

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Implementation of Convolutional Neural Networks on a Versal Device

Tiago David Sousa Ramos

Master in Electrical and Computer Engineering

Supervisor: João Paulo de Castro Canas Ferreira

July 31, 2023

Abstract

Autonomous driving has recently gained much attention with improved efficiency and safety. Object detection algorithms are vital to these systems. They provide the vehicle with a perception of its surroundings. However, these algorithms can be costly regarding computation and power demand. Because autonomous devices have small hardware resources and some constraints regarding energy usage, implementing ML models on edge devices is a problem. Therefore, FPGAs are gaining interest due to their hardware reconfigurability and low power consumption. Some frameworks like Vitis AI have been developed to accelerate these detection models on FPGA-like boards.

This research aimed to convert CNNs into hardware descriptions and evaluate their performance and energy efficiency. The PointPillars and Voxel R-CNN models were selected as representatives of state-of-the-art 3D object detection networks, and their suitability for deployment in resource-constrained embedded systems was assessed.

A Versal AI Core VCK190 board was used, and the Vitis AI framework was tested as the implementation tool. The PointPillars network was run, making use of the Vitis AI Library. Regarding the Voxel R-CNN model, only a part of this network was quantized, compiled, and accelerated. It was implemented with and without total DPU support.

We found that the DPU presents good values in terms of performance when compared to the GPU. However, there are some timing bottlenecks when transferring the data between the CPU and the DPU. Regarding power usage, the Versal device uses far less energy than the GPU.

The results of this work show that Vitis AI and the DPUs are powerful tools to aid the implementation of CNNs. Future works may explore problems encountered in this project, such as the unsupported sparse convolutions so that a full Voxel R-CNN model could be implemented on a Xilinx device and the optimization of the data transfer between the DPU and the CPU.

Acknowledgements

As a chapter of my life is closing, I want to thank some people I consider essential throughout my academic life and who helped me do this project.

First, I want to thank my supervisor, Professor João Canas Ferreira, for all our meetings and the advice that was always important for maintaining discipline and guiding me through the right path.

For all the people working in room I223, I appreciate the excellent environment and fellowship towards me since my first day there.

I am grateful for all the friends I made on this journey. We have been together since the first day and have endured the same struggles throughout these years.

Last, I thank my parents and my dearest Nicole for always believing in me and motivating me in the hard times. We shared a lot of memories, and a great part of my academic life, I owe it to her.

Tiago Ramos

This work is supported by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project n° 047264; Funding Reference: POCI-01-0247-FEDER-047264].

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Goals	2
1.4	Document structure	2
2	State of the Art	5
2.1	Convolutional Neural Networks	5
2.1.1	Convolutional layers	6
2.1.2	Pooling layers	7
2.1.3	Fully connected layers	8
2.1.4	Activation functions	8
2.2	LiDAR	10
2.3	3D Object detection models	11
2.3.1	Point cloud-based methods	11
2.3.2	Grid based methods	11
2.3.3	Point-Voxel based methods	12
2.4	Field-Programmable Gate Array	12
2.4.1	Versal architecture	13
2.5	Vitis AI framework	15
2.5.1	Vitis AI workflow	16
2.5.2	Deep Learning Processing Unit (DPU)	16
2.5.3	Model Zoo	17
2.5.4	Quantizer	18
2.5.5	Compiler	19
2.5.6	Profiler	19
2.5.7	Library	19
2.6	Summary	21
3	Design and Implementation	23
3.1	Setup and tools used	23
3.2	PointPillars acceleration	24
3.2.1	Implementation description	25
3.2.2	Inference on the board	26
3.3	Voxel R-CNN acceleration	27
3.3.1	Implementation setup	28
3.3.2	Implementation description	29
3.3.3	Inference on the board	32

3.3.4	Evaluation	34
4	Results and Evaluation	37
4.1	PointPillars	37
4.2	Voxel R-CNN	39
4.2.1	CPU and DPU implementation	41
4.2.2	Total DPU implementation	42
4.2.3	Power measurement	42
5	Conclusion and Future Work	45
A	DPUCVDX8G Resource Utilization	47
B	Voxel R-CNN Inspect Result	49
	References	51

List of Figures

2.1	Simple CNN structure.	6
2.2	Representation of a simple convolution.	7
2.3	Representation of max and average pooling operation.	7
2.4	Sigmoid function plot.	8
2.5	Tanh function plot.	9
2.6	ReLU function plot.	9
2.7	Leaky ReLU function plot.	9
2.8	An example of how a LiDAR system works.	10
2.9	Simple structure of a point based detection model.	11
2.10	Simple structure of a grid-based detection model.	12
2.11	Structure overview of the Versal Series.	13
2.12	Illustration of the AI engines composition.	14
2.13	Overview of Vitis AI integrated development environment.	15
2.14	DPU naming scheme.	17
2.15	DPUCVDX8G top-level block diagram.	17
2.16	Simple representation of the quantization process in an ANN.	18
2.17	Simple representation of the compilation process in an ANN.	19
2.18	Diagram of Vitis AI programming APIs.	20
2.19	Block Diagram of Vitis AI Library.	22
3.1	Versal VCK190 setup.	24
3.2	Overview of the PointPillars network.	25
3.3	Overview of the model zoo PointPillars directory.	26
3.4	Overview of the Voxel R-CNN structure.	27
3.5	Top view of OpenPCDet working directory inside Vitis AI's environment.	28
4.1	Example of BEV detection of PointPillars using Vitis AI Library samples.	37
4.2	Example of 3D detection of PointPillars using Vitis AI Library samples.	38
4.3	Cropped image from inspecting total DPU implementation of the 2D backbone of Voxel R-CNN.	41
4.4	Power usage of GPU during Voxel R-CNN inference.	43
4.5	Power usage of Versal ACAP during inference.	43
B.1	Image result from inspecting the 2D backbone from Voxel R-CNN with total DPU not supported.	49
B.2	Image result from inspecting the 2D backbone from Voxel R-CNN with total DPU supported.	50

List of Tables

2.1	Main resources and capabilities of each Versal series.	14
3.1	Main characteristics of Versal AI Core VC1902.	23
4.1	Profiling summary of DPU while running a PointPillars performance test.	38
4.2	Profiling summary of CPU functions while running a PointPillars performance test.	38
4.3	Accuracy of Voxel R-CNN using float model of 2D backbone part.	40
4.4	Accuracy of Voxel R-CNN using quantized model of 2D backbone part.	40
4.5	Profiling summary of CPU functions regarding the partial DPU compatible model.	41
4.6	Profiling summary of DPU resources regarding the partial DPU compatible model.	41
4.7	Profiling summary of DPU resources for the total DPU compatible model.	42
4.8	Profiling summary of CPU functions for the total DPU compatible model.	42
A.1	DPUCVDX8G resource utilization depending on architecture used.	47
A.2	Peak theoretical performance of each DPUCVDX8G architecture.	48

Abbreviations and Symbols

ACAP	Adaptive Compute Acceleration Platform
AD	Autonomous Driving
AI	Artificial Intelligence
ANN	Artificial Neural Network
AP	Average Precision
API	Application Programming Interface
ARM	Advanced RISC Machine
AV	Autonomous Vehicle
BEV	Bird's-Eye-View
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Compute Unite
DL	Deep Learning
DPU	Deep Learning Processing Unit
DSP	Digital Signal Processing
FC	Fully Connected
FF	Flip-Flop
FLOP	Floating-Point Operation
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GB	Gigabyte
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HLS	High-Level Synthesis
LiDAR	Light Detection and Ranging
LUT	Look Up Table
MAC	Multiply–Accumulate
ML	Machine Learning
NOC	Network-On-Chip
OP	Operator
PE	Processing Element
PL	Programmable Logic
PLB	Programmable Logic Block
RAM	Random Access Memory
RPN	Region Proposal Network
SIMD	Single Instruction Multiple Data
SOC	System-On-a-Chip
VART	Vitis AI Runtime
VLIW	Very Long Instruction Word
XIR	Xilinx Intermediate Representation
XRT	Xilinx Runtime Library

Chapter 1

Introduction

In this chapter, the context of this work will be summarized, as well as the main goals, motivation, and document structure. It serves as a starting point for the following chapters.

1.1 Context

Autonomous driving (AD) is entering a pre-industrialization phase, where it has recently reached a good level of development. It offers a safer and more comfortable driving experience, free from human error. To achieve this, all system levels must be robust and trustworthy [1]. The first challenge such an algorithm faces is localizing the various objects in the environment to perform their respective tasks.

LiDAR (Light Detection and Ranging) sensors are the most popular and essential ones in this regard. By sending laser light to their surrounding and receiving the same reflected waves, they can calculate the object's distance, making an accurate and dense 3D point cloud of the surrounding environment [2]. This active sensor has gained more interest over passive sensors like RADAR and cameras. They are robust in various conditions, less affected by adverse weather conditions, and provide high-resolution, three-dimensional data that accurately represents the environment. However, they can be more expensive than other sensors, making them difficult to adopt on autonomous systems. Their data output can be vast and brutal to process for real-time applications, so efficient algorithms and hardware acceleration techniques must be implemented.

For this to all work together, the autonomous vehicle needs an object detection system that takes the data from the vehicle sensors and transforms it into semantic information. Although 2D object detection methods have been used for several works, 3D methods present more information and accuracy. Therefore, much research has been done in recent years, and many models have been developed with Convolutional Neural Networks (CNNs) widely adopted [3].

Deep neural networks are highly precise and very effective. However, this comes at a high computational cost and power consumption. Using these methods in portable devices and, in this case, autonomous devices with acceptable use of hardware resources can become a problem. For this reason, FPGAs have been gaining more interest over the past years due to their high efficiency

and reconfigurability [4]. They can be used to accelerate these models and obtain some satisfactory results with very few consequences. Some frameworks have been developed to implement and accelerate these CNNs on FPGA, such as Vitis™ AI [5] and FINN [6].

1.2 Motivation

It is just a matter of time until the roads are populated with autonomous vehicles, which will be essential for the transportation industry and society as a whole. According to this study [7], road traffic in America has made people travel 8.8 billion extra hours, resulting in an excess consumption of 33.3 billion liters of fuel, causing an estimated cost of 154 billion euros. Autonomous driving can help to improve this, boosting traffic efficiency and reducing carbon emissions, making the transportation system better, safer, and more eco-friendly.

One of the main challenges in object detection for autonomous driving is to balance the trade-off between the cost and power demand of processing devices and their processing capabilities. This is a significant opportunity for edge and FPGA-based hybrid SoC to contribute to this problem [8]. However, there are some challenges they need to face, like processing vast amounts of data in real-time, having strict power constraints, and being safe and secure against attacks [9].

Since these systems at the edge are far from perfect, some techniques can be used to improve the efficiency of model deployment, like pruning (removing redundant weights) and quantization (converting the baseline 32-bit parameters to fewer bits) [10]. Vitis AI makes implementing these models in Xilinx hardware possible, from being compatible with most popular frameworks like Pytorch, Tensorflow, etc., and running them in Xilinx's DPU (Deep Processing Unit) accelerator.

1.3 Goals

This project was proposed from a partnership between Bosch and Porto University called Theia (THEIA – Automated Perception Driving). It aims to improve the overall state of autonomous vehicles' capabilities with several sub-projects with contributions from different areas of knowledge. This work was in the subproject where the main goal was to "implement and validate perception algorithms based on neural networks in reconfigurable hardware to improve computational and energy efficiency." It is intended to answer the following questions:

- What are the trade-offs between accuracy, real-time performance, and resource utilization when accelerating deep neural networks on an FPGA-like device?
- What are the main differences in accuracy and performance when the inference process is done on the GPU or hardware?

1.4 Document structure

This dissertation is organized into five chapters:

- Chapter 1 provides an overview of the research topic and its motivation. This chapter also states the main objectives of this work.
- Chapter 2 comprehensively reviews the existing knowledge and research in autonomous driving and object detection acceleration. It covers the latest advancements, methodologies, and technologies related to object detection in autonomous driving systems. This chapter also discusses the strengths and limitations of current approaches, identifies any research gaps or challenges, and provides a foundation for the proposed methodology.
- Chapter 3 explains the approach and methodology employed. It details the design and implementation of Voxel R-CNN and PointPillars, including the architecture, training process, and optimization techniques. Additionally, this chapter outlines the integration of the neural networks on Versal VCK190 and explains the software and hardware tools utilized. It concludes with a summary of the methodology employed.
- Chapter 4 presents and analyzes the findings of the research. It showcases the experimental results obtained from the acceleration of the neural networks. This chapter includes a detailed evaluation of the performance of the implemented networks, such as accuracy, precision, recall, and computational efficiency. Power measurements are also included.
- Chapter 5 summarizes the essential findings and contributions of this research. It concisely summarizes the primary outcomes and insights obtained from this work. This chapter also addresses some limitations and challenges encountered during the realization of this project and suggests potential areas for improvement and future research directions.

Chapter 2

State of the Art

This chapter presents state-of-the-art technologies about 3D object detection for AD. It explores the importance of CNNs and their crucial role in this scene. LiDAR sensors are presented together with their key part in capturing 3D information about the surroundings. The main structures for 3D object detection models are also summarized. Furthermore, FPGA technology potential is discussed in addition to Vitis AI framework capabilities for deploying ML models.

2.1 Convolutional Neural Networks

Nowadays, most artificial intelligence systems are based on "Machine Learning" algorithms. These techniques are characterized by their ability to "learn" through analyzing training data, thus becoming more robust and less prone to errors [11]. They aim to automate cognitive tasks like recognizing objects or translating a conversation. These techniques are applicable, especially in high-level tasks such as classifications and regressions. Reliable solutions can be obtained by learning from previous computations and extracting data from giant databases. This is why these algorithms are used in various tasks, including image recognition. In this sense, artificial neural networks are exciting, having a very flexible structure that allows them to be used in multiple contexts.

As the name implies, an artificial neural network can be seen as a mathematical representation of the human brain's workings. To do this, a set of interconnected neurons represent it. In a simplified way, each neuron can be defined as a linear combination between its input and its connection weight (w). This value is added to a bias value and then passed through an activation function, obtaining the output value of the neuron [12]. During the training phase, values of the synapse weights are adjusted so that the neural network's output matches the desired value.

Activation functions introduce non-linearity to neural networks, enabling them to model complex relationships in data. They transform neuron outputs, control gradient propagation, enhance representation power, and influence stability and convergence during training. The choice of activation function depends on the task and desired network behavior.

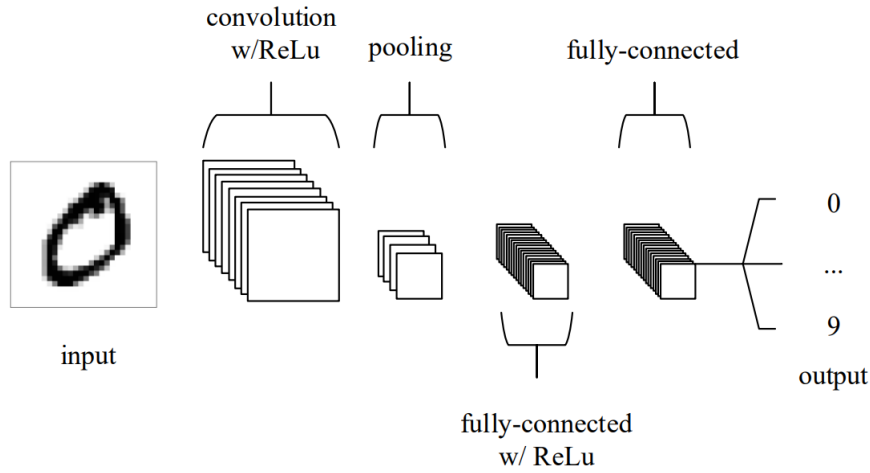


Figure 2.1: Simple CNN structure [13].

There are several types of neural networks with different structures, and the most used by several authors [14] concerning the topic of 3D object detection are convolutional networks. Figure 2.1 represents a simple CNN. This type of ANN has three types of layers: Convolutional, pooling, and fully connected. A CNN architecture is created once these layers are stacked.

2.1.1 Convolutional layers

As its name suggests, the convolutional layer is critical to the operation of CNNs. This layer plays a vital role in extracting meaningful features from input data, allowing the network to learn and understand complex visual patterns. To create a feature map for the supplied image, it essentially convolves or multiplies, using a given kernel, the pixel matrix created for the given image or object.

This process is simplified in figure 2.2. Considering a 2D image represented as a matrix of pixel values (in the middle), a small kernel (on the left side) is placed in the upper left corner of the image. Element-wise multiplication is performed between the kernel and the overlapping region of the image, and the results are added to get a single value. The kernel is then moved to the right by one pixel, repeating this element-wise process. The output of this process is a new matrix (on the right side) called the feature map or activation map.

The real benefit of an activation map is that it reduces the quantity of data that needs to be processed while storing all the distinctive characteristics of an image. A feature detector (kernel) is the matrix with which the data is convolved. By applying multiple filters, convolutional layers can detect different features at different locations, effectively capturing various patterns and edges within the input data [16]. Stacking multiple convolutional layers allows the network to extract increasingly complex and abstract features, allowing for higher-level representation learning.

Additional parameters that can be changed in convolutional layers are stride and padding. The stride determines the step size at which the kernel moves across the input. A higher stride value

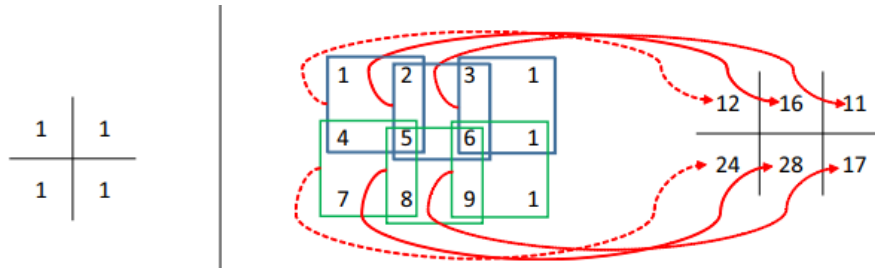


Figure 2.2: Representation of a simple convolution [15].

results in smaller output feature maps, diminishing the spatial dimension. In contrast, padding includes additional border pixels in the input to maintain spatial dimensions in the output feature maps. Stride and padding govern information loss and spatial dimensions during convolution, allowing for control over the granularity and resolution of learned features [17].

2.1.2 Pooling layers

The main objective of the pooling layer is to reduce the dimension, the number of parameters, and the complexity of the model [13]. The most commonly used pooling techniques are max pooling and average pooling [18]. Max pooling selects the maximum value within a pooling window, while average pooling calculates the average value. These operations, represented in figure 2.3, enable the pooling layers to focus on the most prominent features within local regions of the feature maps.

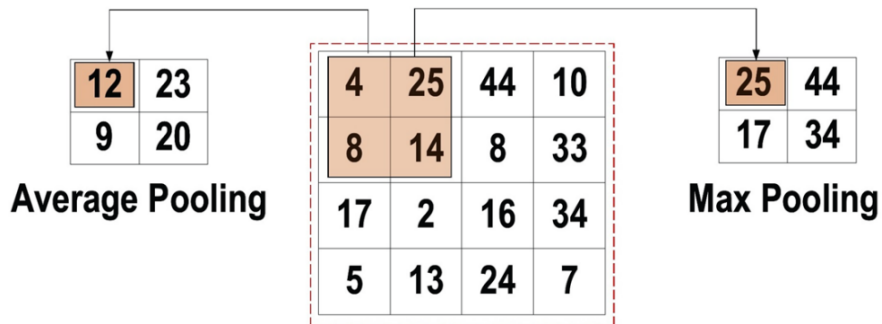


Figure 2.3: Representation of max and average pooling operation [19].

Pooling layers also contribute to regularization in CNNs. By aggregating local features, pooling reduces overfitting by promoting generalization. Spatial aggregation helps the network focus on the dominant features within a region rather than relying too heavily on localized details. This regularization effect enhances the robustness of the network in handling variations and noise in the point cloud data, ultimately leading to improved object detection performance.

2.1.3 Fully connected layers

Fully connected (FC) layers form an integral component of convolutional neural networks (CNNs) and play a crucial role in object detection in autonomous driving. They sit in the back of CNN structures and are in charge of learning intricate feature combinations and capturing global relationships. In the FC layer, neurons are grouped similarly to how they are in a conventional neural network. As a result, any node in a fully connected layer is directly connected to every node in the layer above it and the one below it [17]. The main disadvantage of the FC layer is that it has many parameters associated with its connectivity pattern, leading to increased computational complexity.

2.1.4 Activation functions

In neural networks, activation functions are essential because they introduce non-linearity and allow the network to simulate complex connections between input and output [20]. They choose which information should be sent to the next neuron, much like the neuron model in the brain. If linear activation functions are used, or even no activation functions, it makes the output consistently a linear combination of the inputs, making hidden layers useless [21]. These are some of the most used activation functions:

- Sigmoid has an overall S-shape and the following equation 2.1 and graph 2.4:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.1)$$

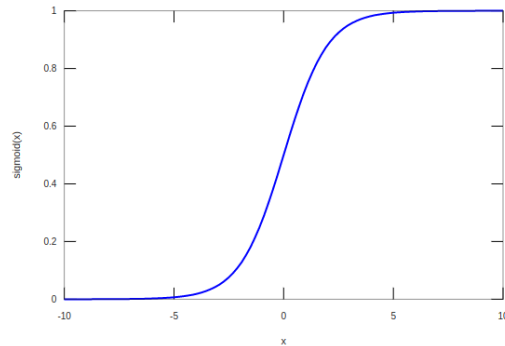


Figure 2.4: Sigmoid function plot.

It is possible to utilize the sigmoid function to solve binary classification problems since it maps input to (0, 1).

- tanh maps a real number to (-1, 1), and since it has a mean value of 0, it can achieve some degree of normalization. It is represented in equation 2.2 and figure 2.5.

$$f(x) = \frac{\exp^x - \exp^{-x}}{\exp^x + \exp^{-x}} \quad (2.2)$$

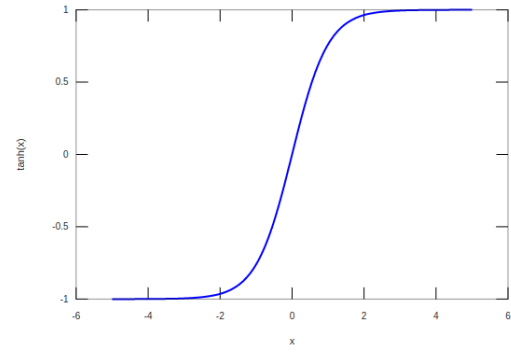


Figure 2.5: Tanh function plot.

- ReLU has the major advantage of accelerating learning and is one of the most used in CNNs. However, there are some drawbacks since it can make previous neurons inactivate. This is because the gradient of ReLU is 0 when x is 0, so the back-propagated error will be null. The following equation 2.3 and figure 2.6 represents it:

$$f(x) = \max(0, x) \quad (2.3)$$

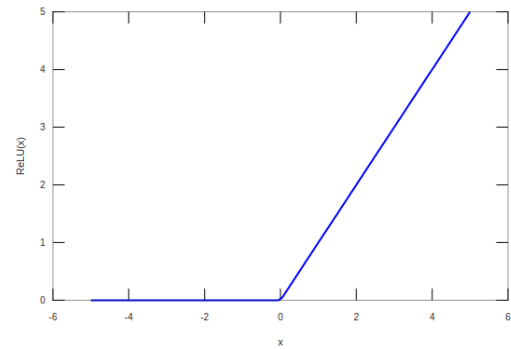


Figure 2.6: ReLU function plot.

- Leaky ReLU is a version of ReLU that aims to reduce neuron inactivation when back propagating. The following equation 2.4 and figure 2.7 can represent it:

$$f(x) = \begin{cases} x, & \text{if } x > 0. \\ mx, & \text{otherwise.} \end{cases} \quad (2.4)$$

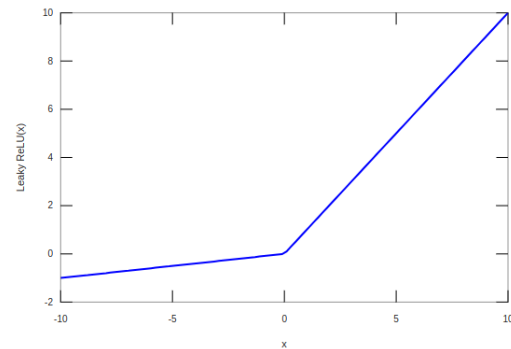


Figure 2.7: Leaky ReLU function plot.

The number m usually is minimal to make a small slope.

2.2 LiDAR

AD relies heavily on LiDAR technology, which offers precise and in-depth 3D data on the surroundings. The importance of LiDAR in AD systems will be discussed in detail in this section, along with the datasets currently being utilized for testing and assessing LiDAR-based object detection algorithms.

Typically, a LiDAR scans its surroundings using one or more laser beams. These are produced by an amplitude-modulated laser diode that emits at near-infrared wavelength. Then after the reflected laser returns to the sensor, it calculates the range of the respective object based on this difference [22]. Following this, a high-resolution 3D point cloud is generated, providing spatial details of the environment. A simple representation of this process is in figure 2.8.

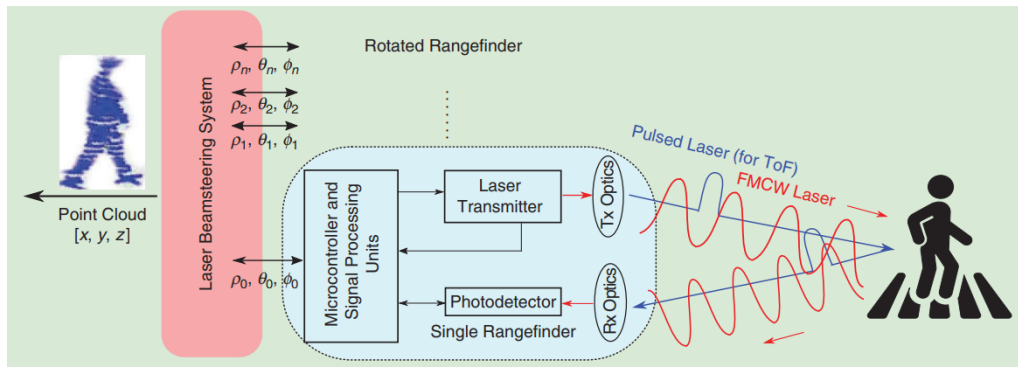


Figure 2.8: An example of how a LiDAR system works [22].

Several datasets have arisen in recent years to assess and benchmark the performance of LiDAR-based object detection systems. These datasets enable researchers to develop and test their algorithms. Various realistic scenarios are available in datasets like KITTI, Waymo Open Dataset, and nuScenes, with varied complexity and annotation quality. These datasets have become priceless tools for creating and evaluating LiDAR-based object recognition algorithms. They are composed of:

- Many high-resolution images providing visual information about the surroundings.
- 3D point clouds
- Calibration parameters for the sensors used
- Annotations that specify the ground truth information of the detected objects.

In conclusion, LiDAR technology serves as a cornerstone in autonomous driving. The number of these sensors present in a vehicle can vary. Some configurations using a single LiDAR and others using multiple have been used [23]. The availability of diverse and well-annotated LiDAR datasets further facilitates the development and evaluation of state-of-the-art algorithms. Continued advancements in LiDAR technology and dataset availability contribute to the progress of autonomous driving systems, bringing us closer to realizing safe and efficient self-driving vehicles.

2.3 3D Object detection models

3D object detection plays a pivotal role in ensuring safe and efficient navigation. It seeks to predict important objects' positions, dimensions, and types. Unlike 2D methods, which have uniformly distributed pixels, range images are sparse and asymmetric 3D representations. Therefore, using traditional CNNs directly to point clouds may not be the best action. Adding to this, there needs to be an inference process capable of use in real-time. As a result, the research community continues to face the issue of creating a model that efficiently manages point cloud data [24].

This section will explain the main approaches for LiDAR-based 3D object detection models, which essentially can be divided into how the data is represented: point-based, grid-based, or point-voxel based.

2.3.1 Point cloud-based methods

As the name indicates, this method processes raw point cloud data without voxelization or grid-based representations, as represented in figure 2.9. These models leverage the inherent geometry and spatial relationships of the point cloud to detect objects in a 3D space. They first take the point cloud through a backbone network that continues downsampling the points and extracting low-dimensional features [25], which are aggregated to obtain high-dimensional features. 3D bounding box predictions are then made in the final stage. Some works based on this method are PointNet [26], PointRCNN [27], and IPOD [28].

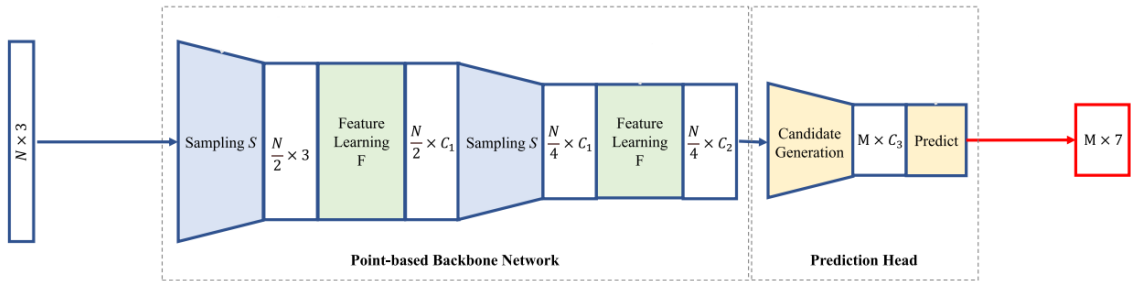


Figure 2.9: Simple structure of a point based detection model [24].

Point-based models also face certain limitations and challenges. They are exceptionally computationally intensive since they act separately on each point. Processing a lot of points might make computing more complex, need more memory, and be very time-consuming [29].

2.3.2 Grid based methods

Unlike Point cloud-based methods that take raw data as input, these involve converting raw point clouds into structured grids like voxels and pillars. This saves memory resources since this dramatically reduces the point cloud's dimension. Then, features are extracted from the points enclosed in each structure instead of each point individually. Examples of this architecture, as represented in figure 2.10, are PointPillars [30], Voxel R-CNN [31], CenterPoint [32], and SECOND [33].

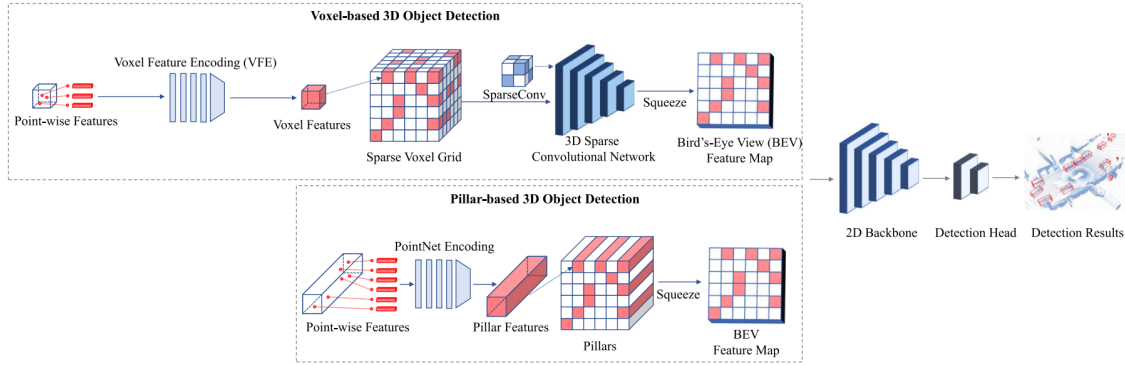


Figure 2.10: Simple structure of a grid-based detection model (adapted from [24]).

Although grid-based models offer advantages in capturing spatial relationships, they have drawbacks. When using voxels and pillars, which are like voxels but with unlimited vertical size, most will be empty because of the point cloud's sparse nature. Therefore, some strategies have to be used to extract features only from the non-empty voxels. Another problem is the grid size since it can significantly impact memory consumption.

2.3.3 Point-Voxel based methods

Point-Voxel-based models have emerged as a hybrid approach that combines the strengths of point-based and voxel-based techniques for 3D object detection. Extracting features from point clouds and voxel grids allows for a comprehensive scene representation. These can be divided into two frameworks, single-stage, and double-stage detection [24].

While this method represents a promising direction with better accuracy, they still have a significant time problem when fusing voxel and point features. Some examples of this architecture are PV-RCNN [34], SA-SSD [35], and STD [36].

2.4 Field-Programmable Gate Array

While state-of-the-art deep learning (DL) models have achieved remarkable performance in object detection, their computational demands pose significant challenges for real-time applications like the ones for AD. This is where the integration of Field-Programmable Gate Arrays (FPGAs) becomes crucial. FPGAs offer unique advantages in terms of parallel processing, flexibility, low power consumption, and hardware customization, making them well-suited for accelerating complex neural network computations. This section overviews FPGAs, their architecture, design flow, and their application in accelerating neural networks.

FPGAs comprise programmable logic blocks (PLBs), configurable interconnects, and I/O interfaces. Flip-flops, lookup tables (LUTs), and specialized hardware blocks are among the resources found in PLBs that make it possible to create customized digital circuits [37]. The configurable interconnects allow for flexible routing of signals, facilitating complex interconnections

within the FPGA.

The hardware description process for FPGAs involves employing hardware description languages (HDL) like VHDL or Verilog, synthesis, placement, and routing. There is also the possibility of using high-level synthesis (HLS) to translate C code to HDL. Designers can specify the behavior and structure of their digital circuits using HDL. The hardware description is transformed into a gate-level representation during the synthesis phase, which is then inserted and routed on the FPGA to establish the physical connections [38].

FPGA-based neural network accelerators leverage specialized architectures using pipelined designs to exploit parallelism and improve computational efficiency. These accelerators can significantly speed up neural network inference tasks [39], making them suitable for applications with real-time or low-latency requirements.

2.4.1 Versal architecture

The Versal architecture is the first Adaptive Compute Acceleration Platform (ACAP) device developed by Xilinx. It offers a highly integrated and adaptable system-on-a-chip (SoC) platform for various applications. Besides having conventional FPGA logic fabric, it offers a processing system and intelligent engines hardened in the board, making it unique. Connecting all this is a hardened network-on-chip (NoC) [40]. Figure 2.11 shows an overview of this structure.

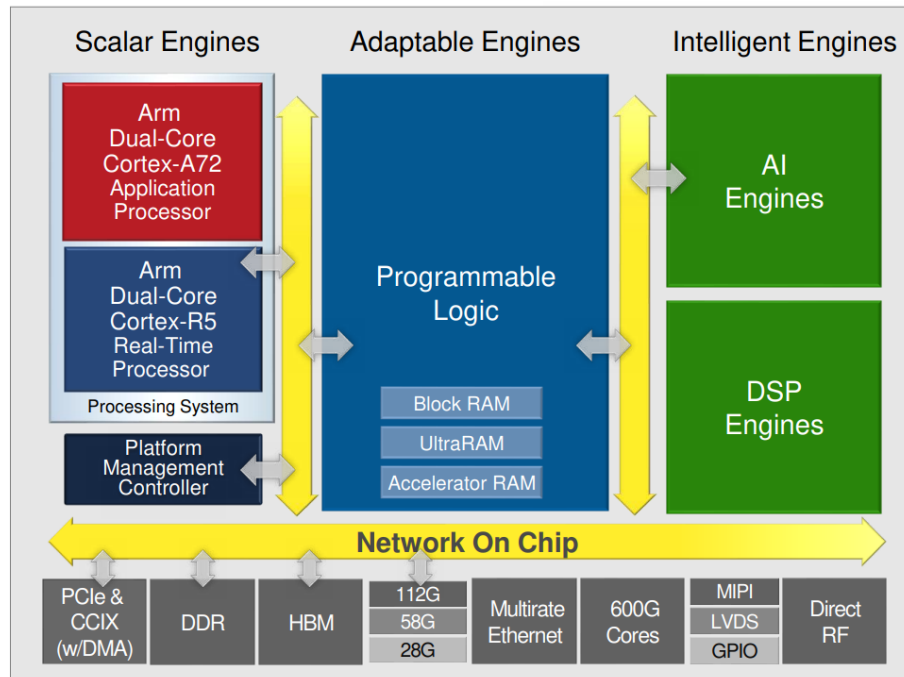


Figure 2.11: Structure overview of the Versal Series [41].

The scalar engines have a dual-core Arm® Cortex-A72 and a dual-core Arm® Cortex-R5F. This programmable software system is optimal for scalar and sequential processing.

The Adaptable Engines are formed of next-generation fast programmable logic and local memory cells. So this structure can support previous designs and newer customizations, being excellent at flexible parallel computations [42].

Regarding intelligent engines, they can be divided into two blocks: artificial intelligence (AI) and digital signal processing (DSP) engines. AI engines were developed because of the increase in ML applications that demand high parallel computing volume. This engine, shown in figure 2.12, is composed of AI tiles that have a single instruction multiple data (SIMD) and very long instruction word (VLIW) processors as well as dedicated memory [43]. Fixed point, floating point, and sophisticated multiply-accumulate (MAC) operations are all handled by DSP engines [44].

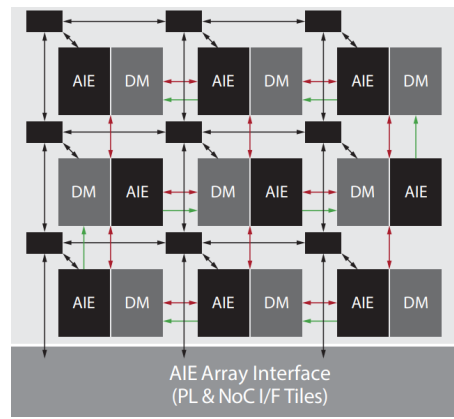


Figure 2.12: Illustration of the AI engines composition [43].

Different versions of the Versal Series have been released with minor differences that target various applications. In table 2.1 are some resources regarding each series. The Edge series focuses on the relationship between AI performance and power efficiency for real-time systems and meets essential security and safety standards. The Versal Core series has a significantly improved AI inference acceleration using AI engines. Prime is the mid-range of this series with a broad range of services. The Premium series offers ground-breaking heterogeneous integration. Lastly, the HBM (High Bandwidth Memory) series provides high memory capabilities with high capacity and memory bandwidth [45].

Table 2.1: Main resources and capabilities of each Versal series (adapted from [46]).

Resources	AI Edge Series	AI Core Series	Prime Series	Premium Series	HMB Series
System Logic Cells (k)	44-1139	540-1968	329-2233	933-7352	3837-5631
DSP Engines	90-1312	928-1968	464-3984	1140-14352	7392-10848
AI Engines	8-304	128-400	N/A	0-472	N/A
I/O	114-530	478-770	316-770	132-780	780

This project will utilize a VCK190 board from AI Core Series.

2.5 Vitis AI framework

HDL and HLS tools enable designers to express their designs at different levels of abstraction. FPGA-specific libraries and frameworks like Xilinx Vitis AI offer pre-optimized functions and APIs for efficient neural network acceleration.

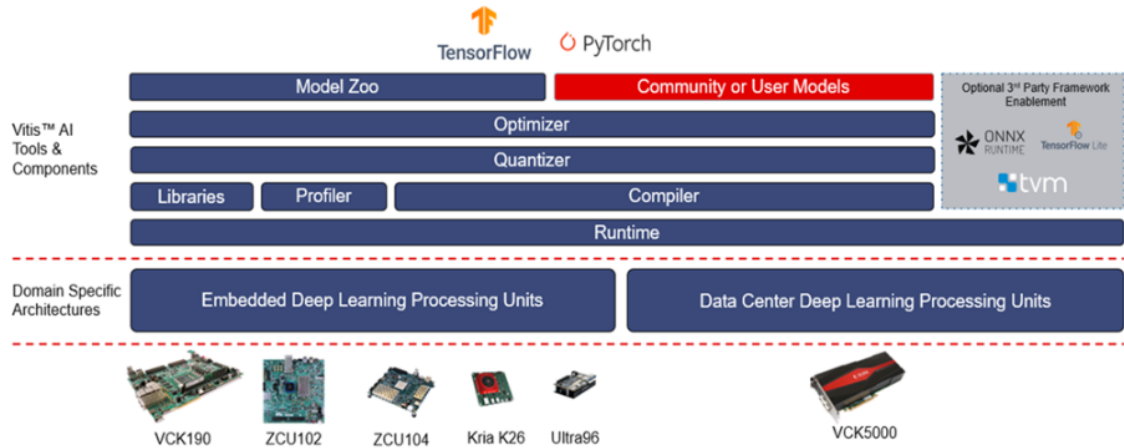


Figure 2.13: Overview of Vitis AI integrated development environment [47].

The Vitis AI workflow, as represented in figure 2.13, simplifies the development and deployment of AI models on FPGAs, using a specialized processor (Deep Learning Processing Unit - DPU) on the FPGA [48]. Developers can leverage the Vitis AI quantizer from model development to quantize and optimize their models, reducing their size while maintaining acceptable accuracy levels. Later it is possible to compile the model and deploy it on the target board. The Vitis AI Runtime provides an execution environment that enables efficient inference on FPGA devices. This end-to-end workflow streamlines the deployment process and ensures seamless integration of AI models with FPGA platforms.

The main components of the Vitis AI platform are:

- **DPU** is a hardware IP (Intellectual Property) block designed specifically for accelerating CNN tasks on Xilinx FPGAs. An IP block is a pre-designed and pre-characterized reusable unit that can be integrated into electronic systems.
- The **Model Zoo** is a repository with multiple DL model examples ready to be deployed.
- **Model Inspector** makes it possible to analyze the model before deployment on the board.
- **Optimizer** is a tool that makes it possible to prune a neural network but is commercially licensed.
- **Quantizer** can quantize and calibrate the model.
- **Compiler** compiles the quantized model.

- **VART** or Vitis AI Runtime is a unified set of APIs to make the deployment on embedded devices quick and efficient.
- **Profiler** can analyze the performance and efficiency of a model running on the PDU.
- **Library** is a collection of high-level APIs and packages designed for effective AI inference using DPUs.

The following sections will go more in-depth about the most critical components.

2.5.1 Vitis AI workflow

Vitis AI has a set of multiple tools, models, and libraries that lets the developer choose different paths accordingly to his objectives [47]:

- **ML and Data Science** - Popular frameworks like PyTorch and Tensorflow are supported, making evaluating ML models accessible.
- **Software Development for Embedded Devices** - Creating custom software for the embedded CPU is possible.
- **Hardware and IP Development** - Developing PL IP blocks for the hardware platform, PL kernel development, functional simulation, and Vivado® for timing, resource utilization, and power.
- **System Validation** - Test the model's performance, timing, and resource usage.

The theme of this dissertation fits in the first and last topics. It is an ML model that needs to be tested and evaluated for all parameters like performance, time, and resource usage.

2.5.2 Deep Learning Processing Unit (DPU)

The DPU is a specialized hardware IP block explicitly designed to accelerate DL inference on FPGAs and is the main component of Vitis AI. At its core, the DPU comprises processing elements (PEs) optimized for matrix multiplication and convolution operations, which are fundamental building blocks of deep neural networks. Each DPU has its unique set of instructions, so it executes the compiled instructions in the *xmodel* file generated by the compiler. This design does not require the construction of a new hardware platform or loading a new bitstream whenever the network is changed [49]. At a very high level, it is possible to say that Vitis AI converts an AI application to a series of DPU instructions.

The naming of the DPU follows the scheme illustrated in figure 2.14.

For the Versal series, the DPUCVDX8G is the general processing engine. This DPU comprises PL and AI Engines that can be configured [50]. Xilinx provides the TRD files, which can be later used to configure different parameters like the number of batch handlers, the number of CUs (Compute Unite), and the frequency of the PL. A top-level diagram is in figure 2.15. Depending

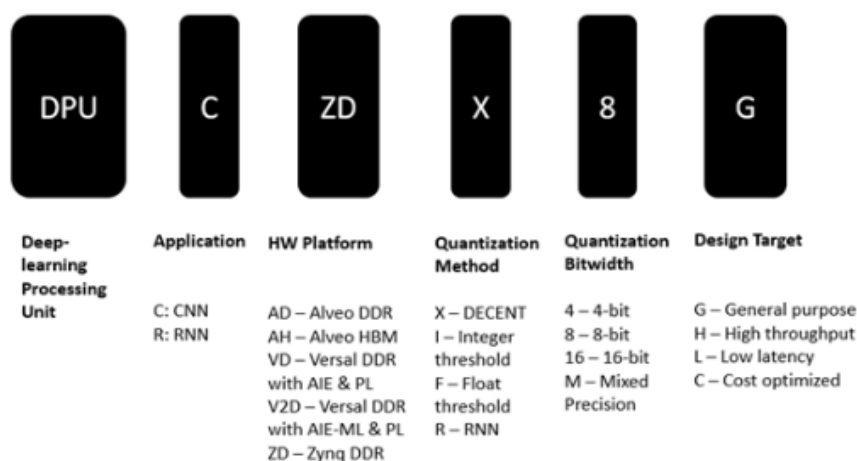


Figure 2.14: DPU naming scheme [47].

on the architecture of the DPU, there is different resource utilization and peak performance. AI engine cores, LUTs, FFs, and memory usage vary from architecture. A table with this data is in appendix A.

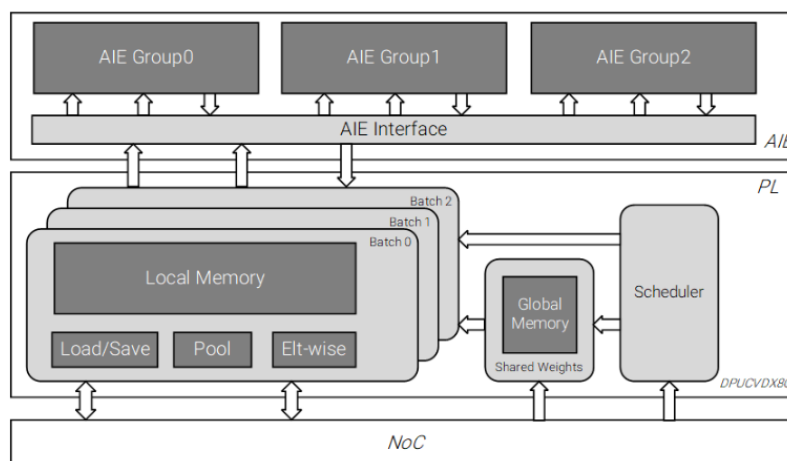


Figure 2.15: DPUCVDX8G top-level block diagram [50].

2.5.3 Model Zoo

In the Model Zoo of Vitis AI, users gain access to a comprehensive collection of pre-trained deep-learning models covering a wide range of tasks and application domains. To facilitate AI development and deployment, the Model Zoo offers diverse models for image classification, object detection, semantic segmentation, and more [47].

Each model in the Model Zoo is accompanied by detailed descriptions, providing insights into their architectural design, key features, and performance characteristics. These models undergo rigorous evaluation and benchmarking, with evaluation metrics encompassing factors like

accuracy, inference speed, and memory utilization.

Researchers and developers can leverage the models from the Model Zoo for their projects, deploying them in the Vitis AI framework or integrating them into specific hardware platforms. The Model Zoo is a dynamic repository, constantly evolving with community contributions and updates. Through the Model Zoo, Vitis AI empowers users with a valuable resource to accelerate their AI development journey and explore the full potential of deep learning.

2.5.4 Quantizer

The Quantizer tool offered by Vitis AI enables efficient and accurate quantization of deep learning models for deployment on FPGA platforms. Quantization is a technique that reduces the precision of model parameters and activations, allowing for lower bit-width representations, like represented in figure 2.16. By employing the Quantizer, developers can convert higher-precision models, such as those trained with 32-bit floating-point precision, into lower-precision representations like the 8-bit fixed-point format. This reduction in precision brings several benefits, including reduced memory footprint, faster inference times, and improved energy efficiency, with an impact on accuracy commonly lower than 1% [51].

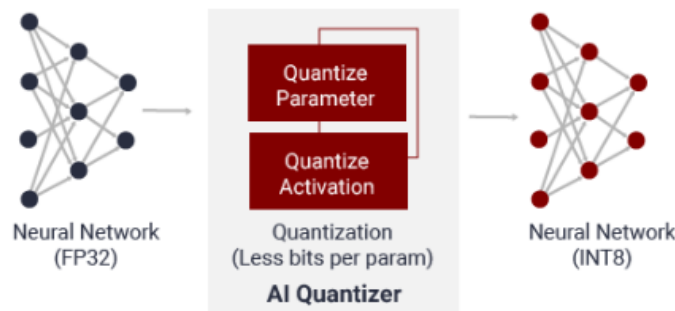


Figure 2.16: Simple representation of the quantization process in an ANN [49].

Before the quantization process, inspecting the model using the Vitis AI Inspector is possible. This tool was released in version v3.0 of Vitis AI and is of good value for developers. With this, we can check if Vitis AI supports all the layers and parameters in a CNN and if they will run on the DPU or the CPU. The main objective is to have all, or the maximum number, of operators on the DPU.

After that, the Quantizer has to do a calibration step. It leverages a subset of the dataset (100-1000 images) used to perform inference to calibrate the activations. This subset does not need to be labeled since no backpropagation will happen.

Following calibration, the model can be effectively quantized into a compiled instructions model called "deploy_model.pb" for TensorFlow and "model_name.xmodel" for Pytorch.

2.5.5 Compiler

As the name suggests, Vitis AI Compiler aims to compile the model generated from the Quantizer to a specific DPU series, as represented in figure 2.17. It transforms the quantized neural network into a list of instructions readable by the DPU. In this process, the Compiler tries to fuse layers and perform some optimizations on the model. An important note is that this process is done to a specific DPU architecture, so other DPUs can not execute the compiled instruction set.

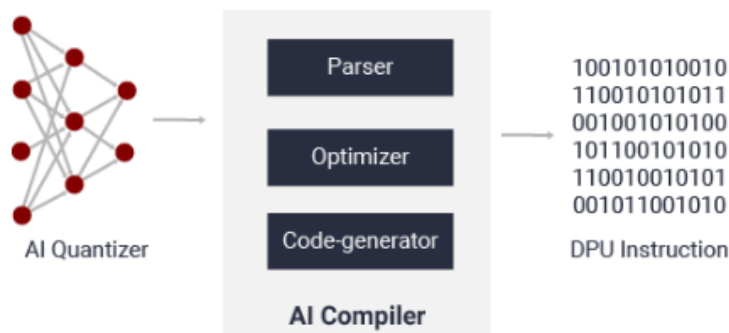


Figure 2.17: Simple representation of the compilation process in an ANN [49].

The compiled *xmodel* can be represented as a Xilinx Intermediate Representation (XIR). This is a graph-based representation of the model to facilitate the deployment of the DPU. When compiling the model, which can use different frameworks, it is first transformed into the XIR format, presenting a unified representation of the model. Several subgraphs are generated depending on the support of the DPU.

2.5.6 Profiler

The Vitis AI Profiler is a powerful tool to analyze and optimize the performance of the running models. By leveraging the Profiler, developers can gain valuable insights into the execution characteristics of their models, enabling them to identify performance bottlenecks, optimize resource utilization, and fine-tune their implementations for maximum efficiency. It is a unique tool that doesn't need to edit any code or recompile the program. It offers comprehensive profiling information, including latency, throughput, memory usage, and resource utilization for different sub-graphs of the running model. Developers can use these metrics to assess their model's efficiency and identify improvement areas, specifically, the time of execution between the CPU and the DPU.

2.5.7 Library

The Vitis AI Library [52] is a comprehensive collection of optimized software functions and libraries provided by Xilinx. It offers a range of pre-built functions specifically designed to accelerate deep learning workloads on FPGA devices. By leveraging the Vitis AI Library, developers can

streamline their development process, improve performance, and speed up their AI application development. It provides a high-level interface that abstracts the underlying FPGA implementation details, allowing developers to focus on the AI model and application logic.

It is built on Xilinx Runtime Library (XRT) and VART and can be split into four main parts, as shown in figure 2.19:

- **Base libraries** give the developer a fundamental programming interface with the "cpu_task" and "dpu_task". These libraries provide an interface to DPU and CPU operations, respectively. To implement post-processing routines, the "xnnpp" library has several functions optimized for each model.
- **Model libraries** aggregates a range of open-source deployment code for various networks using the most popular frameworks like PyTorch and TensorFlow. These code and header files use the foundational Base Libraries to enable inference on DPU and CPU and facilitate post-processing routines.
- **Library samples** can be used to test the networks present at the Model Library. This simplifies evaluating the models and getting values relative to performance and accuracy.
- **Application demos** is a collection of more complicated applications that best show the Vitis AI toolkit's full capability. It works with multiple neural network models in real-time.
- **Graph runner** examples is a group of example programs designed to demonstrate using the Graph Runner API. It simplifies the process of deploying models with multi-unit execution graphs. This API unifies the model into a single graph, so it is unnecessary to implement CPU functions and create a DPU runner for each DPU subgraph.

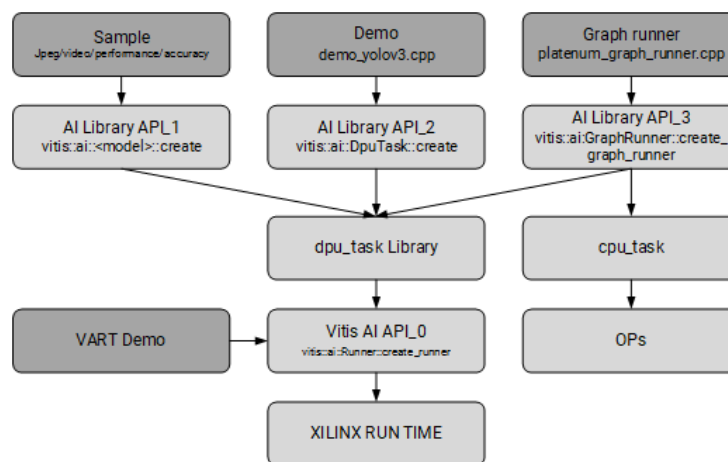


Figure 2.18: Diagram of Vitis AI programming APIs [52].

Four different APIs are available in this toolkit, as shown in figure 2.18. Their usage depends on the specific needs of a CNN and its similarity with other pre-trained models supported by Xilinx.

API_0 is based on VART. It is recommended for more advanced users of AI algorithms. All the pre and post-processing functions must be implemented using this API. Besides that, the tensor objects have to be manually loaded in the DPU runner for inputs and outputs of the several graphs the model might have.

API_1 is based on the Vitis AI Library Sample. It is suggested for a beginner user leveraging AI models that want a good performance. It is a quick way to build applications since it has models and pre and post-processing functions.

API_2 is recommended when a user wants to develop custom pre or post-processing algorithms. Users can write their code using "cpu_task," "dpu_task," and "xnnpp" libraries.

API_3 is proposed when a model that will be deployed is split into several subgraphs. GraphRunner was developed to implement these subgraphs and custom OPs quickly.

In this work, API_1 will be used for the PointPillars implementation and API_0 for the Voxel R-CNN.

2.6 Summary

The Vitis AI workflow will be used to implement and run the chosen neural network models. These implementations will be carried out in a Versal AI Core VCK190 board. The data used comes from the KITTI dataset, and the performance of the models will be measured using the GPU and the Versal ACAP. Timing and accuracy measurements will be made, as well as power consumption.

The evaluation of the implementation using different datasets and the steps that must be made to run the networks present a good starting point for studying the effectiveness of FPGA-based acceleration. It will be possible to take several conclusions from these implementations' obstacles. In addition to that, the Vitis AI tool from Xilinx and the DPU performance will be studied.

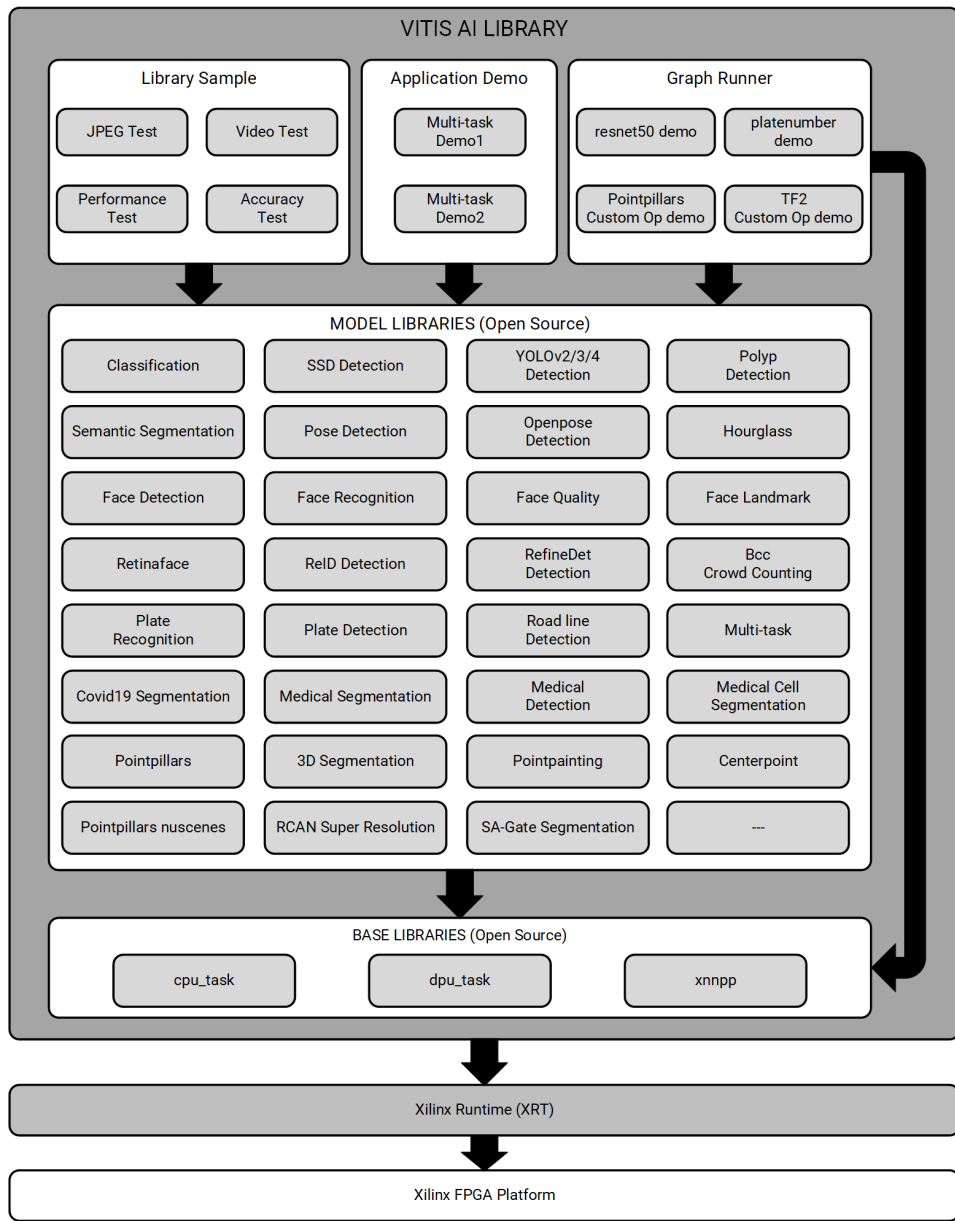


Figure 2.19: Block Diagram of Vitis AI Library [52].

Chapter 3

Design and Implementation

This chapter provides a comprehensive overview of the approaches and techniques employed in conducting the research. The software and hardware utilized will be described in depth.

3.1 Setup and tools used

The board used in this work was a Versal AI Core Series VCK190, part of the core series as specified in section 2.4.1. The main core of this board is a Versal VC1902 with the following characteristics represented in this table 3.1:

Table 3.1: Main characteristics of Versal AI Core VC1902 [53].

AI Engines	400
DSP Engines	1968
System Logic Cells (k)	1968
LUTs	899840
Application Processing Unit	Dual-Core Arm Cortex-A72
Real-Time Processing Unit	Dual-Core Arm Cortex-R5F
Maximum I/O Pins	770
Programmable NoC Ports	28
Integrated Memory Controllers	4

The board setup was relatively straightforward. The boot mode for SYSCTRL and the Versal part was done by switching SW11 and SW1 into the following order (ON, OFF, OFF, OFF). Figure 3.1 shows the board setup and main components. After that, a PetaLinux image provided by Xilinx was passed to the SD Card of the Versal part. To connect to the board, a serial connection was made using a USB-C connector, and after defining an IP, an ssh connection was made using an ethernet cable.

This work also used a desktop computer with an AMD Ryzen Threadripper 1920X 12-Core Processor, GeForce RTX 2070 GPU, and 64GB of RAM. The system runs CentOS Linux version 7 as the operating system, CUDA Version 11.8, and NVIDIA driver version 470.161.03.

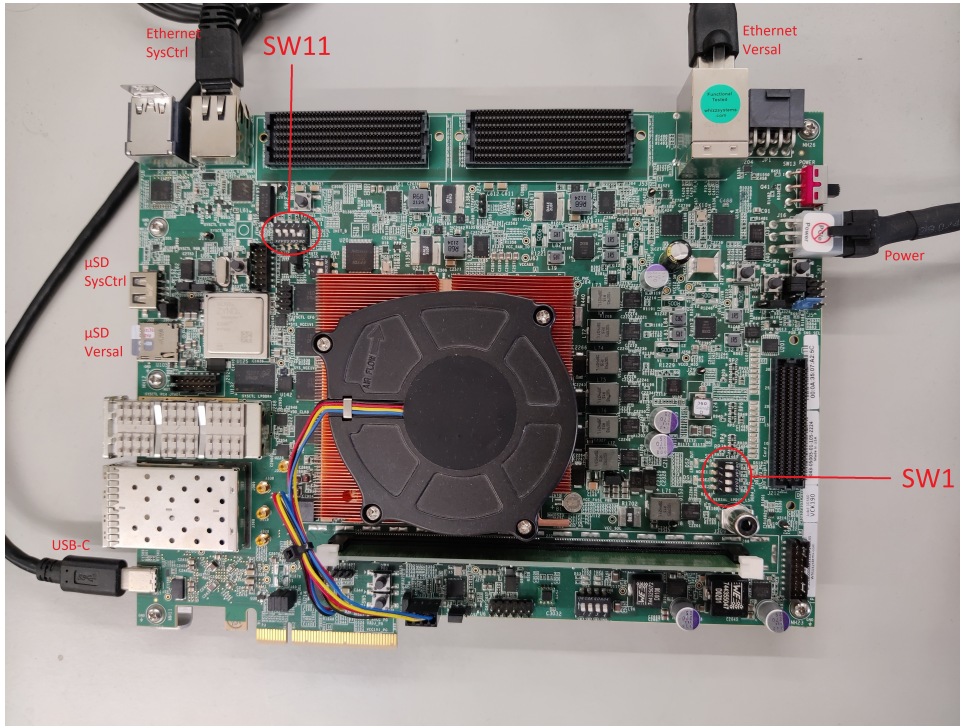


Figure 3.1: Versal VCK190 setup.

Vitis AI was used as the framework to implement and accelerate the ANN on the VCK190. The setup of the docker environment was simple using the provided pre-built docker. The version of Vitis AI is v3.0. The DPU used was the DPUCVDX8G with the architecture C32B6CU1L2S2, which has 32 AI Engines per batch handler, 6 batch handlers, and 1 compute unit. 333 MHz PL frequency and 1.25 GHz AI Engine frequency, making 61.44 Peak Theoretical Performance (TOPS).

As a first contact with the board, the PointPillars algorithm was implemented, taking advantage of the Model Zoo. After that, to measure how easy it is to accelerate a custom CNN using Vitis AI, Voxel R-CNN was chosen to go through all the steps that need to be taken from a bare representation using PyTorch to a running model in the board. OpenPCDet was used to build the Voxel R-CNN model. It is an open-source project focused on point cloud-based object detection in 3D environments. It provides a framework and toolkit for developing and benchmarking object detection algorithms using point cloud data.

3.2 PointPillars acceleration

PointPillars is a state-of-the-art grid-based 3D object detection network. It can be divided into three main blocks as represented in figure 3.2.

The "Pillar Feature Net" part converts the point cloud into a pseudo image so that 2D CNN from the backbone can be used. It utilizes a pillar-based approach for the grid structure. This

efficiently encodes spatial information from point clouds. After that, the pillars are organized into a tensor of size (D, P, N) . D represents each point's (N) values regarding its pillar (P). Due to the sparsity nature of point clouds, some limits were imposed on the number of points per sample and the number of non-empty pillars. Some ReLU functions are then performed, followed by a max pooling operation to generate the learned features. There will then form the pseudo image.

The backbone part is a CNN that processes the pillar-based features extracted by the Pillar Feature Net. It has two main parts: the top-down network and another that upsamples and concatenates the features from the top-down.

The Single Shot Detector (SSD) [54] model is used for the detection head. It performs object detection using diminishing convolutional layers.

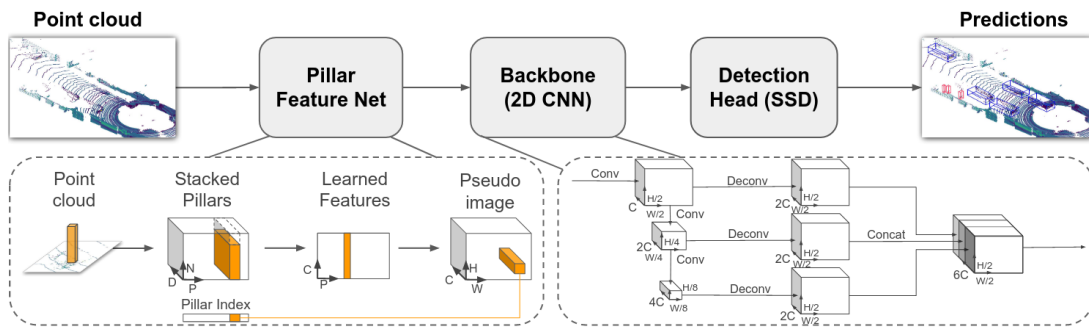


Figure 3.2: Overview of the PointPillars network [30].

PointPillars was chosen because it has a reasonable compromise between detection accuracy and calculation complexity. It is also used in multiple studies and is supported by the Vitis AI Library, making it a good starting point for this project.

3.2.1 Implementation description

Since this was the first approach to the Vitis AI framework, API_1 was utilized as it already has PointPillars and its pre/post-processing routines implemented. It significantly reduces engineering expenses for the intended platform benchmarking and experimentation tasks. This allows an implementation with a significant level of abstraction that can still achieve high-performance levels.

The first step was downloading the corresponding docker environment for this work. Vitis AI has multiple versions of pre-built docker depending on the framework (PyTorch or TensorFlow) and the platform acceleration (CPU, GPU, or ROCm). Pulling a pre-built docker image or building the container from Xilinx recipes is possible.

The second option was chosen, and the environment for PyTorch and GPU was created using the "docker_build.sh" script. This unified script facilitates generating a docker image meeting the developer's requirements. After the download, the container can be run using the shell script "docker_run.sh".

Inside the docker container, and because PointPillars is one of the various models supported in the model zoo, it is possible to download the network and all the associated functions. The model has the following nomenclature `pt_pointpillars_kitti_12000_100_11.2G_3.0`. This means it uses the PyTorch framework, Kitti as the dataset, has an input size of 12000*100 per image, a computational cost of 11.2G floating-point operations (FLOPs), and uses the Vitis AI version v3.0. That said, with the help of the Python script "downloader.py," it is straightforward to download either the compiled *xmodel* file or all the code related to implementing PointPillars on the GPU. The structure of this directory is shown in figure 3.3.

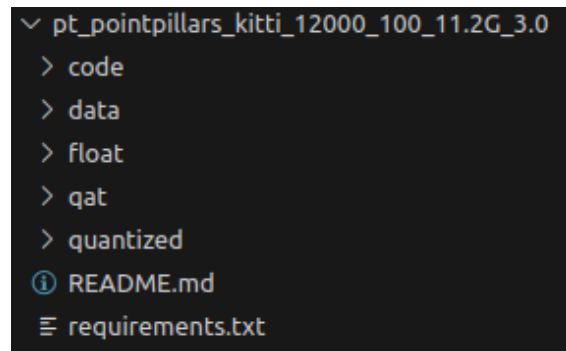


Figure 3.3: Overview of the model zoo PointPillars directory.

The "code" folder contains the code for testing and training the network. It encompasses creating the kitti data info, as well as quantization of the model.

In "data" sits the Kitti dataset that has to be manually downloaded. Kitti has 14999 test images and point clouds, making 80256 labeled objects. This translates to 12 GB of pictures and 29 GB of Velodyne point clouds.

In the "float" folder goes the trained floating-point model.

"Qat" and "quantized" are the destination of the quantization results. These are the quantized model and other vital files generated in the process.

The "requirements.txt" file has all the Python packages needed, and the "README.md" file has the step-by-step procedure to train, quantize, test, and compile the network.

3.2.2 Inference on the board

After all the initial setup and the installation of petalinux on the Versal ACAP, the board was ready to be tested. Usually, when a custom ANN is compiled using Vitis AI and the *xmodel* file is generated, it has to be transferred to the evaluating board to be tested. However, since the model to be evaluated is PointPillars, supported by the model zoo, the compiled PointPillars *xmodel* is already on the board. This is because the board image already includes Vitis AI Runtime packages, VART samples, Vitis AI Library samples, and models. The only thing transferred to the board was a package provided by Xilinx, holding example images and Velodyne point clouds (Velodyne is a leading company specializing in developing and manufacturing LiDAR sensors). This was done

to test the VAI Library, more precisely PointPillars, using the performance and accuracy tests of the "Library Sample."

These Sample Tests are written using C++ and the Model Library as a base foundation, which uses the Base Libraries like "cpu_task" and "dpu_task." Summarizing, they start to generate the network using the supplied compiled models.

```
1 auto net = vitis::ai::PointPillars::create(argv[1], argv[2]);
```

Listing 3.1: Creation of PointPillars network.

The point-cloud data is then stored in an object previously resized to hold the values. These are fed to the network that runs them and then displays the final values.

```
1 auto res = net->run(PointCloud);
2 //...
3 net->do_pointpillar_display(res, flag, g_test, rgbmat, bevmat, rgbmat.cols
    , rgbmat.rows, annoret);
```

Listing 3.2: Run of PointPillars network.

3.3 Voxel R-CNN acceleration

Although implementing the PointPillars using Vitis AI Library is a simple, quick way, and a good starting point for building an application, it does not provide the developer with many options regarding the implementation of the network. In addition, making the application this way does not give a good understanding of how the underlying code is implemented.

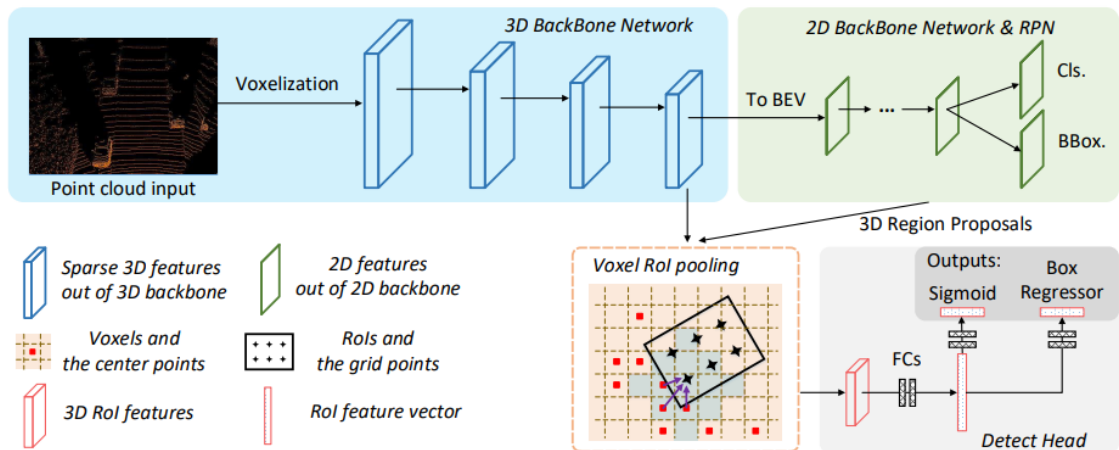


Figure 3.4: Overview of the Voxel R-CNN structure [31].

With that in mind, the next objective was to accelerate a custom ML model and test the main tools provided by Vitis AI. For this purpose, Voxel R-CNN [31] was chosen. It has good results on the KITTI dataset and is well situated in its ranking [55]. Besides that, it served as the base code

for some top 10 ranked models on the KITTI ranking, like SFD [56], VirConv-S [56], VirConv-T [56], and TED [57].

Voxel-RCNN, represented in figure 3.4, combines the strengths of both voxel-based representation and region proposal-based methods to achieve accurate and efficient object detection. The network begins by dividing the point cloud into equally spaced voxels, creating a voxel grid representation. Each voxel encodes spatial occupancy and semantic information. These are fed to the 3D backbone to extract features. Voxel-RCNN then employs the 2D backbone and later 3D Region Proposal Network (RPN) to generate candidate object proposals. These proposals are refined using a fully connected network and classified into specific object categories.

The open-source framework OpenPCDet [58] was used to represent this network most efficiently and quickly. OpenPCDet is designed explicitly for 3D object detection tasks using point cloud data and is Pytorch based. It supports a variety of popular 3D detection architectures, including the official implementation of Voxel R-CNN. The framework provides efficient data loaders, data augmentation techniques, and evaluation metrics for accurate and reliable model training and evaluation. Its user-friendly interface and extensive documentation prove to be a flexible and accessible framework for developing and benchmarking ML algorithms.

3.3.1 Implementation setup

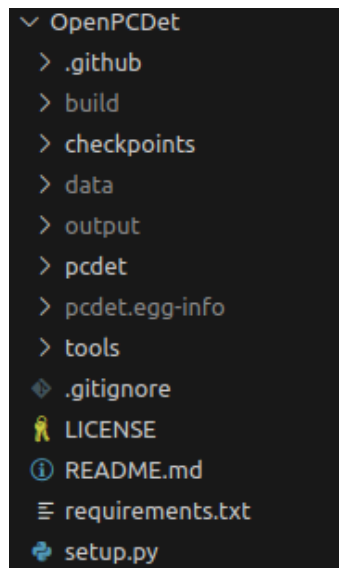


Figure 3.5: Top view of OpenPCDet working directory inside Vitis AI's environment.

The first step was implementing and integrating both OpenPCDet and Vitis AI environments. For this, the OpenPCDet main folder was placed inside the one from Vitis AI. This had to be done to access the tools provided by OpenPCDet inside the running container of Vitis AI. Install the "pcdet" library was done relatively quickly. However, since we were working inside the "xilinx/vitis-ai-pytorch-gpu:latest" container, some package dependencies emerged, making the

need to troubleshoot these. After solving these incompatibilities, our working directory was set as demonstrated in figure 3.5.

In the same way, it was made for the PointPillars network, the Kitti dataset was transferred to the "data" folder, and the data info was generated. The "tools" folder has the main scripts for model benchmarking and training, such as the files "test.py" and "train.py." In "checkpoints," the pre-trained Voxel R-CNN model provided by OpenPCDet with 28 MB size is present. "pcdet" is the main library folder and holds the PyTorch code for the model definition of multiple neural networks and various essential utilities.

3.3.2 Implementation description

In this section, it will be explained step by step how the quantization and compilation of Voxel R-CNN were done. It will describe the main parts of the implementation, the problems that showed up, and all the methods that had to be taken to bypass them.

A Python file was made to implement all the steps related to quantizing and compiling of Vitis AI flow. We started to import all the necessary Python packages. "pcdet" makes possible the dataset and model setup, and "pytorch_nndct" has all the required APIs provided by Vitis AI.

```

1  import torch
2  import torch.nn as nn
3
4  from eval_utils import eval_utils
5  from pcdet.config import cfg, cfg_from_yaml_file
6  from pcdet.datasets import DatasetTemplate, build_data_loader
7  from pcdet.models import build_network, load_data_to_gpu
8  from pcdet.utils import common_utils
9
10 from pytorch_nndct.apis import Inspector, torch_quantizer, dump_xmodel
11 from pytorch_nndct.apis import torch_quantizer, dump_xmodel

```

Listing 3.3: Importing of the most important packages.

The "build_data_loader" function was utilized to load the Kitti data, which presents a convenient and easy way of building the data loader and dataset. The Voxel R-CNN model was then made, loading the parameters from the previously downloaded and pre-trained checkpoint.

```

1  #Load dataset
2  test_set, test_loader, sampler = build_data_loader(
3      dataset_cfg=cfg.DATA_CONFIG,
4      class_names=cfg.CLASS_NAMES,
5      batch_size=batch_size,
6      dist=False,
7      workers=4, logger=logger, training=False
8  )
9
10 #Build the model

```

```

11     model = build_network(model_cfg=cfg.MODEL, num_class=len(cfg.
        CLASS_NAMES), dataset=test_set)
12     model.load_params_from_file(filename=args.ckpt, logger=None)

```

Listing 3.4: Daset loading and building of the network.

All the necessary steps were taken to initialize the Vitis AI flow. However, because of the complexity level of the model implementation, it was impossible to quantize the entire model at once. A dictionary object containing multiple data information is forwarded through the various stages of the neural network. This is due to how the model was built using PyTorch. The Vitis AI Quantizer expects to receive a torch model and a dummy input to quantize the network. However, this dummy input has to be a torch tensor or a list of tensors, resulting in this incompatibility.

To solve this problem, we had to dive deep into the architecture of the torch implementation of Voxel R-CNN. The model is divided into multiple sub-models. Each sub-model forwards the data to the next one. A neural network test was made to determine which models were the slowest. After forwarding the Kitti dataset, the conclusion was that the slowest parts were the 3D Backbone Network and the 2D Backbone Network.

Afterward, knowing which parts of this network would be associated with the CPU or the DPU using the Inspector was possible.

```

1     target = "DPUCVDX8G_ISA3_C32B6"
2     #Inspect the model
3     with torch.no_grad():
4         if inspect:
5             inspector = Inspector(target)
6             inspector.inspect(backbone_module, tensor, torch.device("cuda"
                ), image_format="png")

```

Listing 3.5: Inspection of the model regarding DPUCVDX8G.

While the 2D Backbone was positively inspected, the 3D one was not. This happened because Vitis AI does not support sparse convolutions yet, which means that the 2D Backbone was the only quantizable network of the two.

From the examination of the *png* image generated by the Inspector, it was possible to notice that the padding and transpose functions were not supported by the DPU, resulting in a CPU implementation. This execution is far from ideal and presents a significant bottleneck, making the data be transferred between DPU and CPU because of two elementary functions. It was later discovered that these functions were not being DPU-supported due to a Vitis AI bug.

Solving this problem made the entire neural network supported and able to run on the DPU. Even though this bug was solved and fixed in the later stages of the project, we will still demonstrate the two implementations since we could extract valuable conclusions from both.

That said, the next step was the calibration step. For this, we used the Quantizer and changed the 2D backbone model the NN used to the quantized model. This step was done after 200 data

samples were feed-forward, and two critical files for the following stages were generated. "Base-BEVBackbone.py" is a converted vai_q_pytorch format model, and "quant_info.json" has information about the quantization steps of the tensors.

```

1 #Quantizing calib
2 if calib:
3     quantizer = torch_quantizer(quant_mode='calib', module=
4         backbone_2d_module, input_args=tensor, device=torch.device("cuda"),
5         target=target)
6     quant_model = quantizer.quant_model
7
8     #Change module to quantized
9     model.module_list[3] = quant_model
10
11     #Forward pass with quantized module
12     model.cuda()
13     model.eval()
14     with torch.no_grad():
15         for i, batch_dict in enumerate(test_loader):
16             load_data_to_gpu(batch_dict)
17             with torch.no_grad():
18                 pred_dicts, ret_dict = model.forward(batch_dict)
19             print(i)
20             if i>=200:
21                 break
22
23     quantizer.export_quant_config()

```

Listing 3.6: Calibration of quantized model.

After the calibration, we were ready to test the accuracy of the quantized model. For this, the whole dataset was used, and the benchmark tools from "eval_utils" were utilized, more precisely, the function "eval_one_epoch." This function computes the model's overall performance, logging the evaluation metrics.

Afterward, the *xmodel* file for compilation can be generated. It is dumped into the output folder using the "export_model" function:

```

1 #Run pcdet benchmark
2 logger.info('*****Start testing*****')
3
4 gpu_list = os.environ['CUDA_VISIBLE_DEVICES'] if 'CUDA_VISIBLE_DEVICES'
5     in os.environ.keys() else 'ALL'
6 logger.info('CUDA_VISIBLE_DEVICES=%s' % gpu_list)
7
8 with torch.no_grad():
9     model.cuda()

```

```

8     eval_utils.eval_one_epoch(
9         cfg, args, model, test_loader, 54, logger, dist_test=False,
            result_dir=eval_output_dir
10    )
11
12    if deploy:
13        quantizer.export_torch_script()
14        quantizer.export_xmodel(deploy_check=True)

```

Listing 3.7: Integration of OpenPCDet benchmark tool with the quantized model.

After the *xmodel* file is exported to the "quantize_result" folder, it is ready to be compiled by the Vitis AI Compiler. This is done using `vai_c_xir` in the command line. It takes as input the directory of the DPU architecture configuration file in JSON format present in the docker container and the directory for the *xmodel* to be compiled:

Listing 3.8: Shell command to run the compiler.

```

vai_c_xir \
--xmodel ./quantize_result/BaseBEVBackbone_int.xmodel \
--arch /opt/vitis_ai/compiler/arch/DPUCVDX8G/VCK190/arch.json

```

Finally, the model is compiled and ready to be implemented and tested on the VCK190 board.

3.3.3 Inference on the board

The model had to be inspected before developing the code to make the inference on the board. This is because the quantized and compiled network might be split into some subgraphs running on the CPU and others on the DPU. If this happened, the code for the unsupported functions had to be developed in the board's processor. As already stated, the first implementation had some OPs running on the CPU, which meant the implementation of these functions on the processor.

Even though this problem was solved, this implementation will still be explained as it demonstrates the steps to take when a model is split between hardware and the issues that it provokes. The VART library was used and implemented using Python to run the compiled model on the board. As a first step, the compiled model has to be deserialized and partitioned into several subgraphs. After this, the DPU runners are created:

```

1    #Deserialize xmodel
2    g = xir.Graph.deserialize(model)
3
4    #Get subgraphs
5    subgraphs = g.get_root_subgraph().toposort_child_subgraph()
6
7    dpu_subgraph0 = subgraphs[0]
8    dpu_subgraph1 = subgraphs[1]
9    dpu_subgraph2 = subgraphs[2] #dpu

```

```

10     dpu_subgraph3 = subgraphs[3]
11     dpu_subgraph4 = subgraphs[4] #dpu
12     dpu_subgraph5 = subgraphs[5]
13
14     #Create dpu runners
15     dpu_2 = vart.Runner.create_runner(dpu_subgraph2, "run")
16     dpu_4 = vart.Runner.create_runner(dpu_subgraph4, "run")

```

Listing 3.9: Deserialization of the model and creation of the DPU runners.

The data input for the model to run has to be loaded. For this, several torch tensors from the network running on the GPU were saved into numpy files and loaded into the board:

```

1     for i in range(runTotal):
2         # Check if the file is an npy file
3         if i==0:
4             filename = f"spatial_features_0_5.npy"
5         else:
6             filename = f"spatial_features_{i*6}_{6*i+5}.npy"
7         print(filename)
8
9         # Construct the full path to the file
10        file_path = os.path.join(input_folder, filename)
11
12        # Load the numpy array from the file
13        array = np.load(file_path)
14
15        # Append the array to the loaded_arrays list
16        loaded_arrays.append(array)

```

Listing 3.10: Loading of the input numpy arrays to be run in the DPU.

The DPU runners will then use this list of numpy arrays, executing them asynchronously via the "execute_async" function for every input. Note that this input can be a batch size 6 for every iteration. This is due to the natural structure of the DPUCVDX8G and related architecture used, which has 6 batch handlers. Meaning it can process six samples of data in parallel. This execution is represented in the following fragment of code:

```

1     #Run CPU1 to pad and transpose
2     out1 = transpose(pad2d_subgraph1(input))
3
4     #Run DPU2
5     execute_async(
6         dpu_2,
7         {"ModuleList_0__ZeroPad2d_0__input_1_swim_transpose_0_fix": out1,
8          "ModuleList_0__ReLU_18__1660_fix": out2}
9     )

```

```

10     inp3 = out2.copy()
11
12     #Run CPU3 to pad
13     out3 = pad2d_subgraph3(inp3)
14
15     #Run DPU4
16     execute_async(
17         dpu_4,
18         {"ModuleList_1__ZeroPad2d_0__input_41_swim_transpose_1_fix": out3,
19          "ModuleList_0__ReLU_18__1660_fix": out2,
20          "BaseBEVBackbone__BaseBEVBackbone_1929_fix": out4}
21     )

```

Listing 3.11: Overview of "runDPU" subroutine where DPU runners are executed as well as CPU functions.

Subgraphs 1 and 3 are running on the CPU with "pad2d_subgraph1", "pad2d_subgraph3", and "transpose" being implemented. Another note is that the names of the input and output tensors are added manually regarding the subgraph tree. These names can vary depending on the version of the Vitis AI Compiler.

This implementation was relative to the model, having some layers not supported by the DPU. The model that runs entirely on the DPU followed a similar implementation but without the need to run functions on the CPU.

3.3.4 Evaluation

To draw some conclusions, some evaluations had to be made regarding the performance of the implementations. To measure the execution time on the ARM processor of the VCK190 board, the "time" Python package was used. To test the different execution times of the model running in the GPU, we utilized the "torch.cuda.Event" class. Accuracy and performance benchmark of the quantized model was done by the "eval_one_epoch" functions and the values saved.

```

1
2     if i==0:
3         # Record the start event
4         start_event.record()
5     x = self.blocks[i](x)
6     if i==0:
7         # Record the end event
8         end_event.record()
9
10        # Wait for the events to complete
11        torch.cuda.synchronize()
12
13        # Calculate the elapsed time in milliseconds

```

```
14         elapsed_time = start_event.elapsed_time(end_event)
15
16         print("Elapsed time block 1:", elapsed_time, "ms")
```

Listing 3.12: Time evaluation of different parts of the model running in the GPU.

```
1     #Run DPU
2     print("")
3     print("run DPU")
4     time1 = time.time()
5     runDPU(dpu_2, dpu_4, loaded_arrays)
6     time2 = time.time()
7     timetotal = time2 - time1
8     print(f"Tempo run DPU: {timetotal}")
```

Listing 3.13: Time evaluation of the DPU run.

The Vitis AI Profiler was another helpful tool for acquiring performance and other valuable results. Combining all these tools and benchmark results makes valuable data to be evaluated and conclusions possible to be taken. The next chapter will summarize the main results obtained from this project as well as their analysis.

Chapter 4

Results and Evaluation

This chapter presents the results of implementing two state-of-the-art 3D object detection models: PointPillars and Voxel R-CNN. The primary focus of our evaluation was to assess the performance and efficiency of these models on different hardware platforms, specifically the Versal ACAP device and GPU. For the PointPillars model, we leveraged the pre-trained models available in the Vitis AI model zoo and utilized the Vitis AI library samples.

On the other hand, for Voxel R-CNN, we employed the Vitis AI Flow, which encompassed the entire workflow from quantization to compilation. Additionally, we explored two deployment scenarios: one utilizing the full support of the DPU and the other involving an execution approach with the DPU and CPU.

4.1 PointPillars

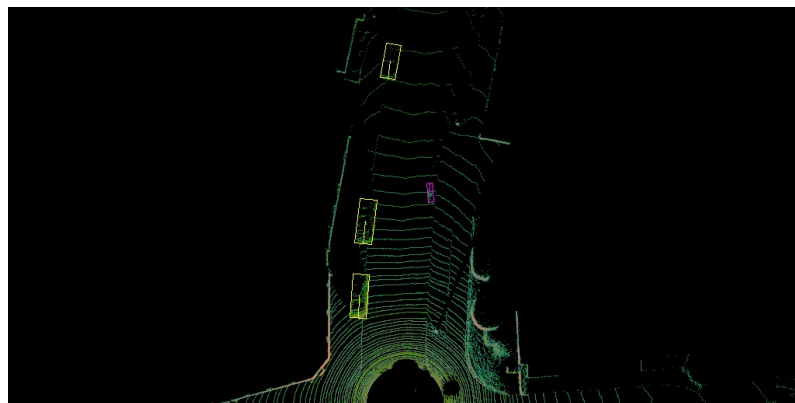


Figure 4.1: Example of BEV detection of PointPillars using Vitis AI Library samples.

As PointPillars inference was the first interaction with Vitis AI Library, the main objective was to get to know Vitis AI and its ready-to-go applications. The first test used was a simple inference on an example point cloud. Besides getting the detected objects in an image file from Bird's-Eye-View (BEV) represented in figure 4.1, we also get the front camera as in 4.2. Output



Figure 4.2: Example of 3D detection of PointPillars using Vitis AI Library samples.

is also written to the terminal with the labeling of the image, the coordinates of the corresponding bounding boxes, and the probability associated with the detection. In this example, we have three cars detected and one person, with an associated probability of 0.9099, 0.9047, 0.7981 for the cars, and 0.8354 for the person.

The performance test was also executed on the VCK190, with each image taking 44.04 ms to process. The total end-to-end time, which includes pre and post-processing, took an average of 39.504 seconds, while the DPU took 23.129 seconds to process. On the other hand, the float model running on the GPU took 55.60 ms to infer each input. This data makes it possible to conclude that the GPU took an extra 11 ms to process each image compared to the DPU. This is a promising result for the implementation of ML model devices on the edge.

Vitis AI Profiler was utilized for more timing results and resource utilization. Table ?? has the main resource consumption for each subgraph of the model on the DPU. It has also values for the performance and the execution time. Table 4.2 has timing values for the CPU functions, which include pre and post-processing and VART functions.

Table 4.1: Profiling summary of DPU while running a PointPillars performance test.

DPU SubGraph	WL	RT	Perf	LdWB	LdFM	StFM	AvgBw
1	0.768	4.720	976.263	0.001	27.466	4.395	6912.242
2	10.159	18.178	3353.273	0.377	310.378	123.833	24481.763

Table 4.2: Profiling summary of CPU functions while running a PointPillars performance test.

Function	Device	Runs	AverageRunTime(ms)
xir::XrtCu::run	CPU	1490	11.419
vitis::ai::DpuTaskImp::run	CPU	1490	16.186
vitis::ai::ConfigurableDpuTaskImp::run	CPU	1490	16.201
vitis::ai::PointPillarsPost::post_process	CPU	745	1.168
vitis::ai::PointPillarsImp::run_2	CPU	745	39.877
vitis::ai::PointPillarsImp::run	CPU	745	39.892

The labeling of table 4.1 and the others from this chapter goes this way:

- Work Load (WL): This metric measures the computation workload in terms of GOP (Giga Operations), where each MAC (Multiply-Accumulate) operations count as two operations.
- Software Run time (SW_RT): It refers to the execution time calculated by the software in milliseconds (ms).
- Hardware Run time (HW_RT): It indicates the execution time of hardware operations in milliseconds (ms).
- Efficiency (Effic): This metric represents the DPU's actual performance divided by its peak theoretical performance, expressed as a percentage (%).
- Performance (Perf): Represents the DPU's actual performance in GOP/s.
- Load Size of Feature Map (LdFM): This metric measures the external memory load size of the feature map in megabytes (MB).
- Load Size of Weight and Bias (LdWB): It represents the external memory load size of the weight and bias in megabytes (MB).
- Store Size of Feature Map (StFM): This metric indicates the external memory store size of the feature map in megabytes (MB).
- Average Bandwidth (AvgBw): It measures the average external memory bandwidth in megabytes per second (MB/s).

Vitis AI Library proves to be a valuable tool with multiple evaluation programs written in C++ ready to use. The Model Zoo provides the developer with all information to implement a specific model into different Xilinx boards. Regarding the profiling of PointPillars, the run-time for the two DPUs was 22.898 ms, and the workload was 10.927 GOP/s. It is possible to deduce that the total run-time of the model was almost twice the time of the DPU inference. The DPU performance being more efficient than the GPU is an encouraging factor for the next implementation. The timing characteristics and utilization will be deeper studied in Voxel R-CNN application.

4.2 Voxel R-CNN

The first result of this project was some accuracy results from running Voxel R-CNN in the GPU. For that, the integrated OpenPCDet library was used. This benchmark resulted in these two tables 4.3 and 4.4.

Intersection over Union (IoU) measures the extent of overlap between the detected object and its real location by calculating the ratio of their intersection area to their union area. It scores from 0 to 1, with 1 being a perfect match. By setting a specific IoU threshold, a minimum overlap value can be defined for a detection to be considered a true positive. This way, setting a higher value for the threshold means a more precise detection, but it can lead to undetected objects. In contrast,

lower values of threshold can result in more lenient criteria, resulting in detections with broader overlaps being considered as positives.

Average Precision (AP) is a commonly used evaluation metric in object detection tasks. It measures the proportion of correct detections among all positive detections. Therefore AP values for higher IoU thresholds tend to be lower than for higher thresholds.

Table 4.3: Accuracy of Voxel R-CNN using float model of 2D backbone part.

IoU Threshold	$AP_{3D}(\%)$			$AP_{BBox}(\%)$			$AP_{BEV}(\%)$		
	Easy	Moderate	Hard	Easy	Moderate	Hard	Easy	Moderate	Hard
0.7	89.1421	84.5415	78.6853	97.8540	89.7324	89.3443	90.2107	88.2890	87.7709
0.5	97.8532	89.7274	89.3430	97.8540	89.7324	89.3443	97.8917	89.7398	89.3670

Table 4.4: Accuracy of Voxel R-CNN using quantized model of 2D backbone part.

IoU Threshold	$AP_{3D}(\%)$			$AP_{BBox}(\%)$			$AP_{BEV}(\%)$		
	Easy	Moderate	Hard	Easy	Moderate	Hard	Easy	Moderate	Hard
0.7	88.6884	84.3910	78.4370	97.3706	89.6580	89.2450	90.0034	88.1876	87.6815
0.5	97.3789	89.6710	89.2636	97.3706	89.6580	89.2450	97.4286	89.6880	89.3000

From the evaluation of these values, we can say that the AP values are consistently lower in the quantized model than in the float model. However, this difference is minimal. Relatively to the inference time for each image, there was a slight decrease from the float model to the quantized model: 61.1 ms to 55.3 ms.

The performance of the 2D backbone part of the Voxel R-CNN model on GPU was also measured. In this way, we could compare the running version of the DPU and the GPU. The average time was **37.594 ms** for an iteration of batch size 6. This value will be compared further on.

Afterward, the Vitis AI Inspector was used, and two images were generated. Depending on the color of each function, it is possible to know which part was assigned to the CPU or to the DPU (blue is given to the DPU, and red is given to the CPU). Examining the two, it is possible to conclude that one is fully DPU-supported and the other partially supported with some functions on the CPU. Both images are in appendix B.

If figure 4.3 is the cropped image from inspecting the model with some layers needing to be implemented on the CPU. As it is possible to see, padding and a permuting operation were assigned to the CPU. Additionally, the convolutional and ReLU functions were associated with the DPU.

In addition, Inspector provides a detailed description of the model and the different layers in text format. During the inspection of the model, some special logging messages are displayed on the terminal. These messages describe Inspector's several stages, from the inspector's start-up, the processing of the layers, to the association of the different subgraphs to either the DPU or CPU. Combined, this makes the Vitis AI Inspector tool a fundamental mechanism. It helps the developer to debug some problems before starting the quantization process.

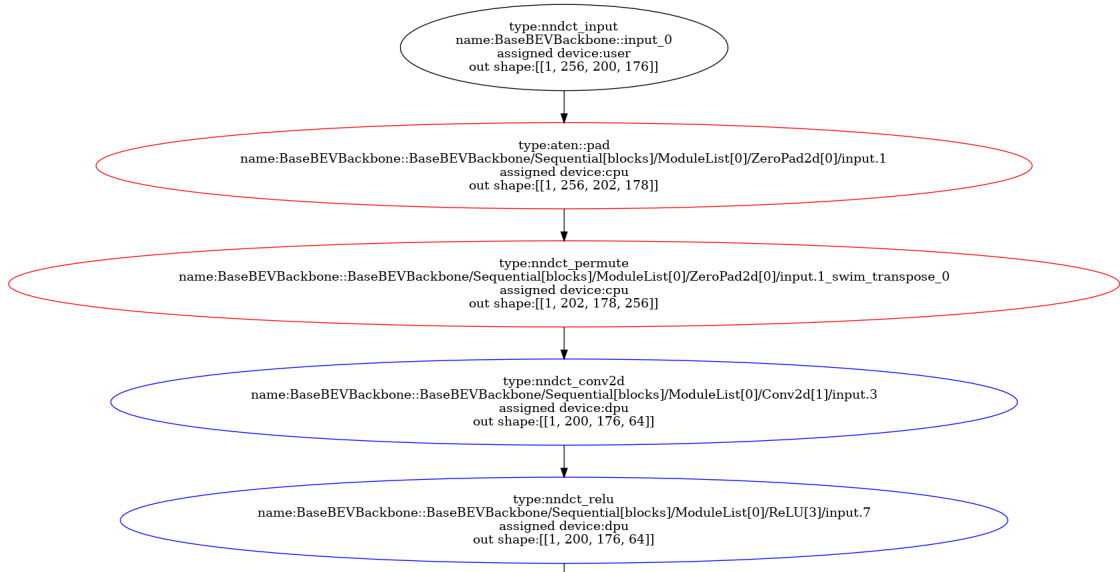


Figure 4.3: Cropped image from inspecting total DPU implementation of the 2D backbone of Voxel R-CNN.

Since we have implemented this network with total DPU support and the other running on CPU and DPU and got valuable results from both, we will split the results in two regarding each implementation.

4.2.1 CPU and DPU implementation

This was the first model to be tested on the VCK190. The "runDPU" function of the Python program takes 12.9246 seconds to process 102 sample data with a batch size of 6. This makes **760 ms** for an iteration of batch size 6. This value of processing is more or less 20x higher than the GPU, which is far from ideal.

Table 4.5: Profiling summary of CPU functions regarding the partial DPU compatible model.

Function	Device	Runs	Average Run Time (ms)
var::TensorBuffer::copy_tensor_buffer	CPU	85	35.159
xir::XrtCu::run	CPU	34	7.620

Table 4.6: Profiling summary of DPU resources regarding the partial DPU compatible model.

DPU SubGraph	WL	SW_RT	HW_RT	Effic	LdWB	LdFM	StFM	AvgBw
1	23.371	8.449	8.315	29.0	0.317	117.715	77.344	23495.946
2	16.020	6.924	6.803	24.3	0.845	64.591	90.234	22881.391

However, from analyzing the table 4.5, it is possible to notice that a massive part of the time is consumed on processing data transfer between DPU and CPU. The function associated with this transfer is "var::TensorBuffer::copy_tensor_buffer", and is almost 5 times slower than the

function associated to the DPU run (xir::XrtCu::run). The corresponding subgraphs running on the DPU, and represented as 1 and 2 in table 4.6, in the GPU take, on average, 20.219 ms and 15.016 ms. This makes the DPU almost 2.4x and 2.2x faster than the corresponding layers on the GPU.

4.2.2 Total DPU implementation

Regarding the inference with total DPU compatibility, the total run-time for the "runDPU" task was 6.6998 seconds, making it **394 ms** to process each iteration of batch size 6. It is almost 2 times faster than the previous implementation. This is because no CPU layers are associated, with the data only needing to be transferred twice for each run. However, it is still very slow compared to the GPU implementation.

Table 4.7: Profiling summary of DPU resources for the total DPU compatible model.

DPU SubGraph	WL	SW_RT	HW_RT	Effic	LdWB	LdFM	StFM	AvgBw
1	39.390	15.030	14.898	27.3	1.161	181.055	167.578	23479.874

Table 4.8: Profiling summary of CPU functions for the total DPU compatible model.

Function	Device	Runs	AverageRunTime(ms)
vart::TensorBuffer::copy_tensor_buffer	CPU	34	182.644
xir::XrtCu::run	CPU	17	14.982

The output from the Profiler is in tables 4.7 and 4.8. As in the previous model, there is a significant bottleneck when transferring the data between the DPU and the CPU. The data processing in the DPU is more than 2x times faster than the GPU.

4.2.3 Power measurement

The power consumption from GPU and the Versal ACAP was also measured since these are relevant topics in edge applications, specifically AV.

A Python script was made to measure the power intake of the GPU. This script used the "pynvml" library to query the power usage of the GPU during the Voxel R-CNN forwarding. This data was later plotted in figure 4.4.

To compute the power consumption of the Versal device, the Versal Adaptive Power Tool was used. This instrument comes pre-installed on the system controller on the Pynq device. It consists of a Python library, "poweradvantage.py," that provides a way to measure the power system of the embedded platform. This way, the script calculates the power independently from the Versal device, making it possible to execute programs on the Versal device and simultaneously measure power statistics. It is possible to get multiple measurements for voltage, power, and current for different parts of the ACAP. However, the device's total power consumption during the model forwarding was plotted and represented in figure 4.5.

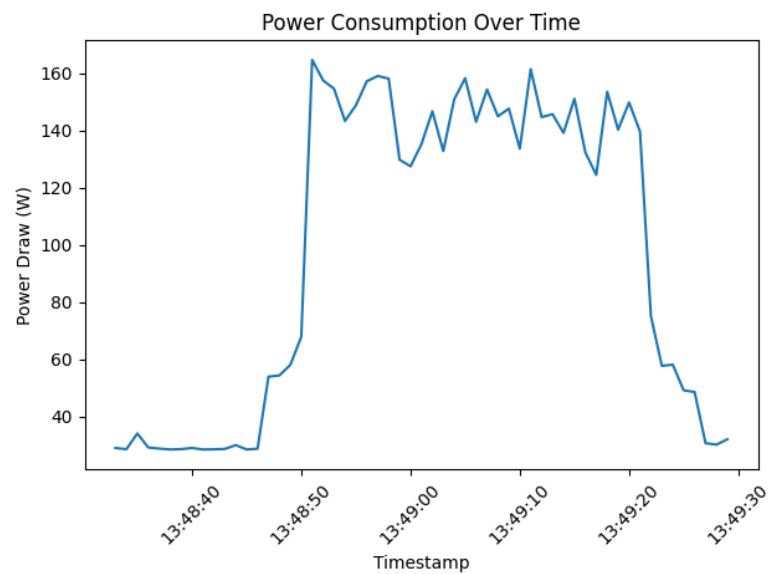


Figure 4.4: Power usage of GPU during Voxel R-CNN inference.

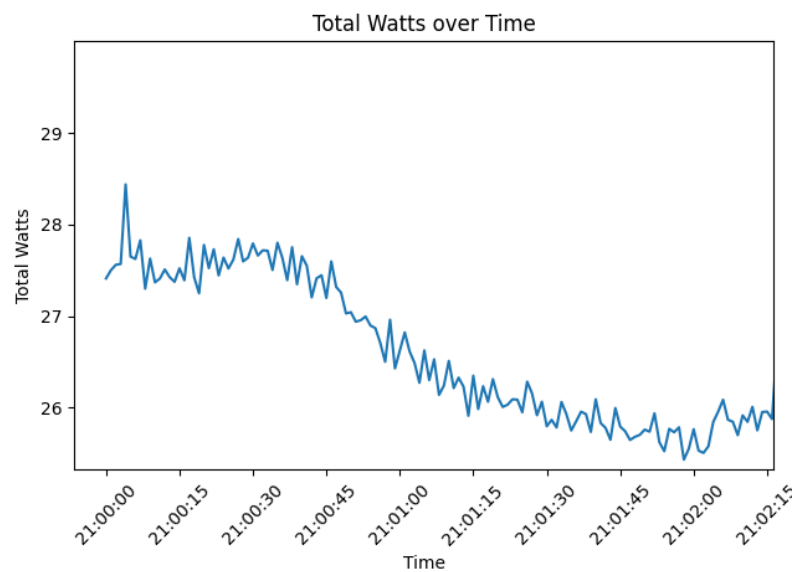


Figure 4.5: Power usage of Versal ACAP during inference.

Comparing both graphics, it is evident that the VCK190 uses less power than the GPU. When making inferences, the GPU goes up to around 140 W, and the Versal device only consumes around 26/27 W and never exceeds the 29 W mark. For 30 seconds of inference time, the GPU consumes around 4140 J of energy, while the ACAP only uses 804 J. This makes the Xilinx device far more energy efficient.

Chapter 5

Conclusion and Future Work

This work investigated the implementation of two ANNs on the Versal AI Core VCK190 board. These were the PointPillars and Voxel R-CNN models, which are relevant in AD. Vitis AI workflow was used for this, and key findings and future directions can be given.

Implementing the PointPillars network was pretty straightforward using the Vitis AI Sample Library. This provides a great starting point for developers that want to implement their applications on a Xilinx device. It includes support for multiple models and has some tests already implemented. These implementations provide excellent results, as seen in the PointPillars application.

The first problem encountered using Vitis AI was not supporting sparse convolutions. This was a massive obstacle since the 3D backbone part of Voxel R-CNN was one of the slowest parts. Besides that, a fully implemented 3D object detection model on edge becomes impossible. A potential future work could be to implement an IP block capable of processing this type of convolution on the PL hardware.

Regarding the Voxel R-CNN implementation, the quantizer proves to be an efficient tool. The accuracy of the network was almost the same, and the performance slightly increased with the FPS going to 18.083Hz from 16.366Hz on the GPU.

The 2D backbone part inference on the board got a massive increase in the time of processing per data. This increase in time was 20 times higher than the identical part running on the GPU. However, this performance difference was due to data transfer processing between the CPU and the DPU. This was a significant bottleneck, as the average runtime of the total DPU implementation was 14.982ms. This is two times faster than the GPU inference, which takes 37.594 ms. A potential area to study and to be improved is finding a better way of data transferring between the DPU and the CPU.

Comparing the two implementations regarding the Voxel R-CNN, the total DPU implementation was two times quicker. Total DPU compatibility is always an objective to aim for when deploying an ANN.

The power consumption is an essential advantage of the Versal device over the GPU. While the GPU consumes around 140W when running the model, the ACAP only uses around 27W, which

is five times less. This utilization also translates to energy usage, with the GPU having 5x times more.

The main objectives of this work were completed, which were the implementation of ANNs on a Versal device and the study of their performance and energy efficiency. Vitis AI and DPU proved to be reliable tools for accelerating DL models but with much room to improve. Some operations might still not be supported, like sparse convolutions, and some bugs can be encountered, leading to unsupported DPU OPs.

Appendix A

DPUCVDX8G Resource Utilization

Resources utilization and peak performance for the DPUCVDX8G regarding different architectures.

Table A.1: DPUCVDX8G resource utilization depending on architecture used [50].

[HTML]FFFFFF Architecture	AI Engine Cores	LUT	FF	Block RAM	UltraRAM	DSP	PL NMU
[HTML]FFFFFF C32B1CU1L2S2	32	82942	111248	0	136	139	8
[HTML]FFFFFF C32B1CU2L2S2	64	165721	222528	0	272	278	14
[HTML]FFFFFF C32B1CU3L2S2	96	248486	333805	0	408	417	20
[HTML]FFFFFF C32B2CU1L2S2	64	146223	190065	0	200	273	10
[HTML]FFFFFF C32B3CU1L2S2	96	209657	268933	0	264	407	12
[HTML]FFFFFF C32B4CU1L2S2	128	273736	348300	0	328	541	14
[HTML]FFFFFF C32B5CU1L2S2	160	338367	428383	0	392	675	16
[HTML]FFFFFF C32B6CU1L2S2	192	403866	507308	678	343	809	18
[HTML]FFFFFF C64B1CU1L2S2	64	93233	132511	0	136	139	8
[HTML]FFFFFF C64B1CU2L2S2	128	186387	265041	0	272	278	14
[HTML]FFFFFF C64B1CU3L2S2	192	280846	397732	0	408	417	20
[HTML]FFFFFF C64B2CU1L2S2	128	157744	215444	0	200	273	10
[HTML]FFFFFF C64B3CU1L2S2	192	222342	298513	0	264	407	12
[HTML]FFFFFF C64B4CU1L2S2	256	286802	381864	0	328	541	14
[HTML]FFFFFF C64B5CU1L2S2	320	355909	469409	0	392	675	16

Table A.2: Peak theoretical performance of each DPUCVDX8G architecture [50].

Architecture	Peak Theoretical Performance (TOPS)
C32B1CU1L2S2	10.24
C32B1CU2L2S2	20.48
C32B1CU3L2S2	30.72
C32B2CU1L2S2	20.48
C32B3CU1L2S2	30.72
C32B4CU1L2S2	40.96
C32B5CU1L2S2	51.20
C32B6CU1L2S2	61.44
C64B1CU1L2S2	20.48
C64B1CU2L2S2	40.96
C64B1CU3L2S2	61.44
C64B2CU1L2S2	40.96
C64B3CU1L2S2	61.44
C64B4CU1L2S2	81.92
C64B5CU1L2S2	102.4

Appendix B

Voxel R-CNN Inspect Result

Images output from inspecting the 2D backbone part model of Voxel R-CNN using Vitis AI Inspector.

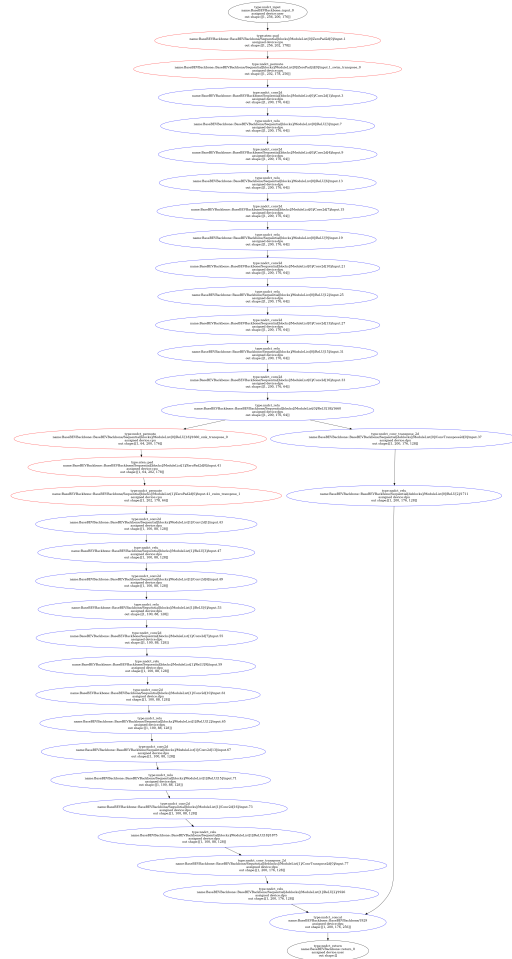


Figure B.1: Image result from inspecting the 2D backbone from Voxel R-CNN with total DPU not supported.

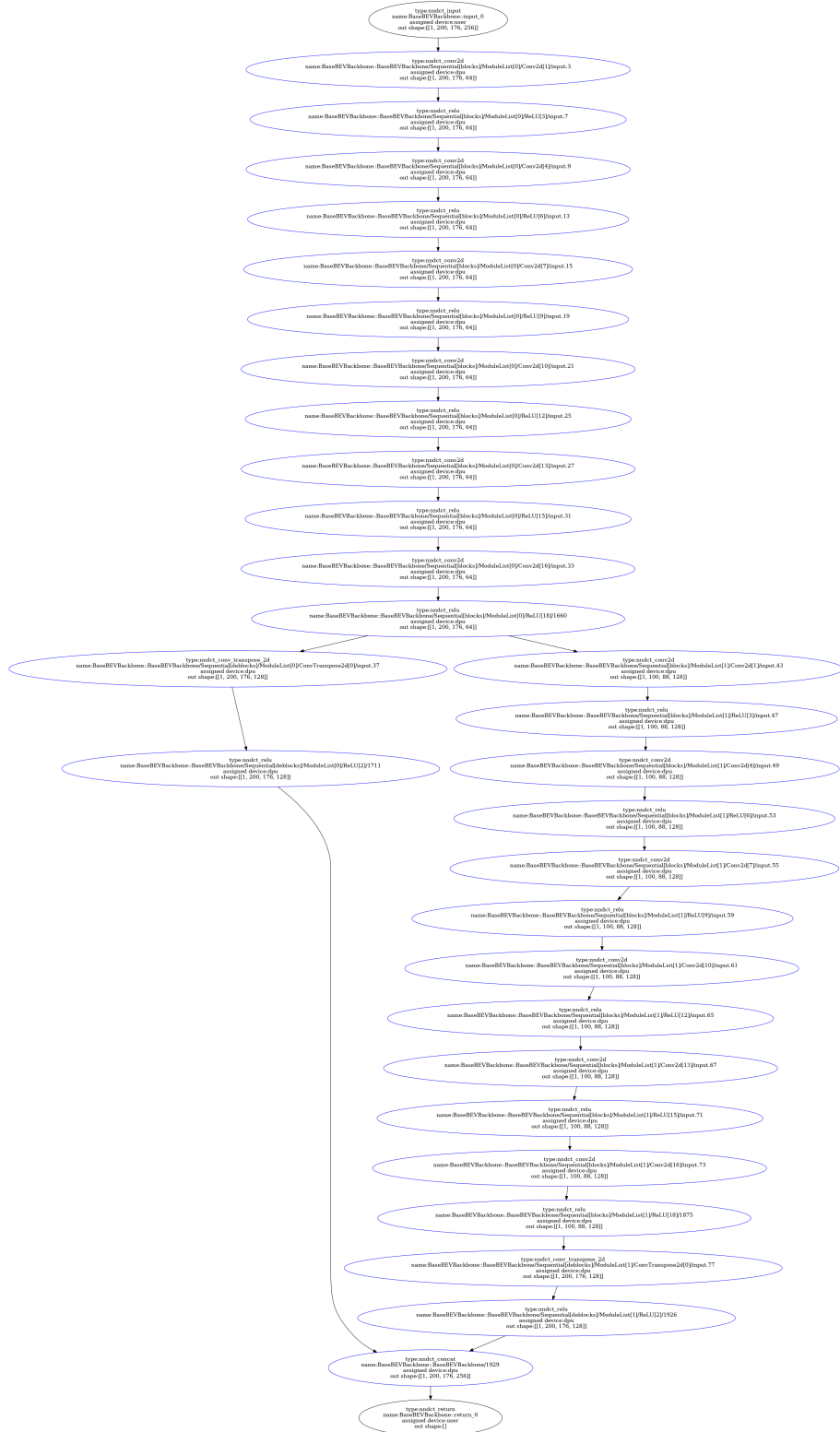


Figure B.2: Image result from inspecting the 2D backbone from Voxel R-CNN with total DPU supported.

References

- [1] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 8:58443–58469, 2020. doi:10.1109/ACCESS.2020.2983149.
- [2] Mahdi Elhousni and Xinming Huang. A survey on 3D LiDAR localization for autonomous vehicles. 5 2020. URL: <http://arxiv.org/abs/2006.00648>.
- [3] Rui Qian, Xin Lai, and Xirong Li. 3D object detection for autonomous driving: a survey. *Pattern Recognition*, 130:108796, 2022.
- [4] Wenjin Huang, Huangtao Wu, Qingkun Chen, Conghui Luo, Shihao Zeng, Tianrui Li, and Yihua Huang. FPGA-based high-throughput CNN hardware accelerator with high computing resource utilization ratio. *IEEE Transactions on Neural Networks and Learning Systems*, 33(8):4069–4083, 2021.
- [5] Xilinx and Inc. Vitis AI github repository. <https://github.com/Xilinx/Vitis-AI>, 2019.
- [6] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’17, pages 65–74. ACM, 2017.
- [7] Sumbal Malik, Manzoor Ahmed Khan, and Hesham El-Sayed. Collaborative autonomous driving—a survey of solution approaches and future challenges. *Sensors*, 21(11):3783, 2021.
- [8] Khan Muhammad, Amin Ullah, Jaime Lloret, Javier Del Ser, and Victor Hugo C de Albuquerque. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems*, 22(7):4316–4336, 2020.
- [9] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.
- [10] Abhishek Balasubramaniam and Sudeep Pasricha. Object detection in autonomous vehicles: Status and open challenges. *arXiv preprint arXiv:2201.07706*, 2022.
- [11] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, 31, 2021. doi:10.1007/s12525-021-00475-2.
- [12] Ran Wu, Xinmin Guo, Jian Du, and Junbao Li. Accelerating neural network inference on FPGA-based platforms—a survey. *Electronics (Switzerland)*, 10, 2021. doi:10.3390/electronics10091025.

- [13] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [14] Shaohua Qi, Xin Ning, Guowei Yang, Liping Zhang, Peng Long, Weiwei Cai, and Weijun Li. Review of multi-view 3D object recognition methods based on deep learning. *Displays*, 69:102053, 2021.
- [15] Jianxin Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 5(23):495, 2017.
- [16] Hamed Habibi Aghdam and Elnaz Jahani Heravi. Guide to convolutional neural networks. *New York, NY: Springer*, 10(978-973):51, 2017.
- [17] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*, pages 1–6. Ieee, 2017.
- [18] Arohan Ajit, Koustav Acharya, and Abhishek Samanta. A review of convolutional neural networks. In *2020 international conference on emerging trends in information technology and engineering (ic-ETITE)*, pages 1–5. IEEE, 2020.
- [19] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.
- [20] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.
- [21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [22] You Li and Javier Ibanez-Guzman. LiDAR for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems. *IEEE Signal Processing Magazine*, 37(4):50–61, 2020.
- [23] Felix Berens, Stefan Elser, and Markus Reischl. Genetic algorithm for the optimal LiDAR sensor configuration on a vehicle. *IEEE Sensors Journal*, 22(3):2735–2743, 2021.
- [24] Jiageng Mao, Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. 3D object detection for autonomous driving: A comprehensive survey. *International Journal of Computer Vision*, pages 1–55, 2023.
- [25] Duarte Fernandes, António Silva, Rafael Névoa, Cláudia Simões, Dibet Gonzalez, Miguel Guevara, Paulo Novais, João Monteiro, and Pedro Melo-Pinto. Point-cloud based 3D object detection and classification methods for self-driving applications: A survey and taxonomy. *Information Fusion*, 68:161–191, 2021. doi:<https://doi.org/10.1016/j.inffus.2020.11.002>.
- [26] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3D classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

- [27] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3D object proposal generation and detection from point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 770–779, 2019.
- [28] Zetong Yang, Yanan Sun, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Ipod: Intensive point-based object detector for point cloud. *arXiv preprint arXiv:1812.05276*, 2018.
- [29] Rui Qian, Xin Lai, and Xirong Li. 3D object detection for autonomous driving: A survey. *Pattern Recognition*, 130:108796, 2022. doi:<https://doi.org/10.1016/j.patcog.2022.108796>.
- [30] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12697–12705, 2019.
- [31] Jiajun Deng, Shaoshuai Shi, Peiwei Li, Wengang Zhou, Yanyong Zhang, and Houqiang Li. Voxel R-CNN: Towards high performance voxel-based 3D object detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 1201–1209, 2021.
- [32] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3D object detection and tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11784–11793, 2021.
- [33] Yan Yan, Yuxing Mao, and Bo Li. Second: Sparsely embedded convolutional detection. *Sensors*, 18(10):3337, 2018.
- [34] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. Pv-rcnn: Point-voxel feature set abstraction for 3D object detection. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10526–10535, 2020. doi:[10.1109/CVPR42600.2020.01054](https://doi.org/10.1109/CVPR42600.2020.01054).
- [35] Chenhang He, Hui Zeng, Jianqiang Huang, Xian-Sheng Hua, and Lei Zhang. Structure aware single-stage 3D object detection from point cloud. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11870–11879, 2020. doi:[10.1109/CVPR42600.2020.01189](https://doi.org/10.1109/CVPR42600.2020.01189).
- [36] Zetong Yang, Yanan Sun, Shu Liu, Xiaoyong Shen, and Jiaya Jia. Std: Sparse-to-dense 3D object detector for point cloud. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1951–1960, 2019. doi:[10.1109/ICCV.2019.00204](https://doi.org/10.1109/ICCV.2019.00204).
- [37] Shubham Gandhare and B Karthikeyan. Survey on FPGA architecture and recent applications. In *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pages 1–4. IEEE, 2019.
- [38] Andrew Boutros and Vaughn Betz. FPGA architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.
- [39] Cong Hao and Deming Chen. Deep neural network model and FPGA accelerator co-design: Opportunities and challenges. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–4. IEEE, 2018.

- [40] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 84–93, 2019.
- [41] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. Xilinx first 7nm device: Versal AI core (vc1902). In *Hot Chips Symposium*, pages 1–28, 2019.
- [42] Xilinx and Inc. Versal, the first adaptive compute acceleration platform (ACAP) (wp505), 2020. URL: <https://docs.xilinx.com/v/u/en-US/wp505-versal-acap>.
- [43] Xilinx and Inc. Xilinx AI engines and their applications (wp506). URL: <https://docs.xilinx.com/v/u/en-US/wp506-ai-engine>.
- [44] Xilinx and Inc. Versal ACAP DSP engine architecture manual AM004 (v1.2.1), 2022. URL: <https://docs.xilinx.com/r/en-US/am004-versal-dsp-engine/Introduction-to-Versal-ACAP>.
- [45] Xilinx and Inc. Versal ACAP technical reference manual. URL: <https://docs.xilinx.com/r/en-US/am011-versal-acap-trm/Introduction-to-Versal-ACAP>.
- [46] Xilinx and Inc. Versal architecture and product data sheet: Overview (ds950). URL: <https://docs.xilinx.com/v/u/en-US/ds950-versal-overview>.
- [47] Xilinx and Inc. Vitis AI user guide UG1414 (v3.0), 2023. URL: <https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Vitis-AI-Overview>.
- [48] Mikhail Lebedev and Pavel Belecky. A survey of open-source tools for FPGA-based inference of artificial neural networks. In *2021 Ivannikov Memorial Workshop (IVMEM)*, pages 50–56. IEEE, 2021.
- [49] Xilinx and Inc. Vitis AI. URL: <https://xilinx.github.io/Vitis-AI/index.html>.
- [50] Xilinx and Inc. Dpucvdx8g for Versal ACAPs product guide. URL: <https://docs.xilinx.com/r/en-US/pg389-dpucvdx8g/Core-Overview>.
- [51] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. *arXiv preprint arXiv:1511.06488*, 2015.
- [52] Xilinx and Inc. Vitis AI library user guide UG1354 (v3.0), 2023. URL: <https://docs.xilinx.com/r/en-US/ug1354-xilinx-ai-sdk/Introduction>.
- [53] Xilinx and Inc. Versal AI Core Series VCK190 Evaluation Kit. Accessed on June 12, 2023. URL: <https://www.xilinx.com/products/boards-and-kits/vck190.html>.
- [54] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. SSD: Single shot multibox detector. In *Computer Vision—ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016.
- [55] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

- [56] Xiaopei Wu, Liang Peng, Honghui Yang, Liang Xie, Chenxi Huang, Chengqi Deng, Haifeng Liu, and Deng Cai. Sparse fuse dense: Towards high quality 3D detection with depth completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5418–5427, 2022.
- [57] Hai Wu, Chenglu Wen, Wei Li, Xin Li, Ruigang Yang, and Cheng Wang. Transformation-equivariant 3D object detection for autonomous driving. *arXiv preprint arXiv:2211.11962*, 2022.
- [58] OpenPCDet Development Team. OpenPCDet: An open-source toolbox for 3D object detection from point clouds. <https://github.com/open-mmlab/OpenPCDet>, 2020.