HARDWARE IMPLEMENTATION OF 3D DATA PROCESSING USING DEEP NEURAL NETWORKS

by

Muhammed Yasin Adıyaman

B.S., Electronics and Communication Engineering, Istanbul Technical University, 2016

M.S., Electronics Engineering, Istanbul Technical University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Electrical and Electronics Engineering

Boğaziçi University

2025

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, Assist. Prof. Faik Başkaya, for his invaluable guidance and unwavering support throughout my PhD journey. His insightful suggestions, constant encouragement, and profound expertise have been pivotal in shaping this thesis. It is no exaggeration to say that this work would not have been possible without his dedicated mentorship and patience.

I am deeply grateful to Prof. Sıddıka Berna Örs Yalçın and Prof. Burak Acar for their invaluable comments and suggestions, which greatly enhanced the progress and refinement of this thesis. My sincere thanks also extend to Prof. Günhan Dündar, whose belief in me and support throughout this journey have been a tremendous source of motivation, and to Prof. H. Fatih Uğurdağ for their constructive feedback and participation in the thesis defense committee.

Furthermore, I extend my heartfelt thanks to the faculty members and staff of the Department of Electrical and Electronics Engineering at Boğaziçi University for their continuous support throughout my academic journey.

Lastly, I owe an immeasurable debt of gratitude to my mother, father, my entire family, and my friends for their unwavering love, encouragement, and support. Their belief in me has been a source of strength and inspiration, for which I am infinitely thankful.

ABSTRACT

HARDWARE IMPLEMENTATION OF 3D DATA PROCESSING USING DEEP NEURAL NETWORKS

Over the past decade, the rapid development of artificial intelligence technologies has led to deep neural networks becoming an integral part of many data processing tasks across various domains. DNNs have transformed the data processing field by automatically learning representations and features directly from the data, outperforming traditional methods, which are based on mathematical modeling with hand-crafted features. One of the areas that have seen the most significant advancements in computer vision since vision data is complex and hard to process with classical hand-crafted methods. In addition to 2D image processing, deep neural networks have proven effective for 3D data processing, which is critical for applications such as autonomous driving, robotics, and virtual reality, which rely on accurate 3D perception to understand spatial relationships and object detection in dynamic environments. In this work, we studied deep neural networks for 3D data processing, especially for point cloud data generated by LiDAR sensors. We also addressed the challenges of point cloud processing, such as irregular structure and high dimensionality. To overcome these issues, we explored 3D scene understanding through 2D image data, specifically tackling the depth estimation and semantic segmentation problems. We also explored fusion methods to effectively leverage the information coming from different modalities. Furthermore, recognizing edge devices' power and latency constraints, we investigated effective hardware implementation and acceleration techniques for designed neural networks.

ÖZET

DERİN ÖĞRENME ALGORİTMALARI İLE 3 BOYUTLU VERİ İŞLEME VE DONANIM GERÇEKLEMESİ

Son on yılda, yapay zeka teknolojilerinin hızla gelişmesi, derin sinir ağlarının birçok veri işleme görevinin ayrılmaz bir parçası haline gelmesine yol açmıştır. Derin sinir ağları, belirli fonksiyonlar ile oluşturulmuş özelliklere dayanan matematiksel modelleme yöntemleri yerine, verilerin yorumlanmasını veriden doğrudan ve otomatik olarak öğrenerek, klasik modellere göre çok daha yüksek performans göstermiş ve bu sayede veri işleme alanını büyük ölçüde etkilemiştir. En büyük ilerlemelerin kaydedildiği alanlardan biri, bilgisayarlı görü alanı olmuştur. Bunun en büyük sebebi görsel verinin karmaşık yapısı ve klasik yöntemlerle işlenmesinin zor olmasıdır. 2 boyutlu görüntü işlemenin yanı sıra, derin sinir ağları otonom sürüş, robotik ve sanal gerçeklik gibi uygulamalar için kritik olan 3 boyutlu veri işlemede de etkili olduğunu kanıtlamıştır. Bu uygulamaların başarısı, değişken ortamlardaki nesnelerin doğru bir şekilde algılanması ve uzamsal ilişkilerin anlaşılması ile mümkündür. Bu çalışmada, özellikle LiDAR sensörleri tarafından üretilen nokta bulutu verileri olmak üzere, 3 boyutlu veri işleme için geliştirilmiş derin sinir ağlarını inceledik. Nokta bulutu işlerken karşılaşılan düzensiz yapı ve yüksek boyutluluk gibi zorluklara da değindik. Bu sorunların üstesinden gelmek için, derinlik tahmini sorununa odaklanarak 2 boyutlu görüntü verileri üzerinden 3 boyutlu sahne anlama yöntemlerini araştırdık. Bununla birlikte, üretilen modellerin kullanılacağı uç cihazların güç ve gecikme süresi gibi kısıtlarını göz önünde bulundurarak, tasarladığımız sinir ağlarının donanımda etkili bir şekilde uygulanması ve hızlandırılması için kullanabileceğimiz yöntemleri araştırdık.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

b	Bias term in neural network layers
C_t	Cell state in LSTM
$\tilde{C}_t$	Cell candidate in LSTM
d_k	Dimensionality of key vectors in attention
f	Activation function
f_t	Forget gate in LSTM
h_t	Hidden state in LSTM
i_t	Input gate in LSTM
K	Key matrix in the self-attention mechanism
P	Random mask used in dropout
Q	Query matrix in the self-attention mechanism
V	Value matrix in the self-attention mechanism
W	Filter or weight matrix in various layers
x	Input vector or data
y	Output of a neural network layer
δ	Threshold used in depth estimation metrics
ϵ	Small constant for numerical stability
μ	Mean value for normalization
σ	Standard deviation or variance for normalization

LIST OF ACRONYMS/ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
AI	Artificial Intelligence
APoZ	Average Percentage of Zeros
ASIC	Application Specific Integrated Circuit
BEV	Bird's Eye View
BN	Batch Normalization
CNN	Convolutional Neural Network
DLA	Deep Learning Accelerator
DPU	Deep Learning Processing Unit
DSP	Digital Signal Processing
FP	False Positive
FPGA	Field Programmable Gate Array
FPR	False Positive Rate
FN	False Negative
GELU	Gaussian Error Linear Unit
GCN	Graph Convolutional Network
GPS	Global Positioning System
HDL	Hardware Description Language
HLS	High-Level Synthesis
IoU	Intersection over Union
LiDAR	Light Detection and Ranging
LN	Layer Normalization
mIoU	Mean Intersection over Union
MLP	Multilayer Perceptron
PL	Programmable Logic
QAT	Quantization Aware Training
RAM	Random Access Memory

RCNN	Region-based Convolutional Neural Network
REL	Relative Absolute Error
ReLU	Rectified Linear Unit
ResNet	Residual Network
RGB	Red Green Blue
RGBD	Red Green Blue Depth
RMS	Root Mean Square
RMSE	Root Mean Square Error
ROC	Receiver Operating Characteristic
RNN	Recurrent Neural Network
SIFT	Scale-Invariant Feature Transform
SLAM	Simultaneous Localization and Mapping
SOTA	State of the Art
SURF	Speeded Up Robust Features
TPU	Tensor Processing Unit
TPR	True Positive Rate
URAM	Ultra Random Access Memory
URAM	Block Random Access Memory
Voxel	Volumetric Pixel

1. INTRODUCTION

Intelligent machines have been one of the biggest dreams of humankind since the invention of the first computer. Deep neural networks are the most promising tool for achieving this goal. Just like many other tasks, we are now at the stage of handing over vision-related jobs too to machines due to the amazing development and success of deep neural networks, especially in the last decade. Most of the ideas currently being used in deep neural networks had been proposed before the 2000s. However, they could not be applied at those times due to a lack of computational power and data. Today, with the power of parallel computing devices like GPUs and TPUs and the constant increase in large amounts of multi-media data, we can feed power-and-data-hungry deep neural networks up to some extent. However, for end devices, power- and speed-related constraints have led us to seek specialized network architectures and efficient hardware implementation of those methods. In this study, we explored the hardware implementation of DNN-based 3D data processing algorithms.

In line with publication ethics and legal requirements, figures and visual content generated within the scope of this thesis—whose copyrights have been assigned to their respective publishers—are included in this document in accordance with the reuse policies outlined on the official websites of those publishers.

1.1. Motivation

Understanding the 3D structure of a scene is crucial for many tasks. For example, self-driving cars can not navigate properly without understanding nearly precise locations of traffic actors, including other cars and pedestrians. The same phenomenon is valid for any auto-pilot systems like self-navigating robots and autonomous drones. 3D object reconstruction is another important task related to scene understanding in terms of input-output relation but on a smaller scale. Keeping the speed of progress in this area, in the near future, it will be possible to generate digital avatars, which can

be fake or digital copies of real humans. Generating digital copies of humans can open a new era in computer games, film industry, and digital marketing. For example, one can try different clothes online or play with his/her own character in computer games. Despite its importance, a sufficient 3D scene or object reconstruction is not readily available after sensor measurements. It is needed to process sensor measurements to extract 3D information. There are two main sensor modalities to attain vision data: LIDAR and camera. Depth information is lost when constructing images due to planar projection. Extracting depth information directly from 2D images is an ill-posed problem even when using stereo cameras. On the other hand, the LIDAR point cloud is sparse and noisy. Both of the sensor modalities (both stereo cameras and LIDAR) provide data that contain 3D shape information. However, both stereo images and point clouds need to be processed to understand the 3D representation of a scene or object and make use of it.

1.2. Contribution

In this thesis, our primary focus is on the implementation of DNN-based methods for 3D visual data processing and reconstruction on FPGA. The main contribution is the successful deployment of both pre-trained models and custom models for various 3D vision tasks, enabling real-time operation on FPGA. We focus on two different tasks for edge deployment: depth reconstruction from 2D images and semantic segmentation from single sweep point cloud data. For each task, we trained several models and chose the best-performing ones to quantize, optimize, and deploy on our FPGA board. For depth estimation on edge:

- We used a pre-trained MiDaSNet model
- We customize the model for edge implementation
- We quantize the model down to a fixed-point 8-bit float representation
- According to the test results, our model achieves SOTA results in the NYUv2 validation dataset compared to the baseline of FPGA-based studies

For semantic segmentation task on edge:

- We selected SalsaNext architecture
- We customize it for edge deployment
- We trained it on the SemanticKITTI and Open Waymo Dataset
- We quantize and optimize it for edge deployment
- According to the test results, our model achieves SOTA results among FPGA based implementations

Alongside these edge deployment studies, we worked on some other algorithms for 3D data processing, especially in the biomedical field. However, due to the complexity of those models, they cannot be implemented on edge, and no further studies have been conducted on them.

1.3. Thesis Outline

The remaining sections of this thesis are structured as follows: In Chapter 2, we introduce challenges related to 3D data processing on edge and conduct a literature review.

In Chapter 3, we introduce different sensor modalities and methods for 3D data processing. Also, we introduce the basics of DNN models and evaluation metrics for different tasks. Also, we introduce common frameworks and summarize the current state of the art.

In Chapter 4, we introduce hardware optimization methods and frameworks commonly used for edge deployments like Vitis-AI, OpenVINO, and HLS4ML. These frameworks are used for model quantization, optimization, and deployment on edge.

In Chapter 5, we present our study about real-time depth estimation on edge in which we utilize the CNN-based lightweight model MiDaSNet and achieve SOTA

results compared to the baseline.

In Chapter 6, we present our study about real-time 3D semantic segmentation from range images using a customized SalsaNext model. According to the results, we achieved SOTA results compared to other FPGA-based implementations.

2. BACKGROUND

The classical method for 3D data processing was based on hand-crafting feature extraction algorithms like SIFT, SURF, or structural features derived by well-parameterized specific functions. However, vision data is too complex to be modeled with these hand-crafted features; thus, classical models were not able to generalize to complex real-world data. On the other hand, the complexity of vision tasks increases, too, which makes it impossible to deal with classical methods. For example, tasks like 3D reconstruction from partial views or tracking objects in dynamic environments require a higher level of abstraction and adaptability, which is not possible with classical methods. Given these factors and the growing success of AI methods, over the past decade, the majority of 3D vision data processing tasks—like most vision-related tasks—have increasingly been entrusted to deep neural networks (DNN).

DNNs have revolutionized 3D data processing by automating feature extraction and learning hierarchical representations directly from raw data. Thanks to increasing computational resources, large-scale datasets, and improved architectures, DNNs outperformed classical methods in a wide range of 3D vision tasks, including classification, reconstruction, detection, and segmentation tasks. These models generalize well to complex, real-world scenarios, and they can capture subtle variations in geometry, texture, or environmental context.

There are many DNN models proposed in the literature for 3D data processing. These methods can largely be classified into four main categories: 3D Convolutional Neural Networks (3D-CNNs), which operate on voxel grids; Graph Neural Networks (GNNs), designed for unstructured point clouds; adapted 2D methods, such as 2D CNNs applied to range images or bird's-eye view (BEV) images; and transformer-based models, which can process both structured and unstructured data. These models show remarkable success compared to the classical methods. Additionally, other techniques, such as multi-view methods, volumetric slicing, and multimodal learning, are also

employed for 3D data processing or representation.

However, the high computational demands of these models limit their deployment on edge devices. As a result, large and complex models are not suitable and can not be deployed for edge applications. Instead, lightweight and optimized models must be used, which leads to an engineering challenge in balancing trade-offs between power consumption, performance, latency, and other constraints. Achieving the right balance requires careful consideration of hardware capabilities, model complexity, and application requirements to ensure efficient deployment in resource-constrained environments.

FPGAs are promising hardware platforms for optimal deployment, offering customizable hardware solutions alongside software-based optimizations. Their flexibility allows them to be tailored for either acceleration, power efficiency, or both. Additionally, the optimizations developed for FPGAs can often be transferred to ASIC implementations, enabling highly specialized architectures for specific operations or models. This makes FPGAs an ideal candidate for real-time and/or low-power deployment in edge applications. There are many techniques to optimize models for FPGA deployment. Among them, quantization, pruning, and pipeline optimization can be utilized to reduce resource consumption without significantly compromising accuracy. These methods are critical for enhancing the efficiency of deep learning models on FPGAs and will be thoroughly explained in Chapter 4.

Typically, the background chapter would be concise and unified, but due to the multidisciplinary nature of our approach, we covered DNNs, 3D methods, and hardware optimization in 3 separate chapters. In this chapter, we will outline the fundamentals of DNN models, the key tasks they address, the evaluation metrics used for these tasks, and the current state-of-the-art solutions for FPGA-based 3D signal processing methods, which is the final task we focus on in this thesis.

2.1. Deep Neural Networks

Many types of deep neural networks have been proposed in the literature. However, almost all of them use some basic building blocks to generate more complex structures. In this section, the basic building blocks of DNNs are explained briefly.

2.1.1. Basic Layers

There are several common types of layers and methods used in DNNs, including convolutional, dense, and pooling layers, along with dropout, regularization, and activation functions. In this section, we briefly describe these layers and techniques.

2.1.1.1. Convolution Layer. A convolution operation is used to extract information from the neighborhood. It can be in the form of 1D (e.g. for speech recognition), 2D, 3D, or 4D (e.g. for action recognition from 3D video). Parameter size for this layer is dependent on kernel sizes and number of input-output channels. CNN applies the following operation:

$$y = f(W * x + b) \tag{2.1}$$

where $W * x$ denotes the convolution operation, W is the filter, x is the input, b is the bias, and f is the activation function.

2.1.1.2. Dense Layer. It is a classical all-to-all connected layer. Assume in the first layer we have n nodes, and in the second layer we have m nodes. Then, there are nxm connections. Dense layers are generally used at the output layer or at the layers close to the output. Mathematically, this can be expressed as:

$$y = f(Wx + b) \tag{2.2}$$

where W is the weight matrix, x is the input vector, b is the bias term, and f is the activation function.

2.1.1.3. Activation. Activation functions are used to introduce nonlinearity to the network, which is essential for learning. Depending on the layer, there are several common activation functions: sigmoid, tanh, ReLU, and softmax. Generally, at the output, ReLU is used in case of regression or sigmoid or softmax is used in case of classification. On the middle layers, generally, ReLU and GELU are used to avoid vanishing gradients in backpropagation. On the other hand, to avoid exploding gradients, ReLU6 can be used instead of ReLU. The mathematical expressions for ReLU, ReLU6, sigmoid, and tanh are given below:

$$\begin{aligned} \mathrm{ReLU}(x) &= \max(0, x) && (2.3)\\ \mathrm{ReLU6}(x) &= \min(6, \max(0, x)) && (2.4)\\ \sigma(x) &= \frac{1}{1 + e^{-x}} && (2.5)\\ \tanh(x) &= \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. && (2.6) \end{aligned}$$

2.1.1.4. Dropout. To avoid overfitting, dropout is used. It simply applies a mask on the previous layer output by zeroing some (probably random) elements. Dropout applies as follows:

$$y = Px \quad (2.7)$$

where P is the random mask (consisting of zeros and ones) shaped the same as the output neural vector, and f is the activation function. During inference, all neurons are used, and their activations are scaled by $mean(P)$.

2.1.1.5. Normalization. Batch normalization(BN), which normalizes features on batch dimension, is commonly used in NNs to avoid overfitting. Alongside BNs, recently,

layer normalization (LN) has gained more popularity and is used in transformer-based architectures and CNNs. Normalization applies as:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{2.8}$$

where μ is the mean, σ is the variance, and ϵ is a small constant for numerical stability.

2.1.1.6. Pooling. Pooling layers are used to quantize off non-relevant information and generally to reduce feature dimensions. Two main types are average pooling and max pooling. It is applied like convolution except that instead of using some learnable kernels for calculation, average or a maximum of the current receptive field (whose size is kernel size) is calculated for the next layer as:

$$y = \max(x) \tag{2.9}$$

$$y = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{2.10}$$

2.1.1.7. Attention. The attention mechanism is used to filter features, mostly channels. Applying the gating mechanism, relevant features are expected to pass to subsequent layers while nonrelevant features are filtered out. Currently, almost all SOTA models use attention mechanisms in both CNN-based and transformer-based architectures. The general attention mechanism applies as:

$$\text{Attention}(x) = \text{softmax}\left(g\left(x\right)\right) f\left(x\right) \tag{2.11}$$

where g and f represents feedforward networks for attention and main datapath.

2.1.2. Complex Layers

Alongside the basic blocks, there are some complex building blocks commonly used in neural networks: residual layer, LSTM, and transformer block.

2.1.2.1. Residual Layer. A residual layer is a fundamental building block of deep neural networks, introduced as part of the ResNet architecture. It uses skip connections between layers to address the vanishing gradient problem by ensuring that gradients flow through the network more easily during backpropagation, enabling the training of much deeper networks. The simplest residual connection is mathematically represented as:

$$y = F(x) + x \tag{2.12}$$

where x is the input, $F(x)$ is the transformation applied by the layers in the block, and y is the output.

2.1.2.2. LSTM. Long Short-Term Memory (LSTM) is designed to learn and capture long-term dependencies in sequence data by utilizing memory cells that can store information over long periods. These cells are regulated by gates (input, forget, and output gates) that control the flow of information into, through, and out of the cell. The LSTM is particularly effective for processing sequential data of varying lengths, such as in time-series forecasting, natural language processing, and speech recognition. However, it can also be adapted for various tasks by slicing input features, provided that slices adequately represent the information needed for output generation.

The structure of an LSTM cell is governed by the following key equations:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.13}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.14}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (2.15)$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \quad (2.16)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (2.17)$$

$$h_t = o_t \cdot \tanh(C_t) \quad (2.18)$$

where f_t, i_t, and o_t are the forget, input, and output gates, respectively, and C_t represents the cell state.

2.1.2.3. Transformer Block. Introduced in [1], it has revolutionized AI tasks, especially natural language processing models. Transformers use self-attention to allow each sequence element to attend to all other elements, enabling the model to efficiently capture local and global dependencies in the data. Besides, unlike RNN-based models such as LSTMs, transformers support parallel processing of sequence elements, leading to significantly faster training times. The core elements of the transformer are multi-head self-attention layers and feedforward layers, both of which are combined with layer normalization and residual connections.

The self-attention mechanism is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2.19)$$

where Q, K, and V represent the query, key, and value matrices, and d_k is the dimensionality of the key vectors.

2.2. Evaluation Metrics

Throughout the thesis, different evaluation metrics will be used for various tasks. In this subchapter, we will outline the commonly used evaluation metrics for classification, semantic segmentation, and depth estimation tasks.

2.2.1. Classification

In classification tasks, the goal is to assign a predefined category to each input sample. Common evaluation metrics for classification tasks are:

- *Accuracy*: Measures the ratio of correctly predicted labels to the total number of samples. It is defined as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.20}$$

  where TP are true positives, TN are true negatives, FP are false positives, and FN are false negatives.
- *Precision*: Represents the proportion of correctly predicted positive observations to all predicted positives:

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{2.21}$$

- *Recall (Sensitivity)*: Measures the proportion of correctly predicted positive observations to all actual positives:

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{2.22}$$

- *F1 Score*: The harmonic mean of precision and recall, used when the balance between precision and recall is important:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{2.23}$$

- *Area Under the ROC Curve (AUC)*: For binary classification tasks, the network result is a single number, generally between 0 and 1. There should be a threshold to decide which class to assign, positive or negative. For different predefined threshold values, the numbers TP, FP, TN, and FN and their ratios will change.

The Receiver Operating Characteristic (ROC) curve is a plot of True Positive Rate (TPR) vs False Positive Rate (FPR) where both values are dependent on threshold θ and can be calculated as:

$$\mathrm{TPR}(\theta) = \frac{\mathrm{TP}(\theta)}{\mathrm{TP}(\theta) + \mathrm{FN}(\theta)} \tag{2.24}$$

$$\mathrm{FPR}(\theta) = \frac{\mathrm{FP}(\theta)}{\mathrm{FP}(\theta) + \mathrm{TN}(\theta)} \tag{2.25}$$

and AUC is the area under the ROC curve as:

$$\mathrm{AUC} = \int_0^1 \mathrm{TPR}(\mathrm{FPR})\, d(\mathrm{FPR}) \tag{2.26}$$

which is between 0 and 1 and ideally close to 1.

2.2.2. Semantic Segmentation

Semantic segmentation aims to assign a class label to each pixel in an image. Common evaluation metrics include:

- *Mean Intersection over Union (mIoU)*: The IoU for each class is calculated by dividing the intersection between the predicted positive and the ground truth positive labels by the union of the two. The mean IoU is the average over all classes:

$$\mathrm{IoU} = \frac{|\mathrm{Prediction} \cap \mathrm{Ground\ Truth}|}{|\mathrm{Prediction} \cup \mathrm{Ground\ Truth}|} \tag{2.27}$$

$$\mathrm{mIoU} = \frac{1}{N} \sum_{i=1}^{N} \mathrm{IoU}_i \tag{2.28}$$

where N is the number of classes.

- *Dice Coefficient*: For binary classification, the most commonly used metric is the Dice coefficient, especially for imbalanced data, and it can be generalized to multi-class segmentation problems too. For a single class, the formula is:

$$\text{Dice} = \frac{2 \cdot |\text{Prediction} \cap \text{Ground Truth}|}{|\text{Prediction}| + |\text{Ground Truth}|}. \tag{2.29}$$

2.2.3. Depth Estimation

Depth estimation involves predicting the distance from the camera to objects in the scene. Common evaluation metrics include:

- *Root Mean Squared Error (RMSE)*: Measures the difference between the predicted depth map and the ground truth depth map as:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(d_i - \hat{d}_i)^2} \tag{2.30}$$

  where d_i is the ground truth depth and $\hat{d}_i$ is the predicted depth.
- *Mean Absolute Error (MAE)*: The average absolute error between predicted depth and ground truth depth:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|d_i - \hat{d}_i|. \tag{2.31}$$

- *Relative Absolute Error (REL)*: Measures the relative error between predicted and ground truth depths:

$$\text{REL} = \frac{1}{N}\sum_{i=1}^{N}\frac{|d_i - \hat{d}_i|}{d_i}. \tag{2.32}$$

- *Threshold Accuracy (δ)*: Measures the percentage of predicted depth values that fall within a threshold of the ground truth. It is commonly expressed as:

$$\text{Threshold Accuracy} = \text{Percentage of pixels where } \max\left(\frac{d_i}{\hat{d}_i}, \frac{\hat{d}_i}{d_i}\right) < \delta k \quad (2.33)$$

  for thresholds $\delta 1 = 1.25$, $\delta 2 = 1.25^2$, and $\delta 3 = 1.25^3$.

2.3. Challenges

Training, optimization, deployment, and both pre-and post-processing of 3D data models for edge deployment present numerous challenges. Several of these challenges have been addressed in various works discussed throughout this thesis. A summary of key challenges is provided below:

- Neural networks are slowly learning and data-hungry models. For instance, to teach a DNN to recognize the dynamics of rare events like car accidents, it needs to be trained on numerous examples from a variety of environmental conditions. Since car accidents are infrequent and generating such data in the real world is expensive and impractical, one possible solution is to use virtual data from simulation environments. However, the data distribution in simulations differs from that in the real world, leading to a domain shift problem. There are several domain adaptation techniques, but none have fully bridged the gap between real-world and simulation data. As a result, the performance of a DNN is constrained by the dataset it is trained on, and when tested on data from a different environment or context, it can become unstable.
- Another important challenge is power efficiency on edge devices. Power efficiency is closely tied to memory footprint, model size, and the number of multiply-and-accumulate (MAC) operations. Reducing model size and complexity can help address this issue, but doing so without compromising accuracy is difficult.
- For many perception problems like detection, segmentation, and navigation, there is a need to evaluate all data channels from multiple sensor modalities. Sensor

fusion is a common approach to evaluating all information channels. However, the data types are different and cannot be easily concatenated. Since sensors are positioned in different locations with varying perspectives, both spatial and temporal alignment issues can occur, which may degrade perception accuracy.

- Sparsity and irregularity, especially in 3D data, can add complexity and increase power consumption, memory usage, and latency. Methods such as projecting 3D data onto a plane to reduce dimensionality, downsampling, and quantization or using sparse convolutions can help mitigate these issues. However, applying these techniques without incurring information loss or performance degradation can be challenging.
- Another problem is the computational resources needed to train neural networks. Experimenting with different models, configurations, and hyperparameters demands significant computational power. While some neural iterative approaches are proposed in the literature, where neural networks are used to search for efficient training hyperparameters, these methods can only approximate near-optimal solutions. Achieving truly optimal solutions remains difficult because neural networks are highly nonlinear and cannot be accurately modeled mathematically.
- One potential trade-off exists between interoperability and compactness. Compactness, often achieved through end-to-end approaches, is preferred because it requires less parameter tuning and minimal pre- and post-processing. Conversely, generating intermediate representations allows for the integration of prior knowledge, which can be advantageous.

2.4. Literature Review

There are many works exploring hardware implementation of DNN models for 3D data processing. To mention a few, in [2], FPGA implementation of DNN on LIDAR data is proposed for point cloud segmentation and classification. In this study, point set representation is used to avoid any information loss due to any type of preprocessing or subsampling. In [3], FPGA implementation of VoxelNet architecture

is proposed. In [4], for 3D point cloud semantic segmentation (PCSS) on mobile devices, a graph convolutional network (GCN) based PCSS processor is proposed. In this work, when input is point cloud with 4K resolution, the achievable speed is 54.7 fps, which is higher than the refresh rate of most available LIDAR sensors. However, the results are from simulation, and there is no physical realization. In [5], for drivable region segmentation, an FPGA implementation of a convolutional neural network called ChipNet is proposed. In this work, input is real-time LIDAR data which is mapped to spherical coordinates, and a dense input tensor is generated for CNN input. The proposed architecture works on FPGA with a refresh rate of 17.59 ms. According to test results, FPGA implementation is 43 times faster than CPU and 13 times faster than GPU. There are many studies on 2D image processing on FPGA, too. Among them, [6] proposes an acceleration method based on blocked Winograd-GEMM architecture. The method is applied to the ResNet-18 model implemented on FPGA. The clock frequency of the used FPGA is 200MHz and 383GOPS rate is achieved. On the software side, there are many 3D processing networks proposed in the literature. In [7], PointRCNN is proposed for object detection in two stages from raw point cloud data. In [8], SquueezeSegV2 is proposed. In this work, training 3D data is obtained using a GTA-V simulator. To solve the domain shift problem, domain-adaptation techniques are applied. The training pipeline, proposed in this work, consists of three major components: 1) learned intensity rendering, 2) geodesic correlation alignment, and 3) progressive domain calibration.

3. 3D DATA PROCESSING

3.1. Sensor Modalities

3D data can be obtained from different input devices such as LIDAR, camera, and RADAR, each of which generates different types of data. Besides, one can use prior knowledge about the environment from high-definition maps or temporal data.

3.1.1. LIDAR

LIDAR generates a point cloud where each point is assigned with 3D coordinates in the LIDAR frame. To calculate ranges, LIDAR uses the time of flight information with knowledge of azimuth and elevation angles. Raw data generated by LIDAR does not only consist of the time of flight but also generates the intensity of the returning laser beam. The intensity data can be used to understand the opacity and reflectance of the materials or to detect outliers. To exploit this data, it can be concatenated easily as a separate channel. The main drawbacks of LIDAR are sparsity, low spatial resolution, low frame rate, high cost, and noise. The amount of noise depends on reflected material properties and propagation medium. For example, if the weather is rainy, laser beams are affected by diffraction. Besides, LIDAR sensors only provide occupancy data without texture information. Although there are some RGBD cameras that give occupancy and RGB data together, these devices can be classified as active infrared cameras. Because they simply emit infrared light patterns and measure depth information using stereo infrared cameras. The cost of the devices is influenced by factors such as angular resolution, range, rotation frequency, field of view, precision, and power consumption. Device selection is determined by the specific conditions of use. In the case of self-driving vehicles, the required specifications often exceed those typically available. These requirements include an angular resolution of less than 0.1° in both horizontal and vertical axes, a range of over 180 meters, a vertical field of view exceeding 40°, a horizontal field of view greater than 120°, power consumption below

10W, and a range precision better than 5 cm.

3.1.1.1. Point Set. A point cloud (i.e., point set) is simply an irregular set of points with raw geometry information. The points can have also intensity data which can be directly obtained from sensor measurements. Besides, especially for indoor visualization, some devices like Microsoft Kinect are able to generate RGB data which can be added to point features.

The main drawbacks of the point set are *sparsity* and *irregularity*, especially for outdoor usage. Because the number of beams depends only on horizontal and vertical (or azimuth and elevation) angular resolution, it does not change with the represented volume. This results in *low spatial resolution*, which complicates object recognition for distant objects where there is only some noisy occupation information. Besides, laser beams sent out may not return, causing *missing points* for some angular pixels, which further contributes to sparsity. On the other hand, irregularity introduces complexity when processing point clouds. For instance, performing a convolution operation requires identifying neighboring points, a process that is computationally intensive due to the irregular nature of the 3D data.

3.1.1.2. Voxel Grid. Voxel grid representation solves the irregularity problem. It divides the 3D space into cubic voxel grid cells with the same dimensions and assigns an occupancy score (if there is a point 1, otherwise 0) to each grid. By doing this with limited overall volume, one can downsample point clouds, too.

Using voxel grids, one can efficiently arrange the data in memory and use it without requiring some complex clustering approaches. However, the main drawback of the voxel grid is sparsity since assigning occupancy information to the whole volume results in empty space generally dominating over occupied space. When using convolutional neural networks, this problem can be partially solved by utilizing a sparsity mask [9]. By determining sparsity in the first stage, the same mask can be used over the fol-

lowing stages as well. Another disadvantage, also related to sparsity, is the cubically growing data size with increasing resolution. Consequently, a trade-off exists between computational cost and resolution.

3.1.1.3. Range Image. By projecting a 3D point cloud onto the LIDAR sensor plane, a range image is generated. Each pixel in this 2D representation takes a depth value. Other than having missing laser beams, range images are dense. Since the resolution of a range image is directly related to angular distance both vertically and horizontally, cylindrical or cubic projection can be used to directly utilize the sensor output without scaling in one or both directions.

Range images can be combined with 2D image processing tools like 2D convolution. However, neighboring pixels often do not correlate (i.e., they do not belong to the same object), which reduces the efficacy of convolution. Moreover, prior information such as high-definition maps cannot be directly utilized, which is significant for self-driving or SLAM applications.

3.1.1.4. Bird Eye View. By projecting the point cloud onto the ground plane (i.e., looking from above like a bird), BEV images are generated. By discretizing the ground plane and quantizing height values, 2D images with height features for each pixel can be obtained. However, similar to voxel representation, empty space dominates, resulting in most pixels being missed, which can be assigned a value of zero (assuming ground).

BEV images enable the efficient use of 2D convolution since the 2D neighborhood in this case directly corresponds to the 3D neighborhood. The main challenge is the computation and memory footprint, which increases quadratically with resolution or area. Sparse convolution [9] can be applied to reduce the computation footprint to some extent.

3.1.2. Camera

The camera generates 2D images with RGB data. Compared to LIDAR, it is prevalent and inexpensive, providing rich texture information that is highly beneficial for object recognition. Cameras can operate at high frame rates, reaching up to 120Hz. Additionally, the abundance of 2D image data available can be utilized to train data-hungry deep neural networks. Using a stereo configuration, 3D information can also be obtained if accurate point correspondences are established. However, the main drawbacks include weak performance in low-light conditions and intrinsic errors caused by focal lenses, flare, and vignettes.

Infrared (IR) cameras, which are stable against changes in ambient light and darkness, offer an alternative. Two types of infrared cameras exist: passive and active. Active IR cameras are typically used indoors in stereo configurations, where an emitter projects IR patterns onto the scene, and the cameras extract 3D data based on local pattern shifts. However, active IR cameras are unsuitable for outdoor use due to noise introduced by ambient light and other IR illumination sources. In contrast, passive IR cameras can be used outdoors to differentiate objects based on temperature information, though they are expensive.

3.1.3. RADAR

Radar, an acronym for ‘radio detection and ranging‘, uses radio waves to measure range or velocity. Compared to LIDAR, it offers a significantly higher detection range and greater robustness to environmental changes. However, it produces sparser data with lower precision due to the use of radio waves instead of laser beams. On the other hand, radar is more cost-effective than LIDAR and provides real-time velocity information through the Doppler effect.

3.1.4. Ultrasonic Sensor

Ultrasonic sensor uses ultrasound frequencies (25kHz - 50kHz) to send chirps (ultrasonic pulses), and by calculating the time of flight from the phase of the returning signal, it determines the distance of the nearest object, which makes it very useful for small distances. However, it is sensitive to noise, and it only gives coarse geometry information consisting of just range without direction. By using a network of sensors, the direction can be determined for the nearest object [10] via trilateration (like in GPS). However, if there are multiple objects, reflections from each object affect each other, which prevents healthy measurements. Therefore, it works if there is only one near object. Nevertheless, ultrasonic sensors are the cheapest among all the abovementioned.

3.1.5. Sensor Fusion

Since each sensor has its own advantages, multiple sensors can be combined using sensor fusion techniques. LIDAR + camera is one of the most common combinations to merge both texture and depth information. Table 3.1 compares LIDAR and the camera.

Sensor fusion methods involve transforming one or more feature spaces into a target feature space. In the context of deep neural networks, this can be achieved at three stages: the input, the output, or within intermediate layers. At the output stage, decisions—such as classifications, segmentations, or detections—generated from the transformed data are combined with decisions from the target data. For instance, to classify an object using both LIDAR and camera data, classification can be performed separately on each data type, followed by an attention mechanism to merge the results, considering both sources for the final decision.

Another approach is to reframe one sensor's measurement in terms of another. For example, in a LIDAR and camera setup, 2D camera data can be transformed to

align with 3D LIDAR data. By concatenating the two, a unified input tensor can be generated for a 3D neural network.

Finally, sensor fusion can also occur at intermediate layers, where both inputs are preprocessed and one is transferred into the feature space of the other. This method appears more promising than the previous two, as it facilitates knowledge sharing during processing, enabling a smoother transition between feature spaces.

Table 3.1. LIDAR vs Camera Modalities.

	LIDAR	*CAMERA*
PROS	Gives direct 3D geometry data accurately	High resolution and rich texture information due to RGB channels
	Invariant to ambient changes in the visible spectrum	Cheap
	Returns intensity as a separate channel which gives extra information about the surface	High frame rate up to 120 fps
CONS	Sparsity: makes it hard to design input architecture	No direct 3D representation
	High cost compared to camera or other sensors (RADAR, ultrasound, GPS)	Stereo configuration is needed which adds complexity
	Weather-sensitive	Weak perception in dark
	A lower amount of data is available to process compared to 2D images	Intrinsic errors: Distortion due to focal lenses, flare, vignette

3.1.6. Prior Knowledge

In scenarios involving multiple actors communicating with each other, information about a shared environment can be transferred between them. For instance, high-definition maps used in self-driving cars are generated and updated using sensor data from multiple sources. Prior data can either be stored offline or updated online as needed.

To incorporate prior knowledge, sensor fusion methods—such as those described earlier—can be applied, depending on the type of prior information and its associated confidence score. An attention mechanism based on confidence scores can facilitate the sharing and utilization of knowledge between multiple actors.

In the context of shape reconstruction, prior shapes (similar to anchor boxes in object detection models like YOLO) can serve as references after classification, with deep neural networks used for fine-tuning. Knowledge sharing is also possible by employing the methods described earlier, combining data from different sensors, viewpoints, or temporal information.

3.2. Literature

3.2.1. Frameworks

In this section, the most used 3D data processing frameworks will be outlined.

3.2.1.1. Autoware.auto. Autoware.auto [11] is all-in-one open-source software for self-driving vehicles based on the ROS2 framework. It provides tools for perception and planning tasks. The classical LIDAR-based pipeline consists of several steps, which can be outlined as point cloud data acquisition from multiple LIDAR sensors - Transforming points and filtering - Fusing multiple clouds - Ground filtering - Object Detection -

Shape Extraction - Classification, and Tracking - Planning. Although Autoware.ai uses classical 3D data processing techniques for perception tasks in default settings, it supports DNN-based methods, too.

3.2.1.2. Torch-Geometric. Torch-geometric is the library for graphical neural networks [12]. Since point cloud (PC) data is sparse and can be represented as a graph, I can use this framework for PC processing. It supports many available convolution algorithms used in the literature.

3.2.1.3. OpenMMLab Frameworks. OpenMMLab is an open-source deep learning toolbox developed by the Multimedia Laboratory at CUHK. It provides a rich collection of modular and extensible frameworks tailored for computer vision tasks, including image classification, object detection, semantic segmentation, and 3D vision. These frameworks are built on PyTorch and offer standardized data pipelines, training procedures, and evaluation tools [13].

OpenPCDet is a point cloud processing framework written in Pytorch. It is currently supporting the most popular open datasets, including KITTI, Open Waymo, NuScenes, and Lyft datasets. According to the team's current plan, they are going to extend support for other important 3D datasets. They provide a unified coordinate system for all datasets before preprocessing [14]. The framework provides a flexible metamodel structure to support various models in the same framework. A template model consists of 4 stages: Backbone3D, Backbone2D, Dense Head, and ROI Head. Each stage nestles a few substages that can be used or passed by different architectures. On the other hand, the framework provides several benchmarks for different model-dataset combinations. Also, pre-trained models are available except for some (model, dataset) pairs due to copyright issues.

MMDetection is a Pytorch-based open-source project for detection. It can also be used for semantic, instance, and panoptic segmentation. It supports multiple standard

datasets, including Pascal VOC, COCO, CityScapes, LVIS. Moreover, it provides a flexible model structure that supports various models and some pre-trained models.

MMSegmentation is another powerful tool within the OpenMMLab ecosystem, designed specifically for semantic segmentation tasks. It supports multiple standard datasets, including Cityscapes, ADE20K, PASCAL VOC, and COCO-Stuff. Besides, it offers a wide array of backbone networks, such as ResNet, HRNet, and Swin Transformer, along with various segmentation heads, making it adaptable to a variety of tasks.

MMAction is another comprehensive toolkit within the OpenMMLab ecosystem designed for video action recognition. MMAction supports a wide variety of datasets, such as Kinetics, UCF101, and HMDB51, and offers pre-trained models, including 3D CNNs, SlowFast, TSM, and I3D, among others. The framework is optimized for both single-frame and multi-frame action recognition, allowing efficient handling of temporal information in video data. It also supports video classification, temporal action detection, and spatiotemporal localization tasks, making it a versatile tool for video understanding. We employed MMAction for a 3D classification task in the biomedical field. However, the model's complexity prevented its deployment on edge devices, and no further investigations were conducted.

There is a 3D version of MMDetection, named MMDetection3D. Built on top of the 2D version, it provides tools to handle 3D object detection tasks using point clouds, stereo images, and LiDAR data. It supports multiple datasets like KITTI, NuScenes, Waymo, and Lyft and multiple models, including PointPillars, SECOND, and others, with options to use pre-trained models for transfer learning or fine-tuning.

Most state-of-the-art models are readily accessible within the MMLAB ecosystem, offering customizable modular structures along with training and evaluation scripts and pre-trained weights. These MMLab frameworks have been utilized for various tasks presented in this thesis.

3.2.1.4. Pytorch3D. PyTorch3D is an open-source library developed by Facebook Research that provides a comprehensive suite of tools and differentiable modules designed to accelerate 3D deep learning research [15]. Built on top of PyTorch, it offers efficient, modular components for handling 3D data such as meshes, point clouds, and volumetric grids. PyTorch3D includes optimized implementations for common 3D operations like rendering, transformations, and sampling, enabling seamless integration of 3D geometry processing within deep learning pipelines. Its differentiable renderer supports gradient-based optimization, making it especially suitable for tasks like 3D reconstruction, shape analysis, and augmented reality applications. The library aims to streamline complex 3D workflows, improving both performance and ease of use for researchers and developers working with 3D data in machine learning contexts.

3.2.2. Methods

3.2.2.1. Classical Methods. There are many tasks that require 3D data processing, including point cloud classification, segmentation, object detection, tracking, and localization. In the classical approach, it is common to preprocess 3D data before complex tasks like classification or segmentation. The preprocessing consists of several steps, including

- Range and angle filtering to eliminate noisy points
- Fusing multiple point sets
- Downsampling point sets to reduce the volume
- Ray-based ground filtering that eliminates ground points according to the point sequence shape along virtual rays delivered from the sensors.

After preprocessing, detection, segmentation, and localization tasks can be realized using hand-crafting mathematical methods.

- Firstly, points must be clustered. The most used method is Euclidian clustering which groups points together according to the distance between them.

- Then, shape extraction is applied using some predefined shapes (like boxes) to fit them to the group of points. Principal Component Analysis (PCA) is used for alignment problems.
- For localization, normal distribution transform (NDT) is one of the most common methods in the classical approach. The NDT algorithm first divides the map into grid cells. The points inside cells are represented by a standard normal distribution with mean and variance. After that, the points are assigned to the nearest point group (grid cell) in NDT. Then, via optimizing the score function, two frames can be aligned to each other after several iterations.

3.2.2.2. DNN-Based Methods. It is worth mentioning that there is a significant transitivity between neural network architectures. For example, transformer networks were initially proposed for NLP tasks. However, currently, they are used in many tasks, including image processing, 3D data processing, etc, and achieve state-of-the art results in many benchmarks. That is to say; it was important to know SOTA models even if they are not currently used in the specific task I am currently working on.

Another model is "Sparse point-voxel convolution NAS", shortly SPVNAS [16]. It achieves some SOTA results in terms of 3D data processing. Simply, it combines voxel convolution with point-wise processing via MLP. The first one is responsible for extracting coarse features, while the latter is responsible for finding finer details.

One of the most used 3D environments is bird-eye-view representation. BEVFusion and BEVFormer networks [17,18] use BEV representation for 3D object detection. As an input, the former uses both multivision and LIDAR data, while the second one uses only multivision to obtain bounding boxes for detection in a 3D environment. In the Waymo challenge, both were state-of-the-art but for different tasks in terms of input setup, as explained above. BEVFusion model structure is given below. The main idea is to project LIDAR and camera features on BEV map and fuse them. Then the fused features are used to obtain 3D object detection. It is possible to use different heads to realize other tasks, too, like BEV-Map-segmentation. The BEVFormer model,

on the other hand, only uses camera data to extract 3D bounding boxes. The model is given below. It uses a spatiotemporal transformer, which exploits spatial features from multicamera images, and it uses previous state info via the temporal-self-attention mechanism.

4. NETWORK OPTIMIZATION FOR EDGE DEPLOYMENT

In this section, the two most used network optimization strategies for hardware-accelerated networks are explained. They are pruning and quantization.

Table 4.1. An overview of CNN acceleration strategies [19].

CNN Acceleration Methods			
Network Structure	*Network Optimization*	*Hardware Accelerator*	
		Platform	*Optimization*
Novel Components	Convolution Optimization	CPU	Lookup Table
Network Architecture Search	Factorization	GPU	Computation Reuse
Knowledge Distillation	Pruning	ASIC	Memory Optimization
	Quantization	FPGA	...

In Table 4.1, CNN acceleration methods are shown. These methods can be divided into three main categories: network structure optimization, network optimization, and hardware acceleration. Network structure optimization includes introducing novel components, performing network architecture search for high-level parameter estimation, and applying knowledge distillation. For network optimization, convolution operations can be optimized (such as using separable convolutions), parameters can be factorized by decomposing higher-rank tensors into lower-rank tensors, and techniques like pruning or quantization can be applied. This report focuses on pruning and quantization methods.

4.1. Pruning

Generally, neural networks are over-parameterized after training. Pruning can be necessary and effective under certain conditions:

- Some parts never function after training.

- Full output is not required, such as in transfer learning.
- Data or computational resources are insufficient to support the large models needed for operation or fine-tuning [20].

Finding optimal parameters for pruning is a challenging task. Applying brute force results in an NP-Hard problem, which is not feasible for DNNs containing millions of parameters [21]. To address this issue, parameter importance is typically estimated, and pruning is performed based on pre-defined thresholds. However, using a single pre-defined threshold can lead to issues, as it may not directly correspond to sparsity. Furthermore, different layers may exhibit varying sensitivity to pruning. Advanced and flexible threshold estimation mechanisms offer a solution to this challenge.

Several methods exist to score parameters for identifying the most prunable ones:

- Absolute values, such as L1 norms, can be used. This method is applicable to single parameters (e.g., specific weights), vectors (e.g., activations), or matrices (e.g., kernels). Its simplicity and effectiveness make it the most common approach in current literature.
- Euclidean norms, such as L2 norms, are applicable to vectors and matrices.
- The total number of nonzero elements in a vector or matrix, often referred to as the L0 norm, can serve as a pruning score.
- The average percentage of zeros (APoZ) can be used. For instance, neurons can be pruned based on their APoZ ratio for specific input data samples.
- Parameters can be ranked by their contribution to activations, with gradients being a useful metric for this purpose.
- Lastly, DNNs themselves can be employed to determine which parameters are prunable.

On the other hand, comparison between parameters represents another key decision point. Parameters can be compared either locally (layer-wise) or globally (network-wise). Pruning can take various forms and methods, which are often categorized into

different groups. Based on scheduling, pruning can be classified as static or dynamic.

Static pruning is performed offline prior to inference. The process begins by identifying the target prunable elements from the pre-trained network. These elements are then pruned, and the network undergoes retraining or fine-tuning in an iterative loop until the desired size or accuracy threshold is reached [19].

For dynamic pruning, the network is pruned during runtime [19]. This approach employs a flexible pruning strategy that generates pruning decisions within the loop. During runtime, the pruning operation is continuously applied. While this type of network offers better adaptability, it carries the risk of catastrophic inference (forgetting). Methods such as inverse-pruning have been proposed to address this issue, though the details are not covered here.

Another way to categorize pruning is based on the pruning targets. These can include weights, channels, activations, kernel filters, or entire layers. The level of sparsity achieved depends on the pruning scope—such as element-wise, channel-wise, shape-wise, filter-wise, or layer-wise—with sparsity generally increasing from left to right [19]. However, element-wise pruning is often avoided, as modern hardware is optimized for parallel computation, and removing individual weights rarely leads to meaningful gains in efficiency or speed.

Several other decisions need to be made during pruning. In terms of scheduling, pruning can be performed at once or iteratively. The first approach is simpler, while the latter generally yields better performance. For post-training, the network can be fine-tuned, which is the most common method, or rewound to an earlier stage, or even trained from scratch. Regarding data usage, pruning can be either data-agnostic or data-aware. Data-agnostic pruning does not use data directly (although it may rely on a loss function), whereas data-aware pruning incorporates data into the pruning process (as seen in the APoZ example above).

Recent pruning results for architectures such as VGG, ResNet, and MobileNet-v2 highlight key trends in model compression [22]:

- A tradeoff exists between accuracy and model size.
- Transitioning to more advanced architectures, such as EfficientNet, generally proves more effective, offering a better accuracy-to-model-size ratio.

It is worth noting that no pruning results found for EfficientNet. This can be attributed to two possible reasons: the network is already compact, making it unsuitable for pruning, or, as the newest model, no pruning studies have been conducted on it yet. In this study, detailed in Chapter 5, pruning was not applied to the model, as it is based on the EfficientNet-Lite3 architecture.

4.2. Quantization

Quantization is another widely used method for network optimization. In the earliest works, quantization combined with clustering and sharing methods for parameters reduced weight storage requirements by nearly four times. In the second stage of evolution, post-training was applied to the quantized model, leading to improved performance. At the third level, quantization was incorporated during training by enforcing parameters to align with one of the quantized levels. In the final stage, KL-divergence was utilized for quantization to optimize the distribution of quantization levels, achieving state-of-the-art (SOTA) results [19].

In the study on LIDAR semantic segmentation on-edge, detailed in Chapter 6, the third approach was applied to enhance the model's resilience to quantization noise. While this method slightly reduced quantization error by +0.1%, it did not improve the overall results, with mIoU decreasing by 2.6%. This decline can be attributed to the model being inherently designed to resist quantization errors. Consequently, quantization-aware training may not be effective for models already robust against quantization noise, as observed in this case. Further details are provided in Chapter 6.

Efficiency improvements achieved through quantization as reported in the literature. According to [19], FPGA implementations primarily occupy low-efficiency regions compared to other ASIC implementations. However, some exceptional works, such as the Arria GX1150, stand out on the graph. Before the adoption of ASIC, it was common practice to test algorithms and hardware implementations on FPGAs, highlighting their critical role.

Quantization offers numerous options. Regarding scheduling, it can be Quantization Aware Training (QAT), as described above. For grouping, quantization can be performed layer-wise or channel-wise. Finally, the resulting parameter length must be determined, with options ranging from binary to N-ary quantization.

4.3. Hardware Implementation

Deep neural networks are composed of similar basic building blocks connected sequentially. Additionally, any interlayer operation involves numerous parallelizable floating-point operations (FLOPs), such as multiply-accumulate operations. These characteristics make neural networks hardware-friendly when suitable architectures are applied.

The main challenges for hardware implementation include achieving high accuracy and speed, ensuring power and memory efficiency, maintaining low latency, providing flexibility for changing workloads, and enabling integration with existing software frameworks [23]. In terms of energy consumption, the cost of off-chip data transfer is orders of magnitude higher than other operations, as shown in Table 4.3.

Furthermore, the number of clock cycles required to read data from various parts of the chip varies significantly, as depicted in Table 4.2. Latency increases as data is accessed further from the center, progressing from CPU registers (1 kB) to L1 Cache (32 kB), L2 Cache (256 kB), L3 Cache (10 MB), and off-chip DRAM (10 GB). The latency in clock cycles for these memory levels is approximately 1, 3, 10, 40, and 200,

respectively [24] [25].

Table 4.2. Data read/write cost for CPU in 45nm process [25].

Memory Type	*Size*	*Speed (cycles)*	*Energy*
CPU Registers	1 KB	1	0.4 – 3.7 pJ
Cache (On-chip SRAM)	32 KB – 10 MB	2 – 40	10 – 100 pJ
Memory (Off-chip DRAM)	10 GB	200	1.3 – 2.6 nJ

Table 4.3. Energy consumption of basic operations for 45nm process [23].

Operation	*Energy (pJ)*
8-b Add	0.03
16-b Add	0.05
32-b Add	0.1
16-b Floating Point Add	0.4
32-b Floating Point Add	0.9
8-b Multiply	0.2
32-b Multiply	3.1
16-b Floating Point Multiply	1.1
32-b Floating Point Multiply	3.7
32-b SRAM Read (8 kB)	5
32-b DRAM Read	640

Neural network operations exhibit a high degree of parallelism, particularly due to the localized nature of computations in convolutional layers. A key metric for evaluating the efficiency of hardware implementations is the ratio of floating-point operations (FLOPs) to memory accesses. This ratio serves as an indicator of the data evaluation efficiency. As discussed in [24], the FLOP-to-memory access efficiency metric can be significantly enhanced for both convolutional neural networks (CNNs) and deep neural networks (DNNs) through the application of blocking techniques.

4.4. Frameworks for Accelerated Inference on Edge

4.4.1. Generic Frameworks

4.4.1.1. ONNX Runtime. ONNX Runtime is a high-performance inference engine designed for executing models in the ONNX format. It is framework-agnostic and supports a wide range of hardware backends. Being cross-platform, it allows seamless deployment of models originally developed in frameworks such as PyTorch and TensorFlow. ONNX Runtime enables model deployment across diverse environments, including CPUs, GPUs, and FPGAs, from vendors like Intel, Xilinx, and Apple.

4.4.1.2. TensorRT. TensorRT is an inference library built on CUDA and specifically optimized for NVIDIA GPUs, particularly those equipped with Tensor Cores. It supports models trained using all major frameworks, including TensorFlow and PyTorch, and is designed to deliver high throughput and low latency for deep learning inference. TensorRT is widely used in applications such as autonomous vehicles, robotics, and other real-time systems where latency is a critical factor. It supports multiple quantization formats, including FP32, FP16, and INT8, enabling efficient and flexible deployment.

4.4.1.3. OpenVINO. The Open Visual Inference and Neural Network Optimization (OpenVINO) toolkit is an open-source platform for optimizing and deploying AI models on Intel hardware, including CPUs, GPUs, VPUs, and FPGAs. Widely used in industrial and healthcare applications, OpenVINO provides numerous optimization techniques such as quantization, layer fusion, and graph pruning. It supports inference with FP32, FP16, and INT8 data types, enabling efficient and high-performance AI deployments.

4.4.1.4. NVDLA. The Nvidia Deep Learning Accelerator (NVDLA) is an open-source framework designed for optimized AI inference on various edge devices, including IoT

sensors, mobile devices, and other battery-powered systems. It provides hardware-based solutions for time and energy-efficient inference. Since NVDLA is implemented in Verilog, it can be deployed on FPGAs and further optimized by integrating deep learning acceleration architectures into custom ASICs and SoCs. It supports the most common quantization levels, FP16 and INT8. Thanks to its modular structure, NVDLA offers flexibility and ease of customization compared to other frameworks. For CNN models it includes optimization techniques such as sparse-weight convolution and batch convolution to reduce memory bandwidth usage. Additionally, it supports Winograd convolution, which can reduce the number of multiplications by approximately ~ 2.25 times for long convolutional kernels with high aspect ratios.

Table 4.4. Comparison of ONNX Runtime and TensorRT.

Feature	*ONNX Runtime*	*TensorRT*
Primary Hardware	CPU, GPU, FPGA, custom accelerators	NVIDIA GPUs
Optimization Target	General-purpose and cross-platform	NVIDIA GPU acceleration
Precision Support	FP32, FP16, INT8, INT4	FP32, FP16, INT8
Deployment Scale	Cloud, edge, IoT	Real-time systems, edge devices
Best For	Compatibility and flexibility	Performance-critical GPU inference
Toolchain	Framework-agnostic (supports ONNX)	NVIDIA-based workflows
Use Cases	Multi-platform AI inference pipelines	Real-time, GPU-optimized deployments

Table 4.5. Comparison of NVDLA and OpenVINO.

Feature	*NVDLA*	*OpenVINO*
Primary Hardware	Custom ASICs/SoCs	Intel CPUs, VPUs, FPGAs, GPUs
Optimization Target	Low-power AI on embedded systems	Intel hardware acceleration
Precision Support	INT8 and FP16 (hardware-limited)	FP32, FP16, INT8
Deployment Scale	Low-power, embedded applications	Edge AI, industrial IoT, cloud AI
Best For	Energy-efficient AI inference	High-performance inference on Intel
Toolchain	Custom implementation	Intel Distribution of OpenVINO Toolkit
Use Cases	Embedded AI for low-power devices	Industrial AI, vision, speech, healthcare

In Tables 4.4 and 4.5, a comparison of generic frameworks for accelerated inference is presented. Among these, NVDLA is predominantly utilized in custom ASICs and FPGAs, as its engine is implemented in Verilog, which is compatible with both platforms.

4.4.2. FPGA-Specific Frameworks

4.4.2.1. HLS4ML. High Level Synthesis for Machine Learning (HLS4ML) [26] is a tool developed for hardware acceleration of deep neural network (DNN) models, enabling the integration of machine learning models into FPGAs. This framework supports customizable parallelization schemes using compilation metaparameters such as the reuse factor, achieving inference rates in the range of MSps (million samples per second) for small networks with highly parallel designs.

In [27], the HLS4ML development team presented a method for real-time event reconstruction at the Large Hadron Collider in particle physics. They implemented a fully connected network with three hidden layers and 4389 parameters on a Xilinx Kintex Ultrascale FPGA. This implementation achieved latencies of approximately 75–150 ns at a clock frequency of 200 MHz while utilizing only about 10% of the available DSPs.

The advantages of HLS4ML are as follows:

- It is open-source.
- It provides tools for numerous commercial FPGA cards from various brands.
- Since the network can be fully mapped to programmable logic (PL), it achieves fast inference speeds at the level of MSps.

4.4.2.2. FPGA AI Suite IP. The FPGA AI Suite IP enables runtime kernel execution on FPGA fabric via AXI interfaces, with a focus on Intel FPGAs. To utilize this suite, an Architecture Description File (ADF) is required to configure the firmware for the programmable logic (PL). Furthermore, model weights and operations must be converted into an intermediate representation compatible with the OpenVINO framework.

The FPGA AI Suite compiler processes the ADF and the OpenVINO intermediate model to generate configuration instructions for creating the Deep Learning Accelerator (DLA) IP. This DLA IP can be integrated into any FPGA project. Once the DLA IP is included and mapped to the PL, a bitstream is generated, enabling the FPGA to be programmed.

Table 4.6. HLS4ML Project Current Status [26, 27].

Backend	*(Q)Keras*	*PyTorch*	*(Q)ONNX*	*Vivado HLS*	*Intel HLS*	*Vitis HLS*
MLP	supported	limited	in development	supported	supported	experimental
CNN	supported	limited	in development	supported	supported	experimental
RNN (LSTM)	supported	N/A	in development	supported	supported	N/A
GNN (GarNet)	supported	N/A	N/A	N/A	N/A	N/A

At runtime, the DLA IP utilizes the precompiled model to process input data efficiently. It employs optimized AXI interfaces to communicate with external memory or other FPGA components, ensuring seamless and high-performance operations.

The FPGA AI Suite currently supports the following functions and layers [28]:

- *Convolutional Layers:* 2D and 3D convolutions with filter sizes up to 28×28 for 2D and $28 \times 28 \times 14$ for 3D, strides up to 15, and padding up to 65,535.
- *Depthwise Convolution:* Similar parameters as 2D convolution.
- *Fully Connected Layers.*
- *Activation Functions:* ReLU, PReLU, Leaky ReLU, Sigmoid, Swish, Tanh, H-Sigmoid, H-Swish.
- *Pooling Layers:* Max pooling with window sizes up to $13 \times 13 \times 13$ and average pooling up to 27×27, with strides up to 4 and padding of 1 or 2.
- *Others:* Softmax (up to 4096 channels), elementwise multiplication, and operations like ChannelToSpace, DepthToSpace, and PixelShuffle.

4.4.2.3. Vitis AI. Vitis AI is an AI inference development platform designed for AMD devices, boards, and Alveo data center acceleration cards. It provides a full stack of software tools for training, optimizing, quantizing, compiling, and running models on the target platform.

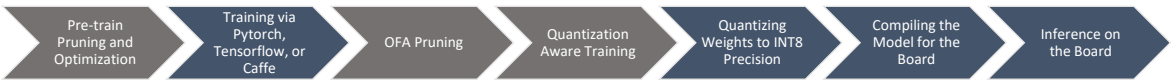

Figure 4.1. Vitis AI Workflow.

The Vitis AI workflow is similar to that of the FPGA AI Suite. To clarify the process, the flowchart in Figure 4.1 illustrates the steps, with mandatory steps highlighted in dark blue and optional steps in gray. The end-to-end workflow includes the following steps:

- *Pre-train Optimizations and Pruning:* Optimize the model before training by removing or replacing unsupported or inefficient layers and operations. Additionally, prune the model before training, which is generally more effective than post-training pruning.
- *Training the Model:* Train the model using one of the supported frameworks (PyTorch, TensorFlow, or Caffe).
- *OFA Pruning:* Optionally, apply Once-For-All (OFA) [29] pruning to the trained model. This method is supported by Vitis-AI, although it typically offers little advantage over pre-train pruning when targeting a single model variant.
- *Quantization Aware Training:* Optionally, fine-tune the trained model by incorporating quantization-aware training. In this step, fake quantization noise is injected during forward propagation to train the model against quantization error while backpropagation remains similar to that of training a floating-point model.
- *Quantizing into INT8 Precision:* Quantize the model weights to 8-bit fixed-point (INT8) precision. This process requires one or more input samples for calibration, converting the floating-point model into a quantized xmodel.
- *Compilation:* Recompile the quantized xmodel for the target DPU architecture

(e.g., using the DPUZCX8DG architecture file). This step uses a precompiled generic xmodel file and a DPU architecture file to generate an xmodel that contains DPU instructions and quantized weights.

- *Inference on the Board:* Before running inference, program the FPGA portion with the required prebuilt firmware provided by Vitis AI for common DPU topologies (such as DPU-3136 and DPU-4096). Activate the firmware using Vitis AI tools. Then, deploy and test the model on the board with the help of runtime libraries like VART and XIR, which manage data movement and the inference process.

Compared to the DLA used in the FPGA AI Suite, Vitis AI employs its counterpart, the Deep-Learning Processing Unit (DPU). The DPU is a specialized IP that can only be mapped to specific boards as it requires dedicated hardware resources in the programmable logic (PL). Compared to the FPGA AI Suite, which focuses on generating an IP core as an FPGA project with optimized configurations, Vitis AI integrates the entire workflow within its platform.

In our projects, we utilized the DPUCZDX8G architecture, which was well-suited for our hardware. The following list outlines the key operators supported by the Vitis AI DPUCZDX8G architecture [30]:

- *Convolution:* Supports both convolution and transposed convolution.
- *Depthwise Convolution:* Supports depthwise convolution and depthwise transposed convolution.
- *Pooling:* Max pooling and average pooling.
- *Activation Functions:* ReLU, ReLU6, Leaky ReLU, Hard Sigmoid, and Hard Swish.
- *Elementwise Operations:* Elementwise-sum and elementwise-multiply.
- *Other Operations:* Dilation, reorg, correlation in 1D and 2D, argmax and max along channel dimension, FCN layer, softmax, concat, and BN layer.

In Table 4.7, common models implemented using both the FPGA AI Suite and Vitis AI frameworks are compared in terms of inference speed on card and TOP-1 accuracy on the ImageNet dataset. For the FPGA AI Suite, results are provided for two predefined DLA architectures: AGX7-FP16-Performance and AGX-Performance-Giant on the Agilex 7 board. The former architecture prioritizes higher accuracy, while the latter emphasizes higher speed. In the table, when model names are the same, the first row corresponds to the most accurate results, and the second row represents the fastest results among the vendors' implementations.

Table 4.7. FPGA AI Suite vs Vitis AI.

Model	*FPGA AI Suite*		*Vitis AI*	
	Accuracy	*Speed (fps)*	*Accuracy*	*Speed (fps)*
MobileNet-v1	71.2	567	68.22	993
MobileNet-v1	71	764	67.8	1038
Mobilenet-v2	74.8	288	71.94	764
Mobilenet-v2	74.7	618	-	-
MobileNet-v3	75.8	234	65.36	1073
MobileNet-v3	72.6	304	-	-
ResNet50-v1	74.4	195	74.36	214
ResNet50-v1	74.1	247	71.67	310

Vitis AI results are obtained from the ZCU102 board using three DPUCZDX8G-B4096 cores in the programmable logic. These results vary based on different pruning ratios, where higher pruning ratios deliver faster performance, and no-pruning configurations yield more accurate results. All implementations are provided by Xilinx.

As observed in the table, Vitis AI achieves superior speed compared to the FPGA AI Suite, while the FPGA AI Suite outperforms Vitis AI in terms of accuracy. This difference arises because Vitis AI employs INT8 precision, whereas the FPGA AI Suite utilizes FP16 precision.

In our projects, detailed in Section 5 and Section 6, we utilized Vitis AI for quantizing, compiling, and running our models on the FPGA board Kria KV260.

4.5. Architectural Methods

Many works are in the literature about the hardware acceleration of deep neural networks. On the network side, for optimal hyperparameters, we can use neural architecture search (NAS) methods. Besides, for an efficient design, we should also consider hardware architecture and mapping. There is a work combining both hardware and software optimization methods. The method is called "Neural Accelerator Architecture Search" or NAAS shortly [31]. In NAAS, the evolution mechanism is used to sample the next possible architectures. For each iteration, the next candidate architectures are selected that are close to the best previous ones and distant from the worst previous ones. The same applies to mapping and accelerator architecture selection. The accelerator design parameters include architectural sizing parameters: L1 and L2 cache sizes of processing elements (PE), number of processing elements, and bandwidth for data transfer; also there are connectivity parameters: number of dimensions, size of dimensions, and PE connections. On the other hand, mapping design parameters include loop orders and tiling sizes. The proposed algorithm describes all parameters numerically (including connectivity parameters) and tries to find the best option by optimizing the previous best candidates. Using the NAAS method, a much lower energy-delay product can be achieved according to the same number of iterations for a random search.

Another SOTA study is the One-For-All (OFA) network [29], which stands somewhere in between neural architecture search and network pruning. It applies progressive shrinking on input size, kernel size, channel size, and layer size. The main contribution of this work is that it trains big models and small models together, nested. Then, it prunes the parts (e.g. detailed feature activations) and uses only the most important parts for the smaller networks. It is possible to extract several networks from the biggest one and use them without retraining.

5. DEPTH ESTIMATION ON EDGE

Depth estimation is essential for many vision applications, including autonomous driving systems and robotics [32]. Like many vision-related methods, depth estimation and 3D scene reconstruction tasks have transitioned from hand-crafted mathematical modeling to black-box modeling via DNNs, which achieve much better performance compared to the former. Besides, as computing resources have increased due to AI breakthroughs, model complexities have also increased. However, since computing capabilities are limited and requirements related to power and latency are constrained, complex models cannot be implemented directly on the edge. There are two options for edge computing: either data acquisition is done on the edge device and model inference is performed by a remote server, or both acquisition and inference are performed on the edge device. The former has disadvantages like high latency, data transfer costs, privacy concerns, and requires complex infrastructure to function properly. On the other hand, the latter suffers from a lack of computing capabilities. To overcome computing constraints on edge devices, DNN models should be optimized in many cases. The most common optimization methods are pruning and quantization of model parameters. In pruning, some model weights are deleted, which changes the model architecture and may cause a significant performance drop. In quantization, the model weights are quantized down to lower bit resolution to reduce the computing workload and improve the speed. In this study, we quantized a CNN-based lightweight depth estimation model, MiDaSNet, down to 8-bit fixed-point representation and deployed it on a Kria KV260 FPGA card by utilizing DPU soft architecture on the programmable logic (PL).

5.1. Literature Review

The artificial intelligence revolution we have been experiencing for over ten years has, as in many computer vision areas, caused the transformation of algorithms in the field of depth perception from classical hand-crafted solutions to AI-based so-

lutions. However, AI models require significant computational power compared to classical hand-crafted models, which is not available for edge devices in many cases. Many solutions are proposed in the literature to overcome problems related to the lack of resources. Among them, FPGA-based solutions offer many advantages, including lower power consumption while preserving higher inference speed, customization, and further enhancement possibilities via ASIC implementation. To mention a few, [33] applied two of the most common CNN-based network architectures, namely ResNet-50 and EfficientNet-B0, that solve the depth estimation problem. [34] implemented a depth-from-motion model using an optical flow algorithm on FPGA, which offers high speed and low resource consumption on hardware; however, the results are motion-dependent and show lower accuracy compared to more sophisticated NN-based methods. [35] proposed FPGA implementation of a plenoptic depth estimation method which uses stereo-matching with recursive census transform to calculate the sparse depth from micro-lens-arrays. Although it results in fast and accurate depth estimation with relatively lower computational power consumption, the depth map calculated from MLAs is sparse, and an increase in pixel density would result in excessive resource consumption and higher delays. [36] proposes a lightweight residual convolutional neural network implemented on FPGA for depth enhancement. As the name suggests, the method requires prior coarse depth information, which limits the study's use case. [37] presents a 1-D matching algorithm to calculate the depth map from image sequences captured from drone cameras. Although the proposed algorithm is fast and achieves lower power consumption, it adds some constraints on the direction of motion and speed, which makes it unsuitable for complex environmental conditions. [38] presented an HW/SW co-design approach to implement DeepVideoMVS [39] network on FPGA, which achieves 60 times faster inference speed compared to CPU with the cost of slight accuracy degradation. [40] proposes FasterMDE, an optimized encoder-decoder network that uses multi-objective neural architecture search to optimize the encoder block for edge implementation. [41] implemented the DepthFCN network on FPGA, optimizing the network architecture for different quantization levels to reduce dynamic power consumption while increasing inference speed. [42] implemented a customized version of a small-sized PyDNet model on Xilinx Zynq UltraScale+ MPSoCs

ZCU102 FPGA board with further optimization via quantization using Vitis-AI DPU architecture.

5.2. Methodology

In FPGA implementation, the model selection should consider many factors, including model size, model complexity, and stability. Firstly, the model size should be small enough to fit in on-chip memory. Secondly, the model architecture should be simple enough to be implemented on FPGA. In our case, we used a specialized deep learning processing unit (DPU) embedded in PL for model inference.

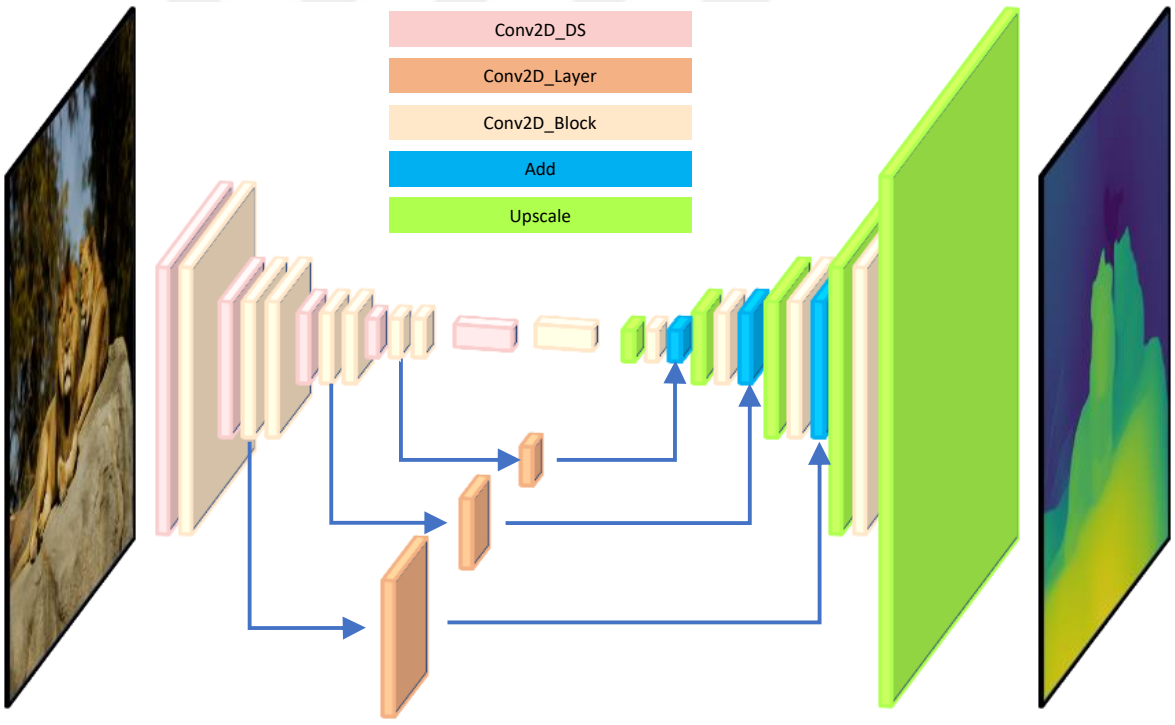

Figure 5.1. MiDaSNet-Small-v2.1 basic architecture. The layer types are color-coded as pink: Conv2D layer with stride=2, orange: Conv2D layer with stride=1, yellow: Conv2D block with stride=1, blue: simple addition, and green: bilinear upsampling with 2.

The DPU architecture supports classical blocks like convolutional layer, dense layer, batch-normalization, and ReLU activation. However, more complex architectures

like transformer blocks are not supported. Therefore, we decided to use a CNN-based lightweight model for our task. On the other hand, the model should be stable for different mediums and under different conditions.

Models trained on a single dataset can be highly unstable when the medium changes. Therefore, we decided to use a model that can perform well for zero-shot tests. Considering all these factors, we selected MiDaSNet-small-v2.1 [43], which is the smallest model among all MiDaSNet models. The selected MiDaSNet model is CNN-based and has approximately 21M parameters. It is trained on 5 datasets, including ReDWeb, MegaDepth, WSVD, DIML Indoor, and 3D movies. These datasets have distinct characteristics in terms of captured environments, image quality, and camera settings, which contribute to the model generalization and, thus, the stability across different mediums. The model's architecture is given in Figure 6.1. The model is a fully CNN model where the left part consists of EfficientNetLite3 [44] encoder as its backbone. The input is processed by the first 9 blocks of the EfficientNetLite3 model, which gradually decreases the image size down to a 1/32 ratio. The right side consists of RefineBlocks, which merge residual paths with the main path via simple addition and use bilinear upsampling with a ratio of 2 for upscaling. Each RefineBlock consists of 5 convolutional layers with residual connections, and they can be fed with 1 or 2 inputs where if 2 inputs are provided, the layer simply adds them up before processing. There are 3 long residual connections via single Conv2D layers. Besides, there are many short residual interconnections both in the encoder and decoder parts. The model has a total of 21M parameters, where 30% of them come from the EfficientNetLite3 feature extractor backbone, which is pre-trained on the ImageNet dataset. The remaining 70% of parameters belong to the residual-refinement head and intermediate convolutional blocks. In Table 5.1, layer counts are given for each ingredient layer type. The model includes 99 two-dimensional convolutional layers, 72 batch normalization layers, 10 ReLU, and 48 HardTanh activation layers.

To implement MiDaSNet on FPGA, we first rewrite the model code, eliminating custom blocks with classical ones while preserving weights and changing layer

types if necessary for a correct conversion. Rewriting the model code to simplify it and avoid dependencies on other libraries allows a smoother conversion by Vitis-AI tools. Thanks to the simplification, we were able to fit the model into a single subgraph, which was partitioned (between CPU and DPU) when direct conversion was applied. Therefore, the bottleneck due to CPU-DPU data transfer is avoided, and the speed is improved. On the other hand, Vitis-AI tools generate buggy results during quantization on ReLU6 activation layers. We found out that $ReLU6 \rightarrow ReLU6$ conversion was falsely realized as $ReLU6 \rightarrow ReLU$, which degrades the $\delta 1$-accuracy by 10%. To heal it, we work around the problem by changing $ReLU6$ to $HardTanH$. With correct parameters, $HardTanH \rightarrow ReLU6$ conversion is possible without any performance degradation. Before quantization, model pruning and additional optimizations for edge devices were considered; however, according to the initial results, the quantization already results in a 3% reduction in accuracy. Further optimizations like pruning or additional quantization to lower bits would likely result in a significant performance drop, which motivated us to focus on achieving high performance without these techniques. Therefore, we quantize the model without further customization and compile it for FPGA implementation with different predefined DPU configurations, including DPUZCXDG-ISA1-b4096 and DPUZCXDG-ISA1-b3136, which are specialized architectures for 2D convolutional layers with different processing bandwidths. For both DPU architectures, the resource consumptions are given in Table 6.5, where each BRAM unit has a memory size of 32 Kb, and each URAM unit has a memory size of 288 Kb. After the quantization down to an 8-bit fixed point representation, we deployed our model on the Kria KV260 board.

5.3. Experiments

We tested our model's zero-shot performance on the NYUv2 [45] dataset. Firstly, to highlight differences regarding the chosen DPU architecture, we implemented our model with different DPU configurations: B4096 and B3136. For both configurations, the model achieves 82% accuracy and close performance on other metrics, with just slightly lower results for the second one. On the other hand, there is a significant

difference in inference speed. The first one achieves 50 fps/67 fps for Python/C++ implementations. The second one achieves 38.2fps/47.9fps for Python/C++ implementations which is 30-40% lower compared to the former one. This is because DPU-B4096 can process a higher volume of data compared to the DPU3136 architecture. The results for both DPU configurations are given in Table 5.3.

Table 5.1. Model ingredients.

Layer	*Sequential*	*ZeroPad2d*	*Conv2d*	*BatchNorm2d*	*ReLU6*	*ReLU*	*Identity*
Count	17	5	99	72	48	10	26

Table 5.2. Resource utilization for different architectures.

Resource	*Available*	*DPU-B3136*		*DPU-B4096*	
		Utilization	*Utilization (%)*	*Utilization*	*Utilization (%)*
LUT	117120	45640	38.97	50867	43.43
LUTRAM	57600	5171	8.98	7063	12.26
FF	234240	79642	34	98797	42.18
BRAM	144	45.5	31.6	67.5	46.88
URAM	64	50	78.13	50	78.13
DSP	1248	566	45.35	710	56.89

Table 5.3. Performance results for different DPU configurations.

Architecture	*δ1-Acc.*	*δ2-Acc.*	*δ3-Acc.*	*Speed (fps)*
DPU-B3136	82.32	96.76	99.28	38.2 / 47.9
DPU-B4096	82.61	96.81	99.31	50.7 / 67.3

The quantitative comparison results with the baseline models are given in Table 5.4. For the comparison, we could not find any FPGA implementation of depth estimation models where zero-shot performance is noted. Therefore, we compare our model with only a few models that are implemented on FPGA and noted performances on the NYUv2 dataset. Alongside that, we added some CNN-based depth estimation models implemented on different devices like CPU, GPU, or other embedded computing platforms. The comparison is based on the image input size, number of floating point operations (FLOPs) in GOP, $\delta 1$, $\delta 2$, and $\delta 3$ accuracy, relative absolute error,

root mean square error, speed in frame-per-second, dynamic power consumption, clock frequency, implemented platform or board, and network architecture.

Table 5.4. Comparison with baseline on NYUv2 dataset.

Architecture	*Input Size*	*GOPs*	*δ1*	*δ2*	*δ3*	*REL*	*RMSE*	*fps*	*Power*	*Freq*	*Platform*
VGG [46]	228x304	23.4	76.9	-	-	-	-	-	-	-	CPU
ResNet50 [47]	384x384	61.8	78.1	95	98.7	0.155	0.66	-	-	-	CPU
ResNet50 [33]	256x256	-	-	-	-	0.168	0.638	-	-	-	CPU
EfficientNet-B0 [33]	256x256	-	-	-	-	0.156	0.625	-	-	-	CPU
ResNet-UpProj [48]	228x304	22.9	81.1	95.3	98.8	0.127	0.573	18.18	-	-	GPU
FasterMDE [40]	-	-	-	-	-	*0.113*	-	33.57	-	-	Jetson Xavier NX
DeepVideoMVS [38]	-	-	-	-	-	-	-	3.6	-	188M	ZCU104
DepthFCN [41]	256x256	*0.66*	76.2	-	-	-	-	*123*	*0.3W*	200M	ZU3EG
MiDaSNet [43]	480x640	3.47	*85.8**	*97.73**	*99.51**	0.117*	*0.467**	0.71	1.4W	1.3G	ARM Cortex A53
Proposed work	480x640	7.43	*82.6**	*96.81**	*99.31**	0.133*	*0.506**	50.74	0.62W	300M	Kria-KV260

As shown in Table 5.4, even though our model has a high number of parameters and FLOPs, it achieves a comparably high inference rate of 50 fps with a relatively low dynamic power consumption of 0.62W. The dynamic power was calculated by subtracting the power measured during the idle phase from the power measured during inference, encompassing the entire board. The whole board consumes 5.12W during idle phase and 5.8W during inference for DPU-4096 architecture. It means that the inference power efficiency is 8.75 FPS per watt, while the computing power efficiency is 1.28 GOPs per watt. On the other hand, MiDaSNet models show the best performance in terms of δ1-2-3 accuracies even without considering zero-shot errors. However, compared to the float model, the quantized MiDaSNet model achieves 3.2% lower accuracy due to the quantization-related errors. It is important to mention that in the original implementation of the float model [43], a slightly higher δ1-accuracy is reported. Accuracy degradation is caused partly due to the difference in batch numbers. In the original paper, 86.6% δ1-accuracy is reported, whereas in our paper, the customized float model achieves 85.8% δ1-accuracy for batch number of 1. We confirmed that there is no performance loss due to the customization applied to the float model before quantization. The customized float model is directly quantized without further optimization, where the quantized model achieves %3.2 lower δ1-accuracy compared to the float model. The quantization process was performed using Vitis-AI tools. During this

process, we observed that the selected sample image batch for the optimized quantization process affects the model's $\delta 1$ accuracy by approximately $\pm 0.5\%$ for different samples.

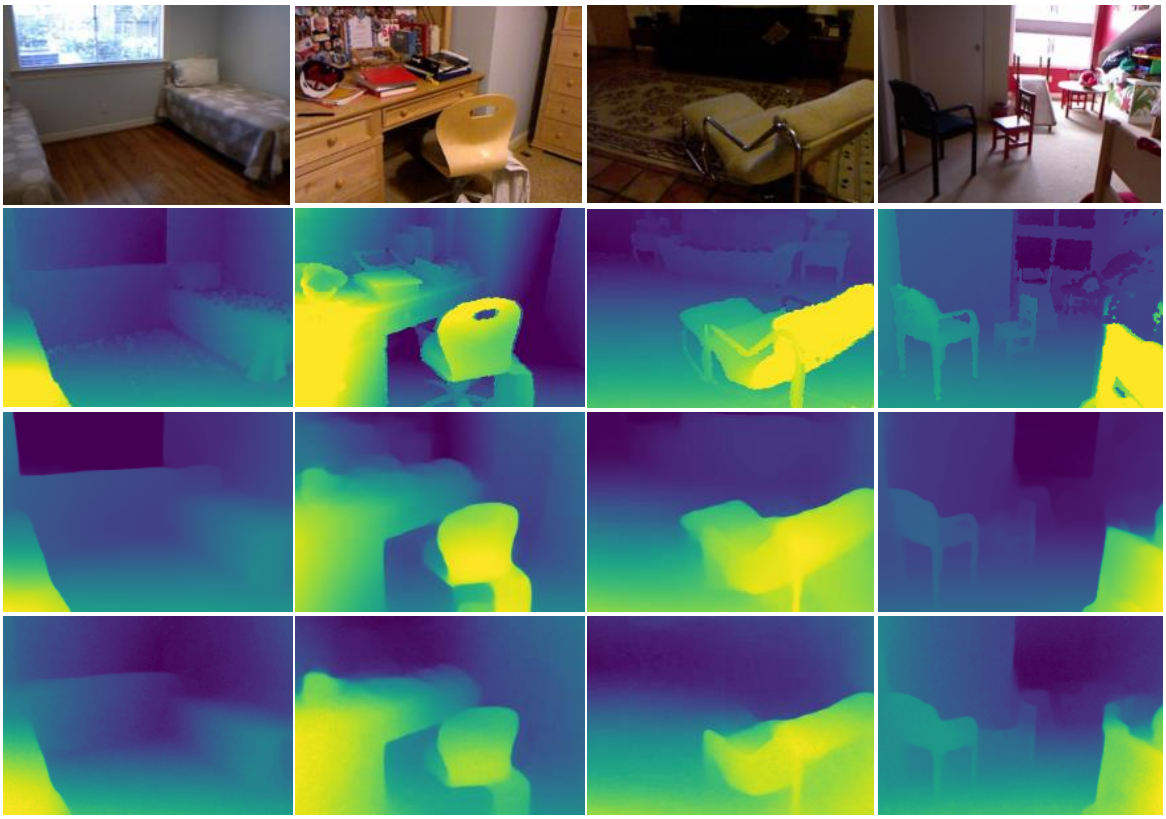

Figure 5.2. Results from NYUv2 dataset. Top to bottom: RGB, Ground truth, MiDaSNet, Quantized MiDaSNet (proposed work)

One might argue that quantizing a deep regression model should result in higher numerical errors than reported. However, the resulting δ-accuracy degradation is not directly linked to the deviation in the model's inference results. Our findings indicate that the average pixel-wise proportional deviation between CPU inference results and DPU inference results is approximately 25%. However, since the model generates a relative depth map, the deviation decreases after removing sample-wise scale and shift errors during post-processing. Moreover, the $\delta 1$-error pertains to pixels falling outside an acceptable range ($< 25\%$) rather than pixel-wise proportional error. For instance, if all pixels exhibit a proportional error of 24% relative to the ground truth depth map, the $\delta 1$-accuracy would be 100%. That is to say, even though the average

proportional deviation between the quantized and float model inference results is high, the δ-accuracies remain similar.

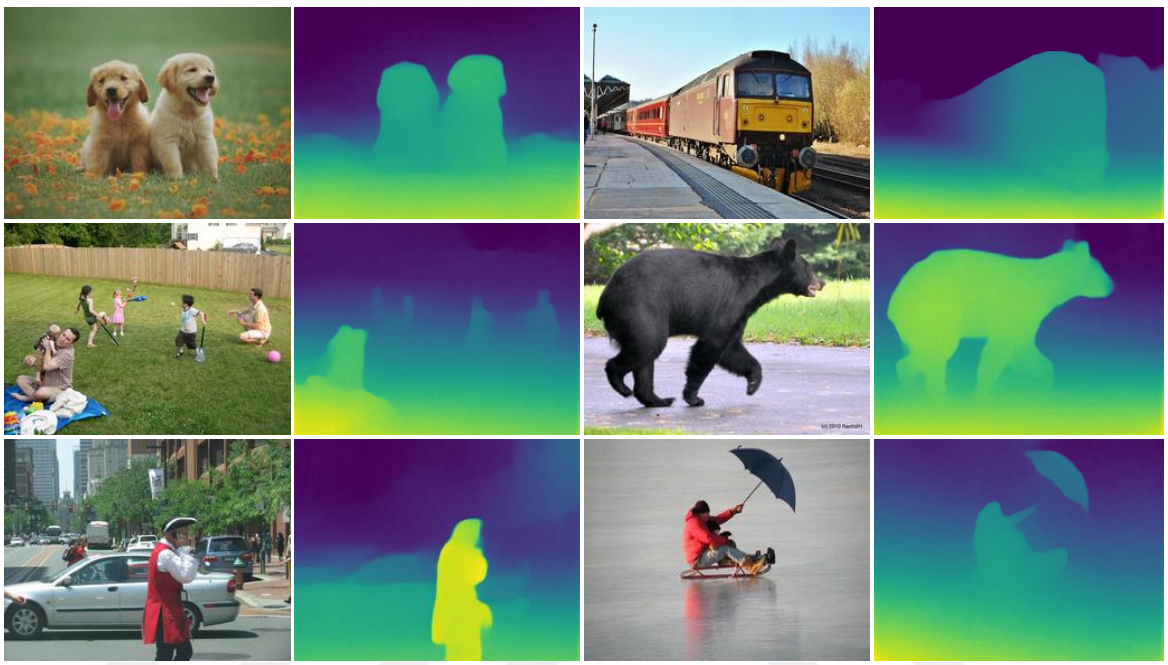

Figure 5.3. Qualitative results from COCO dataset

For qualitative comparison, we share our model's zero-shot inference results on the NYUv2 test dataset in Figure 5.2. As the figure shows, our quantized model's performance is on par with the float model and close to the ground truth depth map. It is noteworthy that there are some apparent differences between CPU and DPU inference results. For example, depth estimation for the first image in Figure 5.2 differs between CPU and DPU. However, there is no visible jitter or dispersed noise in the quantized model's inference result, indicating that the quantization process does not significantly affect the stability of the model. Moreover, the differences partly stem from the post-processing applied before generating the dumped depth images for visualization, such as min-max normalization and cropping. Nevertheless, when evaluating the model's overall performance, we can conclude that it is on par with the float model and produces inference results close to the ground truth. In addition to NYUv2, we present some inference results from the Coco dataset [49] in Figure 5.3 to demonstrate our model's generalization performance across different contexts. Despite

the varied environments of the selected images, the model performs well, thanks to the diverse set of training datasets used for the MiDaS models.

5.4. Conclusion

In this study, we implemented a real-time depth estimation model on the Kria KV260 board. For edge deployment, we chose a lightweight CNN-based model, MiDaSNet-small-v2.1. Since the model is trained on a diverse set of datasets, it exhibits stable performance across different mediums. Besides, quantized to 8 bits and optimized for the board, the model achieves 50.7 fps inference speed and 82.6% zero-shot $\delta 1$ accuracy on the NYUv2 test dataset, with just 3% accuracy degradation compared to the original model. Additionally, our quantized model achieves 96.8% and 99.3% zero-shot $\delta 2$ and $\delta 3$ accuracy, respectively. Compared to the baseline, the model achieves the best $\delta 1$, $\delta 2$, and $\delta 3$ accuracy on the NYUv2 test dataset, even with the zero-shot error. Furthermore, the qualitative test results confirm the model's stable performance across different mediums and datasets.

6. 3D SEMANTIC SEGMENTATION ON EDGE

Point cloud segmentation plays a vital role in vision-based applications such as autonomous driving and robotics [50]. Over time, this field has shifted from traditional hand-crafted techniques to deep neural networks (DNNs), which deliver significantly better performance. This progress has been driven by advances in computational power, enabling the design of more sophisticated models.

However, deploying complex models on edge devices presents significant challenges due to limited computational resources and strict constraints on power consumption and latency. To address these limitations, DNN models must be optimized for hardware efficiency. One effective approach is quantization, which reduces the precision of model weights to lower bit resolutions. This significantly decreases computational requirements, improves power efficiency, and enhances inference speed, making it well-suited for edge deployment.

In this study, we adapted, customized, and trained the lightweight CNN-based SalsaNext model using our proposed multi-head training mechanism. Subsequently, we optimized and quantized the model to 8-bit fixed-point precision before deploying it on a Xilinx Kria KV260 FPGA, leveraging the DPU architecture within its programmable logic (PL).

The main contributions of this work are outlined below:

- *Real-Time Model for Outdoor Robotics:* Developed a robust LIDAR segmentation model optimized for real-time performance in outdoor robotics, ensuring stable and reliable operation across diverse environments.
- *Merging two of the largest LiDAR segmentation datasets:* Proposed a domain adjustment schema to merge two of the most common and largest datasets, SemanticKITTI and Open Waymo Dataset, for autonomous driving.

- *Multiple-Head CNN model based on SalsaNext:* Proposed a multi-head SalsaNext model to train multiple LiDAR segmentation datasets with different characteristics simultaneously without degrading the main head's performance.
- *State-of-the-Art Results on FPGA:* Achieved superior mean IoU and accuracy for point cloud segmentation on the SemanticKITTI dataset, surpassing other FPGA-based solutions.
- *Cost-Effective FPGA Deployment:* Demonstrated that advanced point cloud segmentation models can be effectively deployed on an affordable FPGA platform.
- *Open-Source Implementation:* The implementation will be released as an open-source Python-based GitHub repository, allowing the community to adapt the approach for other CNN-based models compatible with FPGA systems.

The paper is organized as follows: Section 2 reviews related work on point cloud segmentation for edge devices, providing context and background. Section 3 provides an overview of the model architecture, details the dataset domain adjustments, explains the training process, and also elaborates on the quantization and deployment process on the Kria KV260 board. Finally, Section 4 presents the experimental results from deploying the proposed model on a Xilinx FPGA card, showcasing its performance and effectiveness.

6.1. Literature Review

Point cloud segmentation for edge devices faces several critical challenges: managing unstructured data as part of input data representation; balancing complexity and stability in model architectures; achieving hardware efficiency for edge device optimization; and addressing domain shift during multi-dataset training.

6.1.1. Input Data Representation

Existing segmentation methods for point clouds typically fall into four categories based on how they structure the input data:

(i) Point-set-based methods [51, 52],
(ii) Voxel-based methods [53],
(iii) Projection-based methods, such as those using range images or bird's-eye-view (BEV) representations [54–57], and
(iv) Hybrid approaches [58, 59].

Among these, projection-based techniques—especially those employing range images [55, 56]—are particularly attractive for edge deployment because they offer a compact, computationally efficient representation and can leverage mature 2D CNN architectures. Although BEV methods are a viable option, their extensive field of view in outdoor robotics often leads to a significant increase in input size, making them less ideal. In contrast, range images from single-sweep point clouds provide a more concise alternative, which is why our work focuses on range-image-based methods.

6.1.2. Model Architectures

The literature identifies four primary model architectures for point cloud segmentation:

(i) Graph-based models [60–62],
(ii) Transformer-based models [51, 52, 63],
(iii) 3D CNNs [64–66], and
(iv) 2D CNNs [54–57].

Transformer-based models—such as Point Transformer [51] and Range Former [63]—use attention mechanisms to achieve high performance; however, their complexity and reliance on heterogeneous operations can hinder their deployment on edge devices. Similarly, while graph-based approaches like PointNet [67, 68] and KPConv [60] are adept at handling unstructured data, their high computational cost or limited accuracy often makes them unsuitable for edge applications. In contrast, 2D CNNs, suitable for processing projected range images, offer a straightforward and repetitive

architecture that is friendly to hardware implementations while still providing satisfactory performance. Consequently, our approach utilizes a 2D CNN-based model.

6.1.3. Edge Device Optimization

Optimizing LiDAR segmentation models for edge devices necessitates careful consideration of speed and power efficiency. FPGA-based implementations have become a focal point because their programmable logic (PL) units can serve as specialized computing cores, leading to enhanced performance and reduced energy consumption. Both traditional handcrafted algorithms and modern deep neural network (DNN)-based methods have been successfully accelerated on FPGAs. For example, a non-AI approach [69] introduced a real-time FPGA-based ground segmentation technique for LiDAR point clouds using channel-based methods, angular data repair, and a cross-eight-way flood-fill algorithm, achieving significant improvements over CPU and GPU implementations. Later, a parallel processing strategy further refined depth ground segmentation (RDG) on solid-state LiDARs, resulting in a 30.9× speedup compared to CPU-based methods [70]. Despite their speed and power efficiency, these methods often struggle to generalize to the complexities of natural environments compared to AI-based models.

In the domain of AI-based methods, most edge implementations utilize range images processed by quantized 2D CNN models [71–74]. Two widely adopted CNN accelerators for edge applications are NVDLA [75] and DPU [76]. For example, the NVDLA-based method described in [71] introduces a lightweight CNN with just 1.9 million parameters. This approach creates a more compact model by splitting the processing of contextual and spatial features into separate pathways, which are subsequently merged using a feature fusion module. The model, after quantization to INT8 and deployment on a ZCU104 board using the NVDLA architecture, achieved 46.4% mIoU on the SemanticKITTI dataset, with performance improvements in speed (56 fps) and power efficiency (46 times better than GPUs).

On the DPU front, [72] demonstrated a hardware-software co-design implementation of the SqueezeSegV3 [56] model on a VCK190 board, achieving substantial real-time performance and energy efficiency improvements across various customizations of the model compared to GPU-based implementations. Similarly, [73] deployed a lightweight CNN model on the ZCU102 board featuring a customized ResNet34 backbone [77] and an ASPP head [78], which reached 10 fps and an efficiency of 42.5 GOP/W on 2 DPU cores. Additionally, [74] applied channel pruning to the SalsaNext model [55] before training and later quantized the model to INT8, demonstrating high inference rates (up to 97 fps on the VCK5000 board) and improved power efficiency across various Xilinx platforms.

Beyond the training schema (discussed in the next subsection), our FPGA mapping process closely follows conventional DPU-based implementations by employing similar DPU architectures. However, we introduce additional customizations designed to reduce quantization error, as detailed in the Methodology section.

6.1.4. Domain Adaptability

A common challenge with models trained on a single dataset is their instability when faced with domain shifts across different environments. Although multi-dataset training is a potential remedy, significant domain discrepancies—such as those between the Waymo and KITTI datasets—can still adversely affect performance. Previous methods addressing domain shifts [79–82] often suffer from limited generalizability in real-world settings [79, 80] or depend heavily on supervised techniques [81, 82]. To address these issues, our work introduces a simple yet effective double-head training scheme with domain adjustment, specifically designed for range-image-based architectures across multiple LiDAR datasets. This multi-head approach also provides the flexibility to accommodate varying dataset annotations and differing classification requirements.

6.2. Methodology

In FPGA-based systems, selecting an appropriate model requires a careful balance between size, complexity, and stability. The primary trade-off is between the model's size—measured in terms of parameters and MAC operations—and its speed or power efficiency. Although larger models tend to yield higher accuracy, they also incur greater power consumption and increased latency. For efficient hardware mapping, the model must therefore remain both compact and simple.

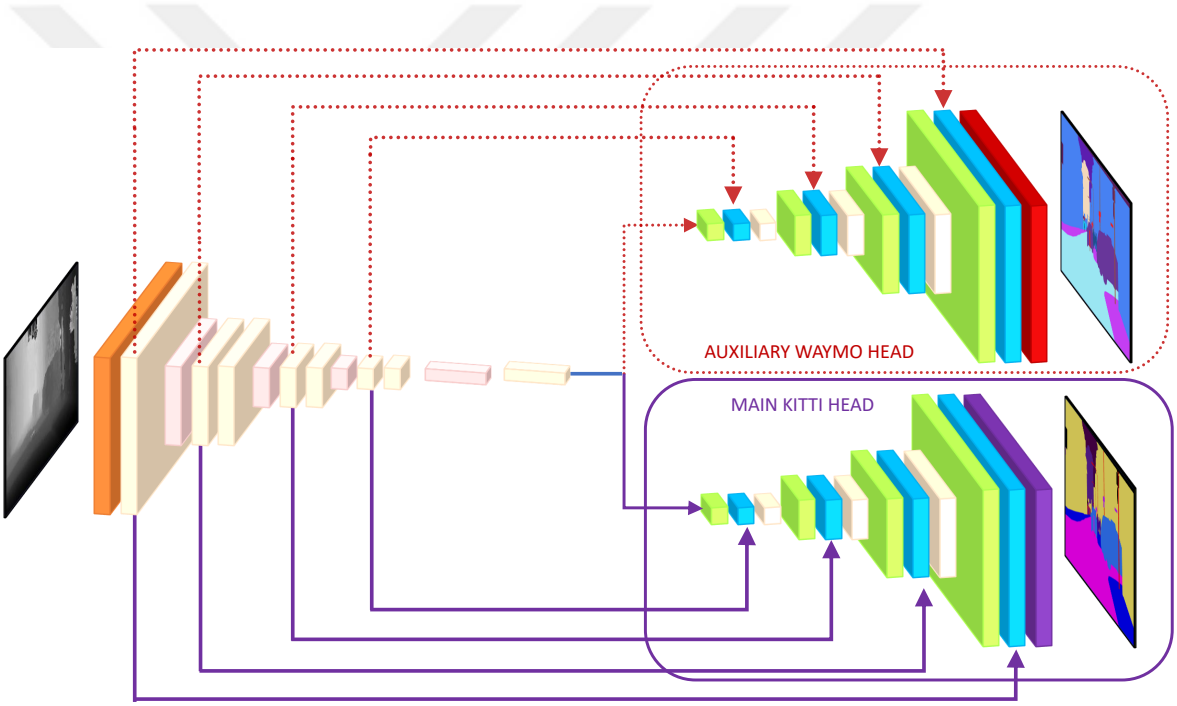

Figure 6.1. SalsaNext model with an additional head for multi-dataset training. The layer types are color-coded as pink: average pooling with stride=2, orange: ResNet context blocks with stride=1, yellow: ResNet blocks with stride=1, blue: simple concatenate, green: bilinear upsampling with a convolutional layer, purple: last conv. layer with 20 channels, and red: last conv. layer with 23 channels.

Given our objective to perform inference using a deep learning processing unit (DPU) embedded in the programmable logic (PL)—which supports only standard operations such as convolution, dense layers, batch normalization, and ReLU activations—we chose a lightweight CNN-based model, SalsaNext [55], specifically designed

to meet these constraints. We further customized the model for multi-dataset training and optimized it for edge deployment.

Table 6.1. The proposed model's layer counts

Layer	*Conv2d*	*ReLU6*	*BatchNorm2d*	*Bilinear Upsample*	*Pooling*
Count	55	54	46	4	4

As illustrated in Figure 6.1 and detailed in Table 6.1, the model follows a fully convolutional design. Its encoder features three residual context blocks followed by four residual downsampling blocks that reduce the spatial dimensions by a factor of 16 and an additional residual block. The decoder comprises four convolutional blocks interleaved with bilinear upsampling layers, ending with a final convolutional layer. Notably, 78% of the model's parameters reside in the feature extractor backbone, while the remaining parameters are allocated to the head. Moreover, the model utilizes atrous convolution to achieve a shallower architecture with a larger receptive field, all without increasing the overall parameter count. In total, the model contains approximately 6.7 million parameters.

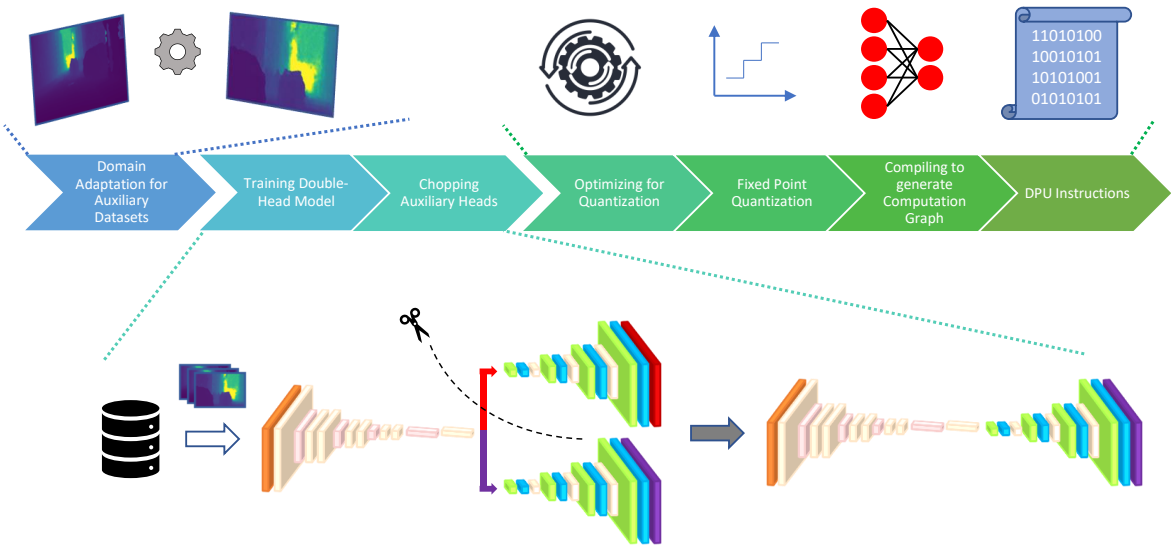

Figure 6.2. The proposed edge implementation pipeline

To enhance the stability and generalization of the model, we proposed a novel training scheme for multi-dataset training for range-image-based LIDAR semantic segmentation models and customized the base model by adding extra heads for each additional dataset (with different labeling conventions). The complete edge implementation pipeline is shown in Figure 6.2.

Table 6.2. Comparison of key features across major LiDAR datasets.

Feature	*nuScenes*	*SemanticKITTI*	*Waymo Open Dataset*
Sensor	Velodyne VLP-32C	Velodyne HDL-64E	Custom 64-beam
Vertical Resolution	32	64	64
Horizontal Resolution	1080	2048	2650
Vertical FOV	-30° to +10°	-24.3° to +2°	-18° to +2.3°
Range	100m	120m	75m
# of Train Sweeps	28,130	19,130	23,691
# of Valid Sweeps	6,019	4,071	5,976
# of Test Sweeps	6,008	20,351	2,982
# of Semantic Classes	32	28	23

There are three major datasets commonly used for LiDAR segmentation tasks: SemanticKITTI, nuScenes-lidarseg, and the Waymo Open Dataset, as summarized in Table 6.2. Due to its low number of channels and horizontal resolution, the nuScenes dataset was not suitable for training the model on projected range images. Consequently, we focused on the SemanticKITTI and Waymo datasets.

However, a significant domain shift exists between these datasets. The Waymo dataset provides range images with intensity values, whereas SemanticKITTI offers point cloud data with intensity values, and the distributions of these intensity values differ considerably. Additionally, although there are overlapping classes in their annotations, they cannot be fully mapped to each other. Thus, domain adaptation is required for both inputs and labels. Therefore, the following adjustments were applied to address the domain shift between Waymo and SemanticKITTI:

- *Class Mapping:* The Waymo dataset contains 23 classes, and the Waymo head was designed to output the same number of classes. In contrast, SemanticKITTI

includes 34 classes, which were mapped to 20 classes as per [55]. The number of channels in the KITTI head was adjusted accordingly.

- *Range Image Regeneration:* Waymo range images were regenerated to match SemanticKITTI characteristics. x, y, and z coordinates were extracted alongside the range image. The regenerated range image was aligned to SemanticKITTI sensor specifications by setting the height from the ground to 1.75 meters, rearranging the coordinate frame, and adjusting the field-of-view (FoV) angles to match SemanticKITTI ($-25°$ to $3°$).
- *Intensity Mapping:* Intensity values from Waymo, which can reach thousands, were clipped and mapped to the range $[0, 1]$ to ensure a similar cumulative distribution function (CDF) as SemanticKITTI.

On the other hand, to adapt the base *SalsaNext* model for FPGA deployment, the following modifications were implemented:

- *Layer Reordering:* Batch normalization layers were reordered with activation layers to optimize the fusion mechanism, applied by Vitis-AI tools (conv+bn+act).
- *Activation Replacement:* If the network feature values exceed the fixed-point range, the related layers are mapped to the CPU. To avoid this, we applied ReLU6 as an activation function throughout the entire network.
- *Upsampling Adjustments:* Since pixel shuffle is not supported by DPU, pixel shuffle upsampling was substituted with bilinear upsampling, followed by convolution, batch normalization, and activation layers.
- *Block Splitting:* The model was divided into encoder and decoder components to separate the feature extractor backbone and facilitate the multi-head architecture.
- *Dual Training:* Two separate decoder heads were trained: one for the KITTI dataset and another for Waymo.
- *Model Simplification:* After training, the Waymo-specific head was removed to create a compact, single-head model.
- *Activation Conversion:* ReLU6 activations were converted to HardTanh to address quantization issues in Vitis-AI conversion tools, ensuring accurate represen-

tation without accuracy loss.

- *Loss Function:* The overall loss was computed as a weighted combination of losses from the two heads, with a ratio of 4 : 1. Each head's loss comprised a class-balanced cross-entropy loss and a Lovász loss [83], following the approach in [55]. For the cross-entropy loss, the weight assigned to the `unlabeled` class was adjusted to `min(class_weights)/50`, enhancing accuracy and awareness of unlabeled regions.

After training, we employed 8-bit fixed-point quantization and compiled the model for specific DPU configurations (*DPUZCXDG-ISA1-b4096* and *DPUZCXDG-ISA1-b3136*). The quantized model was deployed and tested on the Kria KV260 board for both configurations.

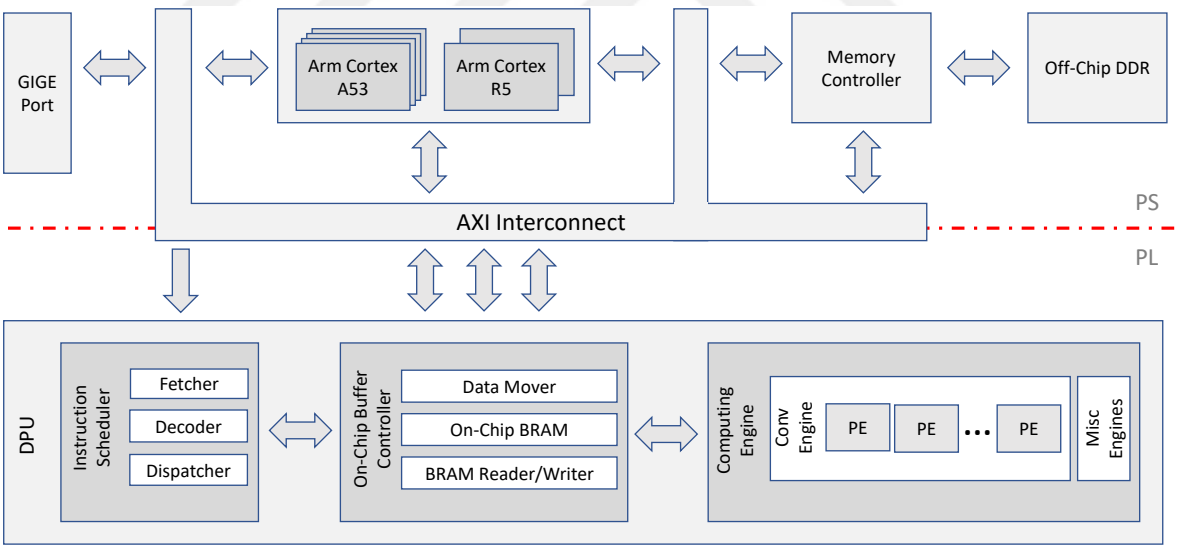

Figure 6.3. DPU Processing Pipeline

Figure 6.3 shows the DPU processing pipeline. For efficient data transfer between DDR and PL (or DPU), 3 strategies are used: Firstly, data is transferred via AXI parallel interface. Secondly, there is a dedicated on-chip memory inside PL made of URAMs and BRAMs. This reduces the need for DDR access. Thirdly, Direct Memory Access (DMA) is used to directly access DDR without CPU intervention.

The inference process begins when the DPU fetches instructions from its dedicated instruction memory, which may reside in on-chip buffers or be accessed via the AXI interface from external DDR memory. These instructions include operation codes for various neural network tasks such as convolution, activation, and pooling. Once the instructions are fetched, the DPU's control unit decodes them to determine the specific operations to be performed. This decoding phase directs the subsequent actions by routing instructions to the appropriate compute units within the DPU.

In parallel with instruction decoding, the DPU initiates data transfers to prepare the operands for computation. Data read operations—managed via Direct Memory Access (DMA) and the AXI interface—load the required input feature maps and filter weights from DDR or on-chip memory (such as URAMs and BRAMs) into the processing units. With the data available, the dedicated multiply-accumulate (MAC) units perform the convolution operations, calculating dot products between input data and weights to produce intermediate feature maps. Once computed, these results are written back to memory, either to local caches or directly to DDR, ensuring that the outputs are available for subsequent processing stages like activation functions or pooling. This continuous cycle of reading, processing, and writing ensures that the DPU maintains a steady flow of data through its highly optimized pipeline, thereby maximizing throughput and minimizing latency.

6.3. Experiments

For qualitative evaluation, we compared the performance of our float model on the SemanticKITTI and Waymo Open datasets, using only the KITTI-specific head for inference in both cases. Figure 6.4 shows the performance of our proposed models on samples from the SemanticKITTI and domain-adapted Waymo validation datasets. In the first SemanticKITTI example, our models—unlike the original model—successfully infer unlabeled regions too, enhancing their overall reliability. Moreover, the double-head model outperforms both the single-head version and the original model on the Waymo sample. For example, while the original model misclassifies pedestrians as other

vehicles, cars, or even buildings, our single-head model correctly labels about half of these instances, and the double-head model nearly classifies all of them correctly.

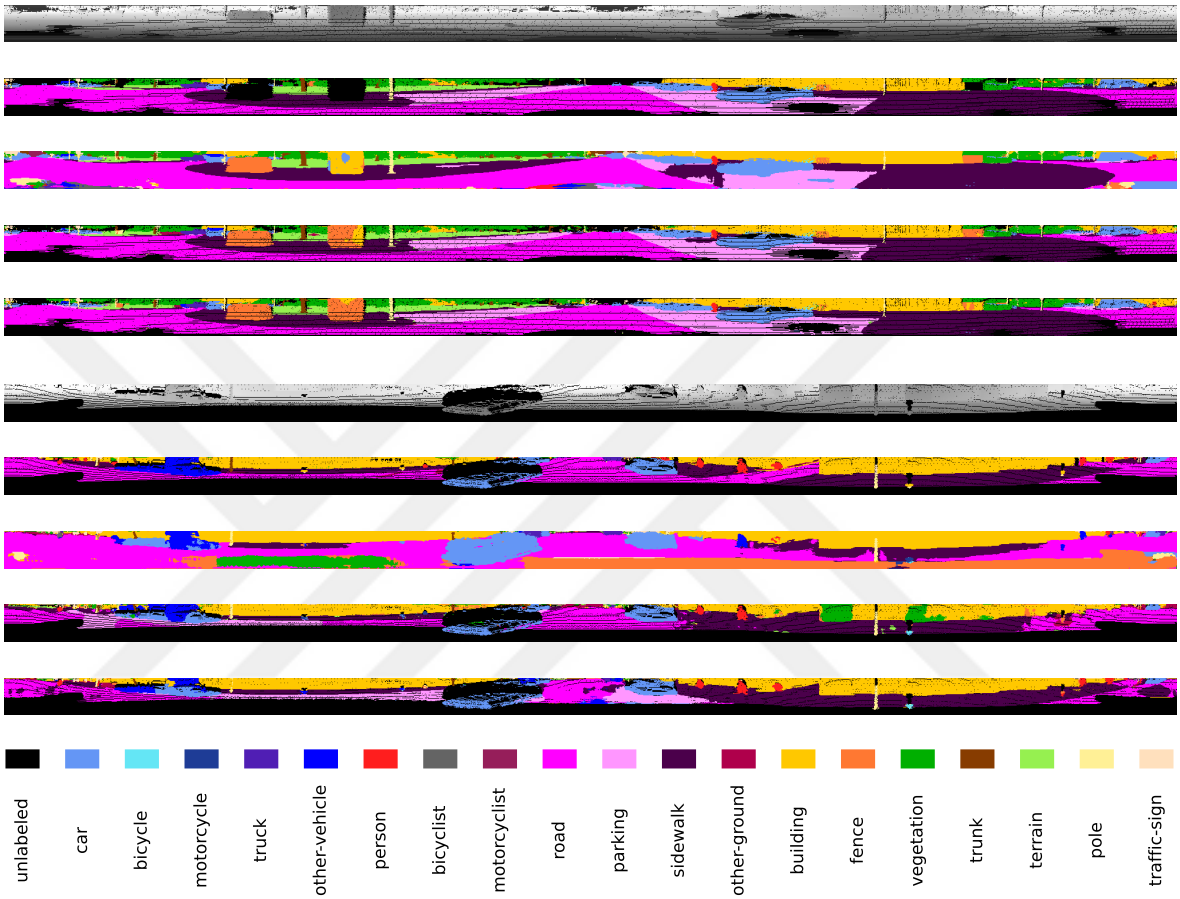

Figure 6.4. First Half: Inference results for a sample from the SemanticKITTI dataset. From top to bottom: 1) Range image 2) Label 3) Inference result from original model [55] 4) Inference result from single head model 5) Inference result from Kitti head of double headed model. Second Half: Inference results for a sample from the Waymo dataset after applying domain adjustment with the same order.

For quantitative comparison, firstly, we evaluated our models' performance against the original model on three datasets: SemanticKITTI, Waymo, and domain-adjusted Waymo datasets. The three compared models include the original SalsaNext [55], and our customized single-head and double-head versions of SalsaNext. As shown in Table 6.3, the double-head SalsaNext achieves the best performance on the domain-adjusted Waymo dataset. Specifically, the performance improvement due to domain

adjustment and double-head training is 12.25% in the mIoU metric and 26.34% in accuracy (or precision) compared to the original model.

Table 6.3. Performance Comparison of original SalsaNext [55] with our customized single-head and double-head versions on SemanticKITTI, Waymo, and domain-adjusted Waymo datasets

Method	*SemanticKITTI*		*Waymo*		*Waymo_DA*	
	mIoU	*Accuracy*	*mIoU*	*Accuracy*	*mIoU*	*Accuracy*
SalsaNext-orig [55]	*61.01*	90.14	9.93	30.16	14.02	43.8
Single-Head [ours]	60.97	90.48	12.5	*31.83*	15.21	46.31
Double-Head [ours]	60.79	*90.71*	*13.2*	31.26	*22.18*	*56.5*

Table 6.4 shows the comparison of classwise IoU scores on SemanticKITTI validation dataset with the baseline float models. According to the baseline, SalsaNext-based models exhibit the best performance in terms of mIoU and accuracy. Our single-head and double-head models achieve approximate results with the original implementation with a small degradation in mIoU (-0.04% for single-head and -0.22% for double-head models).

Table 6.4. Comparison of class-wise IoU scores of float models on SemanticKITTI validation dataset

Method	*Mean IoU*	*Car*	*Bicycle*	*Motorcycle*	*Truck*	*Other-vehicle*	*Person*	*Bicyclist*	*Motorcyclist*	*Road*	*Parking*	*Sidewalk*	*Other-ground*	*Building*	*Fence*	*Vegetation*	*Trunk*	*Terrain*	*Pole*	*Traffic-sign*
SSGV3-K7 [72]	43.8	74.5	22.3	40.9	23.1	16.3	39.7	49.8	0	92.4	39.7	79.0	0.4	78.5	31.2	80.2	46.3	71.9	34.8	33.3
ResNet-GCM [71]	47.9	87.3	19.4	27.6	46.6	28.3	43.5	57.3	*1.6*	91.2	34.1	75.2	2.1	81.3	36.9	78.0	49.4	73.1	41.5	36.3
ResNet-NLA [73]	56.4	92.2	37.5	42.5	72.4	37.5	63.2	75.4	0	92.0	34.2	77.7	8.1	85.0	45.3	84.3	58.7	72.0	50.3	44.1
SalsaNext-orig [55]	*61.0*	94.8	*48.3*	46.4	*79.1*	44.1	*73.2*	*82.9*	0.0	*94.9*	43.0	*81.8*	4.8	*87.0*	53.7	83.4	*67.0*	67.1	*60.2*	47.5
Proposed Work (SH)	60.9	94.3	39.5	51.3	75.3	*47.0*	71.4	79.6	0.0	94.2	*45.0*	81.3	*8.8*	86.9	*54.8*	84.3	65.7	71.4	58.0	49.7
Proposed Work (DH)	60.8	*94.8*	42.7	*55.9*	73.7	45.4	69.3	81.0	0.2	94.1	44.9	81.2	2.4	85.7	51.4	*85.4*	64.8	*74.1*	58.4	*49.7*

To evaluate the impact of different DPU architectures, the model was implemented with two configurations: B4096 and B3136, with their resource utilizations detailed in Table 6.5. Test results reveal that the B4096 configuration exhibits superior performance in terms of speed and accuracy. Specifically, the B4096 configuration

achieves 5.6 frames per second (fps), whereas the B3136 configuration reaches only 3.85 fps, reflecting a 32% reduction in speed. This difference arises because the DPU-B4096 architecture can handle greater data throughput compared to DPU-B3136. Detailed results for both configurations are summarized in Table 6.6.

Table 6.5. Resource utilization for 2 different DPU architectures

Resource	*Available*	*DPU-B3136*		*DPU-B4096*	
		Utilization	*Utilization (%)*	*Utilization*	*Utilization (%)*
LUT	117120	45640	38.97	50867	43.43
LUTRAM	57600	5171	8.98	7063	12.26
FF	234240	79642	34	98797	42.18
BRAM	144	45.5	31.6	67.5	46.88
URAM	64	50	78.13	50	78.13
DSP	1248	566	45.35	710	56.89

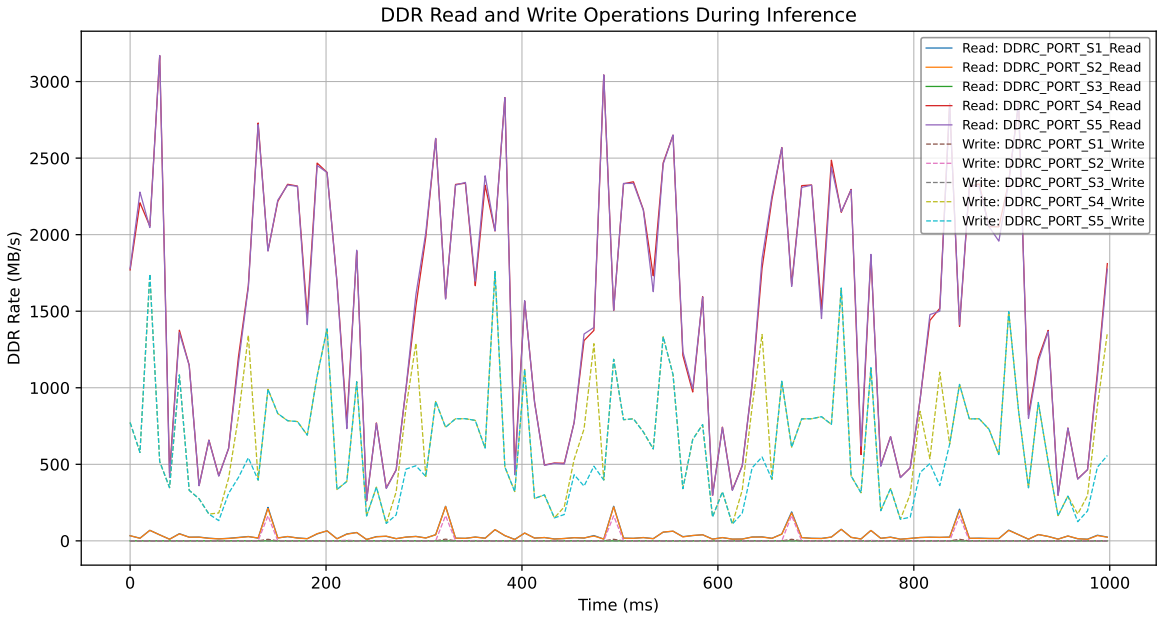

Figure 6.5. DDR read and write rates during inference.

To further evaluate DPU-B4096 architecture, we measured DDR read and write data rates and plotted them in Figure 6.5. As shown from the figure, thanks to direct memory access and parallel AXI interface, for a single port, the maximum read rate exceeds 3GB/s and the maximum write rate exceeds 1.5GB/s.

Table 6.6. Performance results of the proposed double-head model after 8-bit quantization, on SemanticKITTI validation dataset, for different DPU configurations.

Architecture	*mean IoU*	*Accuracy*	*Speed (fps)*
DPU-B3136	54.28	87.78	3.85
DPU-B4096	56.92	88.31	5.6

Table 6.7. Latencies of Inference Phases

Operation	*Fetch*	*Pre-processing*	*Inference*	*Post-processing*
Processor	CPU	CPU	DPU	CPU
Delay (ms)	19.67	50.60	175.44	12.99

The latencies for real-time performance are given in Table 6.7. In terms of data collection, since we cannot reach a 64-channel LIDAR sensor, we read data from the image file on disk which takes 3ms. However, we include sensor readout delays in experimental results. Most of the LIDAR sensors support Gigabit Ethernet (GigE) with a theoretical speed of 125MB/s. Our card also supports GigE. However, due to protocol overhead and other delays, the speed is generally between 80-100 MB/s. We assumed the transfer speed be 80MB/s. Besides we assumed 12 bytes of data packet for each point, and 131k points per sweep. Therefore, we calculated 19.67ms of transfer delay. We included this transfer delay in our calculations. The rest of the latencies are based on the real-time measurements on the board, including preprocessing, DPU inference, and post-processing. According to these results, overall latency reaches 259ms on the board.

Table 6.8 provides a quantitative comparison of our model against baseline segmentation models implemented on FPGA, as well as CNN-based semantic segmentation models executed on FPGA platforms. Our quantized model achieves the highest mIoU, with a total power consumption of 9W for the entire board. Although our inference frame rate is low, it remains within the operational range of most of the commercial LiDAR sensors (5–15 Hz).

Table 6.8. Comparison with the baseline of FPGA-based LIDAR segmentation models on SemanticKITTI dataset. mIoU, Accuracy, Power Efficiency, and Latency metrics are given for both float and quantized models. For DPU-based implementations, results for single-thread, single-DPU configuration are noted except [73] which has 2 DPU cores.

Method	*GOPs*	*mIoU*	*Accuracy*	*Power (W)*	*Speed (fps)*	*Freq. (M)*	*Platform*	*Efficiency (GOPs/W)*	*Infer/E2E Latencies (ms)*
ResNet-NLA [73]	71.4	56.4/-	-	16.8 (all)	10.0	300	ZCU102	42.5	100/-
SSGV3-K7 [72]	-	45/43.8	87/85.5	74 (peak)	4.8	-	VCK190	-	208/-
ResNet-GCM [71]	-	47.9/46.4	-	0.71/5.43 (idle/infer)	56.0	250	ZCU104	-	18/-
SalsaNext-pruned [74]	33.3	58.8/54.2	-	-	6.4	300	KV260	-	-/-
Proposed Work (DH)	125.7	*60.8/56.9*	*90.7/88.3*	5.18/9 (idle/infer)	5.6	300	KV260	78.2	175/259

In Table 6.8 SalsaNext-based models [55, 74] (including our models too) exhibit the best performance in terms of mIoU and accuracy. Additionally, our model achieves a computing efficiency of 78.2 GOPs/W and an inference power efficiency of 1.08 FPS/W. It is worth noting that there is a small, but important difference between our study and [55, 74], Since we changed the activation function from LeakyRELU to RELU6, the numbers during inference cannot explode, which makes the model suitable for fixed-point representation where the scale is limited. Without $LeakyReLU \longrightarrow ReLU6$ conversion we could not quantize our model: Either the tools gave an error, or the generated model graph is partitioned between CPU and DPU which reduces the inference speed and increases the power consumption. Another SalsaNext implementation in [74] achieved to convert the model with LeakyReLU because they applied QAT beforehand. However, before fine-tuning (QAT) their model achieves 58.8 mIoU while after QAT the quantized model achieves 54.2 mIoU. This means an overall 4.6% quantization error in mIoU, whereas our model experiences a lower (3.9%) drop even without QAT, thanks to ReLU6 conversion. We tried the Vitis quantization library to test our model performance with QAT, however, the training process was too slow, and we could not continue on that. However, we could apply QAT using the Pytorch quantization library (work with a similar principle as Vitis tools) to quantize our model to INT8 which resulted in 58% mIoU, meaning an even lower 2.9% quantization error in mIoU when applying QAT.

6.4. Conclusion

In this work, we deployed a real-time point cloud segmentation model on the Kria KV260, a cost-effective FPGA platform, utilizing the lightweight CNN-based *SalsaNext* model. The model was trained on two commonly used large-scale LiDAR semantic segmentation datasets, namely SemanticKITTI and the Open Waymo Dataset. To mitigate performance degradation caused by domain shifts between datasets, we proposed a domain adjustment schema, which is particularly beneficial for 2D range image-based models. Although the domain shift between these datasets is significant, our approach—incorporating proper weighting on loss functions and a double-headed model—achieved a +12.25% increase in the mean Intersection over Union (mIoU) metric on the Waymo validation dataset when cross-inference was applied to the KITTI head, owing to the domain adjustment and double-head training schema.

After training, the model was simplified by removing the Waymo head, resulting in a single compact model optimized for edge deployment. The compact model was further refined and quantized to 8 bits for efficient hardware execution. On the SemanticKITTI validation dataset, our implementation outperformed baseline models, achieving a mIoU of 56.9% and a frame rate of 5.6 fps. This represents only a 3.8% reduction in mIoU compared to its floating-point counterpart. Additionally, the quantized model achieved an overall accuracy of 88.3%, with a minor 2.4% reduction compared to the floating-point version.

Despite the relatively limited computational resources of the Kria KV260, our optimized approach delivered superior performance compared to other FPGA-based models on the SemanticKITTI validation dataset, achieving the highest reported mIoU and accuracy metrics. Qualitative analyses further highlight the model's consistent performance across both datasets, emphasizing its robustness and adaptability.

7. CONCLUSION

This thesis presented a comprehensive study on the hardware implementation of deep neural networks (DNNs) for 3D data processing, focusing on real-time deployment on edge devices. The ever-growing demand for efficient 3D perception in autonomous systems, robotics, and other applications necessitates a shift toward lightweight, optimized solutions capable of meeting the constraints of resource-limited hardware. The primary contributions of this work spanned model development, optimization, and deployment, addressing both depth estimation and semantic segmentation tasks.

7.1. Summary of Contributions

The first major focus was on depth estimation using 2D images. By adapting the MiDaSNet model for edge deployment, the architecture was quantized to an 8-bit fixed-point representation and tailored for FPGA platforms. Experimental results demonstrated that the quantized model achieved state-of-the-art performance on the NYUv2 dataset compared to other FPGA-based approaches. This highlights the feasibility of deploying computationally intensive tasks like depth estimation on power-constrained devices.

The second focus of the study was on semantic segmentation using point cloud data. A modified version of the SalsaNext architecture was implemented and optimized for edge applications. The proposed model was extensively evaluated on the SemanticKITTI and Open Waymo datasets, achieving competitive results against state-of-the-art FPGA-based implementations. The ability to process 3D point cloud data in real time showcases the effectiveness of the approach in scenarios such as autonomous navigation and robotics.

Beyond individual tasks, this work explored the broader challenges of deploying DNNs on edge devices. Techniques such as pruning, quantization-aware training, and

the use of specialized hardware frameworks, including Vitis AI, played a central role in achieving both computational efficiency and high accuracy. Furthermore, the training and evaluation pipelines were carefully designed to handle diverse datasets and models, enabling seamless adaptation to the constraints of edge implementations.

7.2. Challenges Addressed

The study also addressed several inherent challenges in edge AI deployments:

- *Resource Constraints:* Designing models that balance computational efficiency, memory usage, and inference latency without compromising accuracy is a critical task. This was achieved through systematic optimization techniques, including sparsity-aware operations and quantization.
- *Model Adaptation:* Transforming large, pre-trained models to operate on FPGA platforms involved significant architectural modifications, requiring careful consideration of hardware limitations and computational bottlenecks.
- *Domain Adaptation:* In the semantic segmentation task, domain adaptation techniques were applied to address challenges arising from differences in data distributions across training and deployment environments. This ensured improved model generalization and robustness, particularly when transitioning between datasets with varying characteristics.

7.3. Future Work

This work underscores the importance of bridging the gap between cutting-edge deep learning algorithms and practical hardware implementations. By optimizing models for edge deployment, this thesis contributes to making advanced AI technologies accessible for real-world applications where power and latency constraints are paramount.

Future research can expand on the findings of this thesis in several directions:

- *Multi-Sensor Fusion:* The integration of additional modalities, such as radar and camera data, can significantly enhance perception systems, enabling robust performance across diverse and challenging environments. In our study, we utilized a single-camera or LiDAR sensor setup due to the limitations of our acceleration board, which is among the simplest and most cost-effective options available. However, with more advanced hardware platforms, it is possible to implement sensor fusion techniques to further improve perception accuracy and reliability.
- *Advanced Hardware Acceleration:* Exploring next-generation FPGA architectures and ASIC-based accelerators can open new possibilities for ultra-efficient edge AI deployments.
- *8-bit Floating-Point Quantization:* In this work, we employed INT8 quantization for its simplicity and wide support. Alternatively, 8-bit floating-point (FP8) formats can be used to improve dynamic range, potentially reducing quantization errors—particularly when using unbounded activation functions like ReLU. However, deploying FP8 inference on FPGAs requires a complete redesign of the DPU pipeline, as current vendor-supplied DPU architectures do not support FP8 natively. Moreover, implementing an FP8 engine in hardware would demand significantly more programmable logic (PL) resources than INT8, which is already resource-intensive. It's also important to note that FP8 may offer limited advantages in scenarios using saturating or bounded activation functions such as ReLU6 or HardTanH, where quantization error is inherently constrained.
- *Domain Adaptation for Other Datasets:* In this thesis, the semantic segmentation task was trained using two LiDAR datasets. While this approach demonstrated strong performance, combining multiple datasets could further improve model generalization and robustness across diverse environments. Developing methods to seamlessly integrate and adapt to such varied datasets remains an important direction for future work.

In conclusion, this thesis demonstrated the feasibility and effectiveness of deploying advanced 3D data processing algorithms on edge hardware. By tackling the challenges of computational efficiency, data irregularity, and hardware optimization, it lays a solid foundation for future advancements in the field of real-time 3D perception and edge AI.

REFERENCES

1. Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention Is All You Need", ArXiv:1706.03762 [cs.CL], 2023.

2. Bai, L., Y. Lyu, X. Xu and X. Huang, "PointNet on FPGA for Real-Time LiDAR Point Cloud Processing", ArXiv:2006.00049 [eess.SP], 2020.

3. López, J. G., A. Agudo and F. Moreno-Noguer, "3D Vehicle Detection on an FPGA From LiDAR Point Clouds", *Proceedings of the 2019 2nd International Conference on Watermarking and Image Processing*, pp. 21–26, Marseille, France, 2020.

4. Lyu, Y., L. Bai and X. Huang, "ChipNet: Real-Time LiDAR Processing for Drivable Region Segmentation on an FPGA", *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 66, No. 5, pp. 1769–1779, 2019.

5. Kim, S., S. Kim, J. Lee and H.-J. Yoo, "A 54.7 Fps 3D Point Cloud Semantic Segmentation Processor With Sparse Grouping Based Dilated Graph Convolutional Network for Mobile Devices", *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, Seville, Spain, 2020.

6. Kala, S. and S. Nalesh, "Efficient CNN Accelerator on FPGA", *IETE Journal of Research*, Vol. 66, No. 6, pp. 733–740, 2020.

7. Shi, S., X. Wang and H. Li, "PointRCNN: 3D Object Proposal Generation and Detection From Point Cloud", ArXiv:1812.04244 [cs.CV], 2019.

8. Wu, B., X. Zhou, S. Zhao, X. Yue and K. Keutzer, "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation From a LiDAR Point Cloud", ArXiv:1809.08495 [cs.CV], 2018.

9. Graham, B. and L. van der Maaten, "Submanifold Sparse Convolutional Networks", ArXiv:1706.01307 [cs.NE], 2017.

10. Winner, H. *et al.*, H. Winner *et al.* (Editors), *Handbook of Driver Assistance Systems*, pp. 315–319, Springer, Cham, 2016.

11. Autoware Foundation Contributors, "Autoware: Open-Source Software for Autonomous Driving", 2024, `https://github.com/autowarefoundation/autoware`, accessed on December 2, 2024.

12. PyG Team, "PyTorch Geometric", 2024, `https://github.com/pyg-team/pytorch_geometric`, accessed on December 2, 2024.

13. OpenMMLab, "OpenMMLab: an Open-Source Computer Vision Toolbox", 2018, `https://github.com/open-mmlab`, accessed on December 29, 2024.

14. Team, O. D., "OpenPCDet: an Open-source Toolbox for 3D Object Detection From Point Clouds", 2020, `https://github.com/open-mmlab/OpenPCDet`, accessed on December 29, 2024.

15. Ravi, N., J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson and G. Gkioxari, "Accelerating 3D Deep Learning With PyTorch3D", ArXiv:2007.08501 [cs.LG], 2020.

16. Tang, H., Z. Liu, S. Zhao, Y. Lin, J. Lin, H. Wang and S. Han, "Searching Efficient 3D Architectures With Sparse Point-Voxel Convolution", *Computer Vision – ECCV 2020*, pp. 685—702, Glasgow, United Kingdom, 2020.

17. Liu, Z., H. Tang, A. Amini, X. Yang, H. Mao, D. L. Rus and S. Han, "BEVFusion: Multi-Task Multi-Sensor Fusion With Unified Bird's-Eye View Representation", *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6569—-6576, London, UK, 2023.

18. Li, Z., W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Y. Qiao and J. Dai, "BEVFormer: Learning Bird's-Eye-View Representation From Multi-camera Images Via Spatiotemporal Transformers", *Computer Vision – ECCV 2022*, pp. 1—-18, Cham, Germany, 2022.

19. Liang, T., J. Glossner, L. Wang, S. Shi and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: a Survey", *Neurocomputing*, Vol. 461, No. C, pp. 370—-403, 2021.

20. Yeom, S.-K., P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller and W. Samek, "Pruning by Explaining: a Novel Criterion for Deep Neural Network Pruning", *Pattern Recognition*, Vol. 115, No. 0031-3203, p. 107899, 2021.

21. Liu, J., S. Tripathi, U. Kurup and M. Shah, "Pruning Algorithms to Accelerate Convolutional Neural Networks for Edge Applications: a Survey", ArXiv:2005.04275 [cs], 2020.

22. Blalock, D., J. J. G. Ortiz, J. Frankle and J. Guttag, "What Is the State of Neural Network Pruning?", ArXiv:2003.03033 [cs.LG], 2020.

23. Sze, V., Y.-H. Chen, T.-J. Yang and J. S. Emer, "How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful", *IEEE Solid-State Circuits Magazine*, Vol. 12, No. 3, pp. 28–41, 2020.

24. Pedram, A., S. Richardson, M. Horowitz, S. Galal and S. Kvatinsky, "Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era", *IEEE Design Test*, Vol. 34, No. 2, pp. 39–50, 2017.

25. Wu, B., *Efficient Deep Neural Networks*, Ph.D. Thesis, University of California, Berkeley, 2019.

26. FastML Team, "HLS4ML", 2023, `https://github.com/fastmachinelearning/hls4ml`, accessed on January 13, 2024.

27. Duarte, J. *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics", *Journal of Instrumentation*, Vol. 13, No. 07, p. P07027, 2018.

28. Intel, *FPGA AI Suite Reference Manual*, 2023, `https://www.intel.com/content/www/us/en/docs/programmable/768974/2023-3/reference-manual.html`, accessed on Dec 22, 2025.

29. Cai, H., C. Gan, T. Wang, Z. Zhang and S. Han, "Once-for-All: Train One Network and Specialize It for Efficient Deployment", ArXiv:1908.09791 [cs.LG], 2020.

30. AMD, *DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338)*, 2024, `https://docs.amd.com/r/en-US/pg338-dpu/Features`, accessed on Dec 22, 2024.

31. Lin, Y., M. Yang and S. Han, "NAAS: Neural Accelerator Architecture Search", *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1—6, San Francisco, USA, 2021.

32. Adiyaman, M. Y. and I. F. Baskaya, "FPGA Implementation of CNN Based Depth Estimation Network: MiDaSNet", Research Square: rs-4465711/v1, 2024.

33. Swaraja, K., K. Naga Siva Pavan, S. Suryakanth Reddy, K. Ajay, P. Uday Kiran Reddy, P. Kora, K. Meenakshi, D. Chaitanya and H. Valiveti, "CNN Based Monocular Depth Estimation", *E3S Web Conf.*, Vol. 309, p. 01070, 2021.

34. Aguilar-González, A., M. Arias-Estrada and F. Berry, "Depth From a Motion Algorithm and a Hardware Architecture for Smart Cameras", *Sensors*, Vol. 19, No. 1, p. 53, 2019.

35. Bhatti, F. and T. Greiner, "FPGA Hardware Design for Plenoptic 3D Image Processing Algorithm Targeting a Mobile Application", *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*,

pp. 1—-5, Toronto, Ontario, Canada, 2021.

36. Li, Z., H. Sun, Y. Gao and J. Wang, "A Residual Network and FPGA Based Real-Time Depth Map Enhancement System", *Entropy (Basel)*, Vol. 23, No. 5, p. 546, 2021.

37. Marsi, S., S. Carrato, L. D. Bortoli, P. Gallina, F. Guzzi and G. Ramponi, "An FPGA Realization for Real-Time Depth Estimation in Image Sequences", S. Saponara and A. D. Gloria (Editors), *Applications in Electronics Pervading Industry, Environment and Society. ApplePies 2019. Lecture Notes in Electrical Engineering*, Vol. 627, pp. 269—-275, Springer, Cham, 2020.

38. Hashimoto, N. and S. Takamaeda-Yamazaki, "FADEC: FPGA-based Acceleration of Video Depth Estimation by HW/SW Co-design", *2022 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1—-6, Hong Kong, 2022.

39. Duzceker, A., S. Galliani, C. Vogel, P. Speciale, M. Dusmanu and M. Pollefeys, "DeepVideoMVS: Multi-View Stereo on Video With Recurrent Spatio-Temporal Fusion", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6946—6955, Nashville, Tennessee, USA, 2021.

40. ZiWen, D., L. YuQi and Y. Dong, "FasterMDE: a Real-Time Monocular Depth Estimation Search Method That Balances Accuracy and Speed on the Edge", *Applied Intelligence*, Vol. 53, pp. 24566–24586, 2023.

41. Sada, Y., N. Soga, M. Shimoda, A. Jinguji, S. Sato and H. Nakahara, "Fast Monocular Depth Estimation on an FPGA", *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 783—-792, New Orleans, LA, USA, 2020.

42. Liu, S., S. Zhao, P. Zhang and et al., "Real-Time Monocular Depth Estimation for Low-Power Embedded Systems Using Deep Learning", *J Real-Time Image*

Processing, Vol. 19, pp. 997–1006, 2022.

43. Ranftl, R., K. Lasinger, D. Hafner, K. Schindler and V. Koltun, "Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer", ArXiv:1907.01341 [cs.CV], 2020.

44. Tan, M. and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", ArXiv:1905.11946 [cs.LG], 2020.

45. Nathan Silberman, P. K., Derek Hoiem and R. Fergus, "Indoor Segmentation and Support Inference From RGBD Images", *European Conference on Computer Vision (ECCV)*, pp. 746—-760, Florence, Italy, 2012.

46. Eigen, D. and R. Fergus, "Predicting Depth, Surface Normals and Semantic Labels With a Common Multi-scale Convolutional Architecture", *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2650–2658, Santiago, Chile, 2015.

47. Xian, K. and et al., "Monocular Relative Depth Perception With Web Stereo Data Supervision", *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 311—320, Salt Lake City, UT, USA, 2018.

48. Laina, I., C. Rupprecht, V. Belagiannis, F. Tombari and N. Navab, "Deeper Depth Prediction With Fully Convolutional Residual Networks", *2016 Fourth International Conference on 3D Vision (3DV)*, pp. 239—-248, Stanford, CA, USA, 2016.

49. Lin, T.-Y., M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick and P. Dollár, "Microsoft COCO: Common Objects in Context", ArXiv:1405.0312 [cs.CV], 2015.

50. Adiyaman, M. Y. and F. Baskaya, "FPGA Implementation of Double-head SalsaNext: a CNN-based Model for LiDAR Point Cloud Segmentation", *Journal of*

Real-Time Image Processing, Vol. 22, No. 2, p. 78, Mar 2025.

51. Wu, X., L. Jiang, P.-S. Wang, Z. Liu, X. Liu, Y. Qiao, W. Ouyang, T. He and H. Zhao, "Point Transformer V3: Simpler Faster Stronger", *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1—11, Seattle, WA, USA, 2024.

52. Lai, X., Y. Chen, F. Lu, J. Liu and J. Jia, "Spherical Transformer for LiDAR-Based 3D Recognition", *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 17545–17555, Vancouver, Canada, 2023.

53. Zhou, Y. and O. Tuzel, "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection", ArXiv:1711.06396 [cs.CV], 2017.

54. Aksoy, E. E., S. Baci and S. Cavdar, "SalsaNet: Fast Road and Vehicle Segmentation in LiDAR Point Clouds for Autonomous Driving", *2020 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1—-7, Nevada, USA, 2020.

55. Cortinhal, T., G. Tzelepis and E. Erdal Aksoy, "SalsaNext: Fast, Uncertainty-Aware Semantic Segmentation of LiDAR Point Clouds", *15th International Symposium on Visual Computing (ISVC)*, pp. 207—-222, San Diego, CA, USA, 2020.

56. Xu, C., B. Wu, Z. Wang, W. Zhan, P. Vajda, K. Keutzer and M. Tomizuka, "SqueezeSegV3: Spatially-Adaptive Convolution for Efficient Point-Cloud Segmentation", *Computer Vision – ECCV 2020*, pp. 463—480, Cham, 2020.

57. Milioto, A., I. Vizzo, J. Behley and C. Stachniss, "RangeNet ++: Fast and Accurate LiDAR Semantic Segmentation", *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4291—-4298, Macau, China, 2019.

58. Liu, Y., R. Chen, X. Li, L. Kong, Y. Yang, Z. Xia, Y. Bai, X. Zhu, Y. Ma, Y. Li, Y. Qiao and Y. Hou, "UniSeg: A Unified Multi-Modal LiDAR Segmentation Network and the OpenPCSeg Codebase", *Proceedings of the IEEE/CVF*

International Conference on Computer Vision (ICCV), pp. 21543—21554, Paris, France, 2023.

59. Xu, X., L. Kong, H. Shuai and Q. Liu, "FRNet: Frustum-Range Networks for Scalable LiDAR Segmentation", ArXiv:2312.04484 [cs.CV], 2024.

60. Thomas, H., C. R. Qi, J.-E. Deschaud, B. Marcotegui, F. Goulette and L. J. Guibas, "KPConv: Flexible and Deformable Convolution for Point Clouds", ArXiv:1904.08889 [cs.CV], 2019.

61. Bai, Y., G. Li, C. Yang, Y. Li, Q. Xiao and Z. Li, "Differential Graph Convolution Network for Point Cloud Understanding", *Neurocomputing*, Vol. 597, p. 127940, 2024.

62. Qi, C. R., L. Yi, H. Su and L. J. Guibas, "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space", ArXiv:1706.02413 [cs.CV], 2017.

63. Kong, L., Y. Liu, R. Chen, Y. Ma, X. Zhu, Y. Li, Y. Hou, Y. Qiao and Z. Liu, "Rethinking Range View Representation for LiDAR Segmentation", *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 228–240, Paris, France, 2023.

64. Kleesiek, J., G. Urban, A. Hubert, D. Schwarz, K. Maier-Hein, M. Bendszus and A. Biller, "Deep MRI Brain Extraction: a 3D Convolutional Neural Network for Skull Stripping", *NeuroImage*, Vol. 129, pp. 460–469, 2016.

65. Özgün Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox and O. Ronneberger, "3D U-Net: Learning Dense Volumetric Segmentation From Sparse Annotation", ArXiv:1606.06650 [cs.CV], 2016.

66. Zhou, Y. and O. Tuzel, "VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection", ArXiv:1711.06396 [cs.CV], 2017.

67. Qi, C. R., H. Su, K. Mo and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation", ArXiv:1612.00593 [cs.CV], 2017.

68. Bai, L., Y. Lyu, X. Xu and X. Huang, "PointNet on FPGA for Real-Time LiDAR Point Cloud Processing", *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1—-5, Online, 2020.

69. Zhang, X., Z. Huang, A. G. Gonzalez, W. Jachimczyk and X. Huang, "Stream-Based Ground Segmentation for Real-Time LiDAR Point Cloud Processing on FPGA", ArXiv:2408.10410 [eess.SP], 2024.

70. Zhang, X., Z. Huang, A. G. Gonzalez, W. Jachimczyk and X. Huang, "Parallel Processing of Point Cloud Ground Segmentation for Mechanical and Solid-State LiDARs", ArXiv:2408.10404 [cs], 2024.

71. Xie, X., L. Bai and X. Huang, "Real-Time LiDAR Point Cloud Semantic Segmentation for Autonomous Driving", *Electronics*, Vol. 11, No. 1, p. 11, 2022.

72. Delgado, P. P. F., *Real-time Implementation of 3D LiDAR Point Cloud Semantic Segmentation in an FPGA*, Master's Thesis, University of Minho, 2023.

73. Bai, L., Y. Zhao and X. Huang, "A Near Sensor Edge Computing System for Point Cloud Semantic Segmentation", *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1—-5, Austin, Texas, USA, 2022.

74. Xilinx, I., *Vitis AI Model Zoo: Reference Guide*, 2024, `https://xilinx.github.io/Vitis-AI/3.0/html/docs/reference/ModelZoo_VAI3.0_Github_web.htm`, accessed on December 14, 2024.

75. NVIDIA, *NVDLA: Open Source Deep Learning Inference Accelerator*, 2025, `https://nvdla.org`, accessed on February 2, 2025.

76. AMD, *Xilinx Deep Learning Processor Unit (DPU)*, 2023, `https://docs.amd.`

com/r/3.3-English/pg338-dpu, accessed on February 2, 2025.

77. He, K., X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition", ArXiv:1512.03385 [cs.CV], 2015.

78. Chen, L.-C., G. Papandreou, F. Schroff and H. Adam, "Rethinking Atrous Convolution for Semantic Image Segmentation", ArXiv:1706.05587 [cs.CV], 2017.

79. Yi, L., B. Gong and T. Funkhouser, "Complete and Label: a Domain Adaptation Approach to Semantic Segmentation of LiDAR Point Clouds", ArXiv:2007.08488 [cs.CV], 2021.

80. Wu, B., X. Zhou, S. Zhao, X. Yue and K. Keutzer, "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation From a LiDAR Point Cloud", ArXiv:1809.08495 [cs.CV], 2018.

81. Saleh, K., A. Abobakr, M. Attia, J. Iskander, D. Nahavandi and M. Hossny, "Domain Adaptation for Vehicle Detection From Bird's Eye View LiDAR Point Cloud Data", ArXiv:1905.08955 [cs.CV], 2019.

82. Qin, C., H. You, L. Wang, C.-C. J. Kuo and Y. Fu, "PointDAN: a Multi-Scale 3D Domain Adaption Network for Point Cloud Representation", *Advances in Neural Information Processing Systems*, pp. 7190–7201, Vancouver, Canada, 2019.

83. Berman, M., A. R. Triki and M. B. Blaschko, "The Lovász-Softmax Loss: a Tractable Surrogate for the Optimization of the Intersection-over-union Measure in Neural Networks", ArXiv:1705.08790 [cs.CV], 2018.

84. Charles, R. Q., H. Su, M. Kaichun and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation", *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 77–85, Honolulu, Hawaii, USA, 2017.

85. PyTorch, "3D Deep Learning With PyTorch3D", 2020, `https://youtu.be/Pph1r-x9nyY`, accessed on Sept 9, 2025.

86. Tang, H., Z. Liu, S. Zhao, Y. Lin, J. Lin, H. Wang and S. Han, "Searching Efficient 3D Architectures With Sparse Point-Voxel Convolution", ArXiv:2007.16100 [cs.CV], 2020.

87. Cai, H., C. Gan, T. Wang, Z. Zhang and S. Han, "Once-for-All: Train One Network and Specialize it for Efficient Deployment", ArXiv:1908.09791 [cs.LG], 2020.

88. Mildenhall, B., P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis", *Communications of the Association for Computing Machinery*, Vol. 65, No. 1, pp. 99—106, 2021.

89. Wang, Y., Q. Mao, H. Zhu, J. Deng, Y. Zhang, J. Ji, H. Li and Y. Zhang, "Multi-Modal 3D Object Detection in Autonomous Driving: a Survey", *International Journal of Computer Vision*, Vol. 131, No. 8, pp. 2122—-2152, 2023.

90. Sun, P., H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, S. Zhao, S. Cheng, Y. Zhang, J. Shlens, Z. Chen and D. Anguelov, "Scalability in Perception for Autonomous Driving: Waymo Open Dataset", ArXiv:1912.04838 [cs.CV], 2020.

91. Qi, C. R., Y. Zhou, M. Najibi, P. Sun, K. Vo, B. Deng and D. Anguelov, "Offboard 3D Object Detection From Point Cloud Sequences", ArXiv:2103.05073 [cs.CV], 2021.

92. Ettinger, S., S. Cheng, B. Caine, C. Liu, H. Zhao, S. Pradhan, Y. Chai, B. Sapp, C. Qi, Y. Zhou, Z. Yang, A. Chouard, P. Sun, J. Ngiam, V. Vasudevan, A. Mc-

Cauley, J. Shlens and D. Anguelov, "Large Scale Interactive Motion Forecasting for Autonomous Driving : the Waymo Open Motion Dataset", ArXiv:2104.10133 [cs.CV], 2021.

93. MIT-Han-Lab, "BEVFusion Repository", 2024, `https://github.com/mit-han-lab/bevfusion`, accessed on September 25, 2024.

94. Vision, F., "BEVFormer Repository", 2022, `https://github.com/fundamentalvision/BEVFormer`, accessed on December 20, 2024.

95. Team, O. D., "OpenPCDet: an Open-source Toolbox for 3D Object Detection From Point Clouds", 2020, `https://github.com/open-mmlab/mmdetection`, accessed on September 25, 2024.

96. FastML, "Machine Learning on FPGAs Using HLS", 2023, `https://github.com/fastmachinelearning/hls4ml`, accessed on September 24, 2024.

97. Duarte, J. *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics", *JINST*, Vol. 13, No. 07, p. P07027, 2018.

98. Aarrestad, T. *et al.*, "Fast convolutional neural networks on FPGAs with hls4ml", *Machine Learning Science and Technology*, Vol. 2, No. 4, p. 045015, 2021.

99. Ghielmetti, N. *et al.*, "Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml", *Machine Learning Science and Technology*, Vol. 3, No. 4, p. 045011, 2022.

100. Ngadiuba, J. *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML", *Machine Learning Science and Technology*, Vol. 2, No. 1, p. 015001, 2021.

101. Apex.AI, "Online Lecture Notes of Autoware.auto Project", 2020, `https://gitlab.com/ApexAI/autowareclass2020/-/tree/master/lectures`, accessed

on September 23, 2024.

102. Ghiasi, G., Y. Cui, A. Srinivas, R. Qian, T.-Y. Lin, E. D. Cubuk, Q. V. Le and B. Zoph, "Simple Copy-Paste is a Strong Data Augmentation Method for Instance Segmentation", ArXiv:2012.07177 [cs.CV], 2021.

103. Wang, C.-Y., A. Bochkovskiy and H.-Y. M. Liao, "Scaled-YOLOv4: Scaling Cross Stage Partial Network", ArXiv:2011.08036 [cs.CV], 2021.

104. He, K., G. Gkioxari, P. Dollár and R. Girshick, "Mask R-CNN", ArXiv:1703.06870 [cs.CV], 2018.

105. Ren, S., K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection With Region Proposal Networks", ArXiv:1506.01497 [cs.CV], 2016.

106. Mardirosian, R., `https://www.autonomoustechconf.com/sites/autosensorsconf/files/assets/6D%20LiDAR%20FaceOff%20Ouster_Mardirosian.pdf`, accessed on January 20, 2024.

107. Hochreiter, S. and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, Vol. 9, No. 8, pp. 1735–1780, 1997.

108. Kramer, M. A., "Nonlinear Principal Component Analysis Using Autoassociative Neural Networks", *Aiche Journal*, Vol. 37, No. 2, pp. 233–243, 1991.

109. Iandola, F. N., S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally and K. Keutzer, "SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and ¡0.5MB Model Size", ArXiv:1602.07360 [cs.CV], 2016.

110. Denton, E. L., W. Zaremba, J. Bruna, Y. LeCun and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation", *Advances in Neural Information Processing Systems*, pp. 1279—1287, Montreal, Quebec,

Canada, 2014.

111. Han, S., J. Pool, J. Tran and W. Dally, "Learning Both Weights and Connections for Efficient Neural Network", *Advances in Neural Information Processing Systems*, pp. 1135–1143, Montreal, Quebec, Canada, 2015.

112. Han, S., H. Mao and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization and Huffman Coding", ArXiv:1510.00149 [cs.CV], 2016.

113. Sabour, S., N. Frosst and G. E. Hinton, "Dynamic Routing Between Capsules", *Advances in Neural Information Processing Systems*, pp. 3859—-3869, Long Beach, California, USA, 2017.

114. Huang, Q., Y. Zhang, J. Zheng, G. Shang and G. Chen, "A CNN-Based Real-Time Dense Stereo SLAM System on Embedded FPGA", L. Fang, J. Pei, G. Zhai and R. Wang (Editors), *Artificial Intelligence. CICAI 2023. Lecture Notes in Computer Science*, Vol. 14474, pp. 1—-12, Springer, Singapore, 2024.

115. Tatar, G. and S. Bayar, "Real-Time Multi-Task ADAS Implementation on Reconfigurable Heterogeneous MPSoC Architecture", *IEEE Access*, Vol. 11, pp. 80741–80760, 2023.

116. Xilinx, "Hardware Architecture for NLP Acceleration on the Kria KV260 Vision AI Starter Kit", 2022, `https://xilinx.github.io/kria-apps-docs/kv260/2022.1/build/html/docs/nlp-smartvision/docs/hw_arch_accel_nlp.html`, accessed on January 13, 2024.

117. López, J. G., A. Agudo and F. Moreno-Noguer, "3D Vehicle Detection on an FPGA From LiDAR Point Clouds", *2nd International Conference on Watermarking and Image Processing*, pp. 21–26, Marseille, France, 2019.

118. Haidar, S. A., A. Chariot, M. Darouich, C. Joly and J.-E. Deschaud, "Are We

Ready for Real-Time LiDAR Semantic Segmentation in Autonomous Driving?", ArXiv:2410.08365 [cs.RO], 2024.

119. Xie, X., H. Wei and Y. Yang, "Real-Time LiDAR Point-Cloud Moving Object Segmentation for Autonomous Driving", *Sensors*, Vol. 23, No. 1, p. 547, 2023.

120. Liu, C., J. Zhao and N. Sun, "A Transformer-based Real-time LiDAR Semantic Segmentation Method for Restricted Mobile Devices", *Journal of the Franklin Institute*, Vol. 361, No. 4, p. 106632, 2024.

121. Intel Corporation, *FPGA AI Suite IP Reference Manual*, 2024, `https://www.intel.com/content/www/us/en/docs/programmable/768974/2024-3/about-the.html`, accessed on December 31, 2024.

122. NVIDIA Corporation, *NVDLA Primer*, 2024, `https://nvdla.org/primer.html`, accessed on December 18, 2024.

123. AMD, *Vitis AI Overview*, 2023, `https://docs.amd.com/r/en-US/ug1414-vitis-ai/Vitis-AI-Overview`, accessed on December 18, 2024.

124. Lyu, Y., L. Bai and X. Huang, "ChipNet: Real-Time LiDAR Processing for Drivable Region Segmentation on an FPGA", *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 66, No. 5, pp. 1769–1779, 2019.