early and accurate power estimations, which are critical for making design decisions in large and complex devices. Vivado enhances timing closure and performance optimization through tools like Report QoR Assessment (RQA) and Suggestions (RQS), Intelligent Design Runs (IDR) for automated synthesis and implementation, and machine learning algorithms, collectively streamlining the process of achieving optimal design results. Vivado also provides a comprehensive debugging environment that includes a waveform viewer, a logic analyzer, and a hardware manager that allows for real-time debugging of the design on the FPGA[Xilinx24b].

2.13 Petalinux

PetaLinux is an embedded Linux development solution provided by AMD specifically designed for their FPGA and SoC products, including the Zynq family, Zynq UltraScale+ MPSoCs, and MicroBlaze soft processors. PetaLinux provides a comprehensive set of tools and a reference Linux distribution tailored for AMD's embedded processing systems. It is built on top of the Yocto Project,

which is an open-source collaboration project that provides templates, tools, and methods to create custom Linux-based systems. PetaLinux offers a set of high-level commands that simplify the process of customizing, building, and deploying embedded Linux systems[aoifem23].

PetaLinux integrates seamlessly with AMD's hardware design tools, such as Vivado, to synchronize the software platform with the hardware design. This integration ensures that any changes or updates in the hardware design are reflected in the software platform, facilitating a cohesive development process.PetaLinux provides pre-configured Board Support Packages (BSPs) that include boot loaders, system images, and bitstreams. PetaLinux includes a full system simulator (QEMU) that allows developers to boot and test their Linux systems in a virtual environment before deploying them to physical hardware[Xilinx24a].

2.14 Device Drivers

A device driver is a special kind of software program or module that allows higher-level computer programs to interact with a hardware device. It usually communicates with a computer subsystem of a computer bus connected to the hardware. The Figure 2.10 shows how the device driver, OS,OS and the Device[GFG22a].

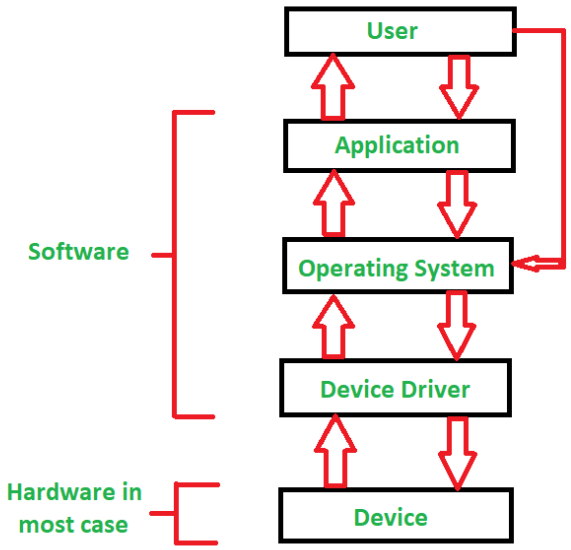


Figure 2.10: Device Driver [GFG22a]

Device drivers often operate with a high level of privilege within the operating system (OS) runtime environment, and some are directly linked to the OS kernel. The operating system and its applications send requests for device access and actions to the corresponding hardware devices through these drivers. In our implementation, the drivers are specifically developed for the TensorFlow Lite runtime to communicate with the hardware accelerator. They operate closely with the hardware without user intervention and abstract the lower-level bit signaling. These drivers define the protocols and messages that enable the computer, OS, and applications to interact with the device and request operations. Furthermore, they manage the communication flow, including

responses sent from the device back to the computer, ensuring reliable and efficient hardware utilization [ASG24]. This design is essential for seamless integration of the accelerator, as it allows the runtime to offload computation without requiring manual intervention or detailed knowledge of the underlying hardware signaling.

2.15 PQ-Box 150

The PQ-Box 150 as in Figure 2.11 is a mobile power analyzer, power meter, and transient recorder. It is designed to quickly measure the source of power disturbances. This Class A device is designed for measurements in low-, medium-, and high-voltage networks and is ideally suited for mobile measurement operations in harsh environments and very confined spaces. The PQ-Box 150 can measure voltage, current, power, energy, and other electrical parameters with high accuracy and resolution. It also supports advanced features such as harmonic analysis, flicker measurement, and event recording [A. Eberle24].

Figure 2.11: PQ-Box 150 [A. Eberle24]

2.16 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, together with dynamic typing and dynamic binding, make it ideal for rapid application development and as a scripting or glue language for connecting existing components. Python's simple, easy-to-learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed[PSF24].

3 Literature Review

With the increasing integration of AI (Artificial Intelligence) into everyday applications, there is a growing need for machine learning and deep learning tasks to be both more efficient and faster. Traditional solutions rely heavily on cloud computing to perform these tasks due to its computational power. However, this approach introduces significant latency, increased energy consumption, and dependency on constant internet connectivity, making it unsuitable for real-time applications. This shift has led to a strong interest in edge computing, where data is processed closer to the source rather than being offloaded to the cloud, thus reducing latency and improving data security[CR19].

Deep learning models are computationally intensive, requiring vast amounts of processing power and memory to achieve accurate results. While cloud computing offers the necessary infrastructure, it introduces undesirable latency and costs, making it less than ideal for real-time machine learning and data analysis. Moreover, energy constraints in edge devices compound these challenges, as edge devices are designed to operate under strict power consumption limits[ZR20].

To address this, edge computing has emerged as a promising alternative. It processes data at the edge, reducing latency, improving response times, and enabling real-time AI applications. However, edge devices have limited processing power, making them inadequate for handling high-complexity tasks without additional support[LZZC19].

To overcome the limitations of edge devices, hardware acceleration has been widely adopted. Among the most effective accelerators are Field Programmable Gate Arrays (FPGAs), which offer a balance between low power consumption, high performance, and customizability. FPGAs enable parallel processing, drastically reducing execution time for tasks such as machine learning inference and signal processing[SCYE17].

Processing at the edge minimizes latency and cloud dependency, improving responsiveness and privacy for real-time AI workloads compared to cloud offloading, which suffers from network latency and energy overheads in sustained transfers. Deep learning on constrained devices is limited by compute throughput and memory bandwidth; even when cloud offers scale, round-trip latency and recurring costs are incompatible with tightly bounded real-time inference and energy budgets at the edge. FPGAs provide parallelism, custom datapaths, and on-chip memory structures for data reuse, achieving higher energy efficiency than general-purpose CPUs/GPUs on targeted kernels; recent edge systems report multi-x energy gains in real-time CNN inference on Zynq/UltraScale+ boards. Application studies report real-time deployment on Xilinx ZCU102 with YOLOv7 at 36.5 GOP/s/W, underscoring the power-performance advantage of FPGA inference versus embedded alternatives while maintaining responsiveness for streaming workloads[JC24].

PetaLinux supports custom Linux distributions on Zynq platforms to host runtime stacks, drivers, and user-space frameworks for deploying edge AI applications with managed device interfaces and optimized DMA pathways to PL accelerators. Case studies show end-to-end deployments

that connect camera sensors to FPGA CNN pipelines for in-situ inference, demonstrating the practical integration of OS, I/O, and accelerators for real-time analytics. Quantization and pruning remain key to fitting models within BRAM/URAM and off-chip bandwidth limits, improving throughput per watt while retaining acceptable accuracy on edge tasks; these steps complement systolic or vector-MAC datapaths synthesized via HLS. Efficient tiling, line-buffered convolutions, and GEMM-based lowering of CNN layers are routinely employed to maximize data reuse and reduce expensive external DRAM traffic on FPGA accelerators[JC24].

SECDA-TFLite provides an open-source toolkit that instantiates the SECDA methodology within TFLite's Delegate API, enabling rapid prototyping, simulation, and integration of custom FPGA accelerators for TFLite ops with reduced setup cost. The toolkit supplies end-to-end flows, including delegate integration, benchmarking, hardware automation scripts, and SystemC-HLS co-simulation; recent presentations document streamlined development via dev-containers, TensorFlow integration, and board automation to shorten accelerator bring-up. The GitHub repository documents setup with Dev Containers/Docker, TensorFlow fork checkout, and environment provisioning, along with scripts to build, launch, and profile accelerators integrated as TFLite delegates; reference accelerators (vector-MAC, systolic) for GEMM/conv, tutorials, and benchmarking are included under a permissive license with ongoing updates. Beyond CNNs, SECDA-TFLite has been used to implement custom delegates for specialized layers such as transposed convolution, demonstrating average 1.9× speedups over dual-thread ARM baselines and improved GOPS/DSP versus prior edge-FPGA designs in end-to-end DCGAN/pix2pix evaluations[ZR20].

4 Methodologies

This chapter describes the methodologies used to achieve the objectives of the project. The work is broadly divided into three main parts: Data & Model Pipeline, Hardware Acceleration, and System Integration. The Data & Model Pipeline process involved data extraction from PQBox150, data preprocessing, development of time series model, and conversion of the model to TensorFlow Lite format. The Hardware Acceleration process involved creating fully connected hardware accelerator using Vitis HLS, interfacing the hardware with the processing system in Vivado, and building a Linux kernel using Petalinux. The System Integration involved development of custom tensorflow lite delegates to access the hardware accelerator from the tflite runtime, and development of C++ drivers to access the hardware accelerator from the Linux kernel.

4.1 Data & Model Pipeline

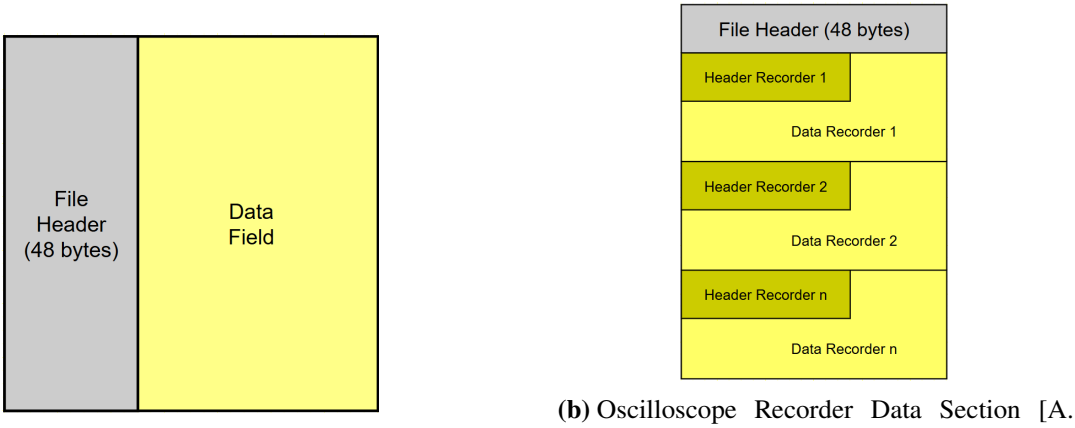
In this section, we discuss about data acquisition, data preprocessing, model development, and model optimisation.

4.1.1 Data Acquisition

Data is acquired from the PQ-Box 150 power quality analyzer. A PQ-Box 150 measurement includes various files in the binary .pqf extension, which can be differentiated according to the type of data stored in them. These include configuration data, cyclic measurement data, event data, and recorder data. In this thesis, we use oscilloscope recorder data, which is a subset of recorder data [A. Eberle24].

Each oscilloscope recorder file contains voltage and current waveform data for all three phases (L1, L2, L3) and the neutral (N) conductor. The data is stored in a binary format, which must be converted into a human-readable format for further processing. The file is divided into a header and a data section as shown in Figure 4.1a. The 48-byte header contains metadata about the measurement, such as the device ID, timestamp, and file ID. The data section of the oscilloscope recorder is further divided into the header recorder and the data recorder as shown in Figure 4.1b. The header recorder contains information about the measurement, such as the recorder trigger ID, the size of the data, and the timestamp of the trigger. The data recorder contains the actual three-phase current and voltage data, stored in float32 format within a structure [A. Eberle24].

Python script is developed to extract the data from the .pqf files. The script reads the binary data from the file, extracts the header information, and then reads the data section to extract the three-phase current and voltage data. The extracted data is then stored in a .csv file for further processing and analysis.



(a) PQ-Box 150 File Format [A. Eberle24]

(b) Oscilloscope Recorder Data Section [A. Eberle24]

Figure 4.1: File formats and oscilloscope recorder structure of PQ-Box 150

4.1.2 Data Preprocessing

The extracted data from the .pqf files is prepared and processed for model training. The dataset consists of multiple timestamps of three-phase voltage and current data. Each timestamp of a channel contains 4096 datapoints of float32 data, representing the voltage or current waveform over 10 cycles.

First, the timestamp samples are extended using edge padding, where the last value is repeated until the nearest multiple of 410 is reached, thereby maintaining the input window size of 410. This ensures that there are no abrupt changes in the waveform, which could affect the FFT results.

Next, the Fast Fourier Transform (FFT) is applied to each channel to convert the time-domain data into the frequency domain, providing a suitable starting point for generating the labels as in Figure4.2. The FFT is computed using the NumPy library in Python, which provides an efficient implementation of the algorithm.

Subsequently, harmonic binning is applied to the FFT results to reduce the dimensionality of the data, with each bin covering ±5 Hz around the corresponding harmonic frequency. The first 50 harmonics are considered for each channel, as they contain the most significant information about the waveform. The resulting binned FFT data is then used as the labels for model training.

The binned FFT data is further decomposed into its magnitude and phase components. The time-domain data, as well as the magnitude and phase components, are normalized separately using Min-Max scaling to ensure that all features are on a comparable scale, thereby improving model convergence during training.

Finally, based on the selected channels, the relevant features are concatenated to form the final input and label datasets. A sliding window approach(Figure 4.3) is then applied with a window size of 410 samples and a stride of 410 samples. This creates non-overlapping segments of data, where each window contains 410 samples from each selected channel (i.e., one cycle of time-domain data). The corresponding labels are the binned FFT magnitude and phase components of 51 samples (i.e., 50 harmonics + the DC component) derived from the 10-cycle segment of the corresponding channel at that timestamp.

The input dataset thus consists of the selected time-domain channels, while the label dataset contains the corresponding magnitude and phase components of the binned FFT data. The preprocessed dataset is then split into training, validation, and test sets for model development and evaluation.

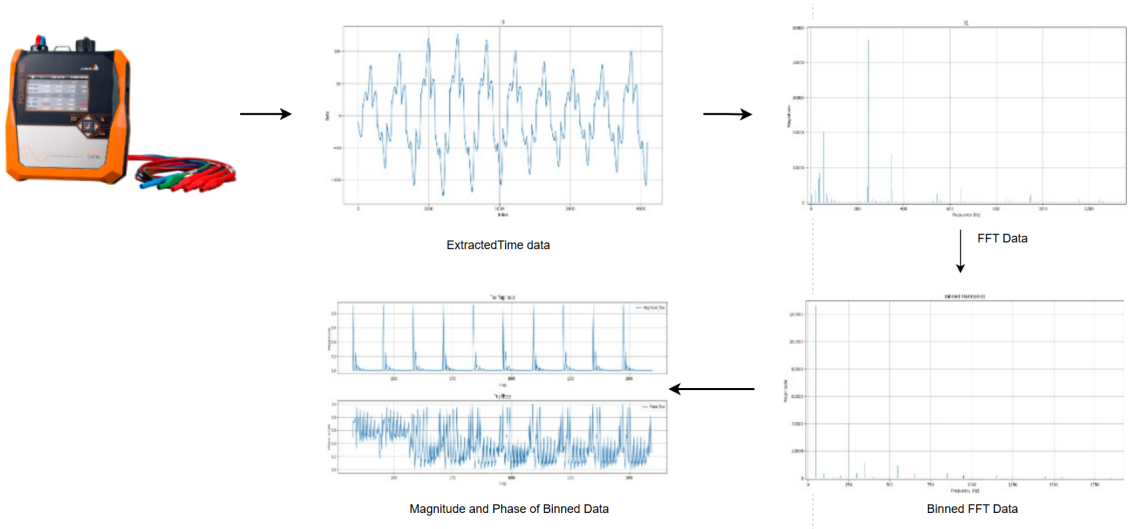


Figure 4.2: Data Extraction and Preprocessing Pipeline

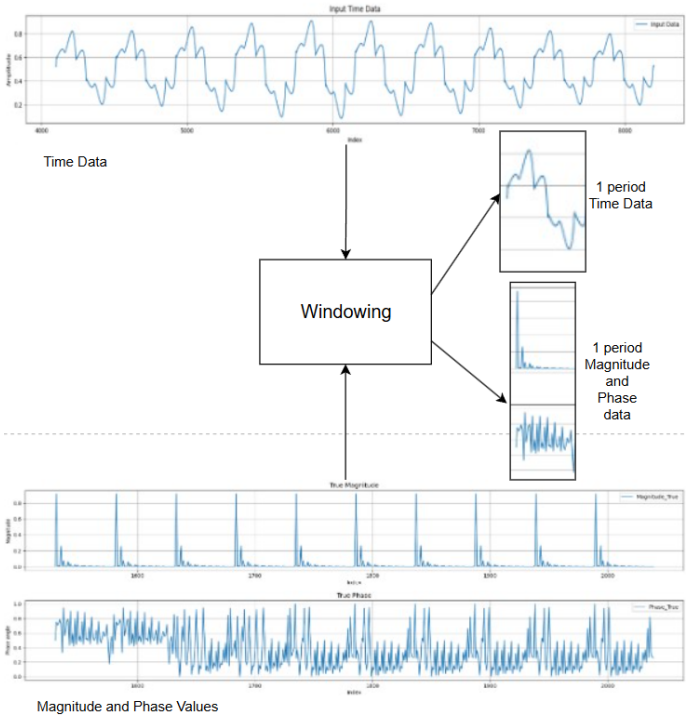


Figure 4.3: Sliding Window Approach

4.1.3 Model Development

A time-series model is developed to predict the binned FFT magnitude and phase components of a 10-cycle signal from the corresponding 1-cycle time-domain input. In this thesis, a dual-branch LSTM-based model is implemented using the TensorFlow and Keras libraries in Python as shown in Figure 4.4. The architecture is designed to effectively capture temporal dependencies in the time-series data while processing the magnitude and phase components separately. This separation allows each branch to specialize in learning patterns relevant to its respective output, thereby reducing the risk of overfitting.

Each branch begins with one-dimensional convolutional (Conv1D) layers, which are responsible for extracting local temporal features from the input sequence. These convolutional layers act as feature detectors, learning short-term patterns such as sudden variations or localized fluctuations in the signal. Max-pooling layers are applied after convolution to progressively reduce the temporal dimension and retain only the most relevant features, which also helps control model complexity.

The extracted features are then passed to stacked Long Short-Term Memory (LSTM) layers. The LSTM units are particularly effective for modeling sequential data because they can capture both short- and long-range temporal dependencies through gated memory mechanisms. In the magnitude branch, smaller LSTM units are employed in deeper stacks to progressively refine the representation and focus on fine-grained amplitude variations. In contrast, the phase branch uses more LSTM layers to capture the more complex, long-range dependencies inherent to phase information.

Following the LSTM layers, multiple fully connected (dense) layers are included. These dense layers integrate the learned temporal representations and perform higher-level abstractions, enabling the network to combine features in a non-linear fashion. Batch normalization layers are interleaved to stabilize training by normalizing intermediate activations, while dropout is applied to reduce overfitting by randomly deactivating neurons during training.

The final output of each branch is produced by a dense layer with 51 units, corresponding to the binned FFT magnitude or phase components. A sigmoid activation function is used in the output layer to constrain the predictions to a bounded range between 0 and 1. This design choice is justified by the fact that the FFT magnitude and phase values are normalized during preprocessing, making the sigmoid activation an appropriate fit. Using sigmoid ensures numerical stability and prevents unbounded predictions, which could otherwise occur with linear activations. It also avoids introducing sparsity effects that could arise with ReLU, which is less suitable for continuous-valued regression tasks on normalized data.

The model is trained using the Adam optimizer, with mean squared error (MSE) defined as the loss function for both magnitude and phase outputs. Training is performed for 64 epochs with a batch size of 32, and early stopping is applied to prevent overfitting by halting training once the validation loss ceases to improve. The model's performance is monitored on a validation set during training, and the best-performing weights are saved based on the lowest validation loss. Finally, the trained model is evaluated on an unseen test set to assess its generalization capability.

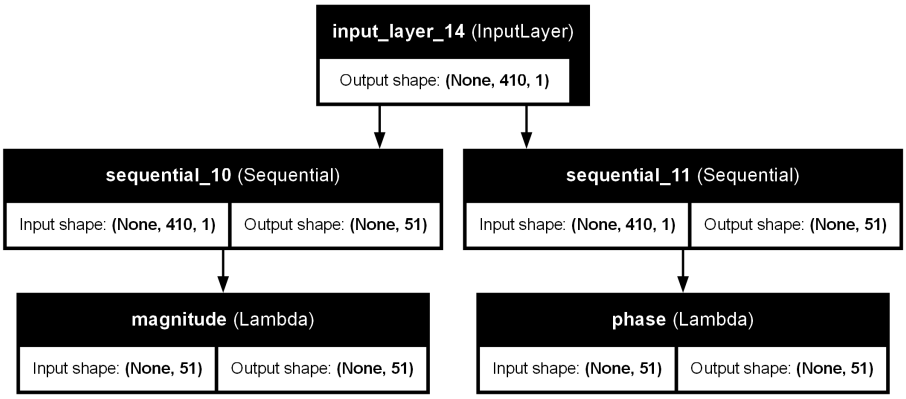


Figure 4.4: Model Architecture

4.1.4 Model Optimisation

The trained model is optimized for deployment on edge devices using TensorFlow Lite (TFLite). TensorFlow Lite is specifically designed for mobile and embedded systems, enabling efficient inference with reduced memory footprint and computational overhead. The optimization process involves converting the trained Keras model into the TFLite format, followed by validation to ensure minimal accuracy loss.

The conversion is carried out in Python using the TensorFlow Lite Converter API. For simplicity, the inference input type is kept as the default float32 format. While float32 is slower during inference and consumes significantly more memory compared to lower-precision formats such as int8, it offers a straightforward implementation and evaluation pipeline. This makes it suitable for prototyping and ensuring functional correctness before moving on to more aggressive quantization strategies.

During conversion, the script first checks for the presence of problematic layers. These are typically dynamic layers (e.g., layers that rely on dynamic tensor shapes), which are not directly supported by standard TensorFlow Lite operations. If such layers are detected, the script adds the TFLite Flex Delegate to the converter. The Flex Delegate enables execution of unsupported TensorFlow ops within TFLite, but at the cost of increased model size, additional dependencies, and incompatibility with custom delegates. For deployment scenarios where lightweight models and hardware accelerators are critical, Flex Delegate is generally avoided unless absolutely necessary.

To further minimize incompatibility issues, the model is then converted into a static-shape version, based on user-specified input dimensions. Static shapes ensure that the converted model is predictable, easier to optimize, and more portable across devices, as opposed to dynamic models that may fail on certain inference engines. Once the static-shape model is generated, it is carefully compared against the original Keras model to verify that there are no significant losses in accuracy or changes in behavior.

Finally, the conversion process is tested using multiple methodologies, ranging from producing basic lightweight TFLite models (with minimal dependencies) to more complex models that include additional operators and Flex Delegates. The final converted TFLite model is then validated on the test dataset to confirm that its accuracy remains comparable to the original Keras model.

Future optimizations, such as post-training quantization to int8 or mixed precision formats (e.g., float16 quantization), could further reduce the memory footprint and improve inference speed. However, for the scope of this thesis, float32 inference is maintained to simplify deployment and ensure a reliable baseline for edge device testing.

4.2 Hardware Acceleration

In this section, we discuss about Accelerator Design, Hardware Integration, and System Software Support.

4.2.1 Accelerator Design

An FPGA-based hardware accelerator was developed using Vitis HLS to implement the computation of the fully connected (FC) layer, one of the key components in neural network architectures. The FC layer performs a weighted summation of inputs, adds a bias, and applies an activation function. Since this operation is extensively used in deep learning models and is computationally demanding, it was selected as the focus of the hardware implementation in this thesis.

The accelerator is designed to support a maximum input size of 32 elements and an output size of 32 elements. It is implemented with float32 precision and incorporates the ReLU activation function. Inputs, weights, and biases are read from block RAM (BRAM), and the computed outputs are written back into BRAM. The computation logic is implemented without advanced optimizations such as pipelining, loop unrolling, or parallelism. This choice was made intentionally to keep the design minimal, focusing on feasibility and correctness rather than maximum performance.

As a result, the accelerator establishes a functional prototype for hardware-software co-design. While the lack of optimizations results in higher latency and reduced throughput, the design successfully demonstrates the practicality of mapping neural network operations onto FPGA hardware. This prototype provides a baseline for future work, where optimizations such as pipelining, quantization, and parallelism can be incorporated to significantly improve efficiency and throughput.

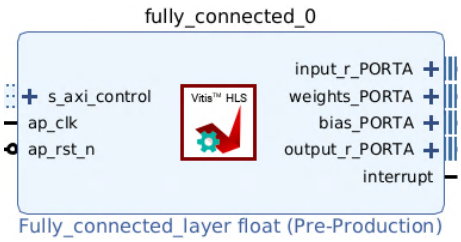


Figure 4.5: FCC IP Block Diagram

4.2.2 Hardware Integration

Integration of the fully connected hardware accelerator was performed using Xilinx Vivado. The process involved creating a block design that connected the Zynq Processing System (PS), the custom FC accelerator IP(Figure4.5), and the required memory and communication infrastructure.

To provide fast and deterministic local storage, dedicated BRAM blocks were instantiated for each data category—input activations, weights, biases, and outputs. This separation ensured organized data management, reduced contention, and simplified the accelerator's memory access logic. Communication between the PS, accelerator IP, and BRAM modules was facilitated using AXI interconnects, where AXI4 master/slave interfaces handled data transfers, and an AXI-Lite interface supported control and configuration.

Importantly, address spaces were explicitly assigned for the FC accelerator's control registers as well as for each BRAM block. This clear memory mapping allowed the PS software and custom drivers to configure the accelerator, write input data and weights, and fetch computed outputs in a predictable and structured manner.

One notable design choice was the exclusion of Direct Memory Access (DMA) for data transfers between DDR memory and BRAM. Instead, data movement relied solely on the AXI4 interface, which simplified the integration but introduced additional latency and throughput limitations. Although this results in reduced performance, the decision aligns with the prototype-oriented nature of this work, prioritizing simplicity and feasibility over raw speed.

Overall, the block design provides a modular and coherent hardware system in which the PS configures and triggers the accelerator, while data is fetched from and written to the respective BRAMs. The integration illustrates how a custom accelerator can be coupled with the processing system and establishes a foundation for future enhancements, where features such as DMA-based transfers and pipelined dataflows could further improve efficiency.

The final Vivado block design is illustrated in Figure 4.6.

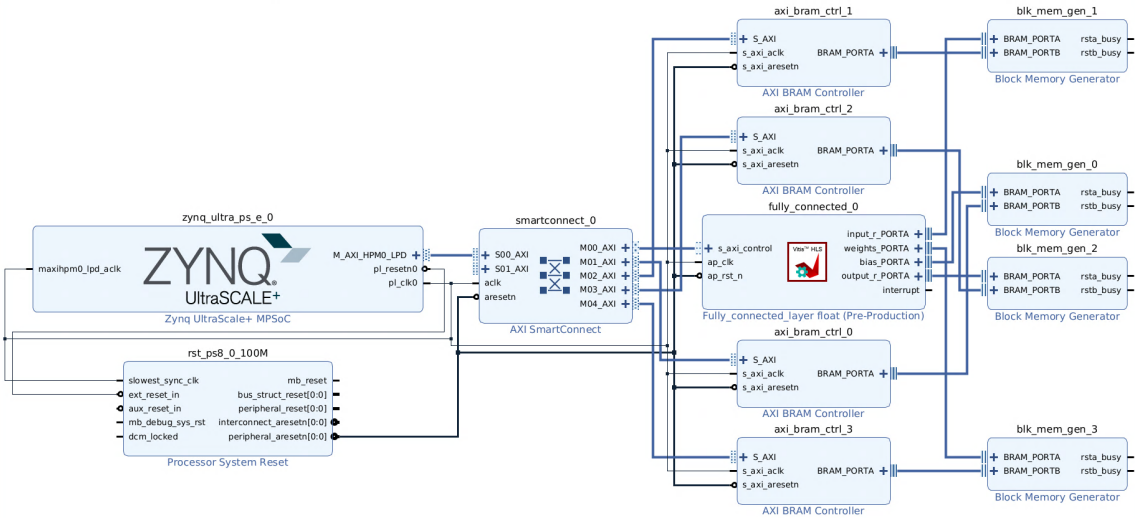


Figure 4.6: FCC Block Diagram

4.3 System Integration

In this section, we discuss about custom TensorFlow Lite delegates, driver development, and end-to-end integration and inferencing.

4.3.1 Custom TensorFlow Lite Delegates

To enable TensorFlow Lite (TFLite) to utilize the custom fully connected (FC) hardware accelerator, a custom TFLite delegate was developed in C++. This delegate acts as a bridge between the TFLite runtime and the FPGA-based accelerator, allowing specific operations to be offloaded to hardware rather than being computed on the CPU. By doing so, the inference process benefits from the specialized parallelism of the accelerator, while maintaining compatibility with the TFLite execution framework.

The delegate implementation follows the standard TFLite delegate architecture and is structured around two main components: the delegate interface and the delegate kernel interface. The delegate interface, which extends from the SimpleDelegateInterface, is responsible for managing the overall lifecycle of the delegate. It is executed during the model loading and planning phase and determines which operations in the model can be supported by the accelerator. Once identified, the supported operators are mapped to delegated nodes, ensuring that these nodes are routed through the hardware accelerator during inference. This component essentially acts as the "planner" for the delegate, performing initialization once at model load time and setting the stage for subsequent execution.

The actual execution of delegated operations is handled by the kernel interface, which inherits from the SimpleDelegateKernelInterface. This interface implements the lifecycle methods required to run the fully connected layer on the hardware. During initialization, the Init method is called once per delegate and receives configuration parameters through TfLiteDelegateParams, setting up the kernel with accelerator-specific configurations. The Prepare method is then called once for each delegated node and ensures that the necessary resources, such as buffer allocations and execution parameters, are correctly set up. The most critical method, Eval, is called every time the node is executed during inference. Within Eval, the logic to offload the computation to the hardware accelerator is implemented. Data is transferred to the accelerator using custom-designed drivers, computation is triggered through control registers, and the outputs are retrieved back into the runtime. This design allows the FPGA accelerator to be seamlessly integrated into TFLite's normal execution pipeline.

During the course of development, an important issue was encountered when running cascaded fully connected layers, where one FC layer feeds directly into another. The default delegate behavior only assigned input and output buffers for each delegated node, without allocating any memory for intermediate outputs. As a result, when a second FC layer attempted to read its input, which should have been the output of the first FC layer, it encountered unallocated memory regions. This led to segmentation faults during inference. The problem does not arise in cases where only a single FC layer is delegated, since the delegate-provided input and output buffers are sufficient. However, for cascaded layers, additional provisions are required to maintain correctness.

To resolve this, a sliding buffer mechanism was introduced into the delegate design. This buffer acts as a temporary storage space for intermediate outputs, ensuring that the results of one fully connected layer are preserved and made available as inputs to the next. In effect, the sliding buffer provides a controlled bridge between cascaded layers, preventing invalid memory access and ensuring that multi-layer architectures can run reliably on the hardware accelerator. Although this introduces a small amount of memory overhead, it is a necessary addition to guarantee correctness and stability in more complex network topologies.

Overall, the custom delegate design demonstrates the feasibility of offloading fully connected layers to an FPGA-based accelerator within the TFLite runtime. By carefully managing the delegate lifecycle and addressing issues such as intermediate buffer allocation, the delegate achieves a robust and scalable integration of hardware acceleration.

The successful operation of the custom delegate relies not only on the delegate framework itself but also on the underlying communication between the TFLite runtime and the FPGA accelerator. This communication is made possible through carefully designed device drivers, which provide the low-level interface to the accelerator hardware. While the delegate defines how operations are mapped and executed within the TFLite runtime, the drivers ensure that data movement, control signaling, and synchronization with the accelerator occur reliably and efficiently. In the following subsection, we present the development of these custom drivers, explaining their role in enabling seamless interaction between the processing system, the accelerator's BRAMs, and the control registers.

4.3.2 Driver Development

To facilitate communication between the TensorFlow Lite (TFLite) runtime and the custom fully connected (FC) hardware accelerator, a Linux kernel device driver was developed in C++. This driver acts as the low-level interface that enables the TFLite delegate to interact with the FPGA-based accelerator, handling tasks such as data transfer, control signaling, and synchronization between the processing system and the hardware. Without this driver layer, the delegate would be unable to issue commands to the accelerator or manage the flow of input and output data during inference.

In this thesis, two types of drivers were developed: BRAM drivers and IP drivers. All drivers use memory-mapped I/O through the /dev/mem file system, which allows user-space applications to directly access specific physical memory regions corresponding to the accelerator's block RAMs and control registers.

The BRAM drivers are responsible for managing the data buffers that reside in the FPGA's block RAMs. Dedicated APIs were designed to support key operations, including writing weights to the weight BRAM, writing biases to the bias BRAM, writing input activations to the input BRAM, reading outputs from the output BRAM, and clearing the contents of the BRAMs when required. By abstracting these low-level memory operations into clean APIs, the driver simplifies the interaction between the software and the accelerator's local memory system.

The IP driver, on the other hand, is responsible for configuring and controlling the accelerator itself. It provides APIs to write values into the IP's control and data registers and to read back the status of execution from these registers. This allows the software to determine whether the accelerator is idle, running, or has completed a computation. To make the interface user-friendly, a higher-level API was created that encapsulates these detailed operations. Through this single function, the user can provide input data, trigger computation, and read back the results, without manually managing individual control registers or synchronization steps.

Execution tracking between the software and the accelerator was implemented using a polling-based technique rather than relying on hardware interrupts. This design decision was made to keep the system simple and minimize development overhead, as interrupts would require additional complexity in the driver and synchronization logic. While polling introduces extra latency and

is less efficient compared to interrupt-driven execution, it provides a straightforward and reliable mechanism suitable for rapid prototyping. Given the focus of this work on feasibility and proof-of-concept, the trade-off of increased latency was accepted in favor of simplicity and faster development turnaround.

4.3.3 End-to-End Integration and Inferencing

The final step of the project involved integrating all components to enable end-to-end inference using the custom fully connected (FC) hardware accelerator within the TensorFlow Lite (TFLite) framework, as illustrated in Figure 4.7. This stage required careful coordination between the data preprocessing pipeline, the TFLite model with the custom delegate, and the Linux kernel drivers that interface with the FPGA-based accelerator.

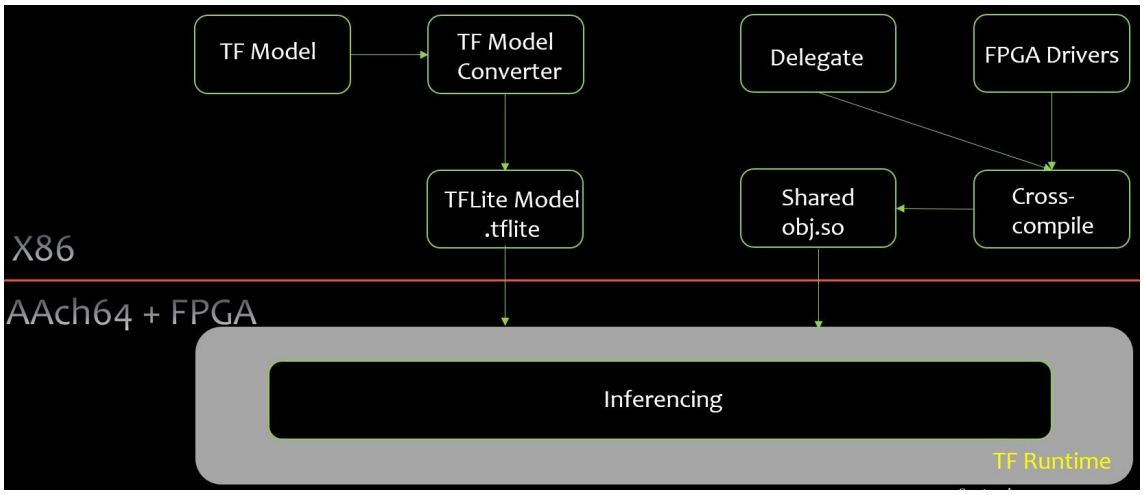


Figure 4.7: Integration and Inferencing Process

To accomplish this, a PetaLinux image was created and built for the Zynq UltraScale platform. From the block design described in the earlier section, the FPGA bitstream was generated and incorporated into the PetaLinux system. Alongside this, a shared object library containing both the custom delegate and the device drivers was cross-compiled for the ARM64 architecture using a TensorFlow Docker image as the build environment. This ensured compatibility with the ARM Cortex-A53 cores on the Zynq device. The generated TFLite model files were then copied into the PetaLinux root file system, ensuring that all components necessary for deployment were available on the target device.

Once the system was prepared, the bitstream was loaded into the Zynq FPGA to configure the programmable logic with the custom FC accelerator. An inference script was developed to complete the integration. This script was responsible for loading the TFLite model, preparing the input data in the correct format, and invoking the TFLite interpreter with the custom delegate enabled. When executed, the interpreter dispatched relevant operations to the FPGA-based accelerator through the delegate and driver stack, while other operations continued to run on the ARM cores.

In addition, the TensorFlow Lite benchmarking tool was also cross-compiled for the ARM64 architecture. This tool was deployed to the target system to measure and analyze the inference performance of the custom accelerator against CPU-only execution. The detailed results of these benchmark experiments are presented in the Results section of this thesis.

5 Results

5.1 Experimental Setup

The project was developed on a Zynq UltraScale+ MPSoC development board, which integrates a dual-core ARM Cortex-A53 processor together with programmable logic (PL) fabric, enabling heterogeneous hardware–software co-design. The toolchain was standardized to ensure compatibility and reproducibility, with the following versions being used throughout the development process: Vitis HLS 2023.2, Vivado 2023.2, Vitis IDE 2023.2, and PetaLinux 2023.2. For software development and cross-compilation tasks, Python 3.10 was employed. In particular, the tensorflow/build:latest-python3.10 Docker image was used to cross-compile both the custom delegate and the associated drivers for the ARM64 architecture, ensuring that the binaries could be executed on the Cortex-A53 cores of the Zynq platform.

Experimental trials were conducted on TensorFlow Lite (TFLite) models containing fully connected (FC) layers with varying input and output sizes as well as different levels of cascading (i.e., single vs. multiple FC layers connected sequentially). To evaluate performance, TensorFlow's benchmarking tool was cross-compiled for ARM64 and deployed on the target system, enabling a direct comparison of inference latency between CPU execution and FPGA-accelerated execution. Additionally, TensorBoard was utilized during the training phase to visualize the evolution of training and validation loss and accuracy, helping to monitor convergence and mitigate overfitting.

On the hardware side, resource utilization, timing, and latency metrics were obtained from the Vitis HLS synthesis reports, while power consumption estimates were derived from the Vivado power analysis reports. Furthermore, scheduling reports were analyzed to understand the pipeline efficiency and identify bottlenecks in the hardware accelerator design.

Model training was performed on a high-performance workstation equipped with an 8-core Intel(R) Xeon(R) E5-2640 CPU @ 2.50 GHz and 40 GB of RAM, providing sufficient computational capacity for iterative experimentation. The training datasets were derived from single-phase current and voltage signals captured using a PQ-Box 150 power quality analyzer. These inputs were chosen to capture diverse power quality scenarios and ensure the model's robustness under varying operating conditions.

The fully connected hardware accelerator was designed and synthesized using Vitis HLS, while system-level integration was carried out using Vivado for block design creation and PetaLinux for building the embedded Linux environment. The custom delegates and device drivers, developed in C++, provided the critical software bridge between TensorFlow Lite and the FPGA-based accelerator. On the machine learning side, the predictive model was implemented using TensorFlow and Keras in Python, with preprocessing and data handling automated through Python scripts.

5.2 Experimental Results

5.2.1 Model Performance

Accuracy and loss metrics

As shown in Table 5.1, the model achieved a low mean squared error (MSE) and mean absolute error (MAE) for the magnitude component, whereas the phase component exhibited a higher error rate. This discrepancy can be attributed to the inherent complexity of phase information. Unlike magnitude, phase data is more variable and highly sensitive to noise, which makes it more challenging for the model to learn stable and accurate representations.

Figure 5.1 presents the input time-domain signals along with the corresponding magnitude and phase labels. Figure 5.2 further illustrates the model's predictions, where it can be observed that the magnitude predictions closely track the ground truth, while the phase predictions show larger deviations.

The training behavior is shown in Figures 5.3 and 5.4. The model converged well for magnitude, as reflected by the steady decrease in loss, whereas the phase loss remained relatively high throughout training. This indicates that the model struggled to capture the cyclic and discontinuous nature of phase data. Such discontinuities often introduce learning challenges for neural networks, leading to reduced prediction accuracy.

Since the primary goal of this thesis was to establish a proof-of-concept for hardware acceleration, the higher error in phase predictions was accepted as a limitation of the current model architecture and training approach. Future work may explore specialized loss functions or phase-unwrapping techniques to better handle phase data.

Dataset	**Magnitude Loss**		**Phase Loss**		**Total Loss**
	MSE %	**MAE %**	**MSE %**	**MAE %**	
Training	0.01	0.2	4.75	14.99	4.76
Validation	0.01	0.19	7.21	21.14	7.22
Test	0.01	0.2	7.35	22.4	7.37

Table 5.1: Model Performance Metrics

Impact of TensorFlow Lite conversion

The converted TFLite model was evaluated on a test dataset to compare its performance with the original Keras model. A small accuracy drop of 0.5% was observed, while the model size decreased significantly from 2.94 MB to 805 KB. The average inference time of the Keras model on the target ARM Cortex-A53 processor was 647.431 ms, measured using a custom Python benchmarking script with the `time` module. In contrast, the TFLite model achieved an average inference time of 36.786 ms, measured using the official TensorFlow Lite benchmark tool compiled for the AArch64 architecture. These results demonstrate that the TFLite conversion effectively balances model size and execution speed, with only a minor reduction in accuracy, making it highly suitable for deployment on resource-constrained edge devices. The substantial speedup is primarily due to TFLite's optimized runtime for ARM64 and lower computational overhead compared to the full Keras model.

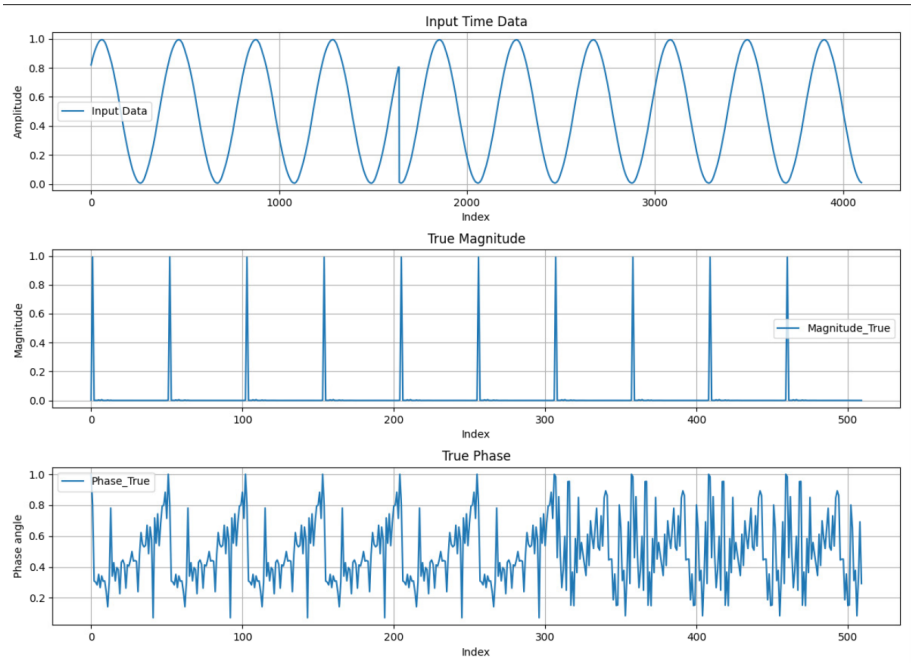


Figure 5.1: Input Data and Corresponding Labels

5.2.2 Hardware Accelerator Evaluation

Resource Utilization:

The table 5.2 presents the utilization of Programmable Logic (PL) resources for the implementation of a fully connected (FC) layer IP block with a sigmoid activation function. This block is designed to execute FC layers with a maximum input size of 32 and an output size of 32, as specified by the user. The implementation uses float32 precision and does not incorporate advanced optimizations such as pipelining, loop unrolling, or parallelism. The on-chip memory usage within the IP is minimal, as the data is primarily stored in external BRAM rather than inside the IP itself.

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	102	-
FIFO	-	-	-	-	-
Instance	-	3	311	579	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	83	-
Register	-	-	105	-	-
Total	0	3	416	764	0
Available	150	240	94464	47232	0
Utilization(%)	0	1	~0	1	0

Table 5.2: FCC IP Resource Utilization Summary

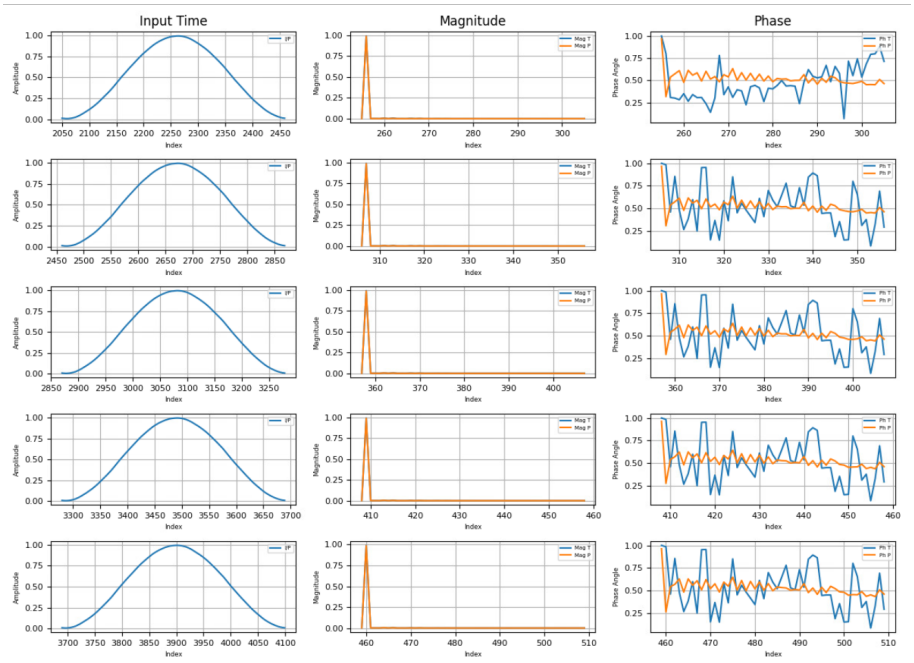


Figure 5.2: Magnitude and Phase Predictions

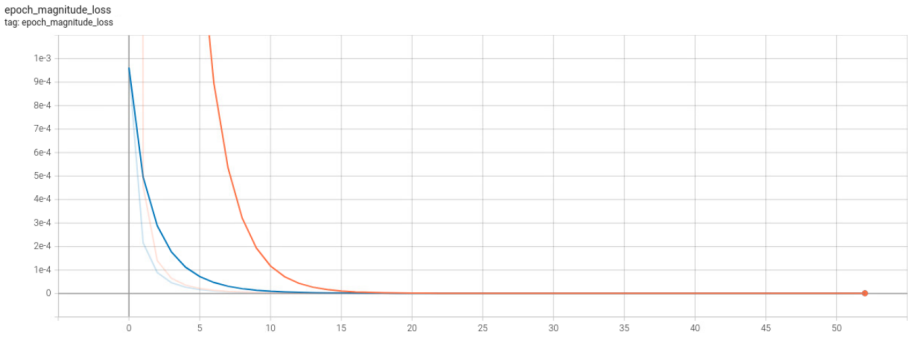


Figure 5.3: Epoch Magnitude Loss

The post-implementation resource utilization summary as showin in Table 5.3 provides a comprehensive overview of the FPGA resource consumption after synthesizing and implementing which includes the resource consumed by the fully connected (FC) layer IP, BRAM blocks, AXI interconnects, and other necessary components for the complete design. As shown in Figure 5.5, the design utilizes a moderate portion of the available resources on the FPGA, leaving ample headroom for future enhancements or additional functionality. The LUT and FF utilization are well within acceptable limits, indicating that the design is efficient and does not overly tax the programmable logic fabric. The BRAM usage is significant, reflecting the reliance on block RAM for storing weights, biases, inputs, and outputs, which is typical for neural network accelerators that require fast local memory access. The DSP usage is minimal, as the current design does not leverage extensive parallelism or pipelining, which would typically increase DSP consumption. Overall, the resource utilization indicates a balanced design that effectively meets the requirements of the FC layer while maintaining flexibility for future optimizations.

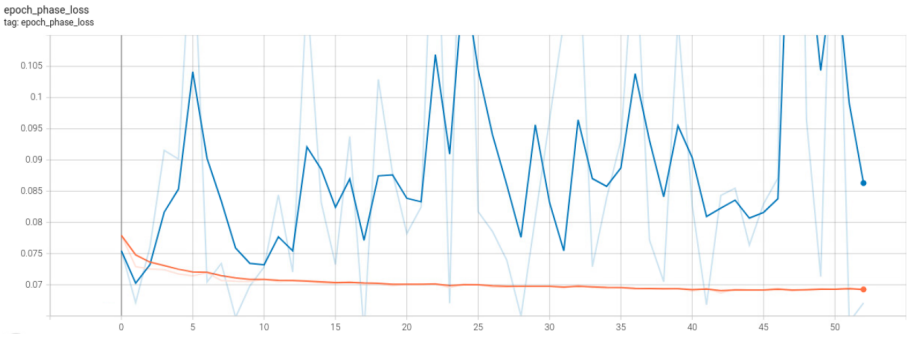


Figure 5.4: Epoch Phase Loss

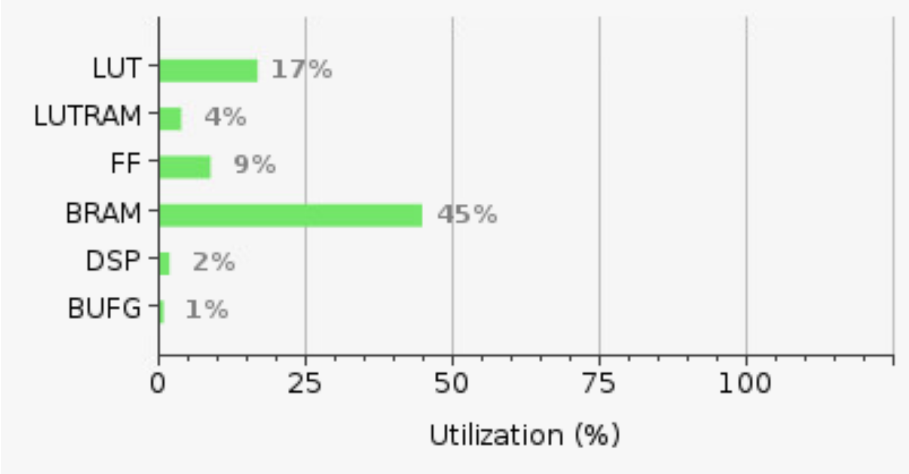


Figure 5.5: Post-Implementation Resource Utilization

Resource	Utilization	Available	Utilization (%)
LUT	7843	47232	16.61
LUTRAM	1146	28800	3.98
FF	8309	94464	8.80
BRAM	67	150	44.67
DSP	4	240	1.67
BUFG	1	196	0.51

Table 5.3: FPGA Resource Utilization Summary

Latency and Timing:

The timing summary in Table 5.4 indicates that the design meets the target clock period of 10.00 ns, with an estimated clock period of 6.713 ns and an uncertainty of 2.70 ns. This suggests that the design is capable of operating at a frequency higher than the target, providing a margin for timing variations due to process, voltage, and temperature (PVT) changes.

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.713 ns	2.70 ns

Table 5.4: Timing Summary

The latency summary in Table 5.5, Table 5.6, and Table 5.7 provides detailed information about the performance of the design. The overall latency ranges from a minimum of 16 cycles to a maximum of 199,681 cycles, with an interval spanning from 17 to 199,682 cycles. This wide variability in execution time reflects the dependence of the Fully Connected (FCC) layer on several factors, including the size of its input and output tensors and the specific data being processed. For example, smaller input vectors or sparse weight matrices lead to shorter execution times, whereas larger inputs or fully dense matrices result in longer latencies.

The absence of hardware optimizations such as pipelining or parallelism further contributes to this variability. In a pipelined design, multiple operations can overlap in execution, effectively reducing the overall latency for sequential inputs. However, since the current design processes operations sequentially, each input must complete before the next begins, which directly impacts the observed interval. Specifically, the minimum interval of 17 cycles is calculated as the minimum latency (16 cycles) plus one additional cycle

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
16	199681	0.160 us	1.997 ms	17	199682	no

Table 5.5: Latency Summary

Module	Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
	min	max	min	max	min	max	
fully_connected_FC_IN_LOOP	10	775	0.100 us	7.750 us	10	775	no

Table 5.6: Instance Detail

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval	Pipelined
	min	max		target	
-FC_OUT_LOOP	15	199680	15–780	-	no

Table 5.7: Loop Summary

Power Consumption:

The power analysis summary in Table 5.8 provides an overview of the estimated power consumption and thermal characteristics of the FPGA design. The total on-chip power is estimated to be 1.996 W, which includes both static and dynamic components. The junction temperature is calculated to be 29.7 °C, indicating that the device operates well within safe thermal limits, given a thermal margin of 70.3 °C (corresponding to a maximum allowable junction temperature of 100 °C). The effective thermal resistance (θ_{JA}) is determined to be 2.4 °C/W, reflecting the efficiency of heat dissipation from the junction to the ambient environment.

The total on-chip power can be expressed as: Total On-Chip Power=Static Power+Design Dynamic Power

It is also referred to as "thermal power," as it represents the total power dissipated on-chip from all supply sources. Static Power: Power consumed by the device when there is no switching activity. It can be further divided into: Device Static: This includes transistor leakage power when the device is powered but not configured. Design Static: Power consumed when the device is configured but idle, with no switching activity. This includes static power in I/O DCI terminations. Design Dynamic Power: This represents the average power consumed due to user logic utilization and switching activity. It reflects the actual dynamic workload executed by the FPGA design.

Parameter	**Value**
Total On-Chip Power	1.996 W
Junction Temperature	29.7 °C
Thermal Margin	70.3 °C (29.5 W)
Effective θJA	2.4 °C/W
Power Supplied to Off-Chip Devices	0 W

Table 5.8: Power Analysis Summary

The power breakdown shown in Figure 5.6 illustrates the distribution of power consumption across various components of the FPGA design. The majority of the dynamic power, approximately 89%, is consumed by the Processing System (PS), which includes the ARM Cortex-A53 cores. This is expected, as the PS handles general-purpose computing tasks and runs the operating system, which typically requires more power than the programmable logic (PL) section.

In contrast, the programmable logic accounts for 11% of the dynamic power, including the custom fully connected (FC) layer accelerator and other logic elements. Notably, the FCC block itself consumes only 3% of the dynamic power, indicating that the hardware accelerator is relatively power-efficient. The remaining power is distributed among other components such as memory interfaces, I/O blocks, and clock management tiles, which contribute to the overall thermal and power profile of the FPGA.

5.2.3 End-to-End system performance

Inference latency (CPU vs Accelerator)

The inference timing statistics in Table 5.9 compare the performance of the system when running inference using only the CPU versus using the CPU in conjunction with the FPGA-based fully connected (FC) layer accelerator. The results are based on 150 inferences, comprising 50 warm-up runs followed by 100 measured runs, which ensures stable performance metrics and mitigates the effects of initial overheads.

As shown in Table 5.9, the average inference time when using only the CPU is 36,785.8 μs, while the average time when utilizing the FPGA accelerator is slightly higher at 36,994.2 μs. This increase is primarily due to the overhead associated with transferring data from the DDR memory to the block RAMs, reading input values from DDR, writing output values back to DDR, and the execution of the FPGA design, which is intentionally kept unoptimized for rapid prototyping purposes. The

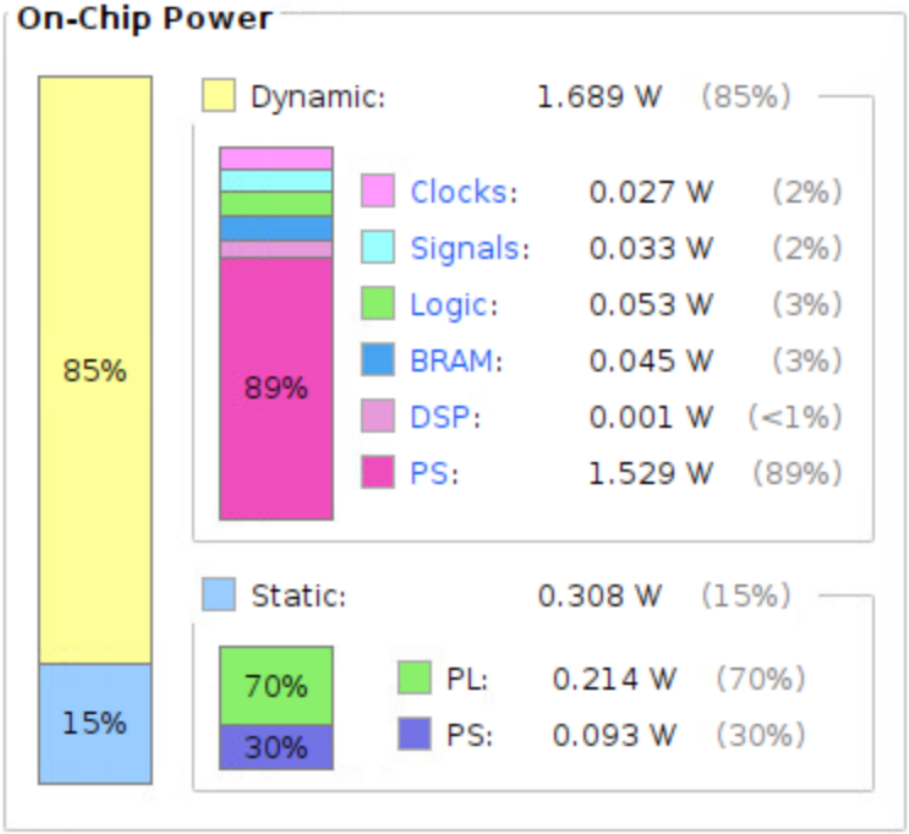


Figure 5.6: On-Chip Power Utilization

minimum and maximum inference times also reflect this trend. Notably, the initialization time for the CPU-only case is 7,835 μs, whereas for the CPU + FPGA case, it is 12,333 μs. This additional initialization overhead arises from configuring the FPGA and loading the weights and biases into block RAM, which are bulkier and require on-time transfers compared to the input activations.

Efficiency gains (speed-up factor)

The speed-up factor is calculated by comparing the average inference times of the CPU-only execution and the CPU + FPGA execution. It is found that the FPGA-accelerated system is approximately 0.57% slower than the CPU-only system. This slight decrease in performance is primarily attributed to the overhead associated with data transfers, since the input values must be read from DDR memory and written into the FPGA's block RAMs, and the output results must be transferred back to DDR. These additional memory transactions introduce significant latency when compared to the direct on-chip accesses of the CPU. Furthermore, the current FPGA implementation is not fully optimized; although FPGAs are designed to exploit massive parallelism, in our design the execution happens sequentially to enable rapid prototyping, which naturally leads to slower performance. The accelerator also runs at a relatively low clock frequency of 100 MHz compared to the CPU's 1.5 GHz, and lacks optimizations such as deep pipelining and parallel computation. Together, these factors cause the CPU + FPGA setup to be slightly slower. However, with improvements in data transfer mechanisms (for instance, using high-throughput DMA) and optimizations in the accelerator design (such as pipelining, loop unrolling, and exploiting

Metric	CPU Only	CPU + FPGA
Count	150	150
First	36878	37037
Current	36736	36985
Minimum	36730	36936
Maximum	37106	37302
Average	36785.8	36994.2
Standard Deviation	73	80
5th Percentile (p5)	36736	36939
Median	36771	36970
95th Percentile (p95)	36929	37249
Init Time	7835	12333
First Inference	37214	37372
Warmup (avg)	36793.6	37004
Inference (avg)	36785.8	36994.2

Table 5.9: Inference Timing Statistics (in μs)

parallelism), the FPGA-accelerated system can be expected to significantly outperform the CPU-only execution. Such improvements will be particularly advantageous for larger models or more complex operations, where the inherent parallelism of FPGAs can be fully leveraged.

5.2.4 Integration and Usability

The TensorFlow Lite delegate demonstrates robust integration with the hardware accelerator, effectively offloading fully connected layers with a maximum of 32 inputs and 32 outputs onto the FPGA. The delegate integrates seamlessly with the TFLite runtime, allowing models with supported layers to leverage hardware acceleration without requiring modifications to the high-level model code. It manages the mapping of operations to the accelerator, coordinates data transfer, and ensures proper synchronization, providing a transparent and user-friendly experience.

The driver layer provides a low-level interface for communication between the TFLite runtime and the FPGA accelerator. The drivers handle memory-mapped I/O, enabling efficient and deterministic data transfer to and from the accelerator's block RAMs (BRAMs) and control registers. By abstracting the complexity of hardware interactions, the driver APIs allow the delegate to focus on high-level operation mapping, improving maintainability and modularity.

The custom delegate has been designed with extensibility in mind, allowing future support for additional operators such as convolutional layers, pooling layers, and activation functions, thereby expanding the range of models that can benefit from hardware acceleration. Similarly, the driver layer is structured to support FPGA-implemented operators, including matrix multiplication, convolution, and activation functions, ensuring seamless integration with the delegate and providing a scalable platform for accelerating more complex neural network operations. Additionally, the system supports hot-swappable FPGA bitstreams, allowing new accelerator designs to be loaded onto the

programmable logic without rebooting the system. This capability enables on-the-fly upgrades of hardware accelerators, ensuring that performance improvements and new features can be deployed rapidly, enhancing flexibility and prolonging the utility of the embedded platform.

Bazel scripts have been developed to integrate the custom delegate and drivers into a single shared object file, which can be loaded directly by the TFLite runtime. This approach guarantees that all necessary components are available at runtime, simplifying deployment and reducing potential configuration errors. By consolidating the delegate and driver into a unified package, the system ensures both portability and ease of use, which are critical for embedded and resource-constrained environments.

6 Future Scope

6.1 Software improvements

Phase unrolling and differencing can be applied to improve the accuracy of phase values by reducing discontinuities and capturing finer variations. Several windowing approaches can also be employed to minimize spectral leakage, thereby preserving the true frequency components of the signal. Instead of edge padding, alternative padding methods may be used to reduce distortions at signal boundaries and better preserve waveform continuity. Furthermore, binning can be performed more effectively by carefully processing the signal so that energy is concentrated within the harmonic bins, thereby reducing the margin of error from ±5Hz and enabling more accurate signal reconstruction.

In this thesis, dual-branched models have been adopted; however, alternative modeling approaches could be explored to further enhance accuracy. During TensorFlow Lite (TFLite) conversion, float32 quantization was used. Other quantization strategies, such as dynamic range quantization, float16, or integer quantization, could be applied to achieve additional reductions in model size and improvements in inference time, particularly for deployment on resource-constrained devices.

6.2 Hardware improvements

The current fully connected (FC) layer hardware accelerator is designed to handle a maximum input and output size of 32. To support larger models, the design can be scaled to accommodate greater input and output dimensions. In the proposed improvement, inputs and outputs can be stored in DDR memory, while weights and biases—which remain fixed for a given model—can be stored in BRAM. This ensures that BRAM is used efficiently for frequently accessed static parameters, while DDR provides scalable storage for dynamic activations.

To further optimize memory handling, Direct Memory Access (DMA) can be employed to enable high-speed data transfers between DDR memory and the FPGA accelerator. Effective use of parallelism and pipelining techniques can also be incorporated into the design to increase throughput and reduce latency, both of which are crucial for real-time inference.

Beyond the fully connected layer, other time-critical operations can also be implemented as custom IP blocks on the FPGA to offload additional computations from the CPU. Furthermore, interrupt-based execution can replace polling to synchronize the CPU and FPGA more efficiently, reducing CPU overhead and improving overall system responsiveness.

7 Conclusion

This work set out to explore hardware–software co-design for accelerating time-series inference on Zynq SoC platforms. A data extraction pipeline was developed to convert PQBox150 time-series data from the proprietary .pqf format into .csv files, making the dataset accessible for further processing. The extracted data was then preprocessed, and a time-series model was created, trained, and subsequently converted into TensorFlow Lite (TFLite) format to enable deployment on embedded systems.

To accelerate the most computationally intensive operation, a custom fully connected (FC) layer hardware accelerator was designed using Vitis HLS and integrated into the programmable logic of the FPGA. Custom Linux device drivers were developed to enable efficient communication between the TFLite runtime and the FPGA-based accelerator, abstracting the low-level hardware signaling and ensuring smooth operation without user intervention. Furthermore, a custom TensorFlow Lite delegate was implemented, allowing the FC layers to be seamlessly offloaded to the hardware accelerator within the TFLite framework.

The complete system was deployed on a Zynq UltraScale+ MPSoC platform running PetaLinux, enabling end-to-end inferencing with hardware acceleration. This approach demonstrated the feasibility of integrating domain-specific accelerators with modern ML runtimes, ensuring compatibility with existing workflows while unlocking the potential for reduced latency and more efficient execution on resource-constrained edge devices.

Bibliography

[A. Eberle24] A. Eberle. *Mobiler Netzanalysator PQ Box 150*. 2024. URL: https://www.a-eberle.de/produkte/mobiler-netzanalysator-pq-box-150/ (cit. on pp. 26, 29, 30).

[AJ20] Abhishek Jain. *Understanding the 1D Convolutional Layer in Deep Learning*. 2020. URL: https://medium.com/@abhishekjainindore24/understanding-the-1d-convolutional-layer-in-deep-learning-7a4cb994c981 (cit. on p. 18).

[AMD23] AMD. *Zynq 7000 SoC Technical Reference Manual (UG585)*. 2023. URL: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/Programmable-Logic-Features-and-Descriptions (cit. on p. 13).

[AMD24a] AMD. *Vitis High-Level Synthesis User Guide (UG1399)*. 2024. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls (cit. on pp. 14, 24).

[AMD24b] AMD. *Vitis Model Composer User Guide (UG1483)*. 2024. URL: https://docs.amd.com/r/en-US/ug1483-model-composer-sys-gen-user-guide/AXI-Interface (cit. on p. 23).

[aoifem23] aoifem. *Introduction to PetaLinux Part 1*. 2023. URL: https://support.xilinx.com/s/article/968413?language=en_US (cit. on pp. 14, 25).

[ARM24] ARM. *Learn the architecture - An introduction to AMBA AXI*. 2024. URL: https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview (cit. on p. 23).

[ASG23] Alexander S. Gillis. *Definition deep learning*. 2023. URL: https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network (cit. on p. 17).

[ASG24] Alexander S. Gillis. *Definition device driver*. 2024. URL: https://www.techtarget.com/searchenterprisedesktop/definition/device-driver (cit. on p. 26).

[BS21] Benoît Steinmann. *Xilinx Vitis HLS introduction*. 2021. URL: https://imperix.com/doc/help/xilinx-vitis-hls (cit. on p. 24).

[CHC19] Choon-Hin Chang. *Is Jitter Good or Bad for Your Device Testing?* 2019. URL: https://www.keysight.com/blogs/en/tech/bench/2019/11/07/is-jitter-good-or-bad-for-your-device-testing (cit. on p. 13).

[CR19] J. Chen, X. Ran. "Deep Learning With Edge Computing: A Review". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674. DOI: 10.1109/JPROC.2019.2921977 (cit. on p. 27).

[DC24] DataCamp. *LSTM Cell*. 2024. URL: https://assets.datacamp.com/production/repositories/6312/datasets/78df4fb88fee52348806f69eb00f17615b7d0c8f/lstm_cell.png (cit. on p. 19).

[GFG22a] GeeksforGeeks. *Device Driver and its Purpose*. 2022. URL: https://www.geeksforgeeks.org/device-driver-and-its-purpose/ (cit. on p. 25).

[GFG22b] GeeksforGeeks. *What is Fully Connected Layer in Deep Learning?* 2022. URL: https://www.geeksforgeeks.org/deep-learning/what-is-fully-connected-layer-in-deep-learning/ (cit. on p. 20).

[Google24a] Google. *LITERT: A Lightweight Inference Toolkit for Edge AI*. 2024. URL: https://ai.google.dev/edge/litert (cit. on p. 20).

[Google24b] Google. *LITERT: Performance and Delegates*. 2024. URL: https://ai.google.dev/edge/litert/performance/delegates (cit. on pp. 20, 21).

[HEINEN24] HEINEN. *FPGA*. 2024. URL: https://heinen-elektronik.de/glossar/fpga/ (cit. on p. 22).

[JC24] Jose Cano. *Accelerating AI at the Edge*. 2024. URL: https://www.doc.ic.ac.uk/~phjk/NANDA24/Slides/Accelerating%20AI%20at%20the%20Edge%20-%20Jose%20Cano.pdf (cit. on pp. 27, 28).

[LZZC19] E. Li, L. Zeng, Z. Zhou, X. Chen. "Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing". In: *IEEE Transactions on Wireless Communications* (2019). DOI: https://doi.org/10.1109/twc.2019.2946140 (cit. on p. 27).

[NI24] NATIONAL INSTRUMENTS. *FPGA Fundamentals: Basics of Field-Programmable Gate Arrays*. 2024. URL: https://www.ni.com/en/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module/fpga-fundamentals.html (cit. on p. 22).

[PSF24] Python Software Foundation. *Python*. 2024. URL: https://www.python.org/doc/essays/blurb/ (cit. on p. 26).

[SCYE17] V. Sze, Y.-H. Chen, T.-J. Yang, J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740 (cit. on p. 27).

[SS20] Pooja Mahajan. *Fully Connected vs Convolutional Neural Networks*. 2020. URL: https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5 (cit. on p. 19).

[SS24] Shubhi Saxena. *Beyond Tasks: Reimagining Workflows in the Age of AI*. 2024. URL: https://medium.com/life-at-quizizz/beyond-tasks-reimagining-workflows-in-the-age-of-ai-3aea6d679941 (cit. on p. 13).

[T23] Taifur. *Path to Programmable III: Training Blog 05 - Inter-communication between PS and PL of Zynq SoC*. 2023. URL: https://community.element14.com/challenges-projects/design-challenges/pathprogrammable3/b/blog/posts/path-to-programmable-iii-training-blog-05-inter-communication-between-ps-and-pl-of-zynq-soc (cit. on p. 13).

[Trenz Electronic24] Trenz Electronic. *MPSoC Modul mit AMD Zynq UltraScale+ ZU2CG-1E, 2 GByte DDR4 SDRAM, 4 x 5 cm*. 2024. URL: https://www.trenz-electronic.de/de/MPSoC-Modul-mit-AMD-Zynq-UltraScale-ZU2CG-1E-2-GByte-DDR4-SDRAM-4-x-5-cm/TE0820-05-2AE81MA (cit. on p. 21).

[Xilinx24a] Xilinx. *PetaLinux SDK*. 2024. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html (cit. on p. 25).

[Xilinx24b] Xilinx. *Vivado Design Suite*. 2024. URL: https://www.xilinx.com/products/design-tools/vivado.html (cit. on p. 24).

[Xilinx24c] Xilinx. *Zynq UltraScale+ Product Selection Guide*. 2024. URL: https://www.mouser.com/pdfDocs/zynq-ultrascale-plus-product-selection-guide.pdf?srsltid=AfmBOoqzEHD5nOr2-FaGtlfuDjcrcYN5-XFDEjarSin86lnFt8J2mvup (cit. on pp. 21, 22).

[ZR20] Zhao, R., et al. *Edge AI: A survey*. 2020. URL: https://www.sciencedirect.com/science/article/pii/S2667345223000196#sec6 (cit. on pp. 13, 27, 28).

All links were last followed on September 15, 2025.