FPGA-Based Hardware Accelerators for Deep Learning in Mobile Robotics

Master of Science in Technology Thesis
University of Turku
Department of Computing
Robotics and Autonomous Systems
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
2023
Yasir Al-Ameri

Supervisors:
Prof. Tomi Westerlund
MSc. Minh Nguyen

UNIVERSITY OF TURKU
Department of Computing

YASIR AL-AMERI: FPGA-Based Hardware Accelerators for Deep Learning in Mobile Robotics

Master of Science in Technology Thesis, 68 p.
Robotics and Autonomous Systems
Turku Intelligent Embedded and Robotic Systems (TIERS) Lab
November 2023

The increasing demand for real-time low-power hardware processing systems, endowed with the capacity to perform compute-intensive applications, accentuated the inadequacy of the conventional architecture of multicore general-purpose processors. In an effort to meet this demand, edge computing hardware accelerators have come to the forefront, notably with regard to deep learning and robotic systems. This thesis explores preeminent hardware accelerators and examines the performance, accuracy, and power consumption of a GPU and an FPGA-based platform, both specifically designed for edge computing applications. The experiments were conducted using three deep neural network models, namely AlexNet, GoogLeNet, and ResNet-18, trained to perform binary image classification in a known environment. Our results demonstrate that the FPGA-based platform, particularly a Kria KV260 Vision AI starter kit, exhibited an inference speed of up to nine and a half times faster than that of the GPU-based Jetson Nano developer kit. Additionally, the empirical findings of this work reported as much as a quintuple efficiency over the Jetson Nano in terms of inference speed per watt with a mere 5.4% drop in accuracy caused by the quantization process required by the FPGA. However, the Jetson Nano showed a 1.6 times faster inference rate with the AlexNet model over the KV260 and its deployment process proved to be less challenging.

Keywords: Edge computing, FPGA, DNN, DPU, Kria KV260, Vitis AI

Contents

List of Figures

List of Tables

List Of Acronyms

APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
BLAS	Basic Linear Algebra Subroutines
BRAM	Block Random Access Memory
CLB	Configurable Logic Blocks
CPLD	Complex ProgrammableLogic Devices
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network models
DPU	Deep Learning Processor Unit
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphical Processing Units
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
KRS	Kria Robotics Stack
LUT	Look Up Tables
Pmod	Peripheral modules
PMU	Platform Management Unit
QSPI	Quad serial peripheral interface

ReLU	Rectified Linear Units
ROS	Robotic Operating System
RPU	Real-time Processing Unit
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessors
SOM	System On Module
SPLD	Simple Programmable Logic Devices
TPU	Tensor Processing Unit
VART	Vitis AI Runtime

1 Introduction

The exponential growth in computational power, which followed Moore's law by doubling the number of transistors in an Integrated Circuit (IC) roughly every eighteen months for at least four decades, has been slowing down since the mid-2000s [1]. One main reason for this faltering in transistor density is the physical size limit of transistors [2]. As transistors approach a nanoscale, they become more susceptible to subatomic interactions such as quantum entanglement and tunneling. These quantum effects cause reliability issues and precipitate an increase in power consumption caused by current leakage [2], [3]. Additionally, sophisticated fabrication techniques are required for developing and manufacturing such small transistors, which increases the overall cost [4]. Consequently, chip manufacturers and software developers shifted their focus to multicore processors in order to keep pace with the increasing demand for processing capabilities driven by Machine Learning (ML) and Artificial Intelligence (AI) applications, such as in robotics and autonomous systems [5].

However, this new design trend faced two main challenges due to the upsurge in the number of cores amalgamated within a single IC. The first challenge is the heat accumulated when all cores operate at their maximum speed [6]. This heat accretion leads to a tremendous increase in energy expenditure, a notable decrease in performance, and might ultimately result in component failure [6]. The second challenge arises from Amdahl's law, which affirms that the achievable speedup of a program is constrained by the sequential parts of the code regardless of the number of cores, as described in (1.1) [7]

$$S = 1/[(1 - P) + (P/N)] \tag{1.1}$$

Where S is the speedup achieved by parallelization, P represents the part of the code amenable to parallel processing, while N denotes the number of cores. With these two challenges, it is evident that relying solely on a multicore design paradigm is inadequate. Therefore, the attention of giant chip manufacturers has been redirected toward custom hardware platforms that are capable of accelerating compute-intensive applications by offloading a specific part of the computation or the entire task from the conventional general-purpose Central Processing Unit (CPU) [8]. The aim of this thesis is to explore the most prominent hardware accelerators that can be used in autonomous mobile robotics and to compare the two commonly used platforms, namely Graphical Processing Units (GPU)s and Field Programmable Gate Array (FPGA)s in terms of accuracy, development process, and power efficiency in relation to operational performance. In our work, we used three different DNNs that were trained and optimized to perform binary image classification in a known environment with predefined parameters. These three DNN models were chosen based on variations in their architectural design including layer count, number of parameters, and expected performance and accuracy. We chose binary image classification over multi-class image classification in this work for three main reasons. The first reason is that binary image classification requires less memory bandwidth and processing power to perform the inference making them suitable for resource-constrained embedded devices. Secondly, having only two classes considerably improves the inference speed, which allows the apparatus to process more Frames Per Second (FPS) and ultimately leads to a reduction in the overall power consumption. Finally, some applications require only two classes, including defect detection, certain surveillance applications, and obstacle avoidance.

1.1 Significance and Motivation

The significance of this thesis lies in its potential to give the reader empirical evidence that supports the use of one hardware accelerator over the other through rigorous testing and evaluation of the platforms. Motivated by the ever-growing availability of hardware accelerators and the increasing demand for efficient mobile robotic systems, where the ability to perform low-power real-time decision-making becomes more imperative, this thesis will address the following research questions:

1. What are the most relevant hardware accelerators for edge computing, with an emphasis on low power and low latency applications?
2. Can FPGAs outperform GPUs in terms of accuracy, inference speed, and performance per watt?

1.2 Related works

The literature shows numerous studies that have been carried out to compare different hardware accelerator platforms, including the work in [5], [9], and [10]. While these studies presented a comprehensive in-depth analysis of various hardware accelerators and how they perform with an array of DNN models, the platforms included in their papers often belonged to different categories and were mostly unsuitable for power-constrained edge devices. Our focus in this paper is to compare two closely related hardware accelerators in terms of cost, power consumption, and intended end application.

Bavikadi et al. [11] implemented a real-time object detection system using a Kria KV260 Vision AI starter kit. In their work, they focused on test benching the accelerator with YOLO V3 and V5, which are pre-trained models for which they used a pre-built Petalinux image provided by Xilinx without performing any Deep Learning Processor Unit (DPU) optimizations. In the scope of our research, we aim to build a flashable

Petalinux image with custom DPU configurations in order to achieve optimal performance for our proposed system. Likewise, Ernesto et al. [12] relied on a pre-built Petalinux image to implement their proposed prosthetic hand system with a KV260 utilizing YOLO V3. In our implementation, the use of object detection in an unknown environment was precluded as a consequence of two main factors. The first factor is that our main focus was on accelerating compute-intensive applications utilizing edge devices while reducing the overall power consumption of the system. The second factor stems from the nature of our intended use case, in which the model can be reduced to include only two classes.

1.3 Contribution

The ultimate goal of this thesis is to progress toward real-time inference in mobile robotics without compromising power consumption. This research, therefore explores viable low-power hardware accelerators and DNN models with the following core contributions:

- Tested the impact of quantization on the accuracy of three DNN models, by leveraging the Vitis AI framework.
- Gathered training, calibration, and validation datasets from the environment that will be used to conduct the experiments.
- Implemented an efficient real-time obstacle avoidance system.
- Compared the performance, power consumption, accuracy, and ease of development of two low-power hardware accelerators: a Jetson Nano and a KV260.

1.4 Structure

The remainder of this thesis is organized as follows:

- Chapter 2 introduces the reader to a comprehensive review of relevant literature. Our goal is to sift through the available hardware accelerators that are viable for edge computing.
- In chapter 3, we describe in detail the top-level design of our proposed system. This chapter additionally delineates the process of configuring the FPGA-based platform in order to prepare it for acDDN model deployment, as well as, the methods we used to optimize, quantize, and compile the DNN models.
- Chapter 4 provides a thorough elaboration of the system, DPU configuration parameters, and the inference process on both the GPU and FPGA-based platforms.
- Chapter 5 presents the results and an in-depth analysis of our key findings.
- Finally, chapter 6 concludes this work and outlines future work directions.

2 Background

Despite the gradual fading of the performance gains promised by Moore's law, improvements in semiconductors have achieved remarkable breakthroughs in digital technologies [13]. These advancements opened the door for a broad range of applications, including AI, autonomy, and mobile robotics. Battery technologies, in contrast, have not attained any substantial leaps in terms of power density and cost throughout the four preceding decades [14]. As a result, systems driven solely by rechargeable batteries are still required to be power efficient. Typically, the four major contributors to power consumption in a mobile robot system are actuators, sensors, processors, and the employed algorithms. This chapter delves into viable solutions to assist an existing robot system in the context of energy efficiency and computing power.

2.1 Hardware accelerators

Hardware accelerators are computing systems that are designed to improve the performance of certain types of computations. Ideally, these accelerators are optimized for specific applications and are capable of offloading computationally intensive operations and executing them concurrently. The most common hardware accelerators include GPUs, Application-Specific Integrated Circuit (ASIC)s, and FPGAs [15].

2.1.1 Graphical Processing Units

Originally built for rapid image rendering, over the past decade, GPUs have been widely used to accelerate computationally demanding applications by offloading certain tasks from CPUs. This heterogeneous computing system provides a number of advantages, including power efficiency, scalability, and flexibility. The architecture of a GPU revolves around a number of Streaming Multiprocessors (SM). These SMs consist of several components, including a Compute Unified Device Architecture (CUDA) core, registers, and shared memory, as shown in Figure 2.1 [16].

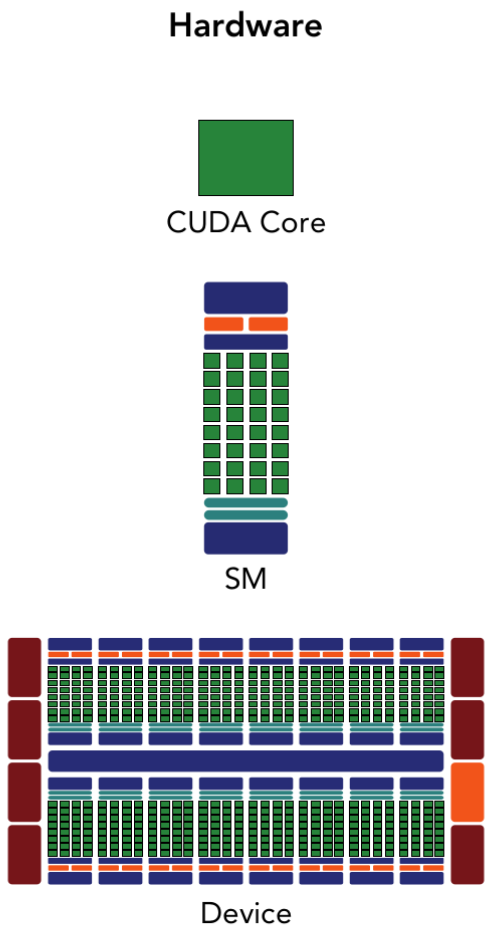

Figure 2.1: The CUDA programming components from a hardware prospective [16]

This figure is reproduced with permission from the book Professional CUDA C Programming *by Cheng et al., published by John Wiley & Sons*

The SMs facilitate the execution of hundreds of threads concurrently, which means a GPU that possesses many SMs is able to run thousands of threads simultaneously. Moreover, instructions within a thread are pipelined to exploit parallelism at the instruction and thread level. As a result, GPUs excel when it comes to computing a massive number of arithmetic operations in parallel, particularly when the same operation must be carried out repeatedly and rapidly. As a parallel computing platform and programming model, CUDA makes use of a Single Instruction Multiple Thread (SIMT) architecture to control threads in groups named warps. These threads execute the same instruction in parallel. On top of threads and warps, there are two other levels of thread grouping within the thread hierarchy in a GPU, a block of threads and a grid of thread blocks, which is the highest level of thread organization as shown in Figure 2.2 [16].

Unlike CPUs, GPUs lack the capability to perform complex branch prediction mechanisms. Consequently, threads in the same warp might not be executing the same instruction within each cycle in parallel as they should. As a result, the threads that host the wrong instructions will be disabled, causing a significant degradation in performance [16]. Similar to the thread hierarchy, CUDA also follows a memory hierarchy. Understating this hierarchy is crucial for achieving efficient data utilization and access. As Illustrated in Figure 2.3, the CUDA memory hierarchy includes the following levels [17]:

1. Registers: Registers are private to each thread and are mainly used to store local thread variables. While registers are considered the fastest memory level within this hierarchy, they are also limited in size. Therefore, utilizing them efficiently can help improve performance considerably.

2. Local Memory: Local memory is also private to each thread, but it is used to store stack allocations and local variables. It has a high access latency when compared to registers and some other memory levels in this hierarchy. Consequently, the use of this memory should be minimized.

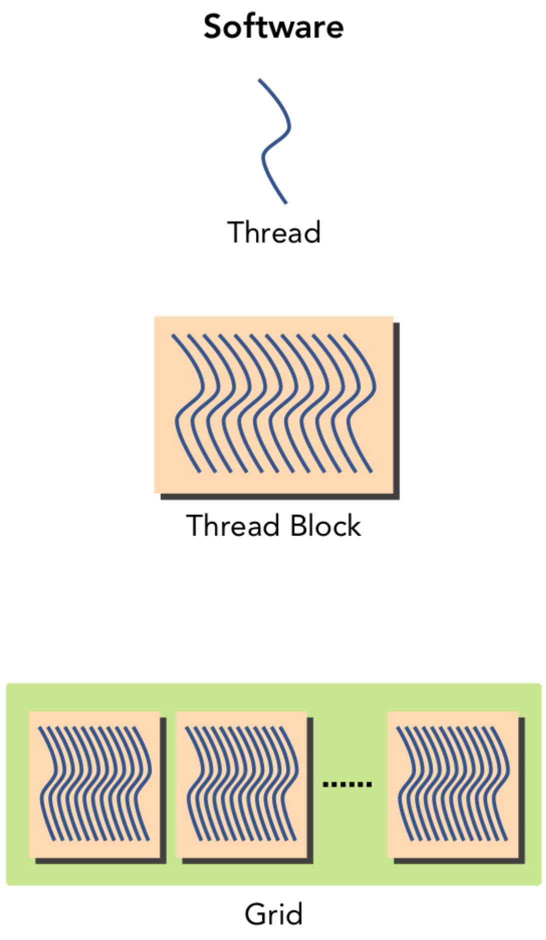

Figure 2.2: The CUDA programming components from a software prospective [16]

This figure is reproduced with permission from the book Professional CUDA C Programming *by Cheng et al., published by John Wiley & Sons*

3. Shared Memory: Shared memory is memory distributed between threads within a thread block. It provides high bandwidth and low latency, making it ideal for sharing data between threads within thread blocks. However, it is relatively diminutive in size than other memory levels aside from registers but can be explicitly managed by the software developer.

4. L1 and L2 Cache: These are two hardware-managed caches that have the task of caching the frequently accessed data from global memory automatically.

5. Constant Memory: This memory is used for storing data meant to be read-only.

6. Texture Memory: Texture memory is also a read-only memory optimized for out-of-order memory access patterns and image

7. Global Memory: Global memory is the most significant memory level in this hierarchy. It provides significantly lower bandwidth and higher latency than all other memory levels. This memory is used for storing data that can be accessed by all processes.

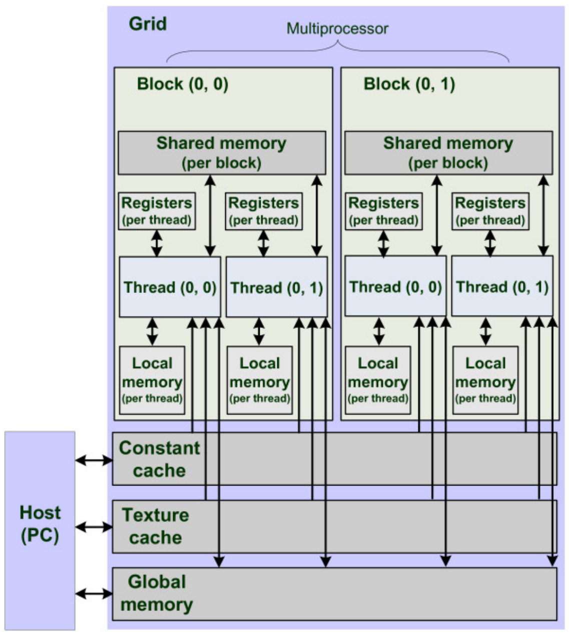

Figure 2.3: An illustration of a typical memory hierarchy in CUDA [18]

This memory hierarchy remarkably improves the memory bandwidth for applications that exploit spatial and temporal locality, which in turn improves the overall performance of the platform for such use cases. However, GPUs lose this leverage with applications that exhibit irregular memory access patterns, making them less efficient [16]. There are two main types of GPUs that are used to accelerate applications in embedded devices. The first type includes integrated GPUs, which are manufactured as part of the CPU. Therefore, they share the memory with the CPU, making them suitable for the size and power-constrained devices that are not required to implement compute-hungry tasks [19]. The second type of GPUs used as hardware accelerators are dedicated GPUs, which have their own processing units, memory, and peripherals and are widely used in high-performance embedded systems and edge devices. One popular computer system that utilizes a dedicated GPU, designed specifically for accelerating AI and machine learning applications, is the Jetson Nano from NVIDIA [20].

2.1.2 Customized hardware solutions

The burgeoning need for low latency, high throughput, and energy-efficient applications propelled by the ever-increasing computational requirements drove the development of hardware acceleration solutions tailored to the unique requirements of different applications. These customized hardware solutions can be divided into two main types in terms of reconfigurability. First are the ASICs, which are designed to serve one sole use case [21]. In other words, the circuit design is permanently wired to meet the requirements of a specific application, which not only improves the performance and power efficiency but also reduces the size, weight, and cost of the end product compared to other hardware solutions. However, due to the significantly high upfront cost and long development life cycle, ASICs are mostly suitable for mass production, where the design has been extensively tested and verified. The Tensor Processing Unit Tensor Processing Unit (TPU) is the most prominent example of an ASIC designed for AI training and inference. Figure 2.4 shows the latest architecture of a TPU chip [22], [23].

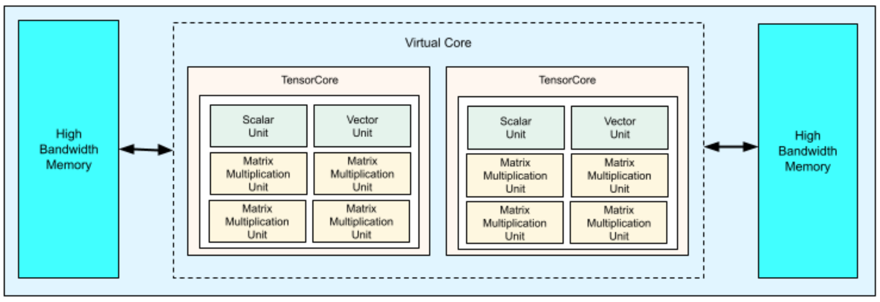
High Bandwidth Memory
Virtual Core
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
TensorCore
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
High Bandwidth Memory

Figure 2.4: The architecture of a TPU chip v4 [23]

The second type of customized hardware solution is the Field Programmable Devices (FPDs), which are integrated circuits that can be programmed or reconfigured by the end users an endless number of times through a Hardware Description Language (HDL) [21]. FPDs include Simple Programmable Logic Devices (SPLD)s, Complex Programmable-Logic Devices (CPLD)s, and FPGAs. FPGAs are widely used in complex designs due to their vast number of configurable logic gates as opposed to SPLDs and CPLDs. Essentially, the architecture of FPGAs is based on a two-dimensional array of Configurable Logic Blocks (CLB)s). Each CLB consists of an array of Look Up Tables (LUT)s, flip flops for storing the results of the LUTs, and a hardware technique that improves the performance of arithmetic operations such as a Full Adder (FA) along with other components [21], [24]. Figure 2.5 highlights some of the most important components of a CLB.

In Addition to CLBs, FPGAs also encompass other components, including communication cores, Digital Signal Processors (DSPs), and Block Random Access Memory (BRAM)s. BRAMs are spread out in columns throughout the FPGA fabric. While these memory units are limited in size, they can be stacked together to deliver bandwidth in the terabyte range [25]. When an FPGA is programmed, a configuration bitstream, which determines the amount of memory required for the application, the interconnections, and the overall behavior of the CLBs, is loaded into the device. These interconnections are established through a configurable routing fabric that encompasses a network of switches and wires. By changing the resistance of the switches through programming, signals can be routed between the desired inputs and output, CLBs, and other components [24].

2.1.3 Hardware accelerators from Xilinx

Xilinx provides a broad range of reconfigurable hardware accelerators for compute-intensive applications, such as image, data, and signal processing [12]. Notable among these de-

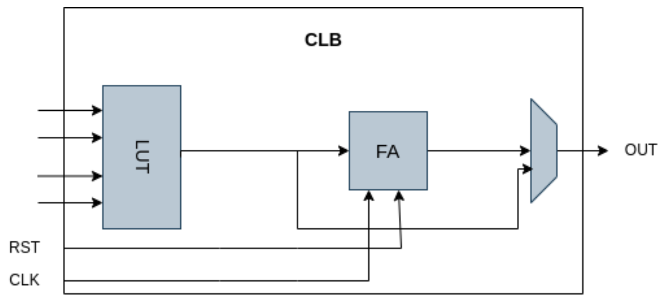

Figure 2.5: The components of a configurable logic block in an FPGA

vices are the Kria KV260 vision AI starter kit and the Kria KR260 robotics starter kit, both of which are built on the K26 System On Module (SOM). This K26 SOM is based on the Zynq UltraScale+ MPSoC architecture, which combines a total of seven processor systems, including an Arm Cortex-A53-based Application Processing Unit (APU) with a maximum frequency of 1333 MHz, an Arm Cortex-R5F-based Real-time Processing Unit (RPU), a Platform Management Unit (PMU) for power management and other subsystem control, and an Arm Mali GPU with a maximum frequency of 600 MHz among other processor systems on this SOM [12], [26]. Additionally, the K26 SOM integrates a Double Data Rate 4 (DDR4) memory, a Quad serial peripheral interface (QSPI) non-volatile memory that hosts the primary boot for the device, a Trusted Platform Module (TPM), and a power controller. The Kria KV260 also offers a number of peripherals that are isolated from the SOM, including a camera connector, one Ethernet interface, a USB 3.0 HUB, and a micro SD card interface that serves as the secondary boot for the device. Figure 2.6 provides a visual representation of the main components of the Kria KV260 [12]. Through the segregation of the SOM from these peripherals, developers can effortlessly customize the kit to make it suitable for different use case requirements.

The primary emphasis of the Kria KR260 robotics starter kit revolves around robotics and visual perceptions [27]. Therefore, the KR260 carrier board adds several General Purpose Input/Output (GPIO) interfaces that are crucial for most robotic applications. These GPIOs include four Peripheral modules (Pmod) interfaces that can be useful for

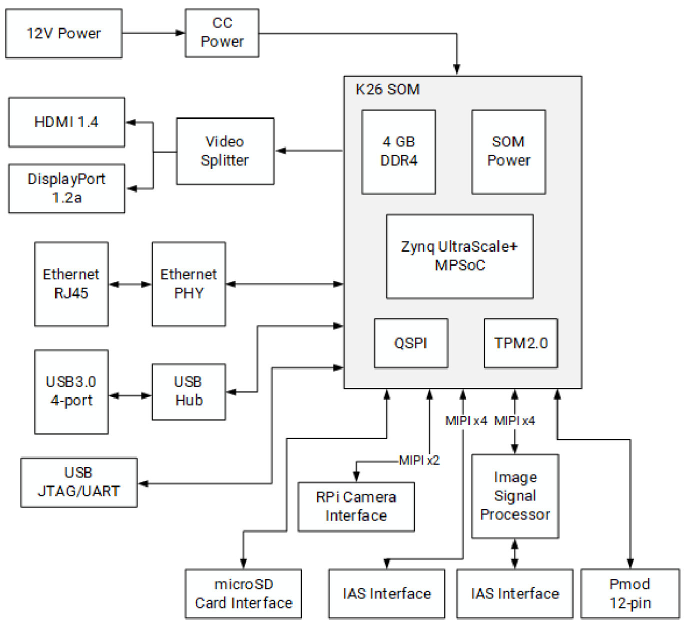

Figure 2.6: A block diagram of the Kria KV260 vision AI starter kit [12].

"© Copyright 2019 - 2022 Xilinx Inc."

connecting up to 12-pin modules, such as servo motors, sensors, and LEDs. In addition to the four Pmods, the KR260 carrier board also offers four Ethernet connectors, a Raspberry PI HAT Header, an SLVS-EC Gen2 X2 lane interface for high-speed, high-resolution CMOS image sensors, and a Small Form Factor Pluggable (SFP) intended specifically for remote image processing in robotics applications. Figure 2.7 presents the block diagram of the K26 SOM along with the carrier board design for the KR260 [27], [28].

2.2 Unified software platforms for FPGAs

Historically, the process of manipulating the hardware design of an FPGA often required expertise in low-level hardware details and HDL [29]. However, the development of unified software platforms allowed software developers to make use of software programming languages for FPGA programming, alleviating the need to focus on hardware intricacies. For example, Xilinx developed the Vitis unified software platforms, which incorporate the components listed below [30]–[32]:

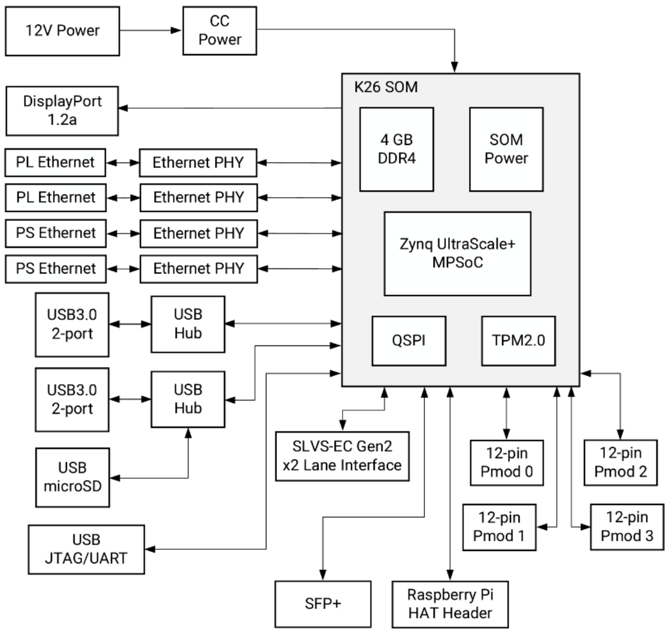

Figure 2.7: A block diagram of the Kria KR260 robotics starter kit [27].

"© Copyright 2019 - 2022 Xilinx Inc."

1. Vitis Accelerated Libraries: These are open-source libraries that can be used without having to apply any modification to the initial code. these libraries include Solver, DSP, Sparse, and Basic Linear Algebra Subroutines (BLAS).

2. Vitis Core Development Kit: The Vitis Core Development Kit includes the tools required for compiling, analyzing, and debugging applications written in C/C++ or OpenCL.

3. Xilinx Runtime library: The XRT allows seamless communication between an application running on a host device and the accelerator.

4. Vitis Target Platforms: With these platforms, the foundational software and hardware architecture is established. This involves components such as custom input and output interfaces along with external memory interconnections.

5. Vitis Model Composer: The Vitis Model Composer provides a set of tools for MATLAB and Simulink integration. These tools facilitate rapid design and verification

by automatic code and test bench generation, expediting the production process on AMD-based platforms.

6. Vitis High-Level Synthesis (HLS): With Vitis HLS, intricate FPGA algorithms can be generated by synthesizing C or C++ code. Translating such high-level code to low-level hardware description code not only enables developers from diverse backgrounds to develop their applications in FPGA platforms effortlessly but also reduces the design validation time.

7. Vitis AI: This platform serves as an extensive solution for AI inference development. As shown in Figure 2.8 [33], it includes efficient DPUs specifically designed and optimized for accelerating deep learning and neural network computations. Vitis AI also provides a complete development kit for fine-tuning, compiling, and profiling the model. This development kit consists of the Vitis AI Runtime (VART), an AI optimizer for pruning, an AI quantizer for converting floating-point models into a fixed digital representation which is more efficient as they require less memory and computing power as well as an AI compiler that is responsible for translating the AI model into an efficient instruction set. Moreover, the Vitis AI platform presents a wide range of AI models, such as the AI Model Zoo, which offers ready-to-use deep learning models, including Open Neural Network Exchange (ONNX), TensorFlow, and PyTorch [31].

2.3 Deep Neural Network models (DNN)

DNNs are artificial neural networks created by stacking together multiple layers of interconnected neurons [34]. These layers are arranged into an input layer, numerous hidden layers, and an output layer. They are responsible for processing the raw input data and passing it to subsequence layers in order to extract relevant attributes [35]. Ideally, each neuron carries out mathematical computations on the data it receives. These mathematical

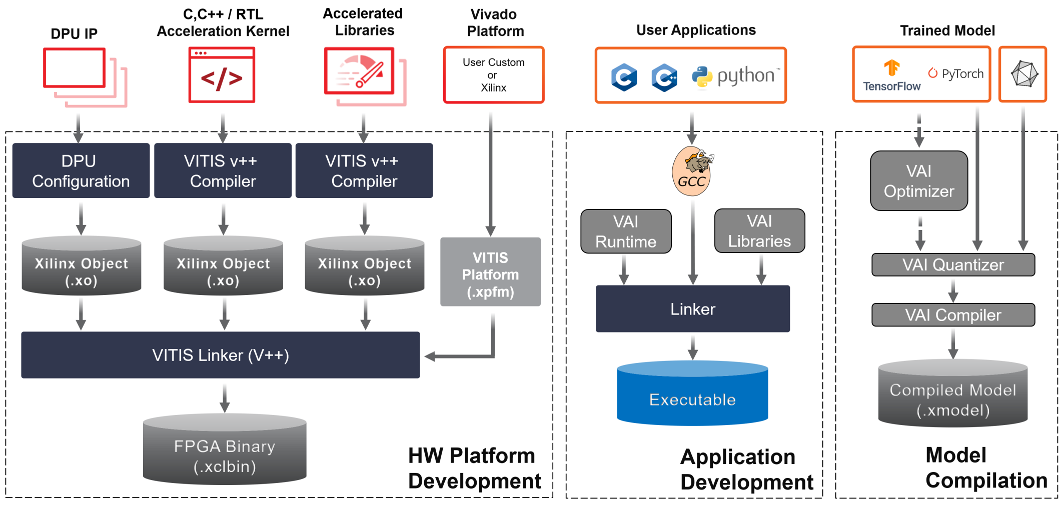

Figure 2.8: An illustration of the Vitis AI integrated development environment [33].

"© Copyright 2022-2023, Advanced Micro Devices, Inc."

operations may include a combination function, as shown in (2.1), which is an outcome of the cumulative result from a dot multiplication of the input and weights as vectors with the addition of some bias to offset the results towards a negative or positive value [34].

$$Combination = \sum_{i=1}^{n}((Inputs_i \times Weights_i) + Bias) \tag{2.1}$$

Following the completion of the combination function computations, an activation function is used on the results in order to introduce non-linearity to the neural network. This activation function will enable the network to learn and detect patterns within the data. Among the most prevalent functions are the SoftMax, Sigmoid, Rectified Linear Units (ReLU), and the Tanh activation functions [36]. These functions mainly differ in their computational complexities, output ranges, and memory requirements, which in turn affects the learning process of the network [34]. For example, the Sigmoid function performs exponentiation, which is known to be computationally intensive as opposed to the ReLU activation function, which only involves element-wise operations as depicted in (2.2) (2.3) respectively [34]–[36].

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$ReLU(x) = \max(0, x) \tag{2.3}$$

The results of the combination and activation functions are propagated through all hidden layers in a sequential flow until the output layer is reached. The final perfections are then produced, after which a backpropagation algorithm takes place in order to minimize any possible prediction errors. Basically, the backpropagation algorithm calculates the gradient loss function and compares it to all parameters in the network in order to adjust the weights and biases accordingly [36] as shown in Figure 2.9 [37].

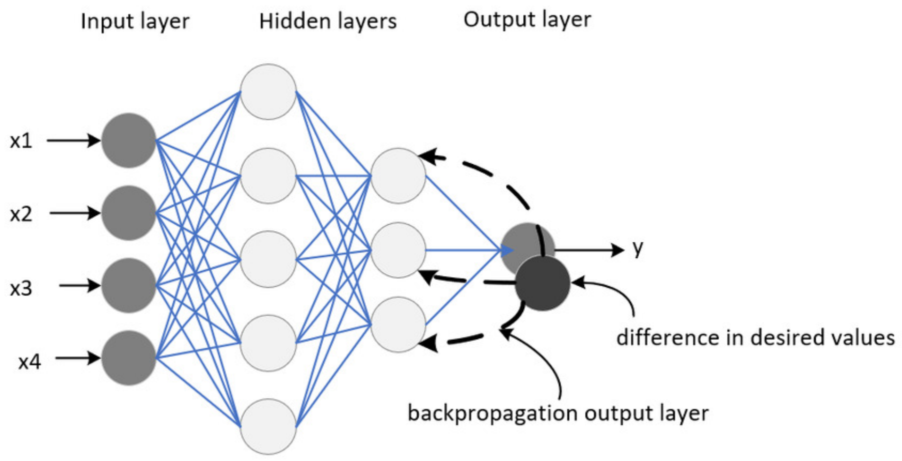

Figure 2.9: An illustration of a basic DNN architecture with backpropagation [37].

Over the past two decades, numerous DNN models have been developed, each of which was made specifically to address particular challenges, such as natural language processing, time-series predictions, and image recognition. The most common DNN models include GoogLeNet, AlexNet, and ResNet. These DNN models mainly differ in the number of layers used, their architecture, as well as some other design principles. For instance, the AlexNet consists of eight layers and utilizes ReLU as the activation function. ResNet on the other hand, allows the use of deeper models such the ResNet-18, ResNet-

50, and ResNet-101 without encountering the problem of vanishing gradients [38].

The DPUs offered by Xilinx have the capability to accelerate a wide range of DNN models due to their high degree of configurability [39]. Typically, these DPUs can be configured in the Vivado development environment to deliver optimal performance for a given DNN model. Xilinx provides different prebuilt DPU designs for each of their multiprocessing platforms. For example, the Zynq UltraScale+ MPSoC is equipped with a DPUCZDX8G, which is tailored to match the resources available on this platform [40]. other DPU architectures include DPUCV2DX8G for VEK280, V70, and Vx2802, DPUC-V2DX8G for VE2302, and DPUCVDX8G for VCK190 [39]. Table 2.1 details the naming convention of these prebuilt designs [40]. For instance, the Kria KV260 and KR260 development kits both possess a DPUCZDX8G, where C represents the application that will run on this accelerator. ZD refers to the hardware platform this DPU is designed for. The number 8 means it is configured for INT8 data types, implying that the trained model is required to undergo a quantizing process to convert from higher precision floating points. And the G indicates that this design is for general purpose computations [40].

Table 2.1: DPU Prebuilt Design Naming [40]

Device	**App.**	**HW Platform**	**Quantization Method**	**Quantization Bitwidth**	**Design Target**
DPU	C: CNN	AD-ALveo DDR	X-DECENT	4: 4-bit	G-General Purpose
DPU	R: RNN	AH-Alveo HBR	I-Integer	8: 8-Bit	
DPU		VD-Versal DDR	F-Float threshold	16: 16-bit	
DPU		VD-Versal DRR DDR with AIE-ML		M-Mixed	
DPU		ZD-Zynq DDR			
DPU	C	ZD	X	8	G

Xilinx also offers predesigned configurations for each DPU design. This makes it easier for developers who are not familiar with FPGA programming and modeling to use the configuration that best fits their application. Table 2.2 shows a list of the possible configurations available for specific DPU designs. The name of the configuration relates to the parallelism applied to different designs. For example, the B4096 configuration indicates that 8 Pixel Parallelism is used with 16 Input Channel Parallelism (ICP) and 16 Output Channel Parallelism (OCP) where the peak number of operations per cycle (OPS) is given by (2.4) and (2.5) [41].

$$Peak\ OPS = PP \times ICP \times OCP \times 2 \tag{2.4}$$

For example:

$$Peak\ OPS\ \text{(B4096)} = 8 \times 16 \times 16 \times 2 = 4096 \tag{2.5}$$

Table 2.2: DPUCZDX8G Possible Configurations

Config.	Freq. (MHz)	Peak Perf. (GOPS)	LUT	BRAM
B1152x1	370	426	28698	117.5
B2304x1	370	852	34379	161.5
B4096x1	350	1400	92630	249.5
B4096x2	330	2700	92630	249.5
B4096x3	333	4100	92630	249.5

2.4 Edge Computing

The evolving demands for compute-intensive applications coupled with real-time decision-making and low-latency requirements in resource-constrained embedded devices elucidates the main reason behind the development of edge computing [42]. Unlike the traditional centralized cloud computing paradigm, edge computing is based on a decentralized computing framework. This distributed computing approach allows the processing of data closer to the source rather than sending it to a centralized data center [43]. In essence, this decentralized paradigm provides the following benefits:

- **Low latency:** Edge computing reduces the time it takes to process data, which is required in real-time applications such as collision avoidance, wearable health monitoring devices, and surveillance applications [43].
- **Reduced bandwidth:** Local processing of the data eliminates the need for large data transfer to remote servers, which reduces the bandwidth cost significantly [42].
- **Improve Security and Privacy:** Without the need to transfer sensitive information to remote servers, which enhances the overall security and privacy [43].
- **Offline Operation:** Edge computing allows applications to function autonomously even for devices deployed in places with limited or no connectivity [44].
- **Hybrid Architecture:** Edge computing offers the capability to incorporate cloud computing when needed. This flexibility enables edge computing frameworks to leverage cloud-based resources while maintaining a low response time, as shown in Figure 2.10 [44].

2.5 The Robotic Operating System (ROS)

ROS is an open-source middleware framework that was designed to ease the development and implementation of robotic systems. It was initially developed by Willow Garage, yet presently managed by the Open Source Robotics Foundation (OSRF) [45]. This framework provides all the libraries and device drivers required by any system, as well as a

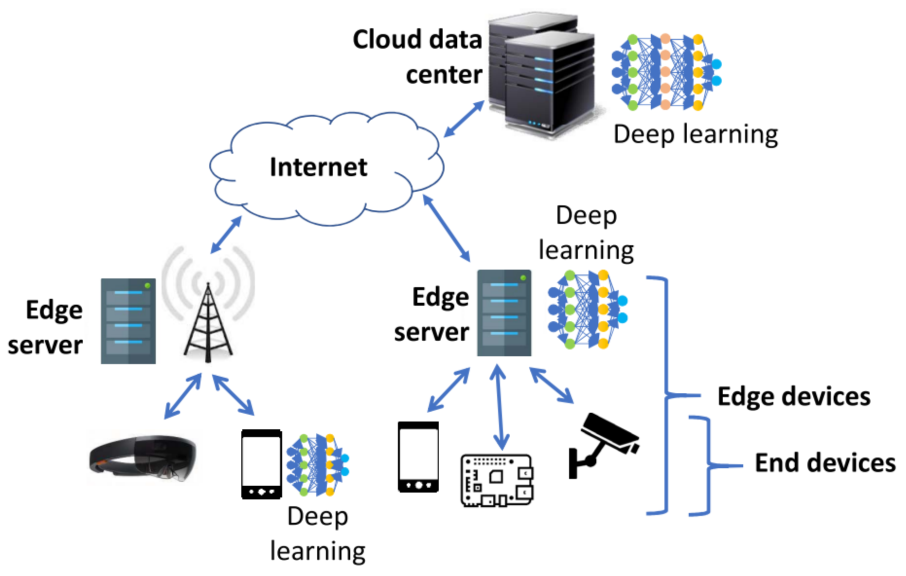

Figure 2.10: A hybrid edge-cloud computing architecture [44].

comprehensive set of tools that enhance its usability, such as a package manager, visual simulations, and documentation. At its core, ROS is based on a distributed network model comprising numerous individual nodes. Each node carries out one or multiple tasks assigned to it, such as reading and processing sensor data, controlling actuators, or extending the functionality of other nodes [46].

ROS nodes are designed to communicate with each other through three communication mechanisms, which are topics, services, and actions. Topics are named channels that facilitate the communication between nodes through a publisher and subscriber model. Publishers and subscribers are application and platform-agnostic. In other words, nodes can send and receive information from other nodes without any limitations or constraints. This decoupled asynchronous communication framework makes topics suitable for real-time data sharing and cross-platform communication. Services, on the other hand, employ synchronous communication mechanisms, which is useful when a node is required to perform an action, such as controlling an actuator, and must receive immediate feedback. Similar to services, Actions are mostly used to control hardware components and require feedback; however, the feedback in actions may be delayed, making them suitable for goal-oriented tasks [46].

Although ROS was initially designed to be platform independent, it runs best on specific operating systems such as Ubuntu 20.04 running on an AMD64 or Intel 64 processor. Nevertheless, over the past five years, a considerable amount of effort and dedication has been invested in integrating hardware accelerators with ROS. Of particular importance is the substantial investment by GPU and FPGA-based platforms. NVIDIA for example, made it possible to install ROS on the Jetson Nano directly [47]. Conversely, running ROS on FPGAs while preserving reconfigurability might not be as straightforward. Therefore, Xilinx developed the Kria Robotics Stack (KRS) in order to facilitate the integration of ROS with their Kria SOM platforms. KRS encompasses the libraries and tools needed to extend the ROS for industrial applications while leveraging the benefits of FPGAs [48].

3 Design overview

This chapter aims to describe the top-level design of the two systems that will be used to assess and compare the performance of the hardware accelerators utilized for obstacle avoidance based on an RGB camera. The first system makes use of the Jetson Nano development kit from NVIDIA, which is based on a GPU for acceleration, while an FPGA serves as the central component for the second system. Additionally, this chapter will provide an overview of the main algorithms employed.

3.1 Integrating an accelerator into the TurtleBot 4 Lite

The chosen platform for the experiments conducted in this work is the TurtleBot 4 Lite. Officially shipped with a Raspberry Pi, the TurtleBot 4 Lite is an ideal platform for carrying out robotics research and development [49]. It is particularly suitable for autonomous task experiments, including perception, localization, and navigation. However, the Raspberry Pi 4 microcontroller is only equipped with a multicore processor that is limited in its ability to perform parallel computing tasks. Therefore, adding an accelerator should improve the performance of the system considerably. Figure 3.1 illustrates the top-level design of the system. Fundamentally, the Raspberry Pi is responsible for controlling the TurtleBot 4 Lite and the hardware accelerator offloads the computationally intensive tasks from the main CPU. Although the Raspberry Pi can be directly connected to the hardware accelerator through the interfaces available on both platforms, a wireless communication method will be used in this experiment.

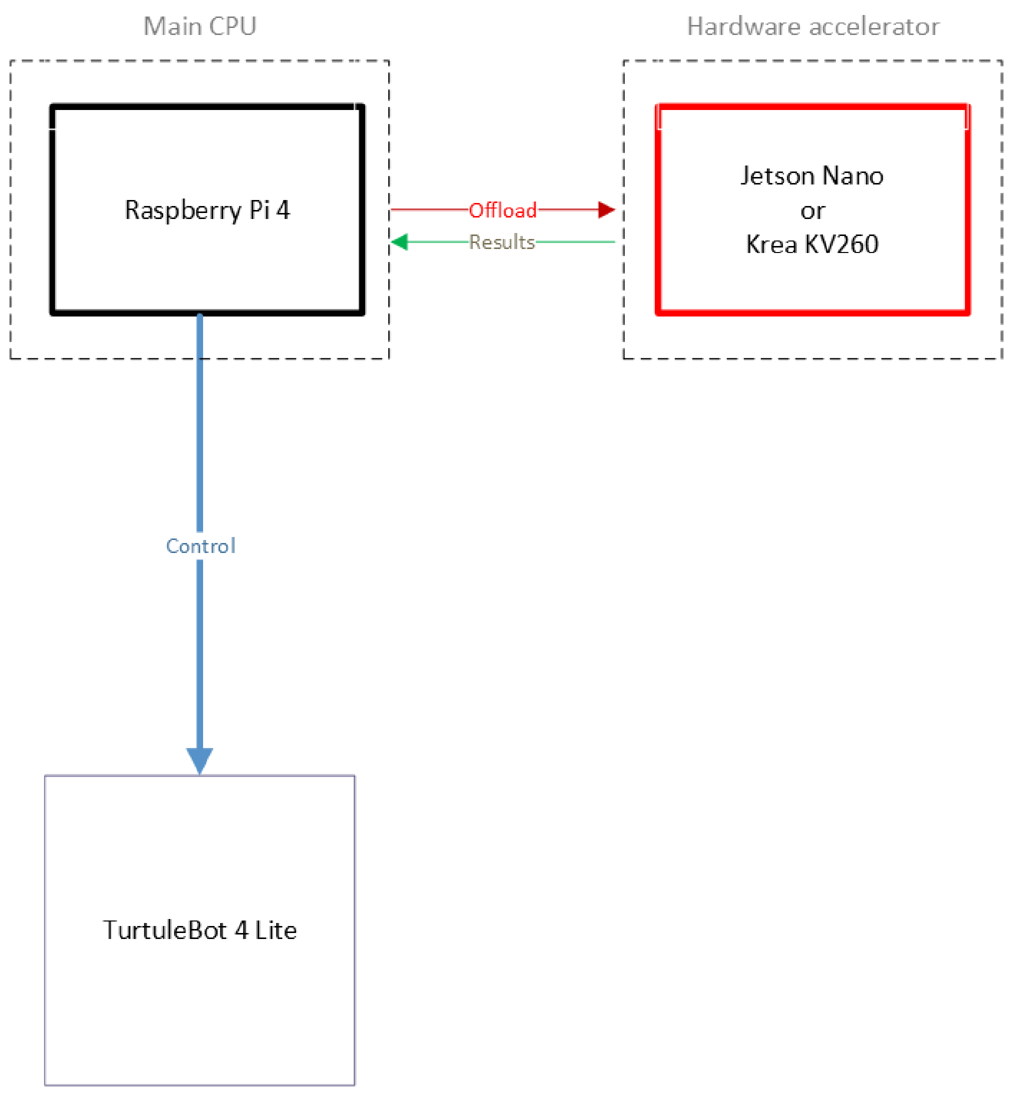

Figure 3.1: The top-level design of the system.

The Raspberry Pi in the TurtleBot is driven by ROS while the Jetson Nano and Kria KV26 typically operate on a Linux-based operating system. Therefore, an MQTT protocol will be used as a communication bridge between ROS and the operating system running on the accelerator. The main task of the system is to avoid obstacles while navigating through a predefined path within a known environment. With these fixed parameters, it should be enough to perform the ultimate goal of the system with a solitary sensor, specifically an RGB camera. This camera can connect directly to the accelerator, as the hardware accelerator handles all image processing tasks. However, having the accelerator as an edge device that is connected to the grid would improve the battery charging cycle of the TurtleBot 4 Lite significantly. It is worth noting here that the Jetson Nano can function in two distinct power modes as described in table 3.1. The first is a five-watt power mode facilitated by the micro-USB interface and the second is through a barrel jack which operates on a ten-watt power mode [50]. While the latter enables the device to operate at peak performance, the former is likely to cause a noticeable reduction in the computational power [50]. The Kria KV260 on the other hand requires a 12V power supply that is capable of providing three amps [51].

Table 3.1: Jetson Nano Power Requirements and Modes [50]

Interface	**Voltage (V)**	**Current (A)**	**Power (W)**		
			Min	**Max**	**Typical**
Micro-USB	5 / 3.3	2.5	5	12.5	10
Barrel Jack	12	4	10	20	20
GPIO Header	5	6	10	30	20

The TurtleBot 4 Lite is powered by a 26 Wh Lithium Ion battery with a nominal 14.4V VBAT. With its capacity to provide ample power, this battery can keep the Raspberry Pi 4 running for 2-4 hours, depending on the workload. The Raspberry Pi 4 outputs 5V and 3.3V through its GPIO pins. Additionally, the TurtleBot also supplies a 12V-3A interface from an additional Printed Circuit Board Assembly (PCBA) as illustrated in Figure 3.2 [52]. Another advantage of not connecting the camera to the accelerator is that the camera is already connected and configured to work on the Raspberry Pi, which makes it easier from a hardware perspective to keep the camera in place and publish the captured images to an image topic through the MQTT broker. The accelerator will then

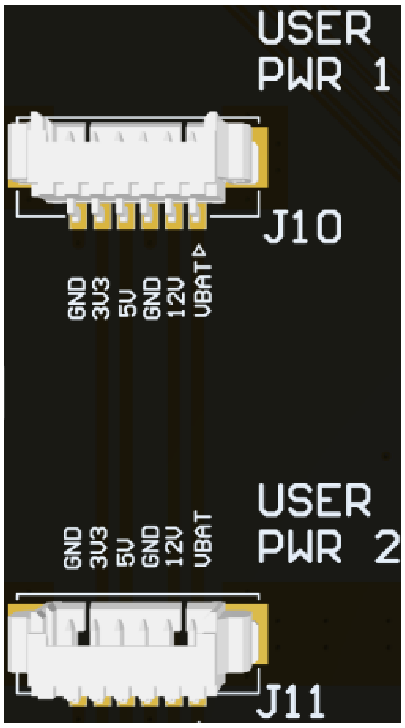

Figure 3.2: Power interfaces provided by the TurtleBot 4 Lite [52].

subscribe to this topic and publish the results back to the Raspberry Pi.

In the configuration shown in Figure 3.3 the Raspberry Pi serves as the backbone of the system, housing all the vital components, including the MQTT broker. This alleviates the workload on the accelerator, allowing it to exclusively employ its resources for the most demanding computational tasks. Alternatively, due to the general-purpose embedded processing capabilities of both target platforms used in this work, in particular, their ability to run ROS, these development kits hold the potential to acquire full control of the entire system, eliminating the need for the Raspberry Pi. for the Jetson Nano, the implementation is straight forward since ROS can be installed with a simple wrapper. However, The Kria KV260 would require more work to be conducted before we can install ROS on it while keeping the ability to accelerate the compute-intensive operations. As mentioned in Chapter 2 for the Kira SOM K26-based platforms KRS must be installed in order to be able to utilize the acceleration features while running ROS. With this setup, the dual-core Arm Cortex CPU, which is embedded into the Kria SOM, takes care of the basic ROS

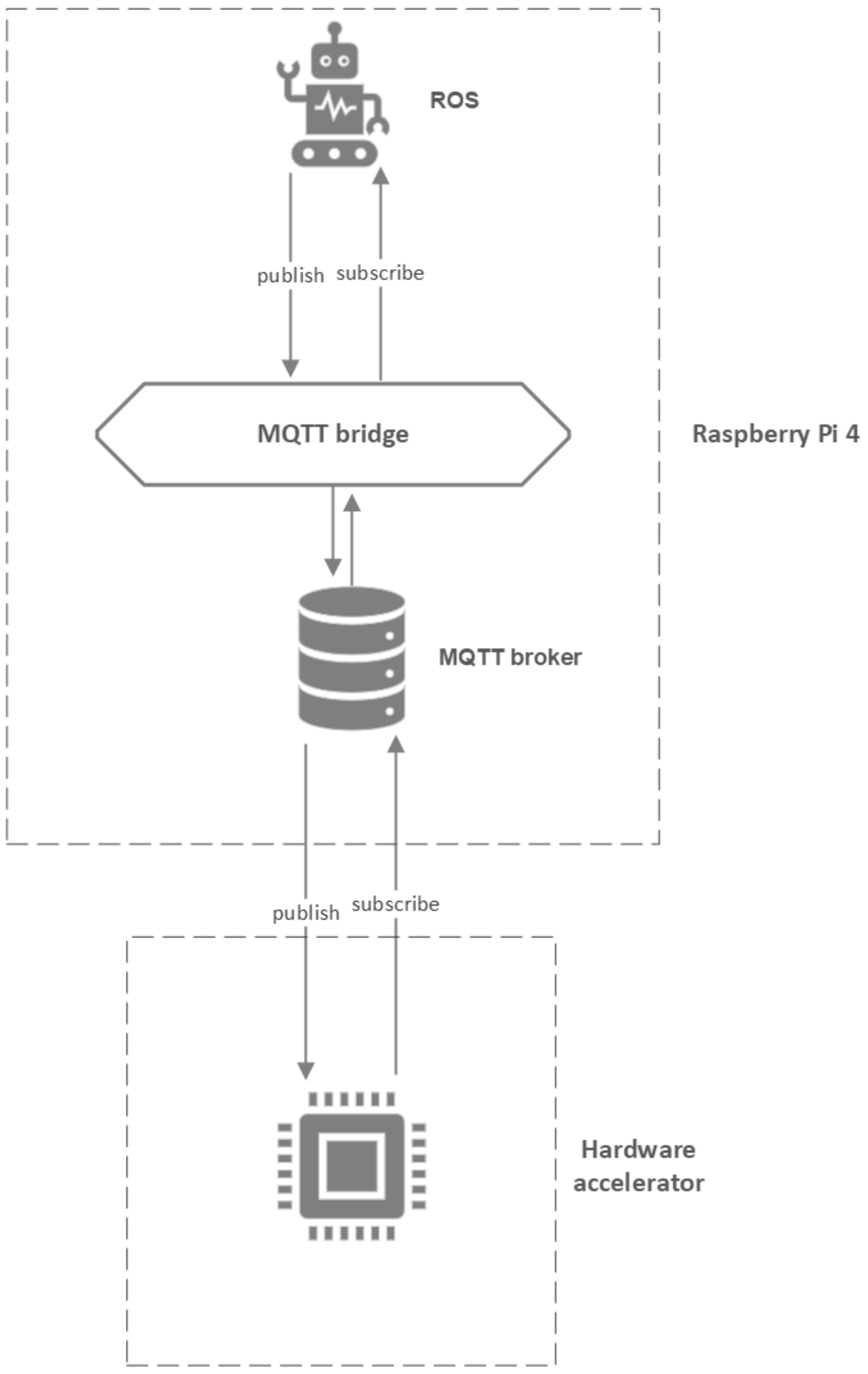

Figure 3.3: The communication method between the Raspberry Pi 4 and the hardware accelerator.

operations while the DPU offloads the applications that require acceleration as depicted in Figure 3.4. Despite the clear advantages, this approach will not be used in this exper-

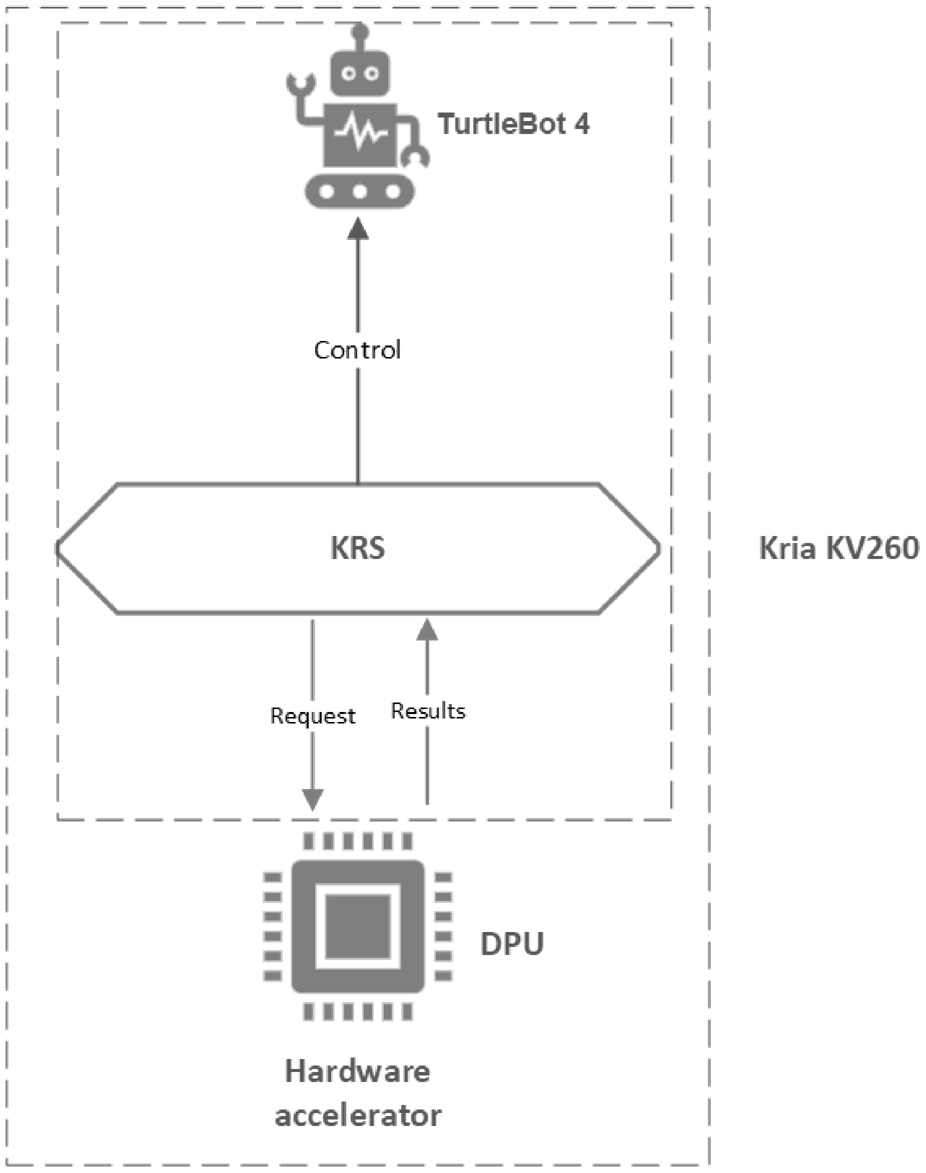

Figure 3.4: The system is fully controlled by the Kria KV260.

iment for three main reasons. First, this work was intended to accelerate the computing of expensive operations on existing systems, that is, to augment these systems not replace them. Second, the KRS is still in its beta release, making it prone to bugs that have not yet been addressed. Moreover, no clear documentation was found for configuring the DPU to run our chosen DNN models after the stack installation. Lastly, powering the Kria development kit from the battery of the TurtleBot 4 directly would theoretically deplete the battery faster which in turn makes it difficult to run the experiments extensively.

Both accelerators used in this paper are developed and optimized for implementing DNN models. However, the FPGA-based platforms have the additional leverage of being able to make use of custom data types due to their high level of reconfigurability. These custom data types are typically of lower precision floating point values or integers. By converting the trained neural network model to this new data type, FPGAs will require less memory bandwidth and computation power which results in a faster inference rate, making FPGAs more efficient for real-time applications. To this end, when using the Jetson Nano, it is necessary to utilize the trained model that has 32 floating point precision values. On the other hand, FPGAs are required to execute three extra steps, which are provided by the Vitis AI platform for this implementation, once the final trained model is generated. The first step involves reducing the number of bits per parameter through a pruning and fine-tuning process to the trained model as demonstrated in Figure 3.5 [53]. Upon the successful completion of the optimization process, the trained model undergoes a compression procedure in which some of the neuron nodes are eliminated in order to prepare the model for the quantization process. This optimization pre-processing of the neural network ensures that the model is streamlined and made more efficient [54].

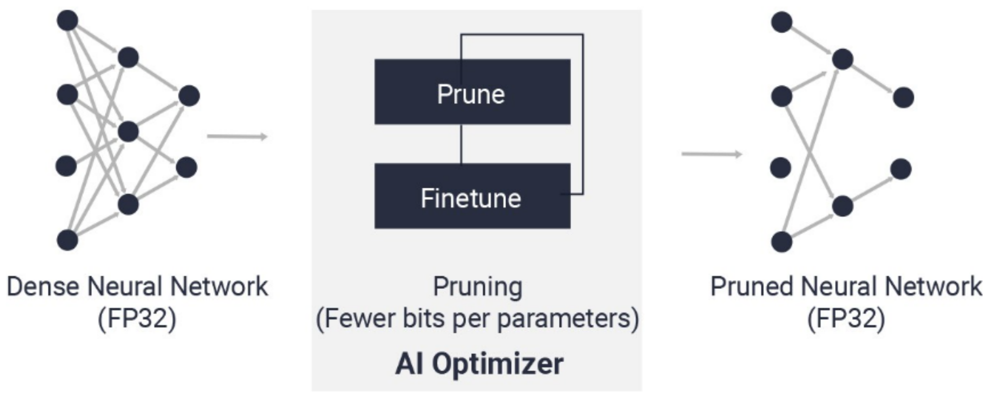

Figure 3.5: The optimization process provided by the Vitis AI platform [53].

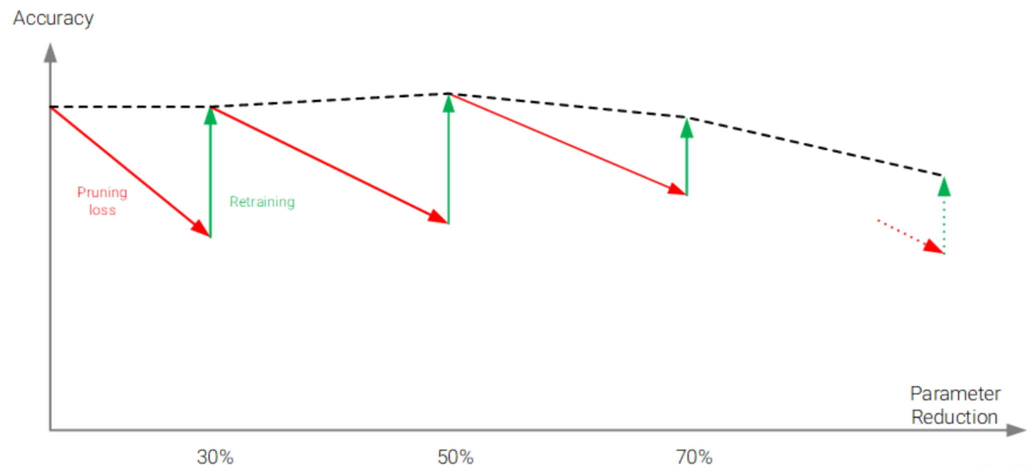

Figure 3.6: Improvements to the accuracy of the compressed model through iterative pruning [53].

The quantization process and result depend solely on the architecture of the DPU, which can be configured to reduce the floating point of the DNN-trained model to a lower precision floating point, INT8, or even other custom data types. For this work, we will configure the DPU to rely on INT8 values. It is worth noting that this modification may lead to a slight reduction in the accuracy of the predictions made by the FPGA. However, it is anticipated that this reduced precision will significantly improve the inference rate, which is important for real-time image processing. Nevertheless, the accuracy loss is not significant due to the iterative execution of the pruning and finetuning of the model as shown in Figure 3.6 [53].

Essentially, the quantization process involves scaling down the weights given to the neural network, through all layers, to a specific number of bits. The outputs of the activation functions are then quantized to the appropriate bit-width accordingly as illustrated in Figure 3.7. At this point, the model has been reduced to the required digital representation but must be further calibrated and fine-tuned, before compiling the model, in order to improve the accuracy of the quantization results. This calibration is done by running the inference several times with a sufficient unlabeled dataset. Algorithm 1 outlines the main

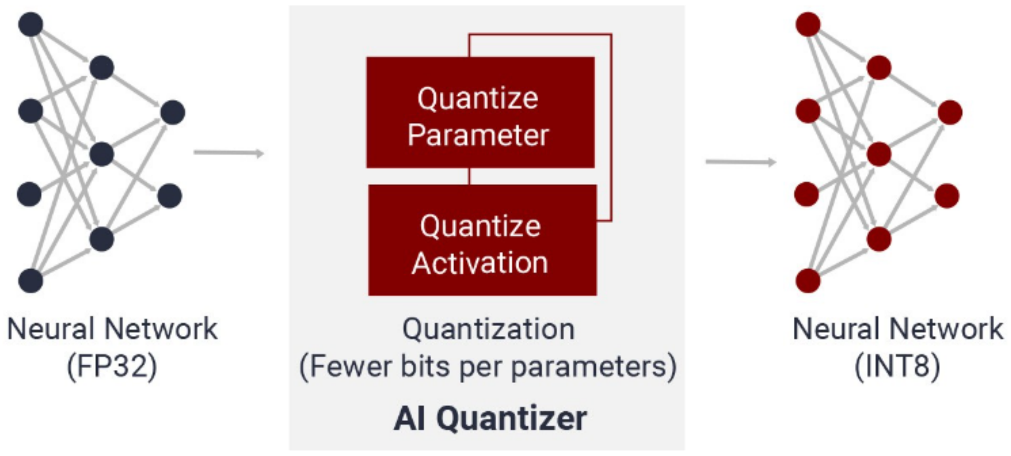

Figure 3.7: The quantization process provided by the Vitis AI platform [53].

"© Copyright 2022-2023, Advanced Micro Devices, Inc."

steps of the quantization process [54].

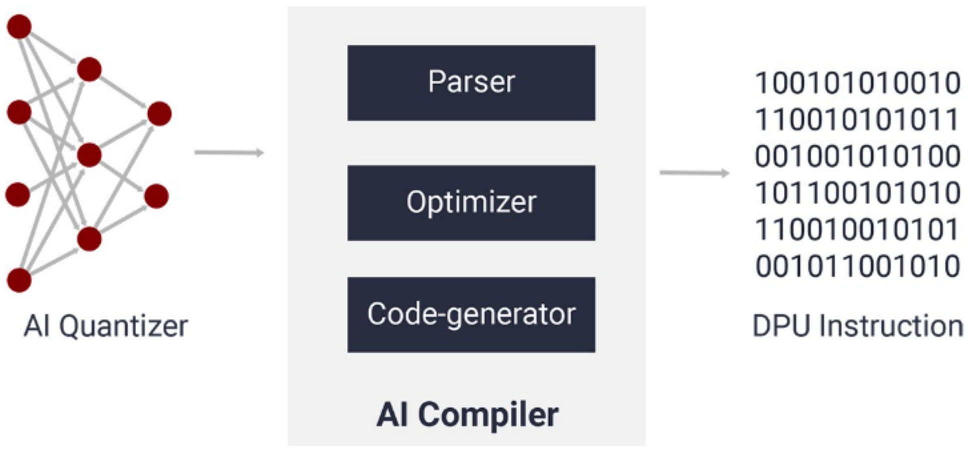

Figure 3.8: The compilation process provided by the Vitis AI platform [53].

"© Copyright 2022-2023, Advanced Micro Devices, Inc."

Before deploying the model, the last step is to compile it using the Vitis AI compiler. This compiler is responsible for parsing, optimizing the calibrated results, and generating the final model that will run on the FPGA as shown in Figure 3.8 [53]. Figure 3.9 provides

Algorithm 1 Quantization Process

```
Require: Trained DNN model, dataset, loader, device
  if $a$ GPU is available on the host device then
      $device \leftarrow CUDA$
  else
      $device \leftarrow CPU$
  end if
  Load model weights from float_model
  $quantizer \leftarrow torch\_quantizer$
  $error\_count \leftarrow 0$
  for images, labels in loader do
      Move images and labels to device                    ▷ GPU or CPU
      $outputs \leftarrow model(images)$
      $error\_count \leftarrow error\_count + sum(|labels - max_(outputs)|)$
  end for
  if quant_mode == 'calib' then        ▷ Calibrate based on the set of unlabeled images
      quantizer.export_quant_config()
  else if quant_mode == 'test' then              ▷ produce the quantized model
      dump the $xmodel$
  end if
                                 ▷ Calculate the accuracy of the quantized model
  $accuracy \leftarrow 1.0 - error\_count/len(dataset)$
```

a summary of the Vitis AI quantization workflow.

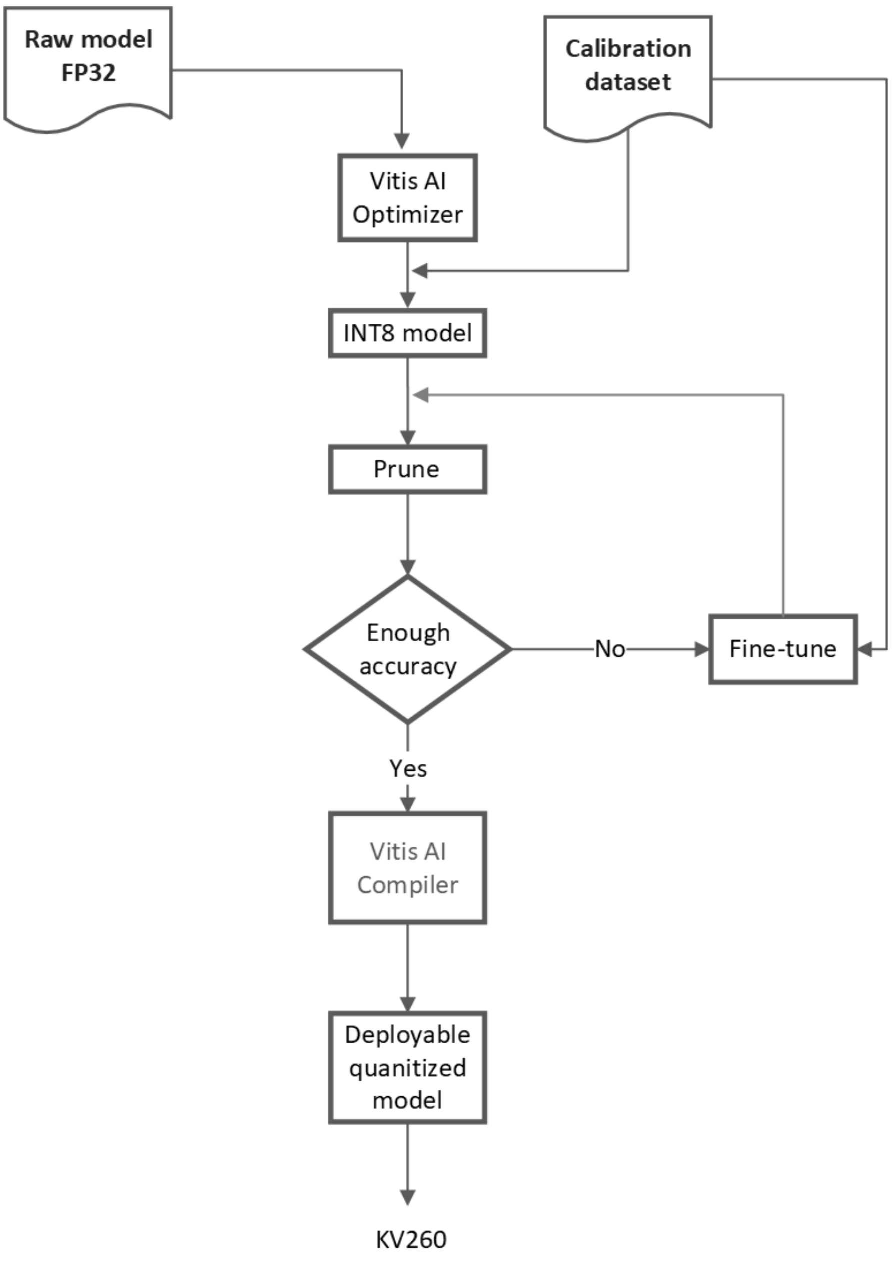

Figure 3.9: The quantization work flow [53].

3.2 Preparing the targeted accelerator

So far we have discussed the necessary steps required to prepare the model in order to run it efficiently on an FPGA-based platform. Ideally, these steps should all have taken place on a host machine, preferably running Ubuntu 20.04 or Ubuntu 22.04. Unlike this FPGA system, the GPU-based platform, which will be utilized in this work, can only run the trained model directly without any additional modifications to the model. Moreover, this GPU-based platform is optimized to run DNN models, by default, without the possibility to apply customizations to the hardware. In other words, the only way to enhance the performance is by experimenting with different DNN models, such as AlexNet, GoogLeNet, and ResNet. To this end, the focus in this section will be on configuring the FPGA for efficient execution of the DNN model.

The FPGA-based platform that we will use in this experiment, is the Kria KV260. Off-the-shelf this edge computing device is only shipped with a primary boot flashed to the QSPI device, as mentioned in section 2.1.3. This primary boot device is exclusively responsible for handling low-level control units, including the first stage boot loader firmware [55]. The secondary boot device for this KV260 is an external micro-SD card that should be flashed with an operating system suitable for the given application. These operating systems are either general-purpose operating systems such as Ubuntu and Petalinux, a flashable prebuilt image, or a custom image tailored for a particular application. Ultimately, the image that will be flashed to the micro-SD card will determine the architecture of the FPGA and DPU. Moreover, this image also defines how the FPGA connects to the DPU, the resources required, and the type of DNN to be used with the DPU.

In order to create a custom Petalinux image capable of running the inference on the target device efficiently, several parameters must be defined. These parameters include information about the platform used, the architecture of the DPU, the frequency, and the amount of memory assigned to each DPU. Initially, we will start by designing the platform in the Xilinx Vivado suit. The three main files required for this platform design are the board design files, the DPU model, and Tool Command Language (TCL) files that are used for task automation. The board files consist of all the specifications and interconnections of the target. These board files are added to the Vivado environment during the installation process of the suit, but can also be added from the Vivado store. The Vivado environment will allow us to create the block design from these files as illustrated in Figure 3.10. This block design represents the HDL code of each block connected together based on the final architecture of the DPU, FPGA peripherals, and other configurations that will allow the system to accelerate the DNN.

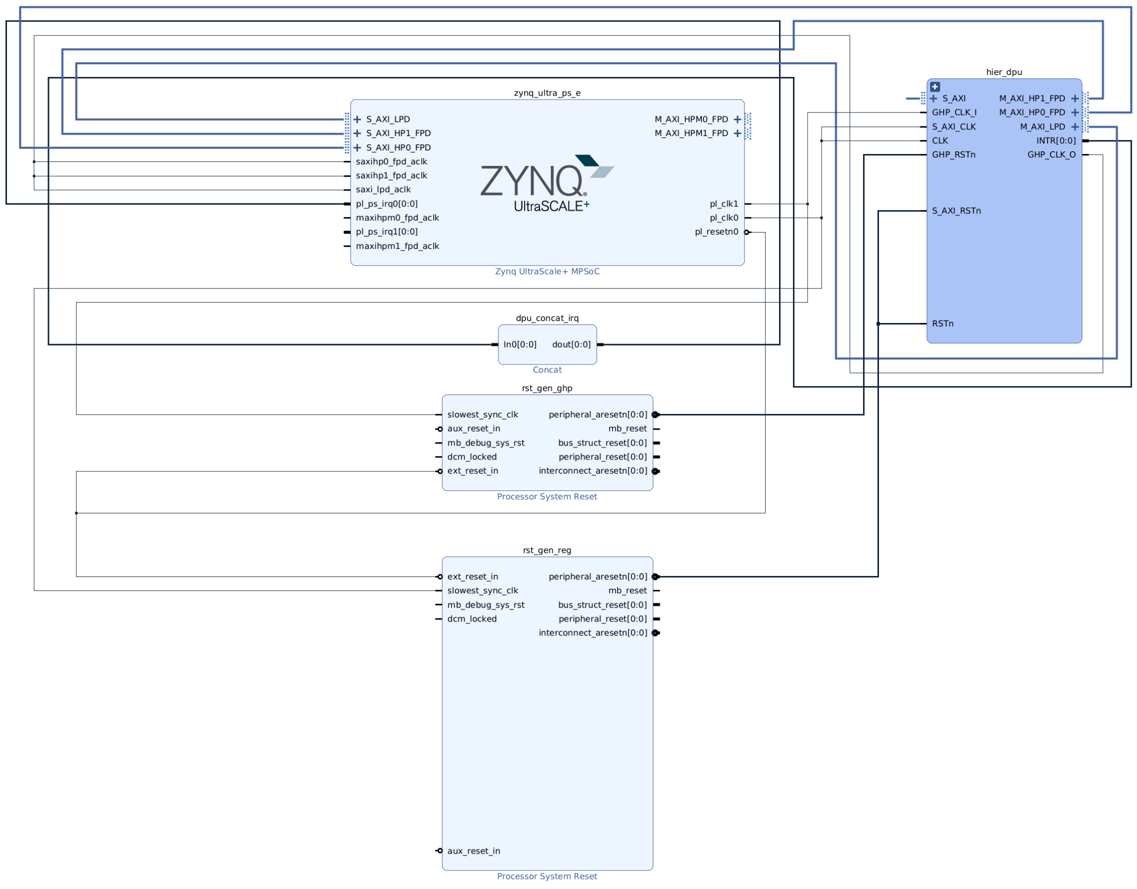

Figure 3.10: The platform design generated by Vivado for KV260.

The second step after creating the block design is to convert the HDL code to a configuration that makes use of transistor logic principles. This is achieved by synthesizing the design, which involves designing digital circuits with transistors and interconnections in a way that enables them to perform the required logical and mathematical operations. These digital circuits are then deployed to the FPGA chip through the implementation process provided by Vivado. Upon completing the implementation, the bitstream is generated, which is the output file that encompasses all the configurations carried out so far. Finally, we need to export the hardware platform as a Xilinx Design Specific Archive (XSA) file, which serves as the foundation of the software components used to build the Petalinux image in the Xilinx Petalinux tool set as shown in Figure 3.11.

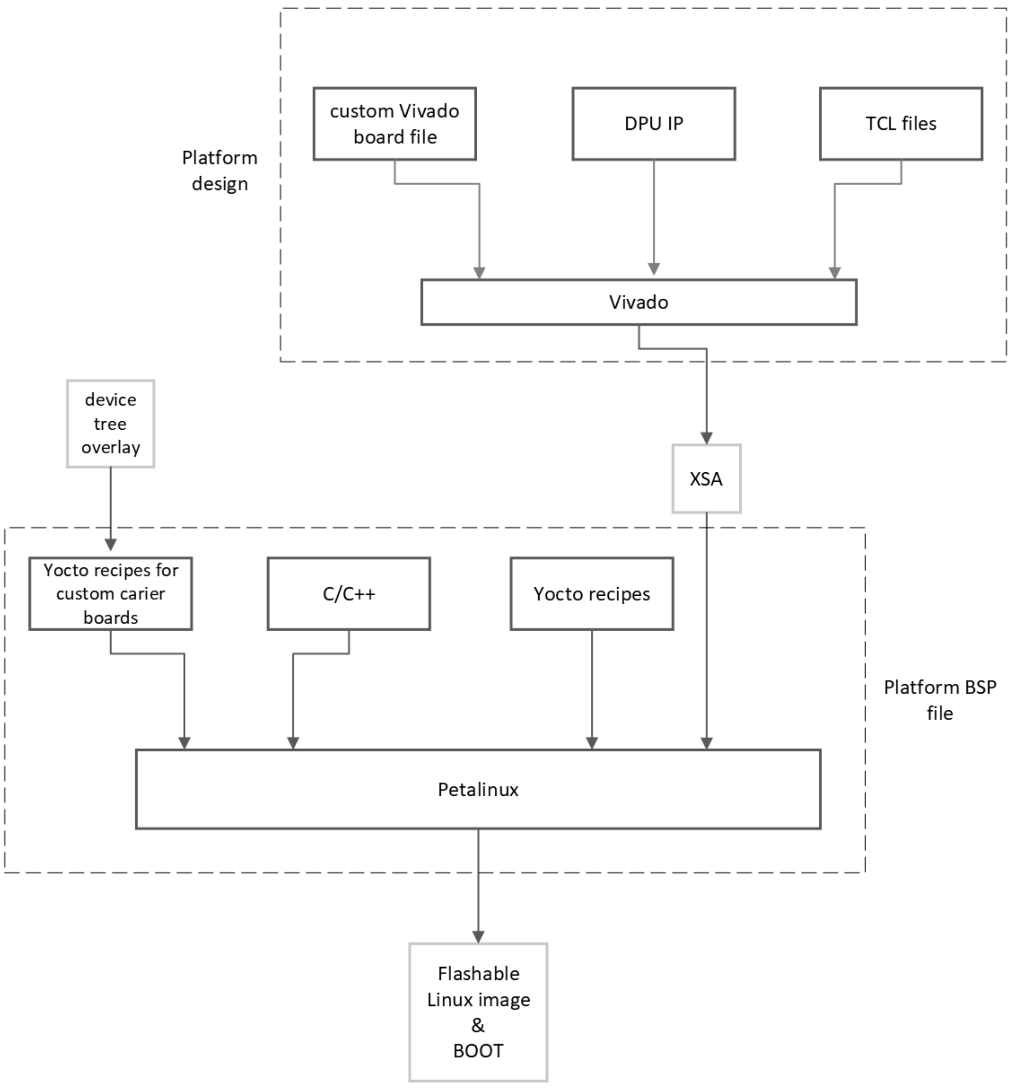

Figure 3.11: Custom Petalinux design flow.

In addition to the XSA file generated by the Vivado environment, the Petalinux tool set will also require a Board Support Package (BSP) file as well as other recipes in order to produce the BOOT file and the flashable Petalinux image. The BSP file contains all the necessary hardware hardware-specific information for both the Kria SOM and the carrier board. This information includes software components and configurations, such as the device drivers and libraries. These components are required for the integration of software with the underlying hardware components.

4 Implementation and Platform Characteristics

The main goal of the system is to avoid obstacles while navigating through a predefined path within a known environment. From this environment, we collected over 130 images of which 100 were used to train the model and the remaining 30 images were designated for the calibration process that took place during the quantization procedure explained in the previous chapter. For this experiment, we employed three DNN models, ResNet-18, AlexNet, and GoogLeNet. we chose these models to represent a variety of architectural complexities and performance characteristics. Both systems were tested with all three models within the same environment for 10 minutes.

The experiments were carried out using two distinct hardware accelerators, a single board GPU-based Jetson Nano from NVIDIA, and a Kria SOM KV260 FPGA-based development kit from Xilinx. Both accelerators were used as edge computing devices in order to augment the capabilities of the TurtleBot 4 lite robotic systems driven by a Raspberry Pi 4 running ROS. Although both hardware accelerators used here are capable of operating the TurtleBot 4 without the need for the Raspberry Pi, we decided to use them with the Raspberry Pi for three main reasons as explained in the previous chapter. The first reason arises from the fact that a hardware accelerator is intended to be used in light of a current system not instead of it. Second, in order to implement ROS on a Kria SOM platform, the image that will be used to configure the DPU and FPGA must be built with KRS, which is still in its beta version with limited documentation. Finally, the Kria KV260 would theoretically deplete the battery of the TurtleBot 4 faster, giving us less time to conduct the test. Both systems established communication with the Raspberry Pi 4 on the TurtleBot via the MQTT protocol, which is possible because ROS is platform-independent.

The only sensor utilized in this work was a stereo-depth OAK-D camera by Luxonis, used as a normal RGB camera. This camera was connected to the Raspberry Pi which was responsible for controlling the TurtleBot, publishing the images captured by the OAK-D camera to a topic, and receiving the results published by the accelerator through another topic. The accelerator only subscribes to the published images by the Raspberry Pi, performs the inference, and publishes the results through MQTT back to the TurtleBot. With this setup, the Raspberry Pi is running two nodes, the first node captures images and publishes them, and the second node subscribes to the results and controls the robot accordingly. The accelerator, on the other hand, is required to execute three threads in parallel, the first thread is responsible for subscribing to the published images and storing them in a designated collection of 7 images. These images are then deleted after being processed, in order to avoid inadvertent re-reading of the same image due to the rapid inference rate by the accelerator. The second thread handles the inference process on the stored images and subsequently publishes the results back to the MQTT broker. The final thread is used to dump the accuracy and power consumption measurements in order to evaluate the performance of the system. Figure 4.1 illustrates the sequential processes and interactions within the system.

In addition to the setup shown in Figure 4.1, we extended the evaluation of the Jetson Nano by connecting the camera directly to it. With this configuration, the Jetson was powered from the battery of the TurtleBot via the barrel jack, leveraging a ten-watt power mode. As discussed in Chapter 3.1, this power mode ensures that the Jetson Nano will function at its highest level of performance. This configuration is expected to give the

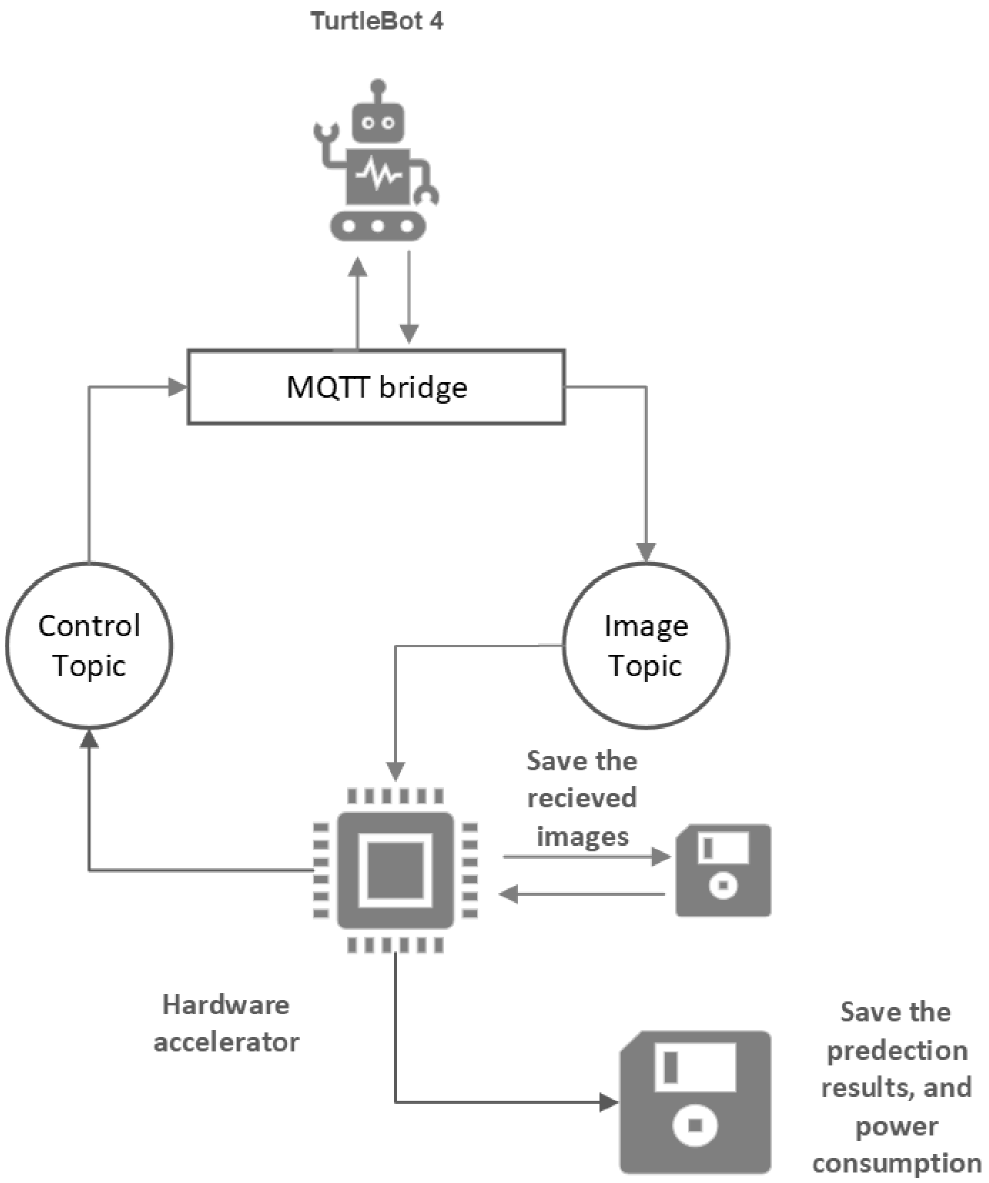

Figure 4.1: System Workflow Diagram

Jetson Nano leverage over the KV260 in terms of network latency, as it will be connected directly to the Raspberry Pi via a LAN connection. Figure 4.2 shows a Jetson Nano mounted on the TurtleBot 4 acting as an edge hardware accelerator while a Raspberry Pi 4 controls the robot using ROS.

The operating system installed on the Jetson Nano was Ubuntu 20.04. Since Jetson is a GPU-based platform, it is not capable of processing low-precision data types. Con-

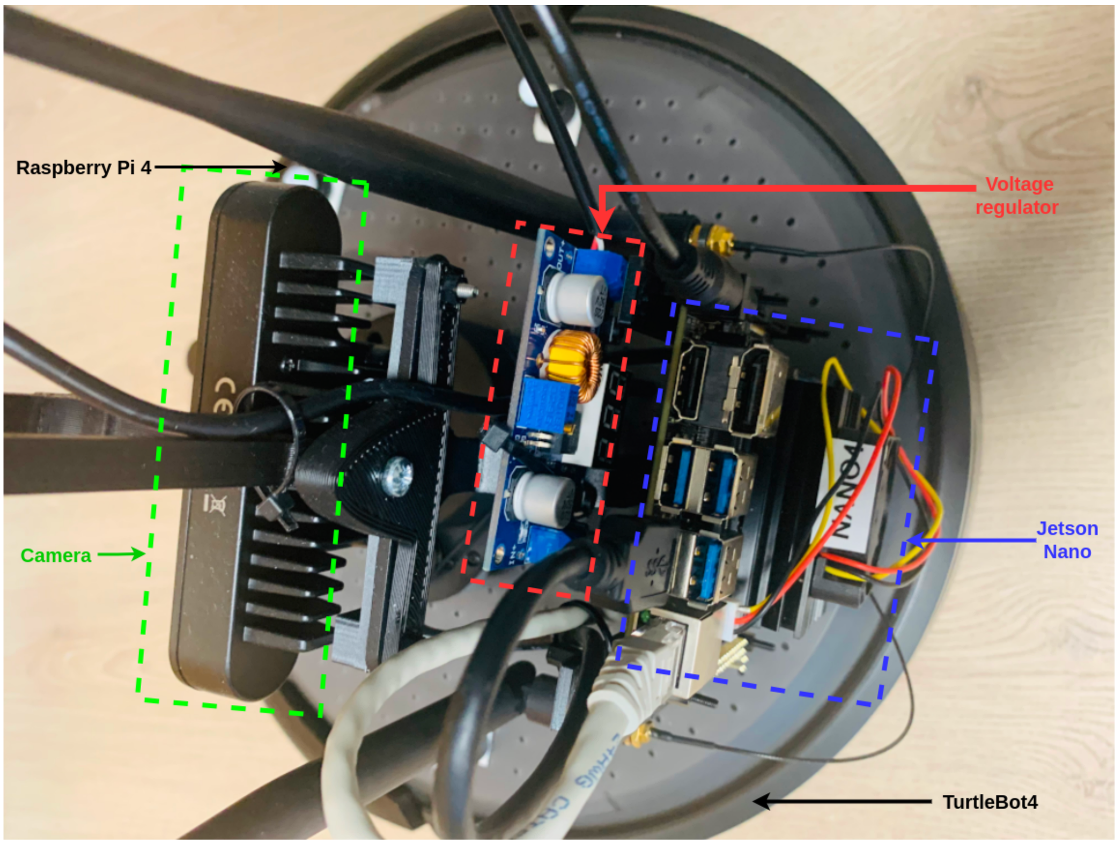

Figure 4.2: The TurtelBot 4 Lite with the Jetson Nano mounted.

sequently, the trained model which was trained and stored with a 32 floating point was used without any additional steps. Conversely, the Kria KV260 is an FPGA-based platform, thus we configured it to harness the computational and memory efficiency offered by lower precision values such as INT8.

For the training process, the models we used were pre-trained on the comprehensive ImageNet dataset. This pre-trained model consisted of thousands of labeled classes, for this experiment, however, we solely needed two classes. Therefore, transfer learning was used in order to replace the classification layer with our custom layer. We divided the collected images from the experimental environment into training and test sets. These sets were then shuffled to generate image batches loaded concurrently through a number of workers. Initially, we opted to train the DNN for 50 epochs, which means the training process will go through our full dataset 50 times. However, we observed that 15-18 epochs gave the best accuracy. Algorithm 2 provides a detailed description of the training process.

From this training procedure, a floating-point model was generated, which is suitable for performing the inference on the Jetson Nano development board. For the Kria KV260 AI vision kit, this floating-point model must be quantized in the Vitis AI environment. Fundamentally, there are three different frameworks available in the Vitis AI environment that can be used for model quantization. These frameworks are, PyTorch, TensorFlow, and ONNX. The choice between these frameworks depends on the trained model. The framework we used for this study was PyTorch. After the successful activation of the PyTorch framework in the Vitis AI environment, we started the quantization process by executing the quantization algorithm discussed in Chapter 3.1 two times. The first set of iterations involved calibrating the weights, biases, and activation values through our dedicated set of calibration images, whereas the second execution aimed at exporting the final quantized model. This quantized model was then compiled within the same Vitis AI environment by specifying the architecture of the DPU. Since we are targeting the Kria KV260, the name of the DPU should be DPUCZDX8G, following the naming convention mentioned in Section 2.3.

Before deploying the compiled model onto the intended device, we created a flashable Petalinux image that contains our custom FPGA and DPU configurations. These configurations included the number of cores, frequency, and activation functions employed in the DNN model in addition to other settings and configurations. At first, Vivado was used to create the FPGA hardware block design by selecting the appropriate hardware components, such as the Zynq MPSoC Arm-core processing system, clock signals, and interrupt peripherals. These components are also available within the prebuilt DPU IPs provided by Xilinx. For this project, DPUCZDX8G IP v4.0 was used with the configurations listed in Table 4.1.

By default, the DPU IP we used to generate this block design establishes the appro-

Algorithm 2 Training Process

```
 1: Load the model pre-trained on ImageNet
 2: Modify the last layer to include only 2 classes
 3: Load training and validation data
 4: Define loss function and optimizer
 5: Set number of epochs
 6: for each epoch do
 7:     Set model to training mode
 8:     for each batch in training data do
 9:         Forward pass-through model
10:         Compute loss
11:         Backpropagate gradients
12:         Update model weights
13:     end for
14:     Set model to evaluation mode
15:     Calculate validation accuracy
16:     Print epoch, loss, and accuracy
17: end for
18: Save model's weights
```

Table 4.1: DPU Configuration Parameters

Parameter	**Value**
DPU_CLK_MHz	304
DPU_NUM	1
DPU_SFM_NUM	0
DPU_URAM_PER_DPU	50
prj_part	xck26-sfvc784-2LV-c
prj_board	KV260
DPUCZDX8G Architecture	B4096
CNN	ReLU + LeakyReLU + ReLU6
RAM usage	Low
DPU_2x clock gating	Enabled

priate connections between the Zynq Ultrascale+ processing unit and the DPU, clock, and interrupt components. In addition to these connections, this DPU IP also provides the most optimal configurations needed when using the Kria KV260 vision AI and starter kit as a hardware accelerator rather than a general-purpose computing system. These configurations include the deactivation of the two full-power high-performance interrupt peripherals while solely activating one low-power port. The intention behind this configuration is to reserve the full-power peripherals exclusively for hardware acceleration use cases. Figure 4.3 shows the final FPGA and DPU block design that was utilized in this experiment.

The Petalinux tool set was then used with the Kria KV260 BSP file supplied by Xilinx, in order to package the hardware design into a flashable image. We then flashed this image to the micro-SD card, which is the secondary boot device on the Kria KV260. With this setup in place, it was possible for us to run the inference on the FPGA-based platform. Essentially, we configured the Raspberry Pi to continually publish images, via

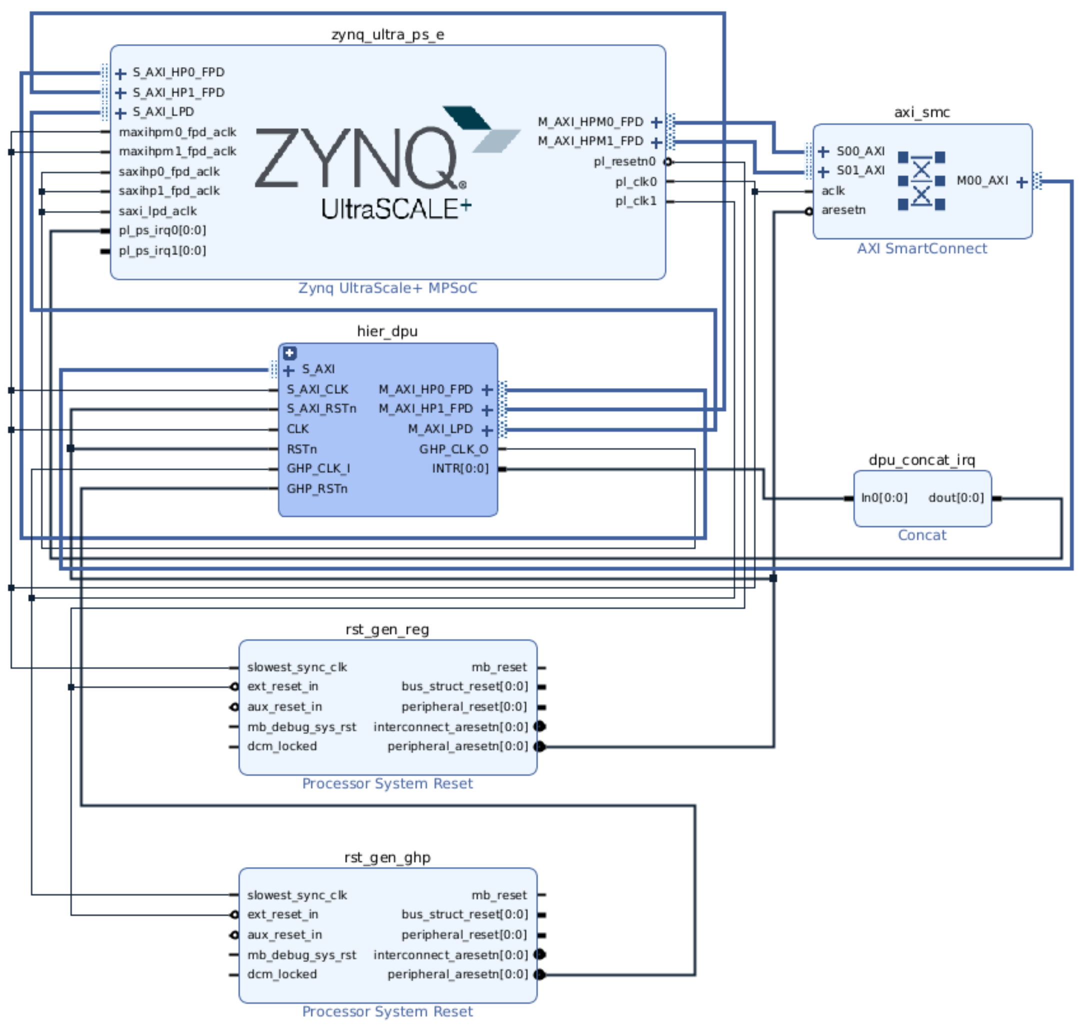

Figure 4.3: The FPGA final block design.

a ROS node, to an image topic at intervals of 0.4 seconds. Since the accelerators are not running ROS, the publishing and subscribing procedure passes through an MQTT bride to an MQTT broker, which can be harnessed by both acceleration systems. With this network, we intended to test the capabilities of both accelerators as edge computing devices. Figure 4.4 shows the KV260 connected as an edge computing device.

Basically, both accelerators continue to run a thread with the sequential steps outlined in Algorithm 3 indefinitely in a loop. Alternatively, we also connected the camera directly to both platforms in which case the MQTT communication method was only used to publish the commands that control how the TurtleBot moves. With an outcome that infers

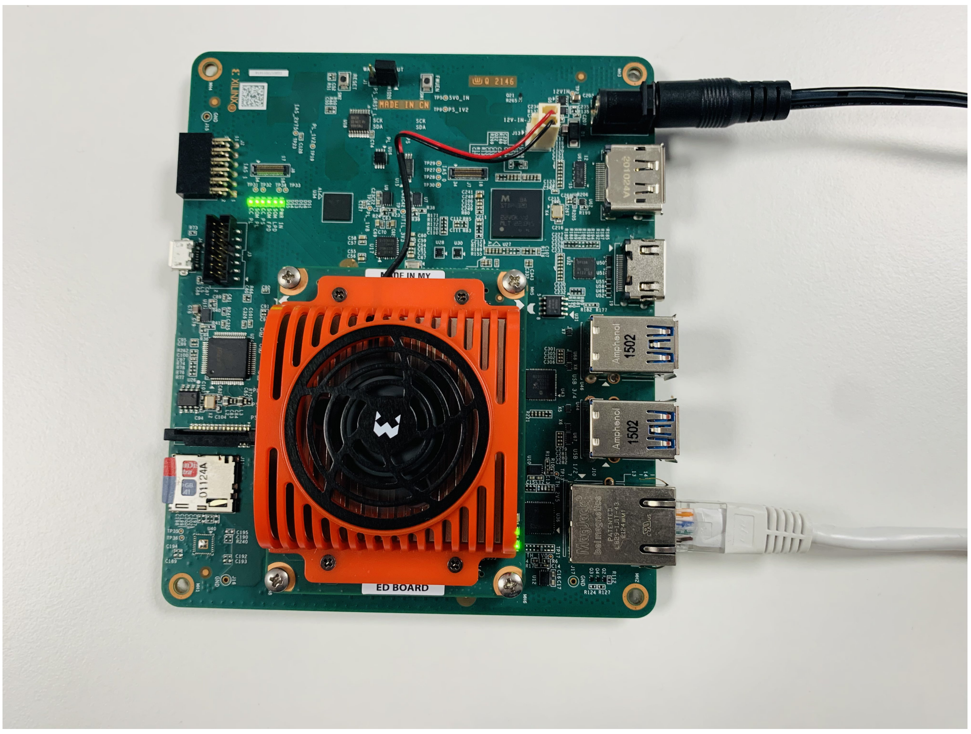

Figure 4.4: The Kria KV260 is used as an edge computing hardware accelerator.

the presence of an obstacle, the published command will force the TurtleBot to rotate in place. Inversely, If the prediction indicates a free path, the accelerator will publish a move forward command. In this work, we considered an object as an obstacle if its distance from the robot was less than one meter due to the exclusive use of an RGB camera which poses a challenge to identify objects at closer distances. Figure 4.5 shows two robot-captured images of an obstacle on the way of the robot. A detailed explanation of the Jetson Nano and KV260 inference process is presented in Algorithms 4 and 5 respectively, while Algorithm 6 outlines the methodology employed to assess the model's accuracy.

Algorithm 3 Handling MQTT Messages

```
1: while True do
2:     Subscribe to the image MQTT topic
3:     while receiving an MQTT messages do
4:         Receive message: $msg(image)$
5:         Save images but overwrite them when seven images have been received.
6:     end while
7: end while
```

(a) (b)

Figure 4.5: Robot-captured images of an obstacle at varying distances: (a) 65 cm, (b) 120 cm

Algorithm 4 Jetson Nano Inference Process

```
1: $model \leftarrow trained\_model$
2: $device \leftarrow cuda$
3: Load model to device.
4: Resize the image to a smaller size and normalized pixel values.
5: Initialize the pipeline
6: Create a color camera source node
7: Set the color order of the image to RGB
8: Set the preview size and color order for the camera
9: Initialize an empty list to store the FPS info
10: Connect to the device and start the pipeline
11: while True do
12:     Convert the RGB frame to a PyTorch tensor
13:     Load the tensor to the GPU
14:     Perform inference using the model
15:     Calculate FPS
16:     Get the predicted class and corresponding label
17:     if Predicted class == Free then
18:         Publish a move forward command to the cmd vel topic
19:     else if Predicted class == Blocked then
20:         Publish a turn command to the cmd vel topic
21:     end if
22:     wait for 0.1 seconds
23: end while
```

Algorithm 5 Kria KV260 Inference Process

```
1: Load the Xmodel.
2: Load and resize the images.
3: Define the data type of the images as INT8.
4: Determine the batch size and number of test images.
5: Initialize counters for processing.
6: while images $\neq$ Zero do
7:     Determine the number of images in the batch.
8:     Initialize buffers (input/output) for the DPU.
9:     for each image within the current batch of images do
10:        Load the image.
11:        Resize it.
12:        $input\_buffer \leftarrow preprocessed\_image.$
13:    end for
14:    Run the DPU with the input buffer.
15:    Wait for the DPU to complete the execution.
16:    Fetch the output
17:    for each output within the current batch do
18:        Get the prediction results.
19:        if Predicted class == Free then
20:            Publish a move forward command to the cmd vel topic
21:        else if Predicted class == Blocked then
22:            Publish a turn command to the cmd vel topic
23:        end if
24:    end for
25: end while
```

Algorithm 6 Model Accuracy

```
$correct\_predictions \leftarrow Zero$
for each image in the blocked image folder do
    Perform inference on the model using the image
    Get the index of the predicted class with the highest probability (pred)
    if the prediction is correct then
        $correct\_predictions \leftarrow correct\_predictions + 1$
    end if
end for
for each image in free image folder do
    Perform inference on the model using the image
    Get the index of the predicted class with the highest probability (pred)
    if the protection is correct then
        $correct\_predictions \leftarrow correct\_predictions + 1$
    end if
end for
$Total\_images \leftarrow blocked\_images + free\_images$
$Accuracy \leftarrow \frac{correct\_predictions}{Total\_images}$
```

5 Experimental Results

The results presented in this chapter were obtained by running the inference of three DNN modes on both the Jetson Nano and Kria KV260 using the system configuration described in Chapter 4. The DNN models we chose for this experiment were AlexNet, GoogLenNet, and Resnet-18. Fundamentally, we based our assessment on four main metrics, throughput, power consumption, prediction accuracy of the model, and intricacies of the development process. But before we delve into the results obtained from the accelerators, it might be useful to evaluate the performance of the Raspberry Pi 4 alone, which is the main driver of the robot being tested. The operating system running on this Raspberry Pi is Ubuntu 20.04, which includes a ROS installation as part of its software suite to control the robot.

Table 5.1: Power Consumption (Watts) of the Raspberry Pi 4 B

Operating State	**Power Consumption**
Idle	5.26 V X (0.56 - 0.63) A = (2.94 - 3.31) W
Running DNN	5.26 V x (0.81 - 0.94) A = (4.26 - 4.94) W

We decided to benchmark the Raspberry Pi in a similar method as the accelerators, in other words, the Raspberry Pi was considered an edge device powered by a 5V adopter rather than the robot's battery. This configuration also allowed us to obtain an accurate measurement of the Raspberry Pi's power consumption, by connecting it to the power adapter through a USB tester, as illustrated in Figure 5.1. The USB tester provides voltage and current measurements, from which the total power consumption is the result of multiplying the two values. Table 5.1 outlines the power consumed by the Raspberry Pi during inactivity periods and when running a DNN model.

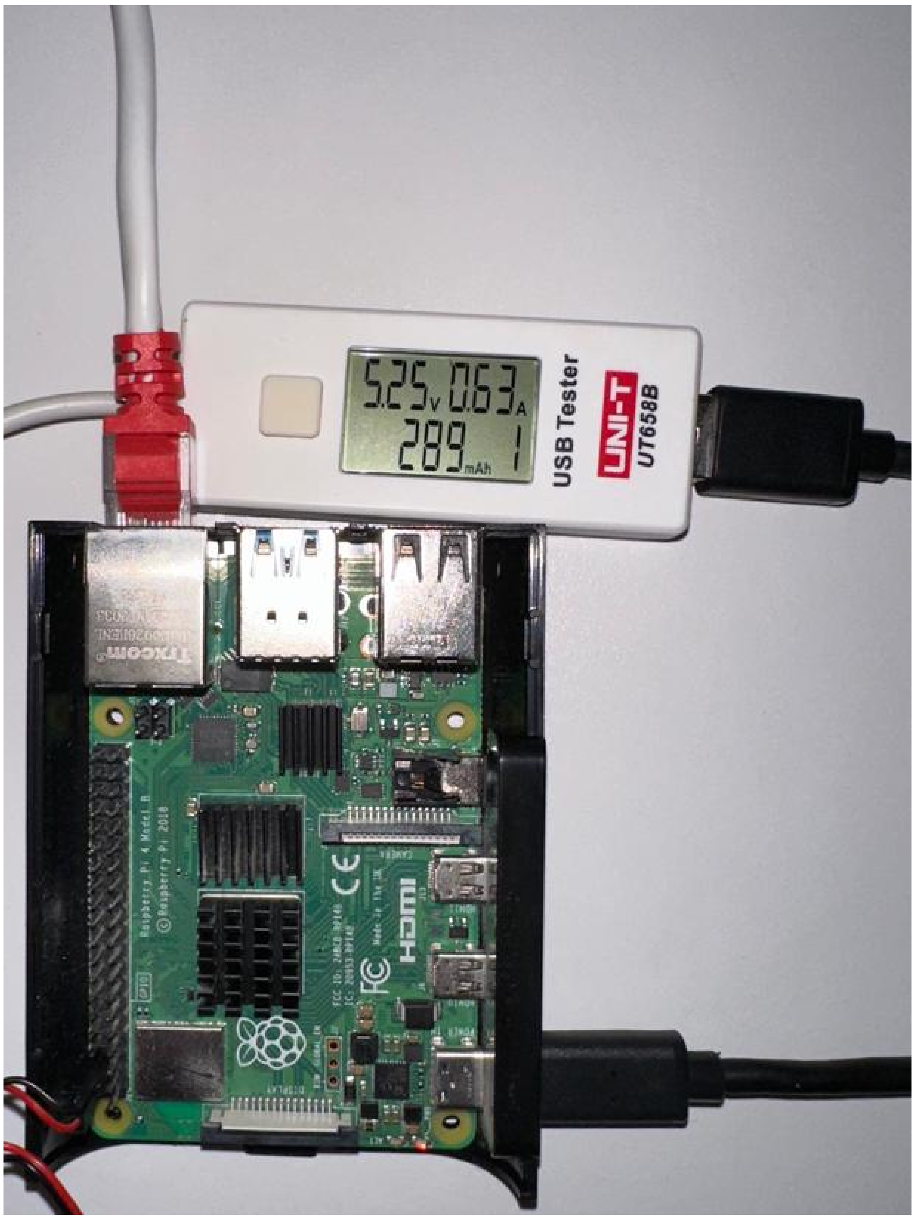

Figure 5.1: Measuring the power consumption of the Raspberry Pi with a USB tester.

Figure 5.2 shows the frames Per Second (FPS) throughput of the Raspberry Pi running the three DNN models. It is evident from this figure that the Raspberry Pi faces significant challenges in running any of the three models alone.

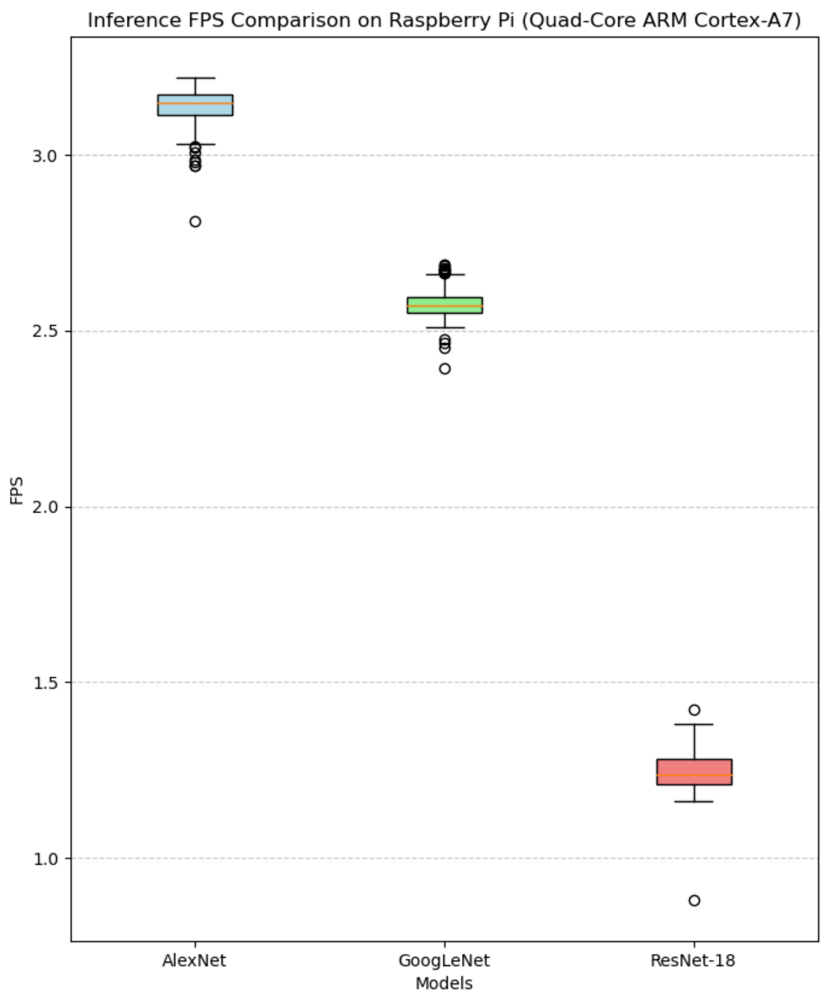

Figure 5.2: The Inference Performance of a Raspberry Pi 4 with AlexNet, GoogLeNet, and ResNet-18 Model.

While the AlexNet DNN model outperformed GoogLeNet and ResNet-18 when running on the Raspberry Pi 4, with a maximum of about 3.5 FPS, the Raspberry Pi is clearly not suitable for real-time inference. Next, we look at the FPS rate of both the Kria KV260 AI Vision development kit, which we configured and optimized to run DNN models, and the Jetson Nano with all three models. Each model was tested for roughly 5 minutes, in which the inference rate was measured in isolation from all other operations, including the process of subscribing to the image topic, saving the images, and publishing the results to the Raspberry Pi. We did this to eliminate any communication delays. Figure 5.3 illustrates the inference rate on the Jetson Nano for both the AlexNet and GoogLeNet convolutional neural networks. Compared to the AlexNet model, the Jetson Nano per-

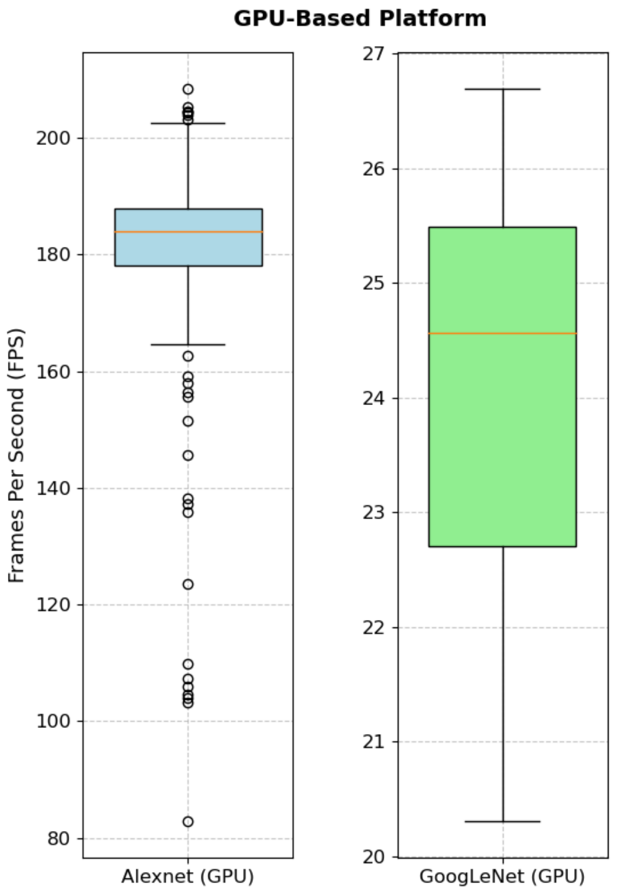

Figure 5.3: AlexNet and GoogLeNet Binary Inference FPS on a Jetson Nano (GPU-Based Platform).

formed poorly when executing the inference with the GoogLeNet neural network despite running the same binary classification within an identical environment. With ResNet-18 used as the DNN model, the Jetson Nano still appeared to have low performance when compared to the FPS speed obtained from the AlexNet DNN architecture. However, ResNet-18 displayed consistent performance with minimal outliers during inference as shown in Figure 5.4.

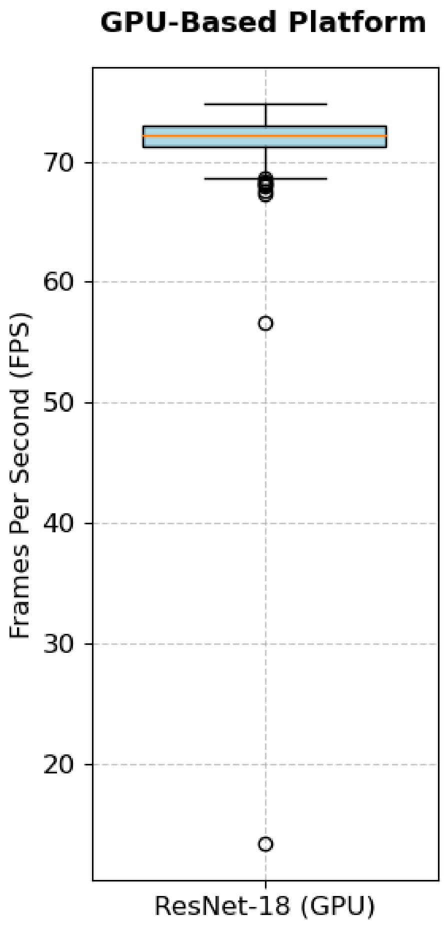

Figure 5.4: ResNet-18 Binary Inference FPS on a Jetson Nano (GPU-Based Platform).

The FPS rates obtained by running the three models on the Kria KV260 showed a significant improvement for both ResNet-18 and GoogLeNet models while offering less throughput with the AlexNet DNN model as depicted in the box plots shown in Figures 5.5 and 5.6.

The reason for this substantial performance fluctuation between the two accelerators with these three DNN models is possibly due to the differences in the number of convolutional layers, the complexity of the activation functions in use, the memory access patterns, and the magnitude of the parameters needed. These factors might have contributed to the increased bandwidth demands leading to a decrease in the number of inferences per second. On the GPU-based platform, AlexNet achieved a remarkably better

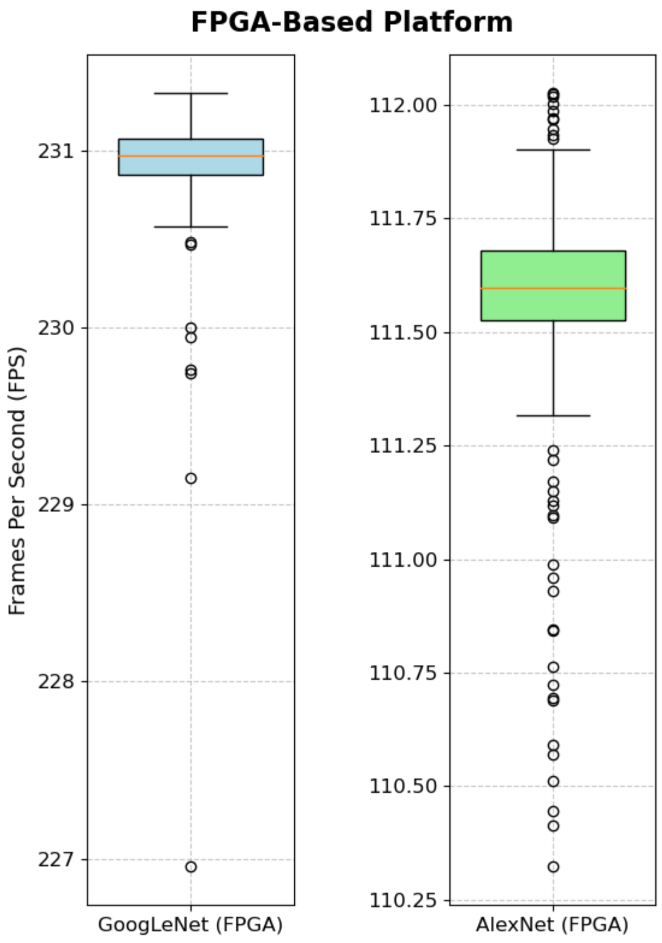

Figure 5.5: The Inference Performance of the Kria KV260 (FPGA-based platform) with the AlexNet and GoogLeNet DNN Models.

FPS rate than that realized by the FPGA-based system, which was clearly caused by the higher bandwidth requirements of the model. This demand for bandwidth forced the Kria KV260 to increase the number and speed of DDR read-write operations as shown in Figure 5.7, potentially leading to more than five times fewer GOPS compared to the DDR read-write rate and GOPS of the ResNet-18 model presented in Figure 5.8 and Table 5.2. This higher bandwidth demand also compelled the CPU on the KV260 to perform part of the computations within 40 and 20 runs as described in Table 5.2.

The appalling performance of the GoogLeNet DNN model on the Jetson Nano was also caused by the layer count and vast number of branches this model possesses. With 48

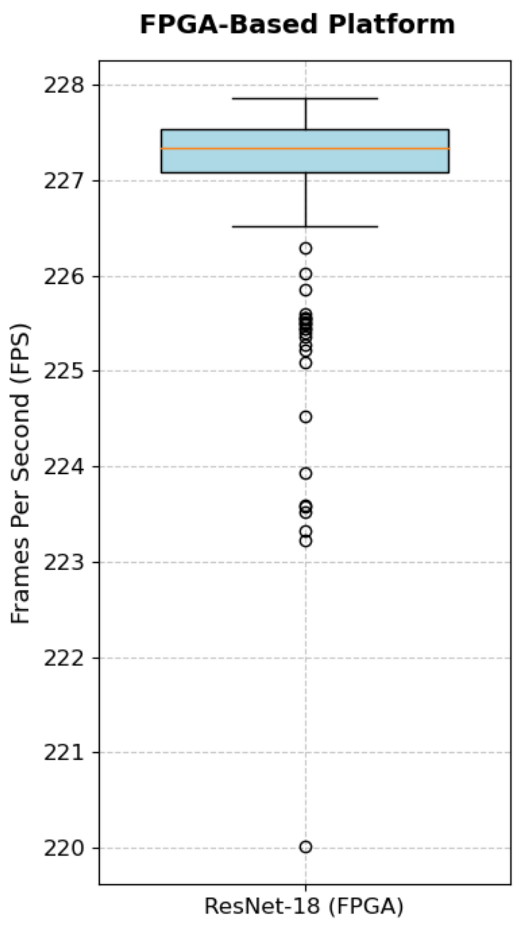

Figure 5.6: The Inference Performance of the Kria KV260 (FPGA-based platform) with the ResNet-18 DNN Model.

layers, GoogLeNet posed a challenge for the GPU on the Jetson Nano to accomplish 30 FPS. The Kria KV260 on the other, delivered outstanding throughput with the GoogLeNe model. The DDR read-write rate was also shown to be considerably lower than both the AlexNet and ResNet-18 models, which allowed for more GOPS as shown in Figure 5.9 and Table 5.2. Although the Jetson Nano outperformed the Kria KV260 with the AlexNet DNN model, there are still two more important metrics before we can deem the platform to be suitable for real-time edge computing applications. These two metrics are power consumption and accuracy in light of the previously presented FPS throughput. In terms of power consumption, the Kria KV260 consumed approximately 5.9 watts when idling, and up to 9.42 watts when performing a DNN inference as shown in Table 5.3.

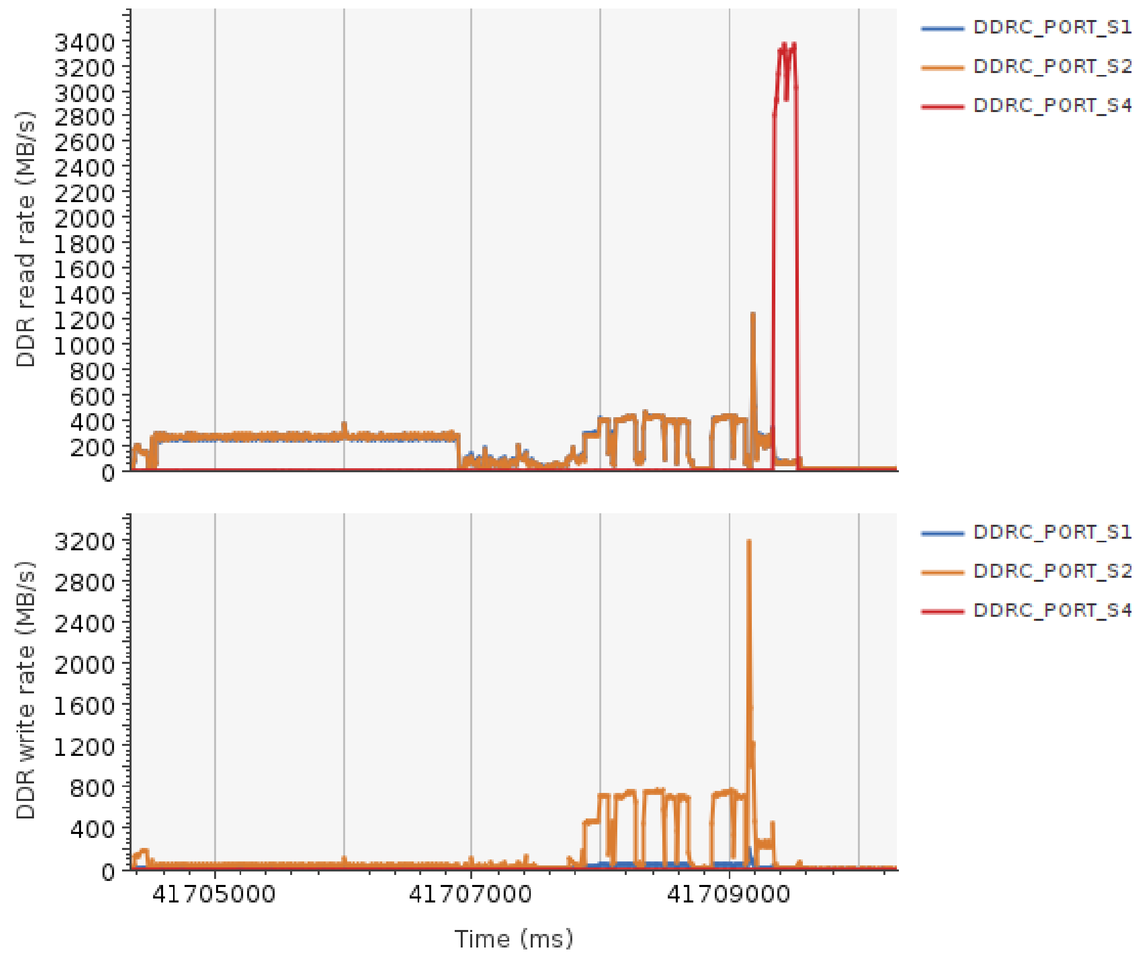

Figure 5.7: AlexNet DDR read-write rate on the Kria KV260.

Table 5.2: Profiling the Kria KV260 with different DNN models.

Metric	AlexNet	GoogLeNet	ResNet-18
Runs	20,40,20	42	20
Computing Unit	DPU+CPU	DPU	DPU
Workload GOP	1.421	3.011	3.633
Performance GOP/s	13.631	36.817	74.466
Mem IO MB	54.529	10.164	11.639
Mem Bandwidth GB/s	6.425	1.527	2.931

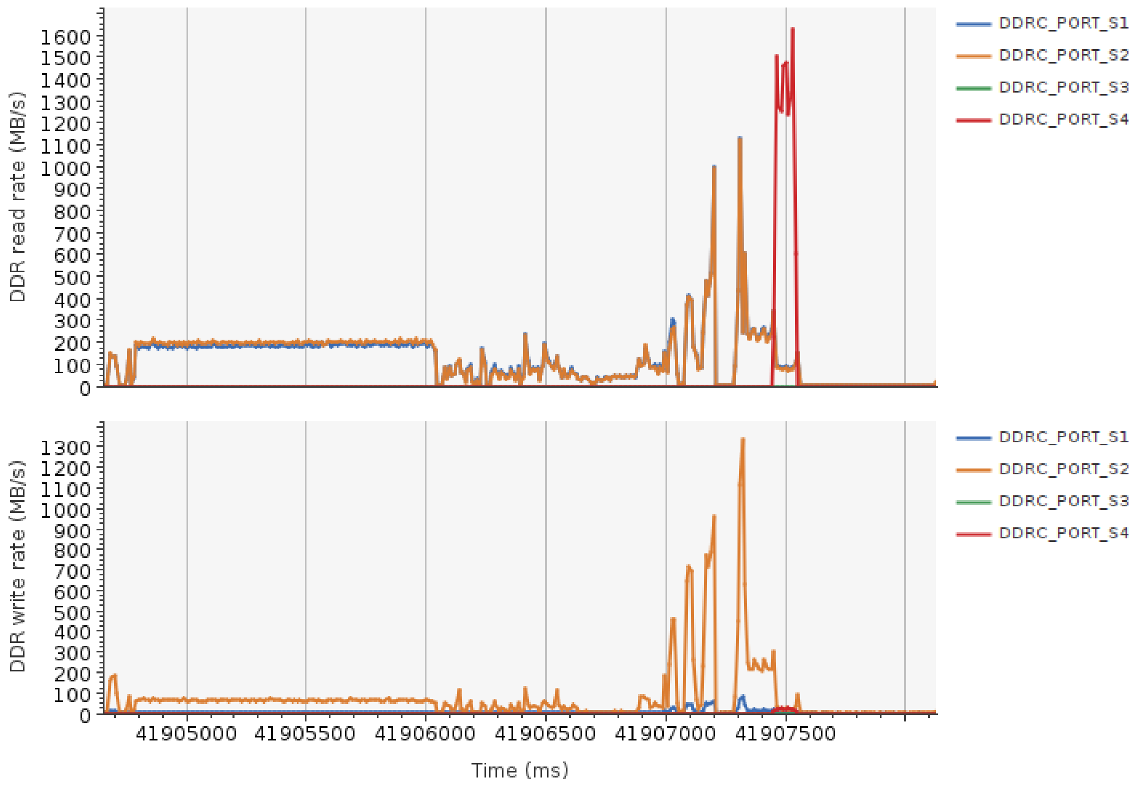

Figure 5.8: ResNet-18 DDR read write rate on the Kria KV260.

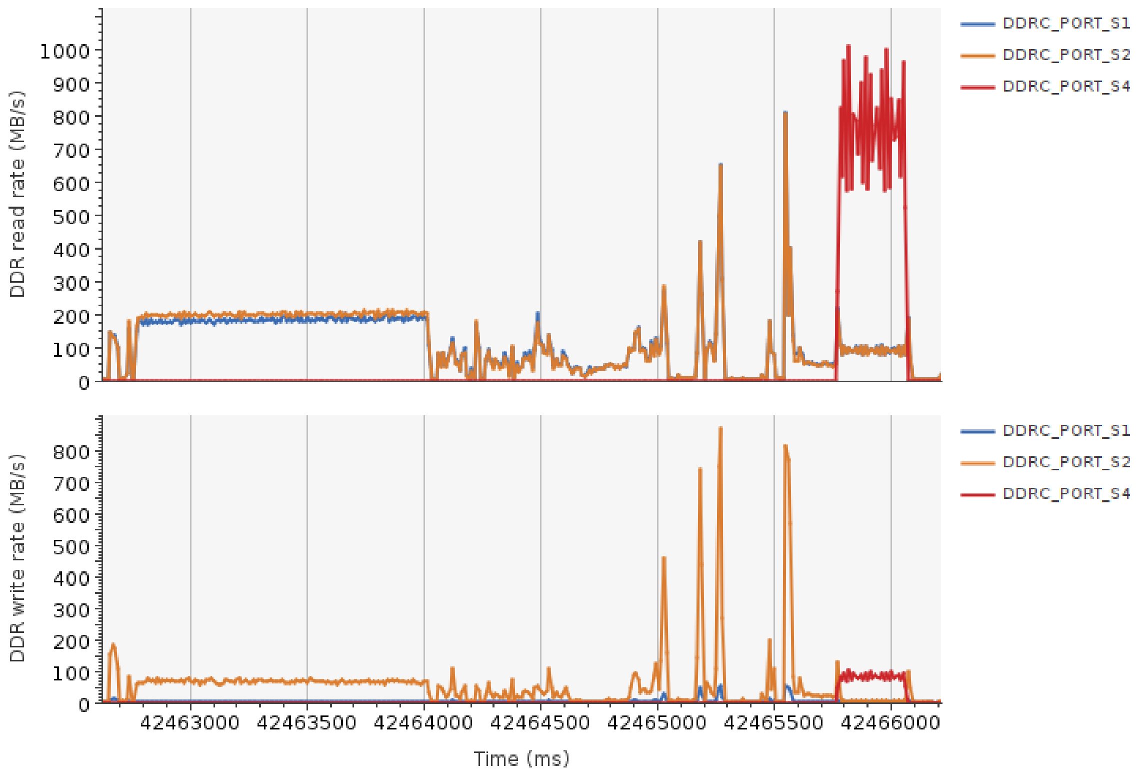

Figure 5.9: GoogLeNet DDR read write rate on the Kria KV260.

Table 5.3: Power Consumption (Watts) of the DNN Models on Jetson Nano and KV260

Metric	**KV260 Power (W)**	**Jetson Nano Power (W)**
Idle	5.9	3.2
Running inference	7.98 - 9.42	7.60 - 8.20

While this consumed power is slightly higher than that consumed by the Jetson Nano under the same power conditions, the KV260 attained better performance per watt than the Jetson Nano considering the average FPS rate for each model as illustrated in Figures 5.10 and 5.11.

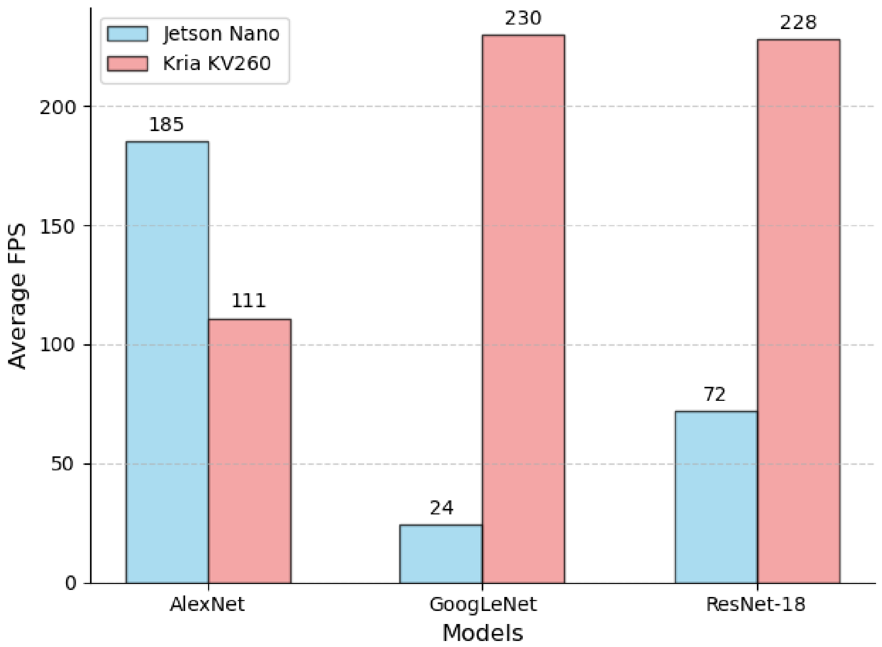

Figure 5.10: Average FPS of AlexNet, GoogLeNet, and ResNet-18 on Jetson Nano and KV260.

Despite the marginally greater performance per watt reported for the Jetson Nano in Figure 5.11, the AlexNet model exhibited the lowest prediction accuracy out of all three models as listed in Table 5.4, yet it should be able to avoid obstacles in real-time due to the high inference rate when compared to the movement of the robot. The reason for this

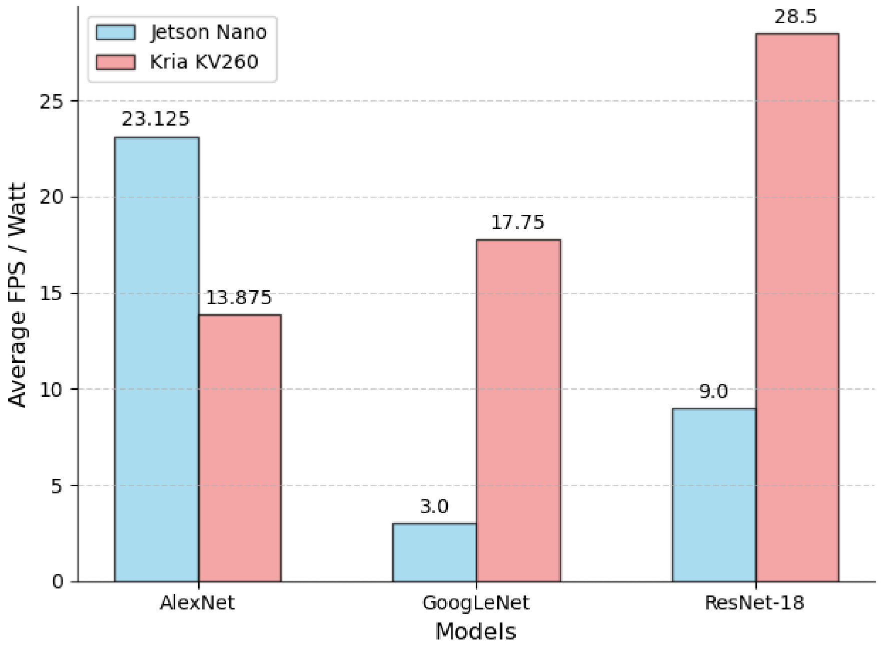

Figure 5.11: Performance per Watt.

Table 5.4: Accuracy Prediction of DNN Models on FPGA and GPU

Model	**KV260 Accuracy (%)**	**Jetson Nano Accuracy (%)**
AlexNet	65.2	68.3
GoogLeNet	74.1	77.5
ResNet-18	79.3	83.9

is that the model would need to make at least 46 incorrect predictions per second in order for the robot to collide, given its top speed of 0.5 m/s. Likewise, ResNet-18, which enabled the Jetson Nano to achieve an inference speed of up to 72 fps, mitigates this risk of collisions. The same, however, cannot be asserted for the GoogLeNet implementation on the Jetson Nano unless the speed of the robot is reduced to 0.2 m/s. This speed reduction resulted in a tremulous movement, particularly noticeable when an obstacle is detected and the robot attempts to avoid it. Unlike the Jetson Nano, the Kria KV260 was capable of maneuvering around obstacles smoothly with all three DNN models. Another noteworthy observation from the accuracy measurements shown in Table 5.4, is that quantizing the model from a 32-floating point to an INT8, utilizing the Vitis AI framework, exerted a negligible impact on the accuracy of all three DNN models tested in this work.

Finally, the process of configuring the DPU, optimizing, quantizing, compiling, and deploying a model on the FPGA, within a Vitis AI environment, did not necessitate any prior knowledge of electronics or HDL. Nevertheless, it was more involved compared to deploying the same model on a GPU-based platform from the perspective of a developer with sufficient CUDA programming skills.

6 Conclusion

In this thesis, we demonstrated the benefits of adding an FPGA-based platform, as a hardware accelerator, to a robotic system driven solely by a Raspberry Pi 4, which is based on an Arm Cortex-A7 CPU. The ultimate goal of this system was to improve the inference speed of a DNN model specifically designed for binary image classification, without compromising on power consumption. Our findings proved that the FPGA-based Kria KV260 was able to achieve a significant 163-fold increase in the inference rate compared to running the same DNN model on the primary controller of the robot, which was a Raspberry Pi 4. Additionally, our experimental results showed a notable improvement of over 6 times in the FPS throughout when running accurate DNN models on the Kria KV260 compared to the GPU-based Jetson Nano evaluation board. Moreover, we discussed the architecture of three different DNN models for real-time binary image classification and substantiated clear correlations between certain architectural components of the models and how they performed on the two accelerator platforms presented in this work. Furthermore, the empirical evidence we provided in this thesis showed that the Kria KV260 was able to offer a performance-to-power ratio exceeding four times that of the Jetson Nano, even though the latter consumed less power than the former. Finally, our work suggested that developing and deploying a DNN model on an FPGA-based platform can be achieved without the need for prior knowledge of hardware-level design.

6.1 Future works

While the work conducted in this thesis has made considerable strides in improving the performance per watt of a robotic system, the presence of several limitations might be worth further examination. One of these limitations stems from the relatively slow speed of the robot, which was possibly caused by the Raspberry Pi's slow publishing and subscribing rate of the ROS node which controls the linear and angular speed necessitating the inclusion of a brief delay to the accelerator. In order to address this issue, we considered making use of the Kria KR260 Robotic Starter Kit. With this platform, we were able to achieve up to ten times the publishing rate of that obtained from the Raspberry Pi 4 alone. However, deploying this development kit on the TurtelBot 4 requires a voltage regulator capable of supplying 12V with at least 2A from the 36 W battery pack in the TurtleBot. Additionally, the KR260 supports ROS2 through the KRS framework. This framework must be built and flashed as an image to the secondary boot device on the KR260, to this end it was not clear to us how to apply the DPU configurations to the same flashable image. Having the KR260 as an additional accelerator would have complicated the system and defeated the main purpose of this work. One solution to this is to design a custom carrier board for the K26 SOM and employ it as a viable alternative to the Raspberry Pi 4.

Another limitation was the communication method used between the accelerator and the robot, which restricted the use of the system to a limited coverage area. Looking ahead, 5G technology could be a compelling area of research for this system. Finally, training the model and applying the optimizations, quantization, and compilation to the DNN model, all took place on a host machine due to the constrained resources on the FPGA-based platform. This limitation not only increases the development time but also limits the use of the model within a restricted environment with predefined parameters. Moving this training procedure onto the FPGA-based platform might yield interesting possibilities.

References

[1] C. A. Mack, “Fifty years of moore’s law”, *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, 2011. DOI: https://doi.org/10.1109/TSM.2010.2096437.

[2] M. Radosavljevic and J. Kavalieros, “Taking moore’s law to new heights: When transistors can’t get any smaller, the only direction is up”, *IEEE Spectrum*, vol. 59, no. 12, pp. 32–37, 2022. DOI: https://doi.org/10.1109/MSPEC.2022.9976473.

[3] J. R. Powell, “The quantum limit to moore’s law”, *Proceedings of the IEEE*, vol. 96, no. 8, pp. 1247–1248, 2008. DOI: https://doi.org/10.1109/JPROC.2008.925411.

[4] R. A. Street, “Thin-film transistors”, *Advanced Materials*, vol. 21, no. 20, pp. 2007–2022, 2009. DOI: https://doi.org/10.1002/adma.200803211.

[5] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini, “A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives”, *Journal of Systems Architecture*, vol. 129, p. 102 561, 2022, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102561.

[6] A. Sethi and H. Kushwah, “Multicore processor technology-advantages and challenges”, *International Journal of Research in Engineering and Technology*, vol. 4, no. 09, pp. 87–89, 2015.

[7] M. D. Hill and M. R. Marty, “Amdahl’s law in the multicore era”, *Computer*, vol. 41, no. 7, pp. 33–38, 2008. DOI: https://doi.org/10.1109/MC.2008.209.

[8] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, “Model compression and hardware acceleration for neural networks: A comprehensive survey”, *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020. DOI: https://doi.org/10.1109/JPROC.2020.2976475.

[9] S. Bavikadi, A. Dhavlle, A. Ganguly, *et al.*, “A survey on machine learning accelerators and evolutionary hardware platforms”, *IEEE Design & Test*, vol. 39, no. 3, pp. 91–116, 2022. DOI: https://doi.org/10.1109/MDAT.2022.3161126.

[10] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, “A common backend for hardware acceleration on fpga”, in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 427–430. DOI: https://doi.org/10.1109/ICCD.2017.75.

[11] G. Rathod, P. Shah, R. Gajjar, M. I. Patel, and N. Gajjar, “Implementation of real-time object detection on fpga”, in *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)*, 2023, pp. 235–240. DOI: https://doi.org/10.1109/ICOEI56765.2023.10125958.

[12] E. Sola-Thomas, M. A. Baset Sarker, and M. Imtiaz, “Fpga-controlled ai vision for prosthetics hand”, in *2023 IEEE World AI IoT Congress (AIIoT)*, 2023, pp. 0520–0524. DOI: https://doi.org/10.1109/AIIoT58121.2023.10174491.

[13] T. N. Theis and H.-S. P. Wong, “The end of moore’s law: A new beginning for information technology”, *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017. DOI: https://doi.org/10.1109/MCSE.2017.29.

[14] F. Schlachter, “No moore’s law for batteries”, *Proceedings of the National Academy of Sciences*, vol. 110, no. 14, pp. 5273–5273, 2013.

[15] F. Schirrmeister, “Chapter 3 - multicore architectures”, in *Real World Multicore Embedded Systems*, B. Moyer, Ed., Oxford: Newnes, 2013, pp. 33–73, ISBN: 978-0-12-416018-7. DOI: https://doi.org/10.1016/B978-0-12-416018-7.00003-1.

[16] T. M. John Cheng Max Grossman, *Professional CUDA C Programming*. John Wiley & Sons, 2014, ch. 3.

[17] Park and Kim, “2d gpu-accelerated high resolution numerical scheme for solving diffusive wave equations”, *Water*, vol. 11, p. 1447, Jul. 2019. DOI: https://doi.org/10.3390/w11071447.

[18] L. Hasan, M. Kentie, and Z. Al-Ars, “Dopa: Gpu-based protein alignment using database and memory access optimizations”, *BMC research notes*, vol. 4, p. 261, Jul. 2011. DOI: https://doi.org/10.1186/1756-0500-4-261.

[19] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, “Performance characterisation and simulation of intel’s integrated gpu architecture”, in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 139–148. DOI: https://doi.org/10.1109/ISPASS.2018.00027.

[20] “Jetson nano brings AI computing to everyone”, NVIDIA Technical Blog. (Mar. 18, 2019), [Online]. Available: https://developer.nvidia.com/blog/jetson-nano-ai-computing/ (visited on 08/12/2023).

[21] A. R. Omondi, *FPGA implementations of neural networks*. Springer, 2006.

[22] M. Rizk, D. Heller, R. Douguet, A. Baghdadi, and J.-P. Diguet, “Optimization of deep-learning detection of humans in marine environment on edge devices”, in *2022 29th IEEE International Conference on Electronics, Circuits and Systems*

(ICECS), 2022, pp. 1–4. DOI: https://doi.org/10.1109/ICECS202256217.2022.9970780.

[23] "System architecture | cloud TPU", Google Cloud. (), [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm (visited on 07/20/2023).

[24] S. Brown and J. Rose, "Architecture of fpgas and cplds: A tutorial", *IEEE Design and Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[25] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses", *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013. DOI: https://doi.org/10.1145/2436256.2436271.

[26] "Kria KV260 vision AI starter kit applications — kria KV260 2022.1 documentation". (), [Online]. Available: https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/index.html (visited on 07/27/2023).

[27] "Kria k26 SOM - xilinx wiki - confluence". (), [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1641152513/Kria+K26+SOM#KR260-Starter-Kit (visited on 07/29/2023).

[28] W. Knitter. "Accelerate your robotics design with the kria KR260 robotics starter kit", Hackster.io. (), [Online]. Available: https://www.hackster.io/news/accelerate-your-robotics-design-with-the-kria-kr260-robotics-starter-kit-89191a42080d (visited on 07/31/2023).

[29] Y. Guan, H. Liang, N. Xu, *et al.*, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates", in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159. DOI: https://doi.org/10.1109/FCCM.2017.25.

[30] V. Kathail, “Xilinx vitis unified software platform”, in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’20, Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 173–174, ISBN: 9781450370998. DOI: `https://doi.org/10.1145/3373087.3375887`.

[31] “Vitis software platform”, Xilinx. (), [Online]. Available: `https://www.xilinx.com/products/design-tools/vitis/vitis-%20platform.html` (visited on 08/01/2023).

[32] “Programming an FPGA: An introduction to how it works”, Xilinx. (), [Online]. Available: `https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html` (visited on 07/26/2023).

[33] “DPU IP details and system integration — vitis™ AI 3.0 documentation”. (), [Online]. Available: `https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-system-integration.html` (visited on 08/08/2023).

[34] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Dive into deep learning”, *arXiv preprint arXiv:2106.11342*, 2021. DOI: `https://doi.org/10.48550/arXiv.2106.11342`.

[35] Z. Wang, W. Yan, and T. Oates, “Time series classification from scratch with deep neural networks: A strong baseline”, in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 1578–1585. DOI: `https://doi.org/10.1109/IJCNN.2017.7966039`.

[36] S. Sharma, S. Sharma, and A. Athaiya, “Activation functions in neural networks”, *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[37] J. Han, D. Wang, Z. Li, N. Dey, and S. Fuqian, “An improved residual-network model-based conditional generative adversarial network plantar pressure image

classification: A comparison of normal, planus, and talipes equinovarus feet", Feb. 2021. DOI: https://doi.org/10.21203/rs.3.rs-262837/v1.

[38] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey", *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. DOI: https://doi.org/10.1109/JPROC.2017.2761740.

[39] D. Agiakatsikas, N. Foutris, A. Sari, *et al.*, "Evaluation of the xilinx deep learning processing unit under neutron irradiation", in *2021 21th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2021, pp. 1–4. DOI: https://doi.org/10.1109/RADECS53308.2021.9954522.

[40] "Introduction • DPUCZDX8g for zynq UltraScale+ MPSoCs product guide (PG338) AMD adaptive computing documentation portal". (), [Online]. Available: https://docs.xilinx.com/r/en-US/pg338-dpu/Introduction?tocId=3xsG16y_QFTWvAJKHbisEw (visited on 08/20/2023).

[41] S. Mahmood, S. Scharoba, J. Schorlemer, C. Schulz, M. Hübner, and M. Reichenbach, "Detecting improvised land-mines using deep neural networks on gpr image dataset targeting fpgas", in *2022 IEEE Nordic Circuits and Systems Conference (NorCAS)*, 2022, pp. 1–7. DOI: 10.1109/NorCAS57515.2022.9934735.

[42] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research", *IEEE Access*, vol. 8, pp. 85 714–85 728, 2020. DOI: https://doi.org/10.1109/ACCESS.2020.2991734.

[43] M. Satyanarayanan, "The emergence of edge computing", *Computer*, vol. 50, no. 1, pp. 30–39, 2017. DOI: https://doi.org/10.1109/MC.2017.9.

[44] J. Chen and X. Ran, "Deep learning with edge computing: A review", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019. DOI: https://doi.org/10.1109/JPROC.2019.2921977.

[45] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild”, *Science Robotics*, vol. 7, no. 66, eabm6074, 2022. DOI: `https://doi.org/10.1126/scirobotics.abm6074`.

[46] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS: a practical introduction to the Robot Operating System.* " O’Reilly Media, Inc.", 2015.

[47] S. M. Sánchez, F. Lecumberri, V. Sati, *et al.*, “Edge computing driven smart personal protective system deployed on nvidia jetson and integrated with ros”, in *Highlights in Practical Applications of Agents, Multi-Agent Systems, and Trustworthiness. The PAAMS Collection: International Workshops of PAAMS 2020, L’Aquila, Italy, October 7–9, 2020, Proceedings 18*, Springer, 2020, pp. 385–393.

[48] “Kria robotics stack (KRS) — KRS 1.0 documentation”. (), [Online]. Available: `https://xilinx.github.io/KRS/sphinx/build/html/index.html` (visited on 08/22/2023).

[49] “TurtleBot 4”, Clearpath Robotics. (), [Online]. Available: `https://clearpathrobotic` `com/turtlebot-4/` (visited on 08/13/2023).

[50] R. Febbo, B. Flood, J. Halloy, P. Lau, K. Wong, and A. Ayala, “Autonomous vehicle control using a deep neural network and jetson nano”, in *Practice and Experience in Advanced Research Computing*, 2020, pp. 333–338.

[51] “Interfaces • kria KV260 vision AI starter kit user guide (UG1089) • reader • AMD adaptive computing documentation portal”. (), [Online]. Available: `https://docs.xilinx.com/r/en-US/ug1089-kv260-starter-kit/Interfaces` (visited on 08/14/2023).

[52] turtlebot. “User interface PCBA · user manual”. (Feb. 8, 2022), [Online]. Available: `https://github.com/turtlebot4-user-manual/electrical/pcba.html` (visited on 08/14/2023).

[53] "DPU IP details and system integration — vitis™ AI 3.0 documentation". (), [Online]. Available: https://xilinx.github.io/Vitis-AI/3.0/html/docs/workflow-system-integration.html (visited on 08/08/2023).

[54] "Vitis AI quantizer flow • vitis AI user guide (UG1414) • reader • AMD adaptive computing documentation portal". (), [Online]. Available: https://docs.xilinx.com/r/1.3-English/ug1414-vitis-ai/Vitis-AI-Quantizer-Flow (visited on 08/15/2023).

[55] "Secondary boot device • kria KV260 vision AI starter kit user guide (UG1089) • reader • AMD adaptive computing documentation portal". (), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1089-kv260-starter-kit/Secondary-Boot-Device (visited on 08/15/2023).

[56] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures", ser. MICRO-46, Davis, California: Association for Computing Machinery, 2013, pp. 86–98, ISBN: 9781450326384. DOI: 10.1145/2540708.2540717.

[57] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks", *Digital Signal Processing*, vol. 73, pp. 1–15, 2018, ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2017.10.011.

[58] T. Ridnik, H. Lawen, A. Noy, E. Ben Baruch, G. Sharir, and I. Friedman, "Tresnet: High performance gpu-dedicated architecture", in *proceedings of the IEEE/CVF winter conference on applications of computer vision*, 2021, pp. 1400–1409.

[59] "Custom carrier card flow — kria™ SOM 2022.1 documentation". (), [Online]. Available: https://xilinx.github.io/kria-apps-docs/creating_applications/2022.1/build/html/docs/custom_cc_flow.html (visited on 08/16/2023).

[60] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it)”, in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14. DOI: `https://doi.org/10.1109/ISSCC.2014.6757323`.