Universidade do Minho
Escola de Engenharia

Pedro Paulo Fontes Delgado

Real-Time Implementation of 3D LiDAR Point Cloud Semantic Segmentation in an FPGA

October 2022

Real-Time Implementation of 3D LiDAR Point Cloud Semantic Segmentation in an FPGA

Pedro Paulo Fontes Delgado

Uminho | 2022

Universidade do Minho
Escola de Engenharia

Pedro Paulo Fontes Delgado

Real-Time Implementation of 3D LiDAR Point Cloud Semantic Segmentation in an FPGA

Master Dissertation

Master in Informatics Engineering

Dissertation supervised by

Sanaz Asgarifar

Victor Alves

October 2022

DECLARATION

Name: Pedro Paulo Fontes Delgado

Dissertation Title: Real-Time Implementation of 3D LiDAR Point Cloud Semantic Segmentation in an FPGA

Supervisors: Sanaz Asgarifar, Victor Alves

Conclusion Year: 2022

Master Designation: Master in Informatics Engineering

Master Branch: Machine Learning and Data Science

University of Minho, _____/_____/_____

Signature: _____

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

List of publications

Delgado, P., Asgarifar, S., Alves, V. Real-Time Implementation of Squeezeseg-V3 Semantic Segmentation Using Vck190 FPGA Board. Submitted to WorldCist'23 - 11st World Conference on Information Systems and Technologies

Acknowledgements

A lot of people have contributed to the successful writing of this thesis. To all of you, I am truly grateful to have shared this journey. Without you, this would have not been possible.

I would like to start by thanking both my supervisors, Sanaz Asgarifar and professor Victor Alves for giving me the freedom to take this exploratory work where I wanted it to go.

I give out all my appreciation to Alexandre Correia for encouraging me to explore a whole new area of study. I truly believe it had a tremendously positive impact on my development professionally and more importantly, personally.

To the FPGA team at Bosch, who received me with open arms and created the best working environment, thank you. Without your amazing knowledge, this work would simply have not been possible.

To my friends, who have supported me and allowed me to escape the loneliest and most stressful hours. Thank you for all the game nights, outings, and the most foolish debates.

To my parents who I profoundly admire and whom I see as amazing role models. Thank you for your open-minded education and amazing childhood. Without you, I would not be who I am.

To Mariana, my sister, and Artur, my nephew, with whom I can always count on for an amazingly good time. Thank you.

To João, my brother, who I also admire greatly. You inspire me to be my best self so you can hopefully have an additional role model.

To my dear partner, Sofia, for being there for me from the beginning of this journey, in the good and the hardest times. Thank you for the endless ideas, and meaningful debates and for always allowing me to trust myself. Your dedication and resilience inspire me. Your happiness drives me.

Pedro Delgado

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, _____/_____/_____

Signature: _____

ABSTRACT

In the last few years, the automotive industry has relied heavily on deep learning applications for perception solutions. With data-heavy sensors, such as LiDAR, becoming a standard, the task of developing low-power and real-time applications has become increasingly more challenging. To obtain the maximum computational efficiency, no longer can one focus solely on the software aspect of such applications, while disregarding the underlying hardware.

In this thesis, a hardware-software co-design approach is used to implement an inference application leveraging the *SqueezeSegV3*, a LiDAR-based convolutional neural network, on the *Versal ACAP VCK190* FPGA. Automotive requirements carefully drive the development of the proposed solution, with real-time performance and low power consumption being the target metrics.

A first experiment validates the suitability of *Xilinx's Vitis-AI* tool for the deployment of deep convolutional neural networks on FPGAs. Both the *ResNet-18* and *SqueezeNet* neural networks are deployed to the *Zynq UltraScale+ MPSoC ZCU104* and *Versal ACAP VCK190* FPGAs. The results show that both networks achieve far more than the real-time requirements while consuming low power. Compared to an *NVIDIA RTX 3090 GPU*, the performance per watt during both network's inference is 12x and 47.8x higher and 15.1x and 26.6x higher respectively for the *Zynq UltraScale+ MPSoC ZCU104* and the *Versal ACAP VCK190* FPGA. These results are obtained with no drop in accuracy in the quantization step.

A second experiment builds upon the results of the first by deploying a real-time application containing the *SqueezeSegV3* model using the *Semantic-KITTI* dataset. A framerate of 11 Hz is achieved with a peak power consumption of 78 Watts. The quantization step results in a minimal accuracy and IoU degradation of 0.7 and 1.5 points respectively. A smaller version of the same model is also deployed achieving a framerate of 19 Hz and a peak power consumption of 76 Watts. The application performs semantic segmentation over all the point cloud with a field of view of 360°.

Keywords: LiDAR, Deep Learning, FPGA

RESUMO

Nos últimos anos a indústria automóvel tem cada vez mais aplicado deep learning para solucionar problemas de perceção. Dado que os sensores que produzem grandes quantidades de dados, como o LiDAR, se têm tornado standard, a tarefa de desenvolver aplicações de baixo consumo energético e com capacidades de reagir em tempo real tem-se tornado cada vez mais desafiante. Para obter a máxima eficiência computacional, deixou de ser possível focar-se apenas no software aquando do desenvolvimento de uma aplicação deixando de lado o hardware subjacente.

Nesta tese, uma abordagem de desenvolvimento simultâneo de hardware e software é usada para implementar uma aplicação de inferência usando o SqueezeSegV3, uma rede neuronal convolucional profunda, na FPGA *Versal ACAP VCK190*. São os requisitos *automotive* que guiam o desenvolvimento da solução proposta, sendo a performance em tempo real e o baixo consumo energético, as métricas alvo principais.

Uma primeira experiência valida a aptidão da ferramenta *Vitis-AI* para a implantação de redes neuronais convolucionais profundas em FPGAs. As redes *ResNet-18* e *SqueezeNet* são ambas implantadas nas FPGAs *Zynq UltraScale+ MPSoC ZCU104* e *Versal ACAP VCK190*. Os resultados mostram que ambas as redes ultrapassam os requisitos de tempo real consumindo pouca energia. Comparado com a *GPU NVIDIA RTX 3090*, a performance por Watt durante a inferência de ambas as redes é superior em 12x e 47.8x e 15.1x e 26.6x respetivamente na *Zynq UltraScale+ MPSoC ZCU104* e na *Versal ACAP VCK190*. Estes resultados foram obtidos sem qualquer perda de *accuracy* na etapa de quantização.

Uma segunda experiência é feita no seguimento dos resultados da primeira, implantando uma aplicação de inferência em tempo real contendo o modelo SqueezeSegV3 e usando o conjunto de dados *Semantic-KITTI*. Um *framerate* de 11 Hz é atingido com um pico de consumo energético de 78 Watts. O processo de quantização resulta numa perda mínima de *accuracy* e IoU com valores de 0.7 e 1.5 pontos respetivamente. Uma versão mais pequena do mesmo modelo é também implantada, atingindo uma *framerate* de 19 Hz e um pico de consumo energético de 76 Watts. A aplicação desenvolvida executa segmentação semântica sobre a totalidade das nuvens de pontos LiDAR, com um campo de visão de 360°.

Palavras-chave: LiDAR, Deep Learning, FPGA

Table of Contents

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations and acronyms

A

ADAM	Adaptive Moment
ADAS	Advanced Driver-Assistance Systems
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit

B

BEV	Bird's Eye View
BNN	Binary Neural Network
BRAM	Block Random Access Memory

C

CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Compute Unit

D

DDR	Double Data Rate
DPU	Deep Learning Processing Unit

F

FLOPS	Floating-point Operations
FPGA	Field Programmable Gate Array
FPS	Frames per Second
FV	Front View

G

GPU	Graphical Processing Unit
GOPS	Giga Operations
GFLOPS	Giga Floating-Point Operations

H

HW	Hardware

I

ICP	Input Channel Parallelism
IoU	Intersection over Union
IP	Intellectual Property

L

LiDAR	Light Detection and Ranging

M

mAP	Mean Average Precision
mIoU	Mean Intersection Over Union

O

OCP	Output Channel Parallelism

P

PE	Processing element
PP	Pixel Parallelism
PS	Processing Subsystem

Q

QAT	Quantization-aware training

R

ReLU	Rectified Linear Unit

S

SAC	Spatially Adaptive Convolution
SIMD	Single Instruction Multiple Data

T

TNN	Ternary Neural Network
TOF	Time of Flight
TOPS	Tera Operations
TPU	Tensor Processing Unit

U

URAM Ultra Random Access Memory

V

VART Vitis-AI Runtime

X

XIR Xilinx Intermediate Representation

XRT Xilinx Runtime

GLOSSARY

Activation function Any mathematical function used in a neural network's layers to introduce non-linearity in the output of the layer's neurons.

Batch (Training) A hyperparameter that defines the number of instances to process before updating the internal model parameters during training of a neural network.

Batch (Inference) The number of instances to inference over in parallel.

Bit-width (Quantization) The number of bits necessary to represent an integer as a binary number.

Computer Vision A field of Artificial Intelligence concerned with the extraction of high-level information from digital visual inputs such as images, videos and point clouds.

Cross-entropy Loss A loss function used to measure the performance of a classification model that outputs probabilities between 0 and 1. Its value increases as the predicted probability diverges from the actual value.

Deep Learning A subset of Machine Learning based on Deep Neural Networks in which multiple layers of processing are used to extract progressively higher-level features from data.

Deep Neural Network In contrast with Shallow Neural Networks, typically referred to as just Neural Networks, Deep Neural Networks have multiple layers between the input and output layers.

Deep Learning Model The resulting weights and biases of a Deep Neural Network fitted to the training data.

Feature An individual measurable property or characteristic of the input. For example, age, gender, weight of a person or the number and location of a specific pattern in an image/point cloud.

Feature Map Each feature map or activation map is the result of convolving an image using a kernel/filter of a CNN and passing it through an activation function.

Filter / Kernel (CNN) Set of learnable weights spatially structured that learn to extract relevant patterns when convolved over the input.

Floating-point representation A system to represent, with a fixed number of digits, real numbers of different orders of magnitude. Differs from fixed-point representation by allowing a variable number of integer and fractional digits.

Loss Function A function that is used to evaluate how well an algorithm models a dataset.

Neural Network Neural Networks are a subset of Machine Learning algorithms that combine a set of progressively learned functions to model the given training data and perform predictions. Each individual function is composed of a set of parameters, namely the weights and biases that are combined through the notion of a neuron and passed through an activation function. The way individual functions are combined is through the notion of layers.

Off-chip memory Memory that resides outside the chip where the computations happen. It has higher access latency but is also bigger than on-chip memory.

On-chip memory Memory that resides in the same chip where the computations happen. It has very low access latency but is also very small.

Point Cloud A set of points in 3D space representing one or more objects in a scene.

1 Introduction

1.1 CONTEXT

LiDAR (Light Detection and Ranging) sensors have been widely recognized as key components for advanced driver-assistance systems (ADAS) and autonomous driving as they enable the tri-dimensional mapping of objects. The additional information extracted by the LiDAR is critical for the central processing unit of the vehicle to perceive the surrounding scenario. Evidence for this trend is the ever-growing adoption of the LiDAR sensor in the sensor suite of autonomous driving solutions in the market [1], [2], [3], [4], [5], [6], [7].

Additionally, the research in autonomous systems has seen dramatic advances in recent years, due to the increase in available computing power and reduced cost in sensing, computing technologies, and price of the necessary hardware, resulting in the maturing technological willingness to produce fully autonomous vehicles. Figure 1 shows the forecast from Statista, a company that specializes in market and consumer data, for the projected sales of autonomous vehicles worldwide from 2019 to 2030. The growth in sales evidences the wide adoption of autonomous driving solutions in the automotive market. Several other sources also estimate the increasing demand and market size of autonomous driving-related products in the automotive market [8], [9].

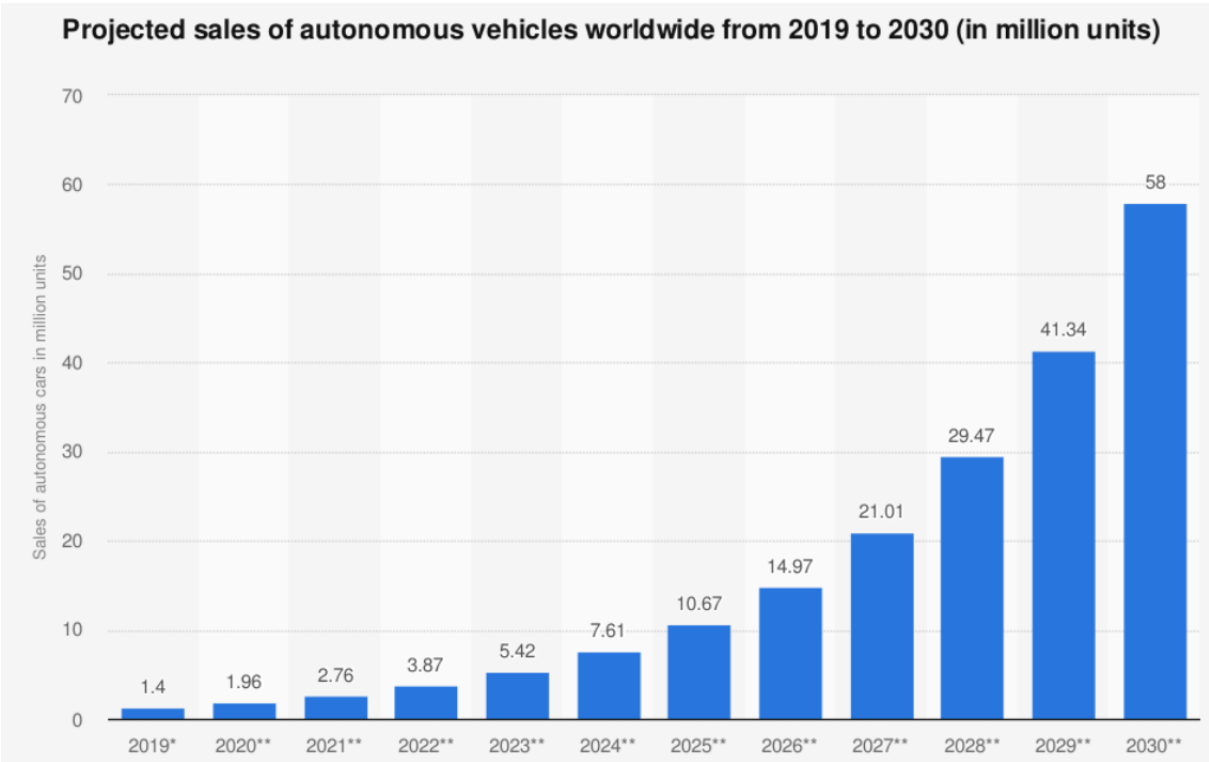

Figure 1. Forecast of the worldwide autonomous vehicles sales from 2019 to 2030. Retrieved from [10].

The core competencies of an autonomous vehicle system are classified into three categories, namely perception, planning, and control [11]. While machine learning algorithms based on deep neural networks have demonstrated great performance in several complex cognitive tasks [12], [13], a significant gap in the energy and efficiency of the computational systems that implement perception algorithms still exists [14]. Most of these algorithms run on conventional computing systems such as Central Processing Units (CPUs) and General-Purpose Graphical Processing Units (GPUs). Alternatives containing embedded hardware solutions, such as Field Programmable Gate Arrays (FPGAs), have started to be explored to help develop solutions that allow more efficient computation of deep neural networks [15], [16].

1.2 MOTIVATION

Several works have demonstrated the capabilities of FPGAs for designing real-time applications that perform deep neural network inference on FPGAs [17], [18], [19]. Some works specifically focus on the implementation of such neural networks using LiDAR data [20], [21], [22]. The possibility to concurrently design hardware and software, a feature of FPGAs, allows for the exploration of more efficient solutions. Not only can one adapt the software for the underlying hardware, but the hardware itself can also be finetuned for the specific application. More, this process can be done iteratively.

With the advent of high-level tools [23], [24] that open the hardware-software co-design research to machine learning and deep learning engineers without FPGA expertise, it becomes possible to increase the efficiency of deep neural networks by reducing power consumption while maintaining the desired framerate.

1.3 OBJECTIVES

The main goal of this work is to train and deploy a deep neural network on an FPGA to perform one perception task using LiDAR sensor data, in the form of a point cloud. Power consumption should be kept to a minimum while maximizing framerate, without losing sight of accuracy metrics. Hardware limitations are also expected to heavily impact the development of a solution and so the search for fitting neural network architectures and consequent computation layers should be carried out with them in mind. A review of the available tools for deep neural network deployment on FPGAs should be conducted. Once a tool is selected, a thorough exploration of the capabilities of the tool, as well as an evaluation of its suitability for the development of a real-time, low-power inferencing application, should be validated.

During experiments, framerate and power consumption should be the target metrics along with the appropriate accuracy metrics of the perception task being solved.

1.4 STRUCTURE OF DISSERTATION

Chapter 2 introduces the technologies and concepts relevant to the understanding of this thesis. Chapter 3 includes a revision of past literature works that include hardware, algorithms, and works with a similar scope to this work. Chapters 4 and 5 contain the two experiments conducted in this thesis. The first explores *Vitis-AI*, the main tool used in this work, by validating its suitability for the objectives of this work. The second implements the proposed solution to the problem this thesis aims to solve. Both chapters 4 and 5 contain individual results, analysis, and respective discussions. Lastly, chapter 6 examines the work and concludes its impact on the problems it aimed to solve. It also contains a section dedicated to suggesting further research opportunities.

2 Technologies And Concepts

2.1 PERCEPTION IN AUTONOMOUS DRIVING

A highly autonomous system must understand its environment by solving highly complex cognitive tasks that allow it to respond to every situation. As a result, self-driving cars rely heavily on software to bridge the gap between sensor information and mechanical vehicle actuation, such as steering and braking. From the collection and processing of sensory data to the control of the vehicle's actuators, there is a system that encapsulates it all.

2.1.1 ADVANCED DRIVER-ASSISTANCE SYSTEMS

ADAS are the electronic systems in a vehicle that use advanced technologies to assist the driver. They use a combination of sensor technologies to perceive the world around the vehicle, and then either provide information to the driver by issuing warnings or actively controlling the vehicle when necessary. To do so, across the desired route, the system should be able to perceive its surroundings and extract high-level information which may be critical for safe navigation. The consequent steps consist in taking the extracted information to plan a set of actions and performing them by controlling the vehicle actuators i.e., the devices that transform an input signal into motion. Following this description, one can distinguish 3 main modules namely perception, planning, and control [11]. Figure 2 presents a diagram of an ADAS containing all 3 modules and their respective interactions.

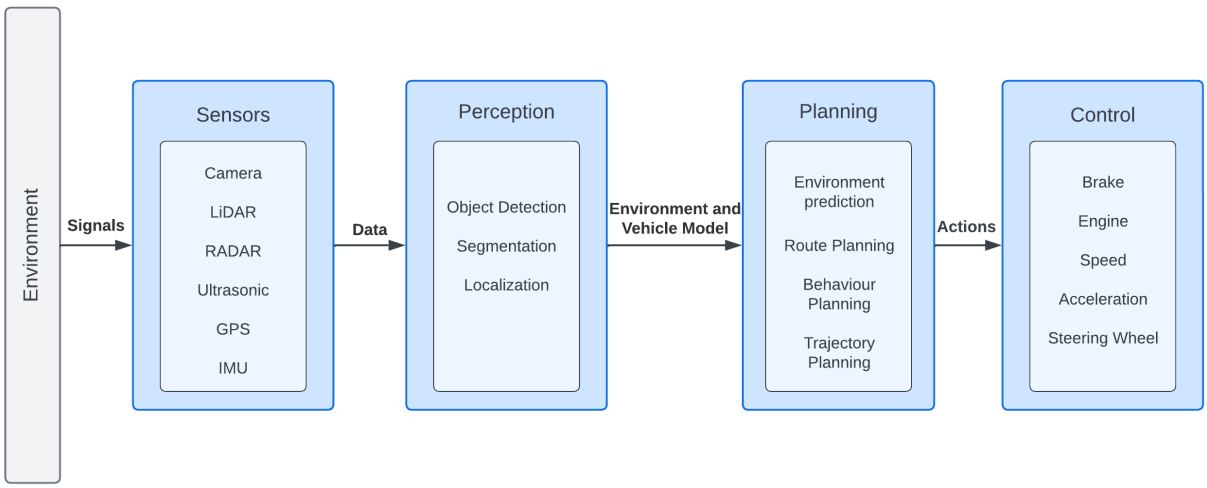

Figure 2. Block diagram of an ADAS.

The perception module directly actuates on the raw sensory data collected by an array of sensors and is responsible for the extraction of relevant features from the multiple sensors' output. These features represent components of the vehicle's surroundings that influence the driving task. Having all these components correctly perceived is necessary for a perception module of a high-level automation ADAS and requires highly efficient and accurate perception algorithms.

2.1.2 ADAS PERCEPTION REQUIREMENTS AND METRICS

Highly accurate, low response time, low energy consumption, and minimal physical size are four fundamental requirements of an ADAS identified in this work. There are other requirements, such as the relevancy of the information provided, that are also fundamental. However, these four are directly linked with performance and efficiency, the focus of this work. Metrics are usually defined to quantify how compliant an application or system is to a set of requirements. Refresh rate, which is linked to the response time requirement, directly depends on the rate at which the sensors can produce the data to be used by the perception module. It is then important to establish a value that considers the LiDAR solutions currently available in the market as well as upcoming solutions. 3.1 defines a specific value for the refresh rate by listing the LiDAR sensors currently available in the market and future solutions.

Contrarily to refresh rate, it is hard to define a maximum value for metrics associated with accuracy, energy consumption, and physical size since they either depend on the current perception algorithms, the available hardware, and the vehicle(s) that the ADAS will target. Therefore, this work will not establish specific values for these metrics. Instead, it will explore and propose solutions that keep these values as close as possible to their optimal values. This should be accomplished by carefully reviewing state-of-the-art solutions and choosing appropriate hardware architectures to deploy the best-suited perception algorithms.

2.2 LIDAR SENSOR

The LiDAR's ability to produce an extremely accurate three-dimensional position of surrounding objects and its innate robustness to exterior lighting conditions has pushed the adoption of this sensor in a large range of automotive perception solutions [1], [2], [3].

2.2.1 WORKING PRINCIPLE

LiDAR is an active remote sensing system. It is active because it generates energy, in this case, light, to collect data about its surroundings and remote because it does so by detecting the energy that is reflected from the surfaces. One of the techniques used in LiDAR to collect depth information is through what is called the time of flight (ToF). In ToF, an emitter fires short laser pulses that reflect off surrounding objects and are captured by the receiver. Since the emitter and receiver are approximately at the same position, it is possible to calculate the distance to the reflecting object using the known speed of light and the delay between the emission and reception of the laser. Usually, LiDAR sensors have multiple emitter-receiver pairs.

2.2.2 POINT CLOUDS

The spatially organized LiDAR data is referred to as a point cloud, a set of points with three-dimensional position and intensity information of the reflecting surfaces in the field of view. Depending on the field of view and resolution of the sensor, point clouds can easily become extremely large, usually, 100k-200k 3D points per frame, which results in a total size of around 1.6MB-3.2MB considering the usual format of 4 floating point values to represent x, y, z coordinates and intensity information. However, if the resolution of the LiDAR is not sufficiently high, the effective range is reduced i.e., objects that are distant from the sensor might become underrepresented or even completely undetectable. Figure 3 depicts a LiDAR's effective range. The same is true for the field of view which delimits, both horizontally and vertically, the surrounding volume scanned. The coordinates of objects that lie outside the field of view will naturally be absent from the point cloud.

Although point clouds are extremely useful to the perception module, several challenges emerge when processing point cloud data, and should be addressed when designing perception algorithms. These challenges include variability in point density, diversified measured intensity, inter-class reflectivity overlap, noise, sparsity, permutation and rigid transformation invariance, and occlusions.

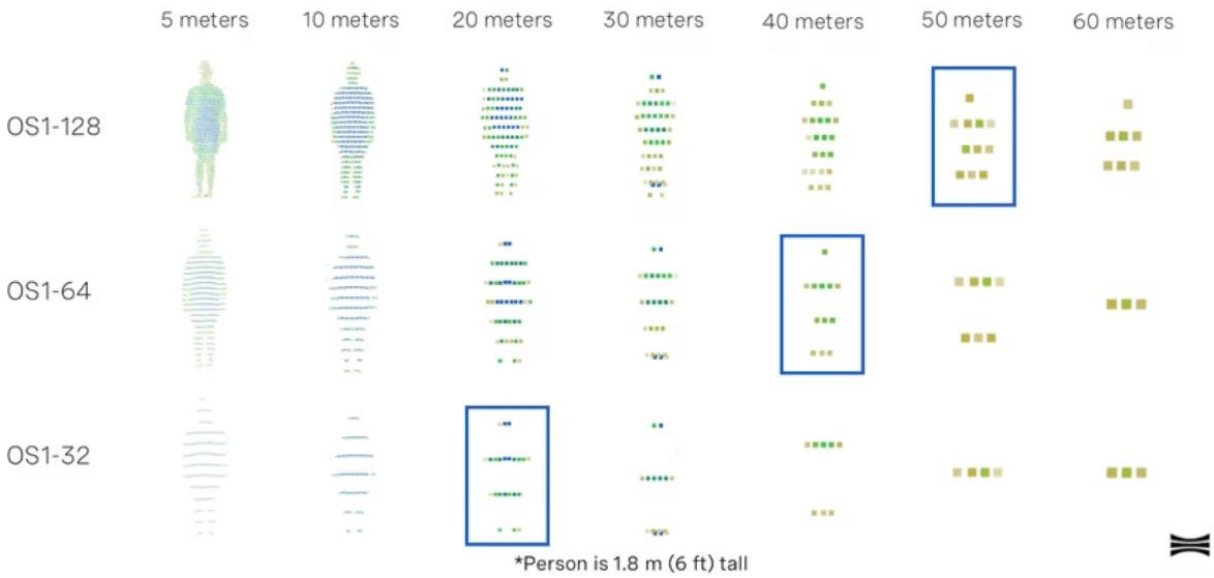

Figure 3. The effective range of Outer's OS1 LiDAR. Adapted from [25].

2.3 DEEP LEARNING IN POINT CLOUDS

Advancements in device capability involving computing power, sensor resolution, and cost-effectiveness, as well as the adoption of highly parallel hardware such as GPUs, have broken most of the barriers to the adoption of deep learning. Also, the increasing availability of high-quality and high-volume datasets highly benefits deep learning models contrary to more traditional approaches that struggle with high-volume data.

2.3.1 THE DEEP LEARNING APPROACH

In the last few years, deep learning approaches have achieved state-of-the-art results in multiple perception tasks involving images, sound, and text. However, a similar level of success for 3D computer vision is only now beginning to take shape, mainly due to the larger amount of data and complexity that point clouds encompass compared to images and the hardware limitations that become even more apparent with data-heavy point clouds.

A typical computer vision pipeline consists of two distinct phases. The first phase usually called feature extraction and more recently, feature learning, consists of the extraction of descriptive or informative patches in the data called features or sometimes also called descriptors. Specifically, in point clouds, features are usually spatial and geometric attributes or relationships between points. The second phase

of the pipeline, which usually consists of a classifier or regressor, is responsible for performing classification or regression or both based on the previously extracted features.

With the adoption of deep learning, both the feature extraction and classification or regression are done "end-to-end" with a deep learning-based model, meaning that the input of the model is the point cloud, and the output is a classification or regression tensor. This leaves out the need for the cumbersome and error-prone process of manual feature extraction that usually leads to poorly generalizable models. Currently, state-of-the-art results on perception tasks in point clouds use end-to-end deep learning models as pipelines. Figure 4 compares both vision pipelines.

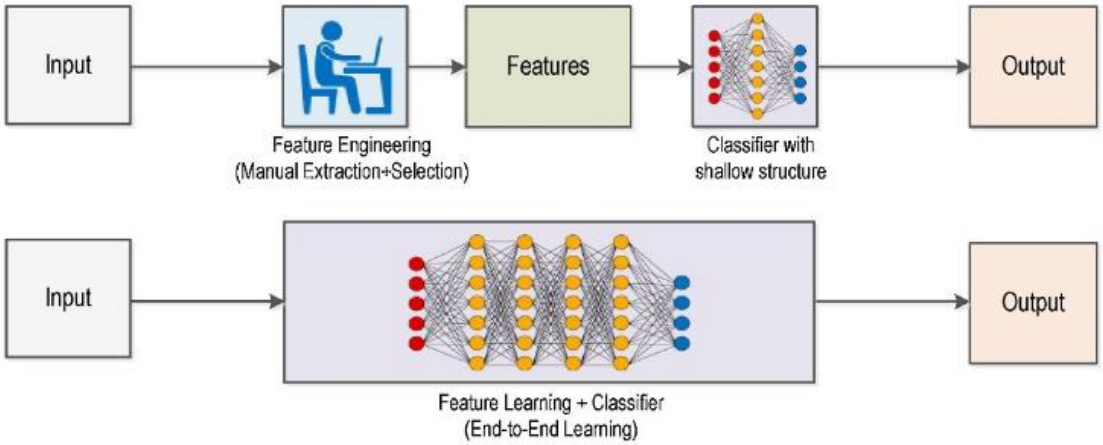

Figure 4. Comparison between a traditional and DL computer vision pipeline. Retrieved from [26].

2.3.2 POINT CLOUD PERCEPTION TASKS

Similarly to 2D computer vision, to evaluate deep learning models on point cloud data, there are a set of established perception tasks. Particularly in autonomous driving perception, this work highlights three.

2.3.2.1 3D OBJECT DETECTION

The goal of 3D Object Detection is to encapsulate every instance belonging to a set of predefined categories in the point cloud with an oriented 3D bounding box and an associated semantic label. As portrayed in Figure 5, the bounding box information can be represented using the coordinates (x, y, z) of the bounding box center, (h, w, l) representing respectively the height, width, and length of the bounding box, θ representing the object's yaw orientation and y_i representing the class the object corresponds to. An assumption made about the bounding boxes is that the objects are on the ground plane, and so their orientation can be described only using the yaw angle.

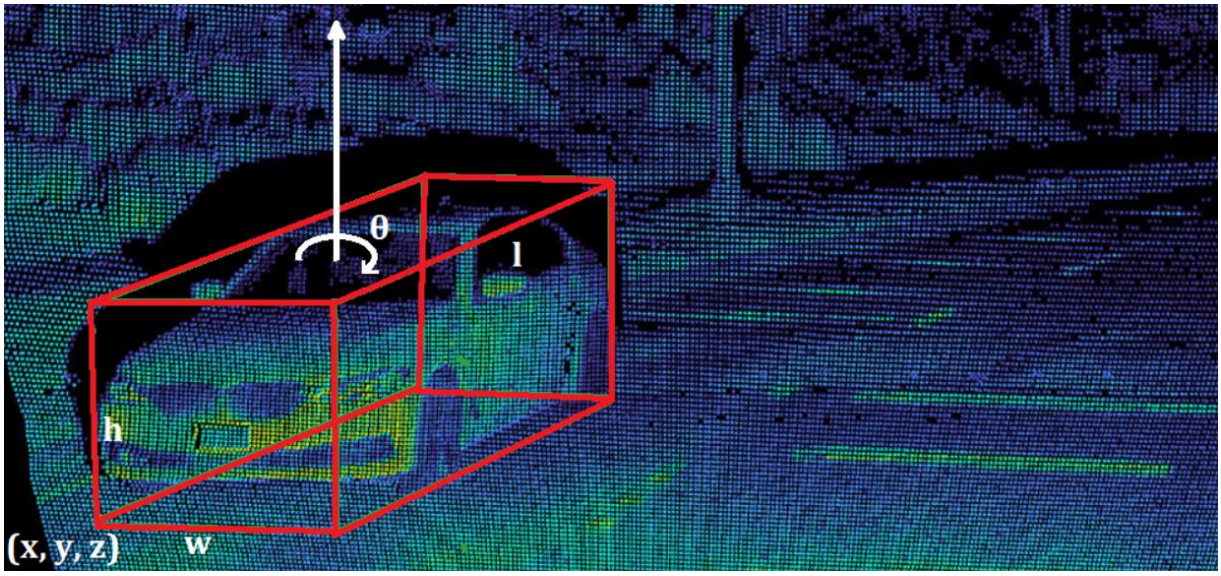

Figure 5. 3D object detection in a LiDAR frame.

2.3.2.2 3D SEMANTIC SEGMENTATION

3D point cloud segmentation aims to label homogeneous regions of a point cloud according to what they are representing. A more formal definition of the task is to assign every 3D point from the point cloud $X = \{x_1, x_2, \dots, x_N\}$ with a semantic or instance label y_i from a set $Y = \{y_1, y_2, \dots, y_K\}$ representing K distinct categories. Segmentation can be subdivided into sub-tasks by the different levels of granularity. At the coarser level, there is semantic segmentation where each group of points is represented by a semantic label such as road, car, or building. This type of segmentation is illustrated in Figure 6. At the intermediate level, there is instance segmentation which is not only trying to distinguish points based on their semantic meaning but also separating different instances with the same semantic meaning. This refers to the case where the objective is to not only identify that a group of points represents a car but to be able to distinguish different cars by assigning each of the groups a different instance label. Finally, the more fine-grained sub-task is part segmentation where several parts of a semantic region are distinguished. For example, from a group of points representing a car, segment the windshield, tires, etc.

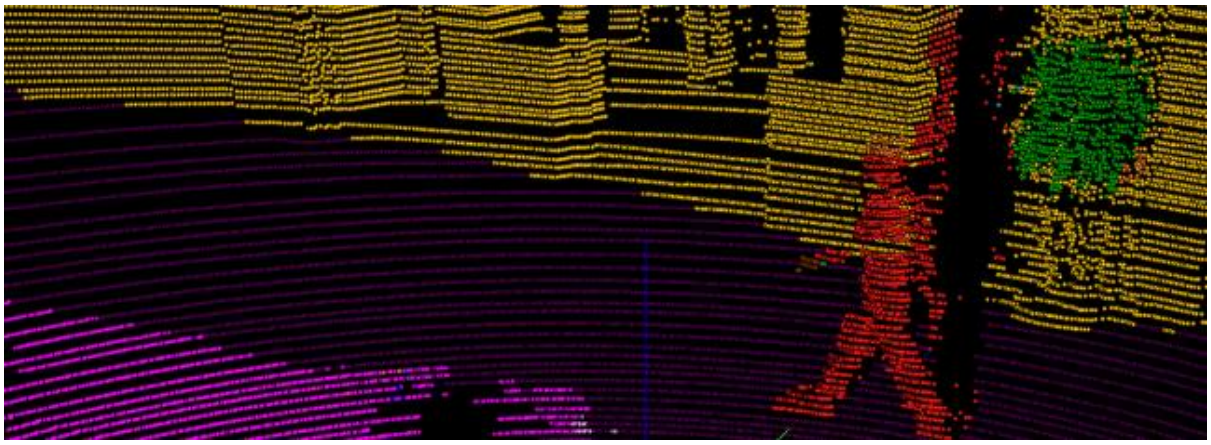

Figure 6. 3D semantic segmentation of a LiDAR point cloud.

2.3.2.3 3D OBJECT TRACKING

Given the locations and labels of a set of objects in a frame, the task of object tracking is to estimate their state in subsequent frames. A naive approach to the problem could be using object detection over all frames, but one obvious problem arises from this solution. If multiple objects are in the frame, the label associated with each object should remain unchanged over the following frames and object detection treats each frame independently. So, there is no way to guarantee that the detector attributes the same label to the detected objects over all frames. To solve this issue, one possible solution is to model the motion of the object i.e., the dynamic object's heading and velocity so that the most likely position in future frames can be predicted, effectively reducing the search space of a detector.

2.3.3 POINT CLOUD REPRESENTATION

Due to the unstructured and sparse nature of the point clouds, some transformations are usually carried out to generate a structured representation. The following sections present and explain the different representations, providing examples of deep learning models that use the representations.

2.3.3.1 PROJECTION-BASED

2D deep learning on images has achieved remarkable results using deep convolutional architectures on tasks such as image classification [27], [28], object detection [29], [30], and semantic segmentation [31]. Besides, well-established 2D datasets containing a lot of data, such as ImageNet [32], are readily available leveraging the application of deep convolutional models pre-trained on these datasets to 2D images. However, the convolution operation is performed on data that is ordered, regular, and on a

structured grid. For this reason, to benefit from the performance of established 2D deep convolutional networks, a natural approach is structuring point clouds in a way that allows the application of 2D convolution operations. One way to achieve this is by performing a projection of the 3D point cloud into a 2D grid. Several projection schemes have been used in different works where predominantly two main schemes are used: Front View (FV) [33], [22] and Bird's Eye View (BEV) [34], [35], [36]. Both are illustrated in Figure 7.

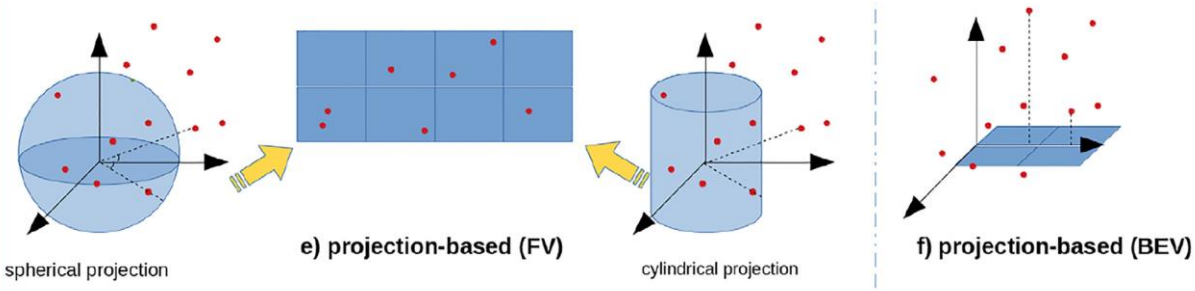

Figure 7. Illustration of spherical, cylindrical, and bird's eye view projections of point clouds. Adapted from [37].

Unfortunately, there is a discretization inherent to the projection operation which results in a loss of information. Because the 2D grid has a limited resolution, several points in the point cloud are likely to end up in the same grid coordinate. There are various ways to deal with this situation. Xu, C. et. al only keep the point with the largest range value $r = \sqrt{x^2 + y^2 + z^2}$. Another possible approach is to combine the x, y, z, and intensity values of all the points through some average or even a small multi-layered perceptron. *PointPillars* [38] is a good example of the latter approach.

2.3.3.2 VOXEL-BASED

A different approach that allows the application of convolutions directly on 3D point clouds is through what is called a voxel-based representation. However, in this approach, the convolution operations used are 3D convolutions.

A voxel is a volume element that represents a specific grid value in 3D space. Voxel-based approaches partition the [L, W, H] 3D point cloud into fixed-sized voxels through voxelization by assigning points in the point cloud to voxels according to their 3D coordinates. The voxel represents all the points assigned to itself by combining features of those points. Figure 8 portrays the voxelization process of a point cloud of an airplane.

However beneficial to the use of convolution operations, a voxel-based representation has some limitations. Firstly, not all voxels will carry important information because point clouds have denser and sparser zones. The sparser zones may contain lots of empty voxels. This results in a memory inefficient representation of the 3D space and wasted computation when applying 3D convolutions [39]. Secondly, because the computational and memory cost increases cubically with the increase in voxel resolution, there is a limit on the total number of voxels, usually around 30^3 [39].

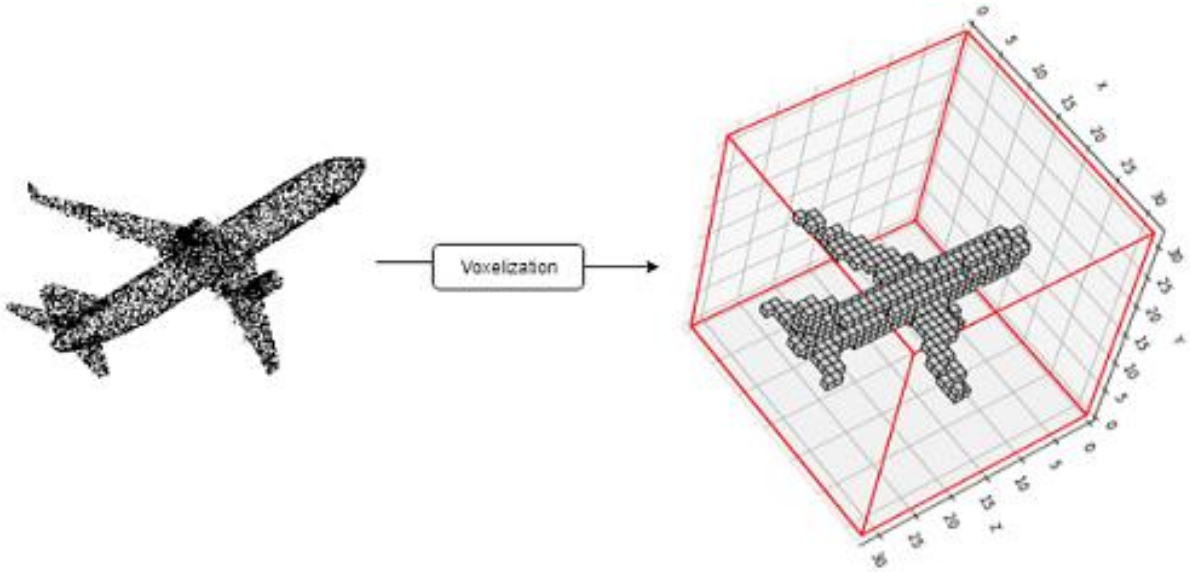

Figure 8. Voxelization of a point cloud using 30^3 voxels. Retrieved from [40].

2.3.3.3 POINT-BASED

Both projection-based and voxel-based representations discretize the point cloud resulting in a loss of information. Contrarily, the point-based approach looks to fully exploit the 3D geometry and shape of the point cloud without information loss.

As noted in the work of Shi, S. et al., projection and voxel-based representations are more computationally efficient, but lose fine-grained localization information, while point-based approaches don't lose so much information, but result in a higher computational cost [41]. Similarly, Deng, J. et al. also suggest that point-based approaches can better retain precise point positions while having a higher computational overhead compared to projection and voxel-based representations [42].

2.3.3.4 GRAPH-BASED

Graph-based approaches convert the point cloud into a graph, as illustrated in Figure 9. The nodes of the graph correspond to the points and the edges represent the relationship between point neighbors

inside a fixed radius. The explicit representation of the relationship between point neighbors through the graph edges is good for modeling the correlation between points in the point cloud [40], so more local spatial correlation features can be extracted from the grouped edge relationships on each node [43]. Recently, more works explore this representation to solve 3D perception tasks [44], [45], [46], [47].

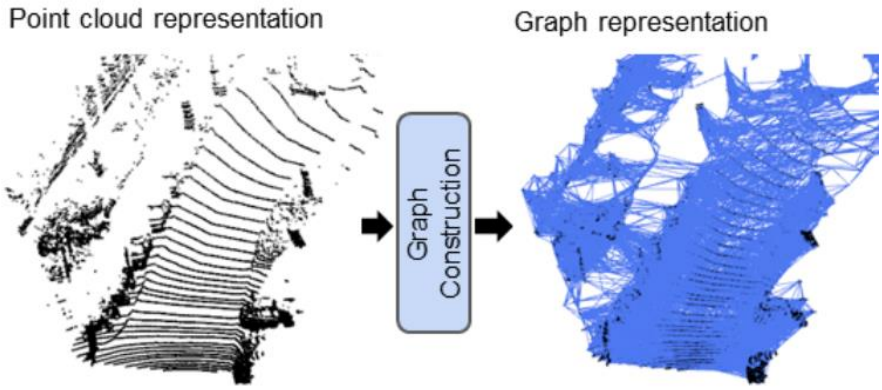

Figure 9. Illustration of a graph representation of a point cloud. Adapted from [48]

2.4 DEEP NEURAL NETWORK COMPRESSION

Although difficult to prove, deeper neural network parameter count has been long observed to be positively correlated with accuracy. From one extreme where natural language processing models have up to billions of parameters [12], to the other where smaller models are designed to fit in embedded hardware [49], there is a need to reduce the size of neural networks, and the computation needed to run them, without compromising accuracy.

2.4.1 QUANTIZATION

Historically most neural networks are trained using 32-bit floating point values. The core idea behind quantization is to reduce the representation of weights and biases, usually to 16-bit, 8-bit, 4-bit, or even 2-bit and single-bit integers. The challenge is to map the set of possible values of a neural network's parameters to a fixed discrete set of integers, effectively minimizing the number of bits required to represent the values. Since activation outputs are usually between 0 and 1 e.g., sigmoid, or at least can be bounded by a low integer value e.g., relu6, the weights of a neural network usually remain within a reasonable small range of values and consequently are good candidates for being represented using lower bit-widths [50].

Besides the obvious reduction in model size, there are added benefits of using lower-bit integer representations such as a reduction in energy consumption and inference latency. Chen, Q. et al.

compared an 8-bit fixed-point adder and multiplier to a 32-bit floating point adder and multiplier concluding that the energy and area of a fixed-point adder and multiplier scale approximately linearly and quadratically respectively with the number of bits used for representation [51]. Also, if the model does not fit on local/on-chip memory, and off-chip memory must be accessed, the lower bandwidth inherent to this access, when compared to local memory access, is a major bottleneck of inference latency [52]. Furthermore, off-chip memory accesses result in orders of magnitude higher energy consumption [23]. For these reasons, model size reduction can decrease inference latency by allowing for the exploration of memory locality. Even if it is not possible to avoid off-chip memory accesses, it is advantageous to have lower bit-width representations since it improves the memory bandwidth i.e., the cost of moving information is smaller. One last advantage of lower bit-width representations is the exploration of Single Instruction Multiple Data (SIMD) [50].

The quantization problem can be seen as the mapping operation of floating-point values in a predetermined range of values to integer values that can be represented with b bits. The quantization and de-quantization operations can be described as

$$\begin{cases} x = S(x_q + Z) \\ x_q = (\frac{1}{S} \cdot x - Z) \end{cases} \qquad \text{(Equations 1 and 2)}$$

where $x \in [\alpha, \beta]$ are the floating-point values, and $x_q \in [\alpha_q, \beta_q]$ are the quantized values. For a b-bit representation, $[\alpha_q, \beta_q]$ would be equal to $[-2^{b-1}, 2^{b-1} - 1]$ and $[0, 2^b - 1]$ respectively when using signed and unsigned integers to represent the quantized values. S and Z are variables that must be derived. Appendix I contains the derivation of S and Z.

Their values are

$$\begin{cases} S = \dfrac{\beta - \alpha}{\beta_q - \alpha_q} \\ Z = round\left(\dfrac{\alpha \cdot \beta_q - \beta \cdot \alpha_q}{\beta - \alpha}\right) \end{cases} \qquad \text{(Equations 3 and 4)}$$

The above quantization mapping is known as affine quantization. In the special case where Z is forced to have the value 0, the name scale quantization or symmetric quantization is given. Figure 10 and Figure 11 respectively depict affine and scale quantization using 8-bit integers.

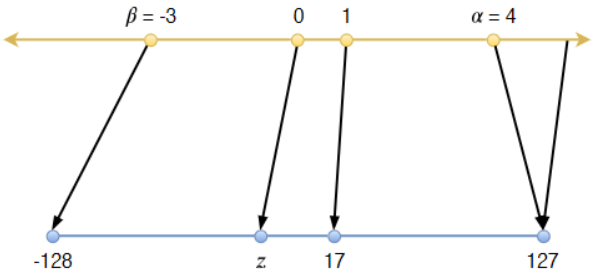

Figure 10. Affine quantization using signed 8-bit integers. Retrieved from [53].

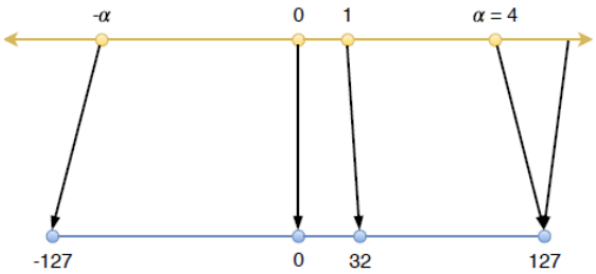

Figure 11. Scale quantization using unsigned 8-bit integers. Retrieved from [53].

There is one potential downside to quantizing neural networks. Usually, an accuracy drop can be observed, especially at lower bit-widths. This is to be expected as the range of values that can be encoded is halved with each removed bit. However, the drop in accuracy is usually not as significant and several quantization techniques have been shown to preserve accuracy, even on the more challenging models to quantize [54]. A review of the proposed quantization techniques and their results can be found in 3.3.

2.4.2 OTHER TECHNIQUES

Pruning, usually used alongside quantization, is the process of removing part of a neural network's parameters, namely the weights while ensuring that the model's performance doesn't drop below a specified threshold. Typically, a pruning pipeline consists of first training a network, then pruning the model according to a specific strategy, and finally fine-tuning the pruned network to compensate for the performance loss. This is done iteratively and in each iteration N number of parameters are removed. However, if the percentage of pruned parameters is high, the matrices representing model weights become sparse. Consequently, matrix operations become harder to accelerate and memory-bound [55].

More techniques have been proposed to reduce neural network size, improve energy efficiency and reduce inference latency, such as low-rank factorization [56] and knowledge distillation [57]. However, like quantization and pruning, those target the neural networks. A different particularly interesting approach is point cloud sampling which consists of sub-sampling the point cloud by preserving the original structure while reducing the number of points. Random sampling and farthest point sampling are the two traditional sampling algorithms [37]. Lang, Itai, Manor, Asaf, and Avidan Shai argue that traditional

sampling approaches do not consider the perception task that the network consuming the point cloud as input is performing. For this reason, they propose a technique that learns task-specific sampling, improving results significantly [58].

3 Literature Review

3.1 AUTOMOTIVE LIDAR REFRESH RATE

LiDAR sensors can usually be configured to operate at different frame rates, allowing them to suit different tasks and scenarios. The maximum frame rate of each sensor is of utmost importance, as it defines the minimal real-time response time that perception algorithms must adhere to. Table 1 lists LiDAR sensors' frame rate as well as information about each sensor's market release year. The selection criteria for the devices detailed in this section prioritizes devices by reputable, industry-leading LiDAR manufacturers - some of which already have commercially available devices, like *Continental*, *Valeo,* and *Ouster* - or startup companies that have established themselves by developing state-of-the-art LiDAR technologies, as is the case of *Innoviz* and *Baraja*. The results from Table 1 show that the frame rate is typically below 30 Hz and that the lower bound, although with some exceptions, is usually 5 Hz. From these values, one may estimate a minimum response time of 33 ms for the perception algorithms. However, it should be noted that higher frame rates result in lower resolutions, regardless of the sensor technology. A lot of the below listed LiDARs allow regulating this resolution/frame rate trade-off by having a refresh rate interval rather than a single value. Examples are *Velodyne's HDL-64E* and *VLS-128*, *Ouster OS2-128*, *Innoviz's InnovizTwo* and *Innoviz 360*, and *Baraja Spectrum HD25*. One can note that in these sensors, substantially lower resolutions result from higher frame rates. Perception systems rely heavily on the resolution of point clouds, especially for identifying small objects and road segments. For this reason, a refresh rate of 10 Hz seems to offer very reasonable resolutions on the listed LiDARs without compromising heavily on frame rate. And the data in the table does suggest that a refresh rate of 10 Hz is widely supported. Hereby, a frame rate of 10 Hz is the reference value for the perception algorithms explored throughout this work.

Table 1. Market released and future automotive LiDAR sensors (references in the table).

LiDAR Sensor	Refresh Rate (Hz)	Angular Resolution (H x V)	Market Release
Velodyne HDL-64E **[59]**	5 - 20	(0.08°- 0.35°) x 0.4°	2007
Velodyne VLS-128 **[60]**	5 - 20	(0.08° - 0.35°) x 0.11°	2017
Ibeo Lux **[61]**	25	0.25° x 0.8°	2018
Ouster OS2-128 **[62]**	10 or 20	(0.7° - 0.18°) x 0.18°	2020
Continental HFL110 **[63]**	25	0.94° x 0.94°	2021
Luminar Iris **[64]**	1 - 30	0.05° x 0.05° *	2022
Innoviz InnovizTwo **[65]**	10/15/20	0.05° x 0.05° *	2022
Innoviz 360 **[66]**	0.5 - 25	0.05° x 0.05° *	2022 (Q4)

Baraja Spectrum HD25 **[67]**	4 - 30	0.04° x 0.0125° *	> 2022
Continental HRL131**[68]**	10	0.05° x 0.075°	2024 **
Valeo Scala GEN1**[69]**	25	0.25° x 0.8°	2024 **

*Highest possible resolutions

**Expected

3.2 DEEP LEARNING HARDWARE

Deep learning models notoriously require lots of computation and memory during inference. With the limited energy consumption in the vehicle and the hardware limitations that it creates, it becomes natural to consider offloading some of the computation outside the vehicle through the network. However, due to network limitations in communication bandwidth, latency, and reliability, only offline tasks, usually consisting of offline model retraining and map generation, can be performed on the cloud [70]. This means that, concerning real-time perception, currently the best solution is to use the paradigm of edge computing which tries to bring the computation as close as possible to the data sources, and the sensors, effectively placing the hardware inside the vehicle. This trend is very visible in the data plot in Figure 12. Given the above-mentioned reasons, it is important to understand the different types of hardware available with especial attention to inference latency, memory, and energy consumption constraints.

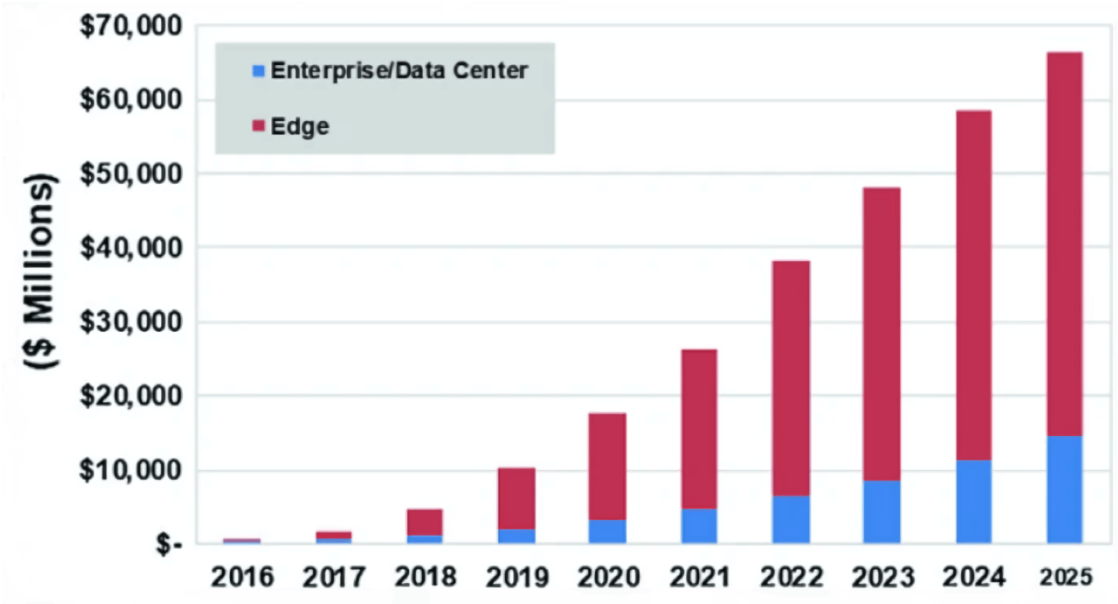

Figure 12. Deep learning chip revenue. Retrieved from [71].

3.2.1 CENTRAL PROCESSING UNITS

CPUs are the most versatile of all the hardware since they can perform almost any type of computation and are unavoidably present in almost every system. This makes them the easiest and less time-consuming hardware to deploy neural network applications on, as less effort is needed to support even the most novel and exotic neural network layers. Even the more common layers, which usually translate into vector to vector or matrix to matrix operations, are supported by low-level linear algebra routines in libraries such as *OpenBLAS* [72] and *Intel MKL* [73].

Most deep learning applications, even when accelerators are present, will inevitably use a CPU for receiving sensor data, data pre/post-processing, or control flow operations. This dependency makes CPUs a strong contender for deep learning inference since there is no latency bottleneck in transferring data like in a CPU-GPU application [74].

However, there is a tradeoff between versatility and resource efficiency. CPUs, being on one extreme of this spectrum are usually not optimized for any specific application. In some cases, where metrics such as energy consumption and inference latency are crucial, specialized hardware is the only solution.

3.2.2 GRAPHICAL PROCESSING UNITS

Although designed for graphical processing tasks, GPUs have become the standard hardware solution for training deep learning models since R. Raina, A. Madhavan, and A. Y. Ng proposed its usage over CPUs, remarkably reducing the training time of models [75]. Their highly parallel nature allows for the efficient computation of linear algebra operations, especially when transferring data in large batches [76], therefore reducing memory accesses outside the GPU and consequently optimizing GPU resource utilization. GPU programming has also become more accessible due to parallel programming tools such as *CUDA* [77] and *OpenCL* [78].

The introduction of tensor cores [79], specially designed for optimizing matrix operations and supporting various lower bit-width representations, further increased the applicability of GPUs for DNN training. Also, Nvidia reports a latency reduction in inference by utilizing tensor cores [80], which opens the usage of GPUs in latency-restricted applications such as ADAS. However, their high energy consumption is a hard limiting factor for their use in such systems. The study conducted by Gawron, H. J. et al. estimates a 3% increase in energy consumption between an autonomous and a non-autonomous vehicle with roughly half of the consumption due to the perception hardware (excluding sensors) [81]. This percentage can become more significant if the cooling of the hardware is considered as noted by

Lin. S. et al. [82]. To address these limitations, various efforts have been made to design and implement mobile GPUs with reduced power consumption, such as *Nvidia RTX* embedded GPU solutions [83] offering as low as 35W maximum power consumption.

3.2.3 APPLICATION-SPECIFIC INTEGRATED CIRCUITS

Similarly to CPUs, GPUs are multi-purpose hardware solutions. This means that, despite the efforts made by graphics card manufacturers to add specialized hardware components such as tensor cores to their cards, GPUs are limited by the fact that they are a multi-purpose solution. Application-Specific Integrated Circuits (ASICs) are, as the name suggests, hardware that is specifically designed to optimize performance for a small set of applications. A well-known example of an ASIC widely used in deep learning is Tensor Processing Units (TPUs) specifically designed by Google for accelerating linear algebra computations [84]. TPUs excel when training models that are heavily dominated by matrix computations but tend to suffer from severe performance degradation when frequent branching or element-wise operations [84]. This inability to perform outside of the specific target application constraints is a typical pitfall of ASICs. Wang, Y. et al. benchmarked *Google's TPU v3* and an *Nvidia V100 GPU* in the training of DL models such as *ResNet-50* [27] and *SqueezeNet* [85], concluding that TPUs consistently provided a considerable speedup in DL model training over GPUs [86].

In the landscape of edge computing, where reduced inference latency and power consumption are the main constraints, there are several ASICs designed to optimize inference latency rather than training time, while keeping the energy consumption low. Examples are *Tesla's Full Self-Driving Chip* [87], and *Mobileye's EyeQ5* [88]. Although achieving fewer operations per second compared to the state-of-the-art general-purpose graphics cards, ASICs designed for edge computing are far more suited for automotive perception due to the high energy consumption of GPUs.

3.2.4 FIELD PROGRAMMABLE GATE ARRAYS

ASICs do not offer enough flexibility to keep up with the rapid evolution of deep learning models as the emergence of new types of layers poses a challenge to specialized hardware, especially since ASICs tend to have a high non-recurring engineering cost and time for design [89]. Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be specifically optimized for a large subset of applications. Contrarily to ASICs, FPGAs are "field" reconfigurable meaning the hardware circuit can be reprogrammed to meet the requirements of the developer even when they change after manufacturing. This allows the

developers to adjust to new model architectures without having to re-design and manufacture a new chip, resulting in a significant reduction in design costs and time to market when compared to ASICS. For these reasons, Intel refers to ASIC prototyping using FPGAs as a standard practice that both decreases development time and accelerates verification by allowing testing of a design on silicon from day one [90]. However, the reconfigurable characteristic of FPGAs introduces significant overhead in raw power performance when compared to ASICs and GPUs. A. Boutros, S. Yazdanshenas, and V. Betz compared three at the time state-of-the-art computer architectures optimized for CNN inference observing an average of 8.7x more area required for FPGA implementations when compared to ASICs. Regarding performance, assuming only raw Tera Operations per Second (TOPs) without considering external memory bandwidth, ASIC implementations were 2.8x to 6.3x faster than FPGAs [89].

Comparing energy efficiency, FPGAs typically provide a lower energy consumption compared to GPUs but still higher than ASICs. Aydonat, U. et al. showed that an implementation of the *AlexNet* network on an *Intel's Arria 10 FPGA* achieved similar results to an *Nvidia TitanX GPU* when considering images per second per watt. The FPGA, although processing approximately 5x fewer images per second, it did so by consuming 5x less energy [91]. Nurvitadhi, E. et al. compared the energy efficiency of a CPU, GPU, FPGA, and ASIC in an implementation of a binary neural network. The results show that the FPGA and ASIC significantly outperform the CPU and GPU in terms of performance per watt [92].

Another advantage of FPGAs is the adaptability to any type of bit-width representation when performing quantization. This allows for testing several possible representations and evaluating the performance, energy efficiency, and inference latency of deep learning models.

Overall, in the ever-changing area of deep learning applications and the tight accuracy, latency, and energy requirements of an ADAS, FPGAs seem to be the most flexible, cost-effective hardware that still offers a very reasonable performance-per-watt.

3.3 DEEP NEURAL NETWORK QUANTIZATION

Introduced in 2.4.1, quantization is becoming a standard procedure when developing deep learning applications that run on embedded hardware. This is evidenced by the quantization toolsets available in widely used and established deep learning frameworks. *Pytorch* offers this functionality through its quantization API [93] and *TensorFlow* offers the *TensorFlow Lite*, a library for deploying models on mobile, microcontrollers, and other edge devices [94]. *Facebook* has also open-sourced its library *QNNPACK* which provides support for quantized neural networks to run on mobile devices [95]. One last

example is *Xilinx*, a company that manufactures and sells FPGAs, which also provides *Vitis-AI*, a tool that supports quantization on their proprietary hardware [24]

3.3.1 QUANTIZATION METHODS

As briefly mentioned in 2.4.1, the main problem when quantizing neural networks is the inherent accuracy drop. Although not as acute as in other mathematical models, given that most current neural network models are overparameterized [96], it can still be very significant. To mitigate this effect, several quantization algorithms have been proposed and fall into one of two main methods, namely quantization-aware training (QAT), and post-training quantization.

In QAT, the quantized model is re-trained to fine-tune the parameters given their quantized values. To do this, the model weights are quantized to the integer values before the forward pass, then the data points are forwarded through the network and the loss with respect to the quantized weights is calculated. A straight-through estimator is used as an approximation of the backward function of the quantization operation and the gradients are added to the floating-point weights [53]. This allows the neural network to adapt to the quantization operation during training and results in less accuracy loss compared to post-training quantization. Figure 13 provides a surface-level illustration of the QAT training procedure.

Ideally, QAT does not need to re-train the model from scratch as it is beneficial to use the pre-trained weights. This allows the QAT to converge faster to a solution and it is common practice in literature [97], [98]. Combining both pruning and QAT, S. Han, H. Mao, and W. J. Dally achieved a 49x model-size reduction on the *VGG-16* model with no loss of accuracy on the *ImageNet* dataset, a 3x to 4x inference latency reduction, and a 3x to 7x increase in energy efficiency on the fully connected layers, also using Huffman coding to encode weights [99]. Nagel, M., et al. used a QAT consisting of cross-layer equalization, range estimation, and learnable quantization parameters to experiment with 8-bit and 4-bit quantization of both weights and activations. Using 8-bit weights and activations, and per-tensor quantization, the authors' solution surpassed the float model baseline in different perception tasks on datasets like *Pascal VOC*, *COCO 2017*, *GLUE,* and *ImageNet*. They also showed the robustness of their pipeline when quantizing weights using 4-bits by staying within 1% of the float model baseline in 5 out of 8 models tested [100].

However, because QAT requires training the model, it can be impractical given that the full dataset and sufficient hardware resources might not be available.

Post-training quantization methods allow quantizing neural networks without the overhead of training the network. This approach has been found to work well for larger models, which have more redundancy

but can struggle on smaller models [101]. It consists in analyzing the model's weights and activations generated by running inference during the calibration process and selecting the correct quantization intervals [54]. Algorithms such as cross-layer equalization proposed by Nagel, M. et al. allow for data-free quantization, meaning that no additional data is needed to quantize the model. The authors demonstrate a top-1 accuracy degradation of only 0.5%, 0.3%, and no degradation respectively on the difficult to quantize *MobileNetV2*, *MobileNetV1,* and the slightly easier *Resnet-18* model using the *ImageNet* dataset when quantizing the model to an 8-bit integer representation [102]. On the other hand, *AdaQuant*, proposed by Hubara, I. et al., uses a small set of unlabeled calibration data. Compared with other post-training quantization algorithms, the authors show that the method is much less susceptible to over-fitting and can be used on a very small calibration set. The *ImageNet* top-1 accuracy of *Resnet* (18, 34, 50, 101), *ResNext*, *Inception-V3,* and *MobileNet-V2* after quantization to 8-bits is higher than the abovementioned two algorithms and depending on the model is usually within a 2% accuracy to the float model baselines [103]. Moreover, the accuracy of the *BERT*-base model on the *SQuAD1.1* dataset achieves an accuracy degradation of only 0.46%. The usage of a calibration dataset is crucial to enable bit-widths lower than 8, such as INT4 in post-training quantization. However, when targeting bit-widths below 8 bits, post-training quantization might not be enough to mitigate the large quantization error [100]. Finally, extreme quantization has been proposed where neural network parameters are represented using only one or two bits, respectively referred to as Binarized Neural Networks (BNNs) [104], [105], [106] and Ternary Neural Networks (TNNs). [107], [108] Particularly, BNNs have a unique advantage since the multiply-accumulate operation, used in dot products, can be done without multiplications or additions when using one-bit representation through *xnor* and bit counting operations, speeding up computations and consuming less power [109].

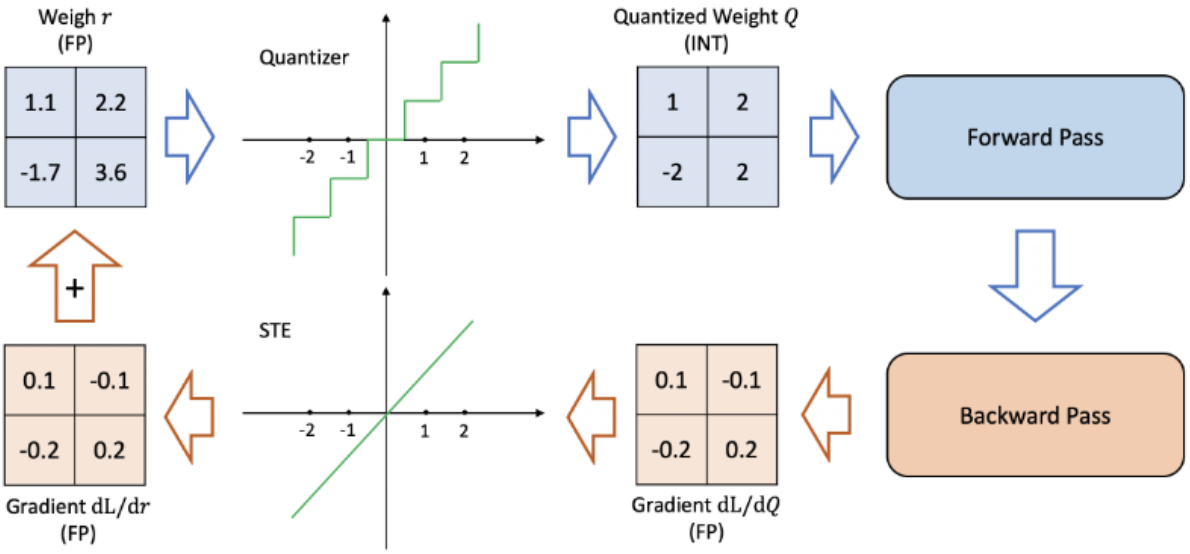

Figure 13. Quantization-aware training with a straight-through estimator. Retrieved from [110].

3.3.2 BENEFITS OF QUANTIZATION

The previous section showed how state-of-the-art quantization methods can maintain accuracy while reducing model size. This section specifically explores previous works that illustrate the benefits of quantization considering energy consumption and inference latency.

Horowitz, M. compared the energy consumption of 8-bit and 16-bit additions and multiplications with the floating point 32-bit baselines on Intel's 95nm processor chips. The results showed a 30x and 18x reduction in energy cost between the 32-bit and the 8-bit and 16-bit additions respectively and an 18.5x reduction between 32-bit and 8-bit multiplications [111]. Hashemi, S. et al. measured the power consumption savings of 16, 8, 4, and 1-bit precisions in a custom hardware accelerator. The authors explored three convolutional architectures containing mostly convolution, pooling, and fully connected layers. Compared to the 32-bit precision floating-point baseline models, the 16-bit, 8-bit, 4-bit and 1-bit models resulted in savings of 60%, 85%, 91%, and 94% respectively [112]. Furthermore, considering the area needed to execute each operation, Gholami, A. et al. showed that 8-bit and 16-bit additions result in a reduction of 116.2x and 62.4x respectively while the 8-bit multiplication result in a 27.3x reduction compared with the 32-bit float baselines [110]. Blott, M. et al. compared the power consumption of several convolutional neural networks on the four different FPGAs using different bit-widths, ranging from 16 to a single bit. The results show that these implementations have very low power consumption, especially at the small bit-widths [23].

Regarding inference latency reduction, S. Kim, G. Park, and Y. Yi measured convolutional neural networks inference speedups of FP16, and INT8 in mobile GPUs. The authors showed a very significant speedup on two of the three hardware targets achieving between 1.5x to 3x inference speedup compared to the 32-bit floating-point baseline [113]. Similarly, Z. Jin and H. Finkel registered speedups ranging from 1.02 to 1.56 of 8-bit precision compared to 32-bit floating-point precision on an *Intel Xeon 4-core CPU* and 1.1 to 2.0 speedups of 16-bit floating-point precision compared to 32-bit floating-point precision on the *Intel Iris Pro mobile GPU*. A merit of this work is that a large variety of neural networks were experimented [114]. Finally, in the work of Nurvitadhi, E. et al., a ternary (2-bit quantization) *ResNet-50* model was shown to have up to 65% better performance per watt (operations per second per watt) compared to a *Titan X GPU* using the *ImageNet* dataset [109].

3.4 DEEP LEARNING ON FPGAS

3.2 highlighted the suitability of FPGAs for deploying deep neural networks. Right after, 3.3 evidenced, through a review of existing works, that low-bit representations can retain accuracy and decrease inference latency and energy consumption.

The following two subsections aim to explore works that involved efforts to implement and deploy deep neural networks on FPGAs.

3.4.1 DEEP NEURAL NETWORK IMPLEMENTATIONS ON FPGAS

Hand-coded FPGA-based accelerator designs require much experience and expertise. It can take a professional hardware developer several weeks just to map a deep neural network to an FPGA, even when using high-level synthesis tools that allow him/her to express the design in an algorithmic level of abstraction using languages such as *C/C++* [115]. For this reason, several older works focus on accelerating only certain layers of neural networks [116]. However, there is still a valid reason for using lower levels of abstraction since it gives the developer more design freedom and optimization opportunities. In the work of Ma, M. et. al, the authors used *Verilog*, a hardware description language, to implement four different convolutional neural networks on FPGAs. Their implementations of *NiN*, *VGG-16*, *ResNet-50,* and *ResNet-152* achieved real-time inference latencies of 7.9 ms, 88.8 ms, 31.82 ms, and 81.8 ms per image on the *ImageNet* dataset with a batch size of one on *Intel Stratix V GXA7* FPGA and 3.8ms, 43.2 ms, 12.7 ms and 32.0 ms on *Intel Arria 10 GX 1150* FPGA. No accuracy metrics were reported [18].

Although there are use cases where the abovementioned approaches are the appropriate solution, they not only increase the costs necessary to implement deep neural networks on FPGAs but also keep deep learning engineers, with no FPGA expertise, from a hardware-software co-design approach to deep learning implementations. For this reason, most recent implementations use tools that interface with established deep learning frameworks. Faraone, J. et al. used the *FINN* library [117] to implement pruned and quantized versions of *AlexNet* (1-bit weights, 2-bit activations) and *TinyYolo* (1-bit weights and 3-bit activations) on a *Xilinx KU115* FPGA. The authors reported a top-1 accuracy of 50.1% and a frame rate of 3797 FPS on ImageNet and a 48.5% top-1 accuracy on *PascalVOC* with a frame rate of 1226 FPS considering the highest pruning percentage tested. The authors however did not specify the batch size nor a baseline float model [19]. Ngadiuba, J. et al. used their library, hls4ml [118], to implement a simple multi-layered perceptron with ReLU activations. They experimented with 1-bit and 2-bit quantizations on MNIST [119] and Jet tagging [120] datasets. The target hardware is a *Xilinx Virtex Ultrascale 9+* FPGA. The results show a 100 ns inference latency, however with only a 3% accuracy drop [121]. Following the previous work, Aarrestad, T. et al. successfully implemented a convolutional neural network. Besides hls4ml, they use *QKeras* [122] to quantize the model and *TensorFlow* pruning API for pruning 50% of the model parameters. The authors compared post-training quantization and QAT using fixed-point representation with 16 bits. With the QAT approach, the model retained the baseline float accuracy down to a bit-width of 4. The latency reported is in the microsecond range [123].

All the previously mentioned works target 2D computer vision tasks. There is a gap in the literature when it comes to FPGA implementations of 3D computer vision models since it combines two fairly new areas of research. Y. Lyu, L. Bai, and X. Huang designed a lightweight convolutional neural network, *ChipNet*, to process LiDAR data and perform drivable region segmentation. The network, quantized with a width of 18 bits, achieved an average precision of 88.29%. To showcase the real-time capabilities of their work, the authors directly sent LiDAR sensor data through UDP to a *Xilinx UltraScale XCKU115* FPGA, used to run the network. From end-to-end, including pre-processing and post-processing of LiDAR frames, the authors reported a latency of 17.59 ms [20]. However, note that the point cloud is sampled in the [-45°, 45°) interval in the azimuth direction. J. G. López, A. Agudo, and F. Moreno-Noguer implemented the convolutional layers of a *VoxelNet* - based model on an *Arria 10 Intel* FPGA using the *leg-up 4.0* framework [124] and the *ModelSim HLS* suite [125] [126]. The accuracy reported is close to the *VoxelNet* baseline float implementation on the *KITTI* dataset. The model parameters are quantized down to a 12-bit representation. The reported inference latency of the convolutional layers is 17.59 ms. However, the batch size is not specified. Finally, L. Bai, Y. Lyu, X. Xu, and X. Huang achieved an end-to-

end implementation of *PointNet* with LiDAR point cloud data in a *Xilinx Zynq UltraScale+ MPSoC ZCU104* development board. The implementation directly receives LiDAR frames via ethernet and achieves inference latencies of 19.8 ms and 34.6 ms in classification and segmentation respectively. However, each LiDAR frame has only 4096 points, much lower than, for example, a *KITTI* dataset frame. The smaller point cloud heavily influences inference latency but could lead to significantly worse accuracy results, which were unfortunately not reported by the authors [21].

3.4.2 HIGH-LEVEL TOOLS FOR DEEP NEURAL NETWORK DEPLOYMENT ON FPGAS

Although the previous examples showed several successful implementations of neural networks on FPGAs with high-level tools, the tools themselves are still at an infancy stage and consequently tend to be rather limited in their scope. For this reason, it is useful to gather a list of the currently available options. Solutions that target cloud-hosted FPGAs, such as *Azure Machine Learning* [127], were consequently not considered. Furthermore, all tools contemplated are currently supported and actively updated. This made the list smaller but hopefully more informative.

3.4.2.1 FINN-R

Developed by *Xilinx Research Labs*, *FINN-R* is an open-source tool intended for design space exploration and the automatic creation of fully customized quantized neural network inference engines on FPGAs. In the authors' words, this tool tries to answer the given question: "Given a set of design constraints and a specific neural network, what is the best possible hardware implementation that can be achieved?" [23].

FINN-R features two converse inference accelerator architectures, represented in Figure 14. The first, referred to as customized *Dataflow Architecture*, is customizable for specific neural network topologies and different bit-widths in weights and activations in each layer, which aims to maximize the use of hardware resources. In this architecture, the computation of layers, the storage of layer weights, and activation maps are all performed in on-chip memory. This has the potential to significantly reduce latency as the amount of off-chip memory accesses is minimized. However, as noted by the authors, this accelerator architecture is not scalable toward really deep CNNs. For these use cases, *FINN-R* also offers a *Multilayer Offload Architecture* in which the layer's weights, and resulting feature maps, are stored in the more abundant off-chip main memory. Both architectures are depicted in Figure 14.

FINN-R uses a frontend module to interface with deep learning frameworks such as *Caffe*, *DarkNet,* and *TensorFlow*. The *Brevitas* tool [128] also from *Xilinx Research Labs*, can be used to perform QAT on *Pytorch* models, extending the reach of the *FINN-R* framework. The frontend module is responsible for translating deep neural networks, quantized in these frameworks, into a common device-agnostic intermediate representation. This representation is quantization-aware, meaning that it has access to the quantization information of each layer which enables mapping to backend primitives optimized for quantized computation.

A series of small subprograms operate on the intermediate representation to perform a series of optimizations such as the "direct quantization", which converts non-quantized layers to fixed-point values. It is also in this phase that the previously mentioned accelerator architectures are generated. Finally, a corresponding high-level-synthesis code is generated.

From the intermediate representation, a backend module creates executable inference accelerators for a selection of platforms, including *PYNQ-Z1* [129], *Ultra96* [130], and *AWS F1* [131].

On the tool's *GitHub* page [132], a list of example neural network accelerators is presented. There, it is possible to verify that topologies such as simple *VGG*-like architectures and small fully connected networks are supported on all targeted FPGA boards with bit-widths of down to 1 or 2 bits. Furthermore, more sophisticated architectures like *MobileNet-v1* and *Resnet-50* are also supported, but not on all target boards. This limitation might indicate problems when deploying models with more complex architectures.

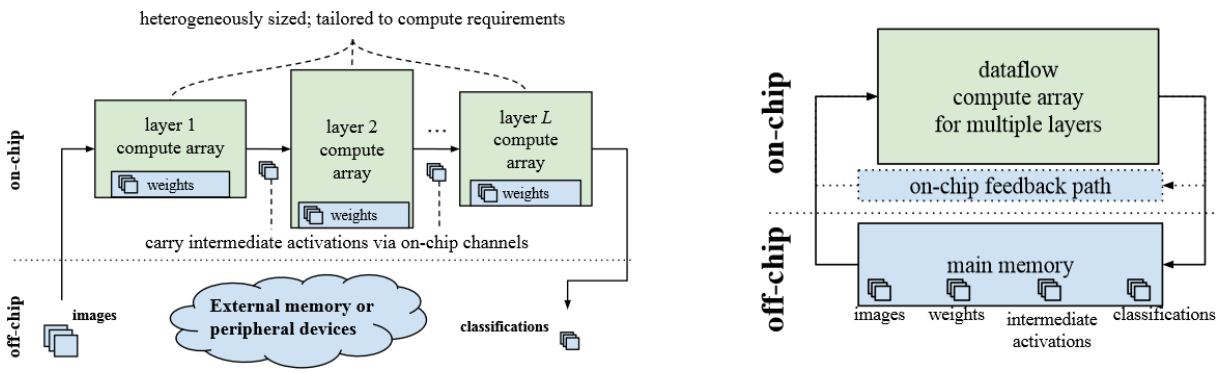

Figure 14. Accelerator architectures: Dataflow Architecture (Left) and Multilayer Offload Architecture (Right). Retrieved from [23].

3.4.2.2 HLS4ML

HLS4ML is an open-source *Python* package designed to interpret and translate machine learning algorithms for implementation with both FPGA and ASIC technologies [118]. Similarly to *FINN-R*, the package first converts the user-specified model into a common internal representation of the network

graph. The developed converters support *QKeras*, *TensorFlow*, *Pytorch,* and *ONNX* model formats. Post-training quantization and QAT are also supported, where the quantization settings of models trained in *QKeras* are propagated to the internal representation. Then, a set of generic optimizations are applied to simplify inference such as the vastly explored optimization of fusing batch normalization layers with preceding dense and convolutional layers [133]. The package also allows pruning the neural networks through what the authors call quantization-aware-pruning, which combines a pruning procedure with training that accounts for quantized weights.

Finally, at a later stage, the internal representation is converted to high-level synthesis code. One key aspect is the explicit support for multiple FPGA vendor high-level-synthesis backends (e.g., Xilinx, Intel, and Mentor).

To evidence the applicability of this tool, the works from Ngadiuba, J. et al. and Aarrestad, T. et al., described in 3.4.1, both use *HLS4ML* to implement neural networks on FPGAs. However, like *FINN-R*, the working examples provided by *HLS4ML* typically target either small networks or larger networks but with simpler topologies. This is further evidence of the premature state of development of such tools.

3.4.2.3 OPENVINO

OpenVINO [134] is an open-source toolkit for optimizing and deploying AI inference. It interfaces with *TensorFlow*, *Paddle Paddle* [135], *Pytorch*, *Caffe*, *ONNX*, and *MXNet* [136] and targets Intel hardware. Unlike all other tools reviewed in this work, *OpenVINO* does not specialize in FPGA deployment, targeting CPUs, GPUs, and VPUs. Instead, it provides an FPGA plugin that allows targeting *Intel Arria* FPGAs. A wide range of examples showcasing supported models, even complex ones, is advertised in the documentation [137]. However, the documentation fails to mention whether these models are possible to deploy on the target FPGAs.

3.4.2.4 VITIS-AI

Vitis-AI is the platform of choice for accelerating AI inference on *Xilinx*'s hardware platforms, targeting both edge and cloud FPGAs. It is designed with high-efficiency and ease-of-use in mind, making it easy for deep learning engineers with no FPGA knowledge to deploy deep learning applications on FPGAs. Like the previous tools, it interfaces with *TensorFlow*, *Pytorch*, *Caffe,* and *ONNX*.

Regarding quantization*, Vitis-AI* provides three different methods to quantize deep learning models, encapsulated into the *Vitis-AI Quantizer*. For post-training quantization, a data-free approach referred to

as quantized calibration, is the simplest and fastest way of quantizing a model. However, for situations where the decrease in accuracy is significant and QAT is not possible, a data-dependent approach is also provided. Similarly to quantized calibration, this approach is referred to as fast finetuning and relies on a small unlabeled calibration dataset. Fast finetuning can achieve better performance than quantized calibration, but it is slightly slower. Both algorithms behind the two post-training quantization methods and respective papers are cross-layer equalization [102] and *AdaQuant* [103]. A brief description of both these works is present in 3.3.1. Vitis-AI also provides QAT for situations where fast finetuning is not sufficient.

As for pruning, *Xilinx* advertises up to 90% pruning of model parameters with a tolerable accuracy loss, through the *Vitis-AI Optimizer*. Unfortunately, pruning-related tools require a commercial license to run.

After quantizing and pruning the model, it is possible to compile the model for the target board of choice using the *Vitis-AI Compiler*. The compiler is responsible for mapping the deep learning model to a highly efficient instruction set and dataflow model called the deep learning processing unit (DPU). It also performs sophisticated optimizations such as layer fusion, and instruction scheduling, and reuses on-chip memory as much as possible. To target the supported FPGAs, a set of DPUs are available. Each DPU is a group of parameterizable IP cores (Intellectual Property cores) which are integrated circuit layout designs that are the building blocks of more complex FPGA logic, analog to libraries in programing. The DPU contains a specialized instruction set that facilitates the mapping of neural networks to the underlying hardware. As an example, the *DPUCZDX8G* high-level architecture is described in Figure 15. The *DPUCZDX8G* first fetches instructions, generated by the Vitis-AI Compiler, from the off-chip memory to control the operation of the computing engine. The computing engine, which is implemented on the programable logic of the target hardware, is where neural network layers are computed. It is composed of processing elements (PEs) that combine the most basic operations such as adders, multipliers, and accumulators. These more basic primitives are combined to create more complex operations such as 2D convolution operations, pooling operations, or concatenations. They are also parameterizable, meaning that each DPU supports variants of the same convolution operations e.g., the kernel size can vary in the interval of 1 to 16.

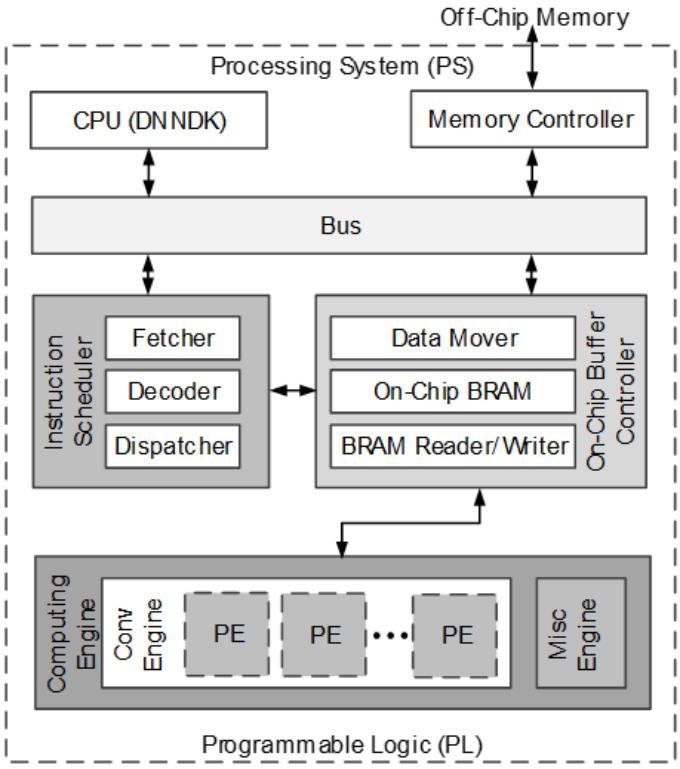

Figure 15. DPUCZDX8G Hardware Architecture. Retrieved from [138].

The *DPUCZDX8G* is optimized for *Zynq UltraScale+ MPSoC* boards and *DPUCAHX8H* is optimized for high throughput applications that use convolutional neural networks at their core. The list of DPUs and respective characteristics such as target hardware is presented in Table 2. Unfortunately, the available pre-built DPUs only support INT8 quantization. Also, without experimentation, it is difficult to understand how adaptable are the pre-built DPUs to neural networks featuring more exotic layers. By analyzing the previous description of the *DPUCZDX8G*, it is expected that implementations of neural networks targeting pre-built DPUs will be limited by the implemented functionality and that to support more operations, new functionality will have to be implemented by the developer. This is a major limitation for deep learning engineers without specific hardware experience. Unfortunately, the other tools suffer from the same limitations. Nevertheless, as the tools mature, the number of supported neural network layers is expected to increase. On this note, *Vitis-AI* does provide support for implementing custom DPUs. However, this approach is outside the scope of this work as it requires deep hardware design knowledge.

Table 2. Vitis-AI pre-built DPUs [113].

DPU Name	Target HW	Application	Quantization Bit-width	Domain
DPUCZDX8G	Zynq UltraScale+ MPSoC	CNN	8	General purpose
DPUCAHX8H	Alveo U50/U280 boards	CNN	8	High throughput
DPUCAHX8L	Alveo U50/U280 boards	CNN	8	Low latency
DPUCADF8H	Alveo U200/U250 boards	CNN	8	High throughput
DPUCVDX8G	Versal ACAP VCK190 board	CNN	8	General purpose
DPUCVDX8H	Versal ACAP VCK5000 board	CNN	8	High throughput

Vitis-AI also provides, through the high-level Vitis-AI Runtime Libraries (VART), a set of APIs that make the data loading, pre-processing, and post-processing that happen on the board CPU, and the model execution, as seamless as possible by abstracting away data transferring between the board CPU and FPGA, as well as data parallelism supported by the DPU. The *C++* and *Python APIs* are available and well documented in the user guide. The developed VART applications can be profiled through the *Vitis-AI Profiler*, reporting individual layer execution time, data transfers as well as data loading and processing.

Contrary to the previously presented tools, *Vitis-AI* provides extensive and detailed documentation. The reader is recommended the *Vitis-AI* user guide [139], a document featuring detailed usage of each of the functionalities through tutorials and example code snippets. However, where *Vitis-AI* mostly distinguishes itself from the rest is the amount of pre-deployed models and their diversity, associated benchmarks, and example applications that can be used to validate the tool. These models can be found in the *Vitis-AI Model Zoo*. The extensive benchmarks of *Model Zoo* models for all the target boards can be found on the *Vitis-AI GitHub* page [24]. The benchmarks provide, for each model, the end-to-end inference latency and throughput. More interestingly for this work, some deployed models are LiDAR-based 3D computer vision models, particularly the *PointPillars* and *SalsaNext* [140] models.

4 VITIS-AI FRAMEWORK EXPLORATION

4.1 EXPERIMENT DESCRIPTION

3.4.2.4 highlighted the suitability of *Vitis-AI* for deploying deep learning models on *Xilinx* boards by referencing the availability of a wide range of deep learning model benchmarks on multiple *Xilinx* FPGAs. Furthermore, the diversity of the benchmarked models is particularly interesting for this work since a small selection of 3D computer vision models is available in the *Vitis-AI Model Zoo*. For these reasons, *Vitis-AI* was elected as the most promising tool for deploying a LiDAR-based model.

However, a typical *Vitis-AI* workflow, from model description to deployment, involves a long list of different components such as the quantizer, compiler, VART APIs, and profiler. It would be unwise to implement a complex 3D computer vision model without first understanding the possibilities and limitations of the tool at hand and validating if the decision to choose *Vitis-AI* is correct given the objectives of this work.

This first experiment aimed to study the Vitis-AI tool by deploying convolutional neural networks on target FPGAs. During the process, it was expected to maximize the exploration of the tool.

4.1.1 OBJECTIVES

The first objective was to understand how *Vitis-AI* interfaces with deep learning frameworks. To do so, the first step consisted in obtaining a quantized model from the model parameters and structure represented in the format of a deep learning framework of choice. Furthermore, during the quantization process, all available quantization methods were experimented with and validated in terms of accuracy metrics and model size. This required the development of quantization code as described in the tool's documentation. Then, the resulting quantized model was compiled for the target DPU, and an application was developed using the VART libraries. Finally, the models needed to be validated considering accuracy, inference latency, model size, and power consumption, making use of the available tools including the *Vitis-AI* profiler. This formed a complete workflow that encompassed all the tools and allowed a precise evaluation of *Vitis-AI*.

4.1.2 DATASET

The dataset used for this experiment was the *CIFAR-10* dataset [141]. It consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The perception task being solved is image classification.

While it is true that *CIFAR-10* is a relatively trivial dataset for today's CNNs, the dataset has a good characteristic for this exploratory analysis. Because it is relatively small, it is possible to perform a deeper and wider exploration of the tool, avoiding long but necessary processes of model training and quantization. Had QAT not been considered in this experiment, model re-training would have not been necessary, and *ImageNet* would have been a more appropriate choice of dataset.

Figure 16. Example of CIFAR-10 images.

4.1.3 DEEP LEARNING FRAMEWORK

In this work, *Pytorch* was selected as the deep learning framework to interface with Vitis-AI. As detailed in the tool's documentation, the workflow that allows the deployment of a deep learning model on an FPGA, given its code description, changes substantially from framework to framework. Studying the intricacies of the workflow of every supported framework would be too time-consuming for the additional benefit.

The choice of *Pytorch* is mainly justified by the recent growth in the usage of the framework when compared to the other heavily used framework, *TensorFlow*. *AssemblyAI* provides very interesting data regarding the comparison of the two frameworks in terms of usage in research papers and *Github* repositories [142]. As can be seen in Figure 17, the graph shows that *Pytorch* has surpassed TensorFlow in terms of new papers. The data is collected from eight top research journals over the past four years. In turn, Figure 18 shows a steady increase in the percentage of *Pytorch Github* projects and a consequent decrease in *TensorFlow* projects. The data is from the well-known website *Papers With Code* [143]. These two trends match what was observed during this work's literature review of 3D computer vision models.

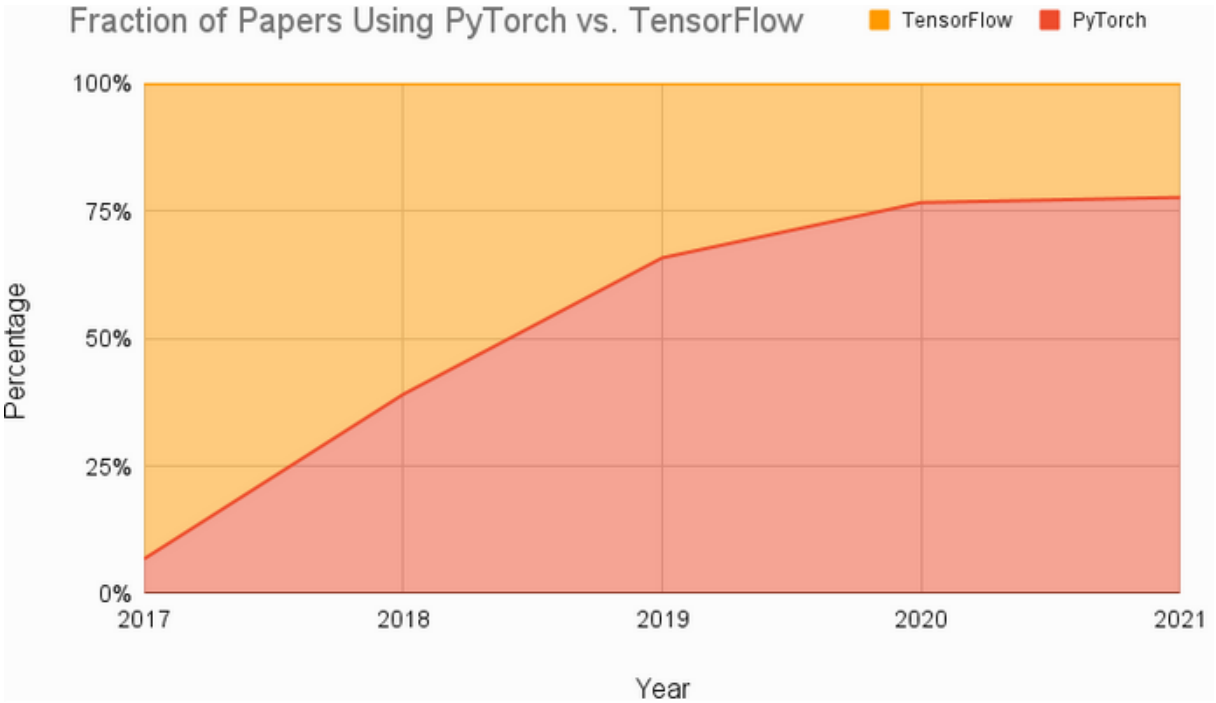

Figure 17. Pytorch and Tensorflow usage in publications. Retrieved from [142].

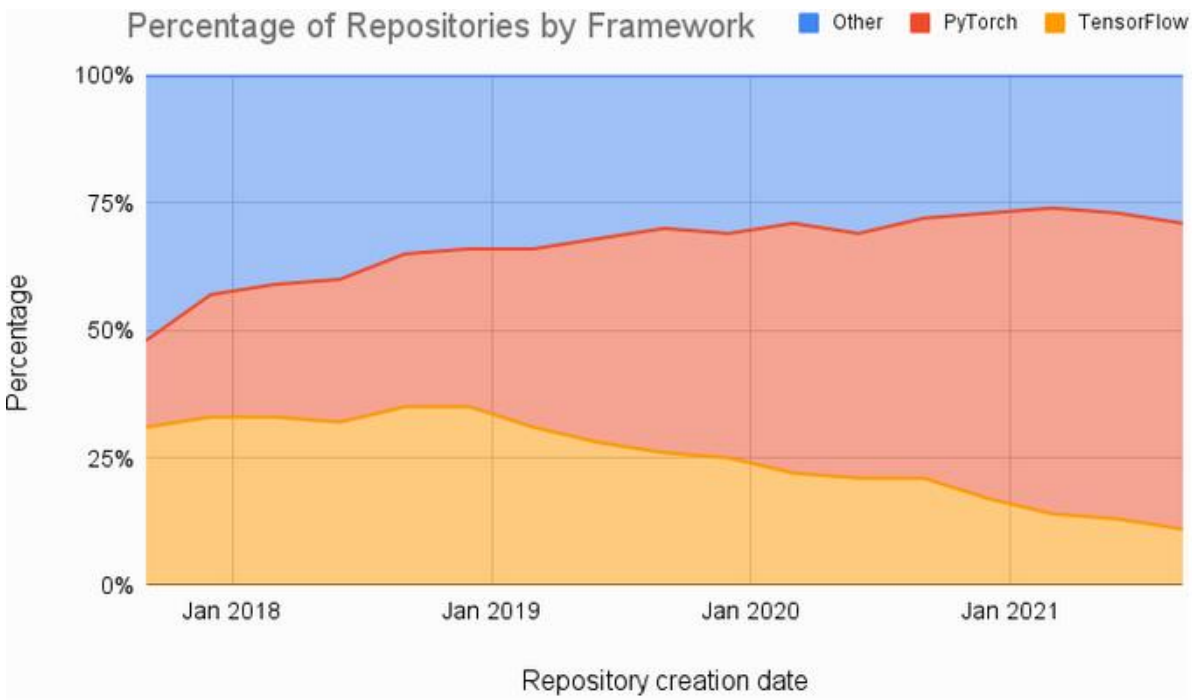

Figure 18. Pytorch and Tensorflow github repository share. Retrieved from [142].

4.1.4 TARGETED DEEP NEURAL NETWORKS

The *ResNet-18* [27] and *SqueezeNet* [85] networks from the *Torchvision* model zoo were selected to be quantized and deployed. Both networks feature a convolutional architecture. *ResNet-18*, as the name suggests, is 18 layers deep and features residual connections. The convolution layers are organized in blocks, each containing two convolutional layers. An average pooling layer is used at the end of the convolution blocks and a fully connected layer produces a 10-element tensor. These elements are then converted into class probabilities through a softmax activation layer. The *SqueezeNet* architecture begins with an isolated *1x1* convolution layer followed by 8 fire modules. Each fire module consists of 1x1 convolution layers followed by 1x1 and 3x3 convolution layers. These convolution layers are followed by ReLU activations. Max pooling layers are used after fire modules 4 and 8. Finally, a convolution layer is used at the end of the fire modules, followed by a global average pooling layer. Likewise, in ResNet-18, a softmax activation layer is used at the end of the network to produce the final probabilities. Both neural network architectures are detailed in Figure 19.

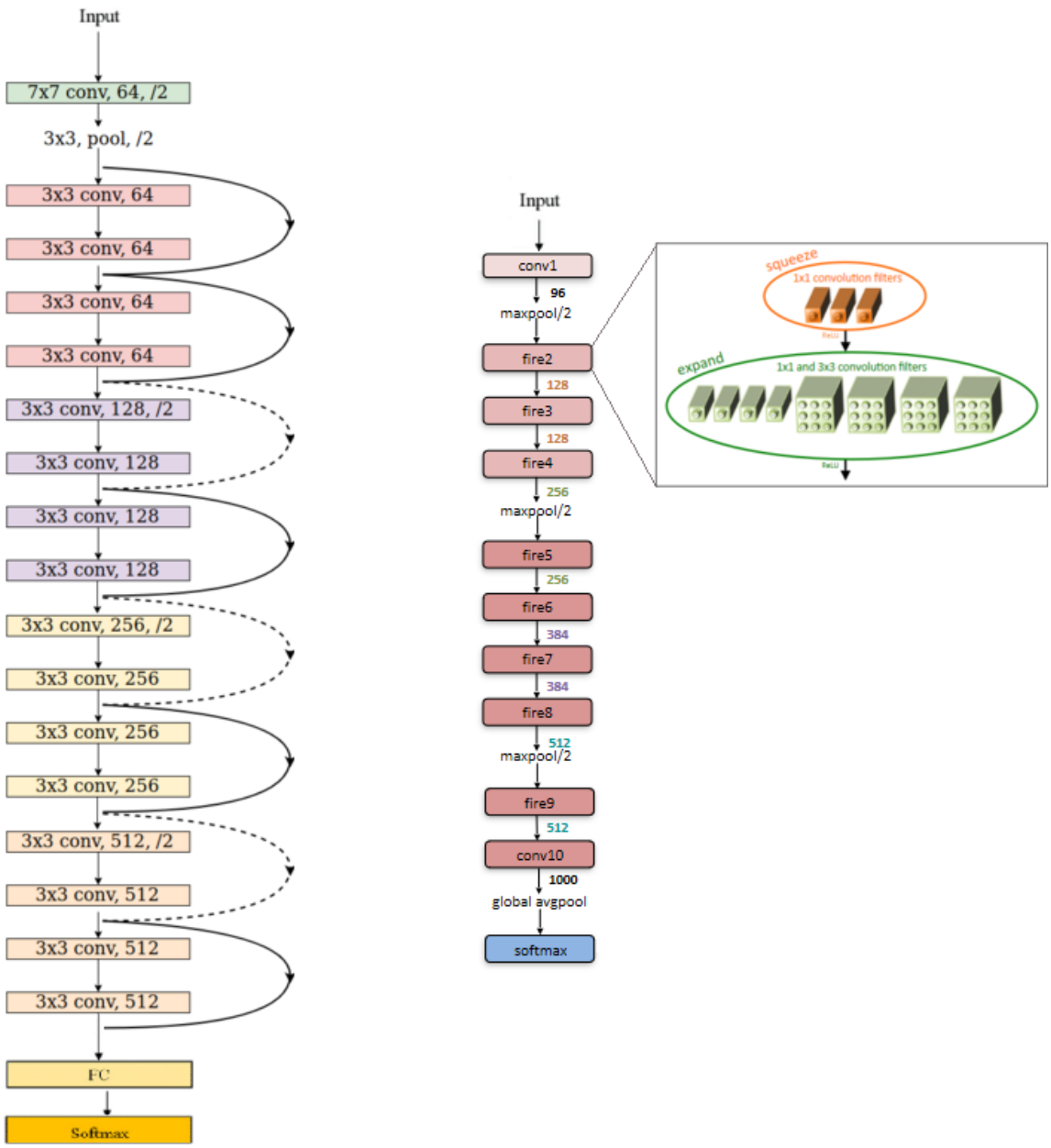

Figure 19. ResNet-18 (Left) and SqueezeNet (Right) architectures. Retrieved from [144] and [85].

Table 3 lists the number of total parameters, as well as the number of floating-point operations (FLOPS) of both networks. Immediately, one realizes that the *ResNet-18* has 15x more parameters and requires 37x more FLOPS to fully compute. Nevertheless, by today's standards, these are still fairly modest numbers.

Table 3. ResNet-18 and SqueezeNet total parameter count and floating-point operations considering Cifar-10.

Model	Input Size (N, C, H, W)	Total parameters	FLOPS
ResNet-18	1 x 3 x 32 x 32	11.18 M	$37.1 * 10^6$
SqueezeNet	1 x 3 x 32 x 32	0.74 M	$1 * 10^6$

4.1.5 TARGETED HARDWARE

Because this experiment prioritized exploration, two different *Xilinx* boards were targeted for model deployment, namely the *Zynq UltraScale+ MPSoC ZCU104* and *Versal ACAP VCK190*, both illustrated in Figure 20 and Figure 21 along with the respective dimensions. More specifically, the deployment focused on the *DPUCZDX8G* and the *DPUCVDX8G*. Table 2 lists all the pre-built *Vitis-AI* DPUs.

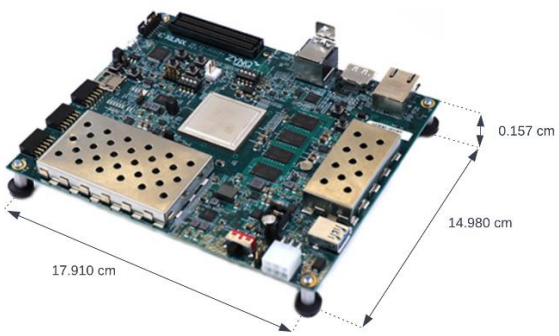

Figure 20. Zynq UltraScale+ MPSoC ZCU104.

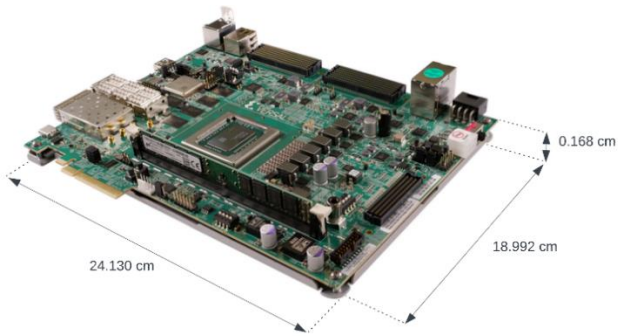

Figure 21. Versal ACAP VCK190.

The performance and efficiency of the DPUs depend on the nature and amount of resources in the underlying hardware. On the processing subsystem (PS) side, both boards feature *ARM Cortex CPUs*. Also, a 2GB and 8GB external DDR4 (Double Data Rate) memory is available for the *ZCU104* and *VCK190* boards respectively. The bandwidth of the *VCK190* DDR4 memory is 33% faster. On the programmable logic side (PS) or FPGA, the *VCK190* also features substantially more resources with 4x more LUTs and slightly more DSP slices. Regarding both on-chip memory types, the *VCK190* features approximately 3x more BRAM and 5x more URAM. AI engines, explained in 4.1.5.2, are only present in the *VCK190* and are used to compute convolution operations.

Table 4. Zynq UltraScale+ MPSoC ZCU104 and Versal ACAP VCK190 resource comparison.

Resources	ZCU104	VCK190
CPU	4×ARM Cortex-A53 @1.5GHz	2×ARM Cortex-A72 1.7@GHz

External Memory	2GB DDR4 (2400 Mb/s)	8GB DDR4 (3200 Mb/s)
CLB LUTs	230K	900K
BRAM	312 (11.0 MB)	967 (34 MB)
URAM	96 (27.0 MB)	463 (130.2 MB)
DSP Slices	1728	1968
AI Engines	-	400

4.1.5.1 DPUCZDX8G

The *DPUCZDX8G* provides user-configurable parameters that allow optimizing resource usage. The name given to a specific combination of parameters is a configuration. It is possible to control DSP slices, LUTs, BRAM, and URAM usage. Naturally, configurations are limited by the amount of available resources. It is also possible to enable functionality to the DPU by activating the ability to compute additional operations that are not active by default such as softmax, average pooling, and depth-wise convolutions.

The *DPUCZDX8G* can be configured with various architectures. Different architectures modify the convolution unit, used to compute convolutions, and enable different levels of parallelism. The level of parallelism can be modified along three dimensions: Pixel Parallelism (PP), Input Channel Parallelism (ICP), and Output Channel Parallelism (OCP). Figure 22 depicts a convolution operation and the three dimensions of parallelism that can be modified. In the image, PP has a value of 2. This means that two pixels of the input feature map are processed at once. ICP has the value 3. So, for every pixel, three values of that pixel along the input channel dimension are processed in parallel. The same applies to each kernel element. Lastly, OCP is 3. Consequently, kernel elements from three different kernels are used at once.

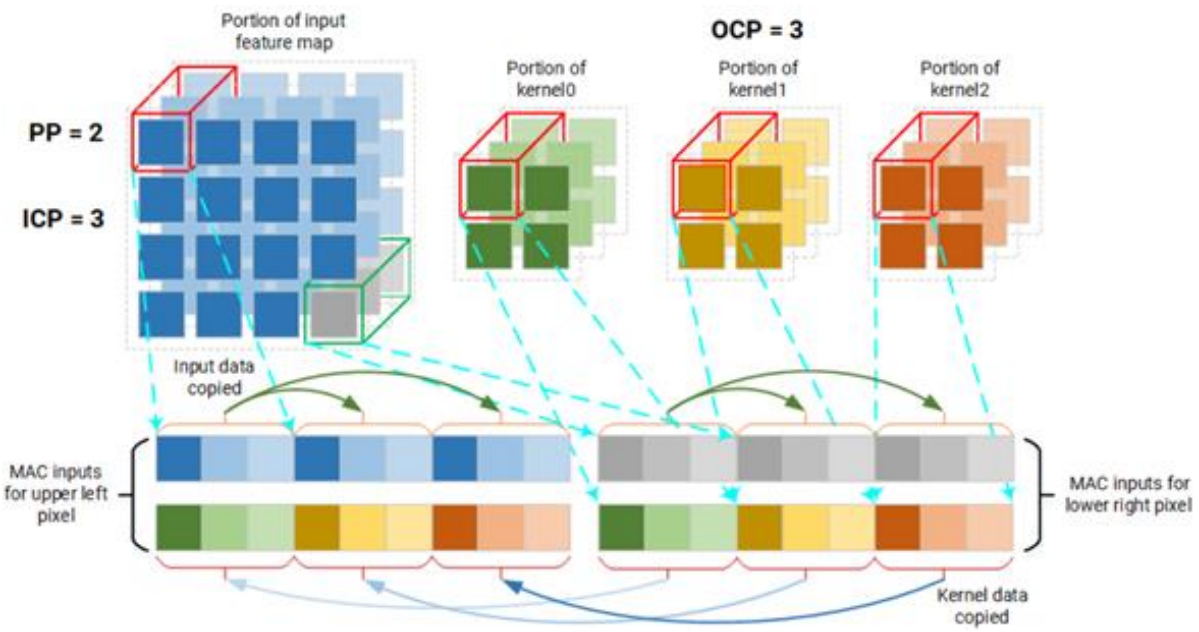

Figure 22. Three parallelism dimensions in convolution operation. Retrieved from [138].

The higher the levels of parallelism, the higher the number of peak operations that can be executed each cycle. The number of multiply accumulates (MACs) per cycle can be given as a function of the parallelism levels:

$$MACs/cycle = PP * ICP * OCP \qquad \text{(Equation 5)}$$

Table 5 lists all existing architectures, the respective parallelism levels, and the peak operations per cycle. Note that a MAC corresponds to two operations, multiply and accumulate, so the peak operations each cycle is equal to double the peak MACs each cycle. Naturally, the increase in the parallelism levels leads to an increase in programmable resource usage.

Table 5. Relationship between DPUCZDX8G architectures' parallelism levels and peak operations per cycle.

Architecture	PP	ICP	OCP	Peak Ops per cycle
B512	4	8	8	512
B800	4	10	10	800
B1024	8	8	8	1024
B1152	4	12	12	1150
B1600	8	10	10	1600
B2304	8	12	12	2304
B3136	8	14	14	3136

B4096	8	16	16	4096

DPUs try to utilize on-chip memory as much as possible during model inference to store intermediate feature maps, weights, and biases. The on-chip memory consists of the BRAM and URAM. It is possible to increase the RAM usage in the *DPUCZDX8G* architectures. Using more RAM can be beneficial because costlier external memory accesses can be reduced, and performance can be improved. It is also possible to choose which type of RAM to use. BRAM only or BRAM+URAM (Hybrid). Furthermore, it is possible to enable higher ram usage which extends the amount of on-chip memory resources available. This parameter is known as "High RAM Usage".

Lastly, it is possible to increase the number of DPU cores used. For the *DPUCZDX8G* specifically, a maximum of 4 cores can be used. More cores can be used to achieve higher performance at the cost of higher programmable resource usage. Table 6 details all the *DPUCZDX8G* configurations explored in this experiment and their respective resource usage.

Table 6. All DPUCZDX8G configurations explored and respective resources.

Architecture	Designation	# Cores	RAM Type	LUTs	BRAM	URAM	DSP
B512	b512x1_hybrid	1	Hybrid	19.7k	15.5	14	66
B1024	b1024x1_hybrid	1	Hybrid	24.1k	41.5	14	130
	b1024x1_bram	1	BRAM only	23.4k	101.5	0	130
	b1024x2_hybrid	2	Hybrid	50.4k	91.5	28	260
B4096	b4096x1_hybrid	1	Hybrid	36.4k	86.5	44	514
	b4096x1_hybrid_high_ram	1	Hybrid + High Ram Usage	48.0k	147	46	706
	b4096x2_hybrid	2	Hybrid	72.7k	177.5	88	1028

4.1.5.2 DPUCVDX8G

Similarly to the *ZCU104* DPU, the *DPUCVDX8G* also provides some configurability. Besides DSP slices, LUTs, BRAM, and URAM usage, it is also possible to control the number of AI Engines (AIEs). The AIEs in the *DPUCVDX8G* perform the convolution operation. The number of batch handlers is also parameterizable. Each batch handler is responsible for handling a batch element and performing the respective computations. A private group of AIEs is available for each batch handler. The number of AIEs per batch handler can be configured to be 32 or 64. The amount of batch handlers is also parameterizable.

Lastly, likewise *DPUCZDX8G*, it is possible to control the number of *DPUCVDX8G* cores, here called compute units. This parameter supports a range of values from 1 to 3. However, it is only possible to increase the number of compute units when the number of batch handlers is 1.

The name of the configuration explicitly contains all the above-mentioned parameter values. For example, *C32B1CU2* means that there are 32 AIEs for each batch handler, a single batch handler, and 2 compute units.

Table 7 lists all the *DPUCVDX8G* configurations targeted in this experiment, the corresponding resource utilization, and the peak theoretical performance per clock cycle measured in tera operations. The peak theoretical performance assumes a 333MHz PL frequency and 1.25 GHz AIE frequency. The values are obtained with the following equation:

$$256 * \text{CPB\_N} * \text{BATCH\_N} * \text{CU\_N} * \text{AIE Frequency} \qquad \text{(Equation 6)}$$

where CPB_N is the number of AIEs per batch handler, BATCH_N is the batch number, and CU_N is the number of compute units.

Table 7. DPUCVDX8G configurations, respective resource utilization and the peak theoretical performance per cycle.

Configuration	AIEs	LUTs	BRAM	URAM	DSP	Peak Theoretical Performance/cycle (TOPS)
C32B1CU1	32	82.9k	0	136	139	10.24
C64B1CU1	64	93.2k	0	136	139	20.48
C64B1CU2	128	18.6k	0	272	278	40.96

4.2 IMPLEMENTATION

The process of deploying the *ResNet-18* and *SqueezeNet* models from the *Pytorch* model description involved a large set of tools of *Vitis-AI*. Figure 23 describes the software and hardware components and respective connections used in this experiment.

Regarding hardware, the proposed setup comprises a Linux-based host server with an *Nvidia GPU RTX 3090* and two *Xilinx* boards containing FPGAs. The connections are realized by ethernet through an ethernet switch. This allows multiple connections to several FPGAs simultaneously.

On a software level, the communications with the FPGAs are done over graphical SSH sessions. For file transfer, the SCP protocol is used. Concerning the *Vitis-AI* tools, the *Linux* host server makes use of a

Docker container with *Vitis-AI* version 2.0. It includes *Conda* environments with a software stack adapted to interface with each of the supported deep learning frameworks. It also contains the *Vitis-AI Model Zoo* and the possibility of using custom models, the latter being the approach used in this experiment. Finally, it provides the *Vitis-AI Quantizer* and *Compiler*. As for the FPGAs, the software tools are made available by a *Petalinux* [145] based image. The tools available are directed towards FPGA deep learning application development. Following a bottom-up view, the DPU is the lowest level of abstraction that interacts with the underlying hardware. Refer to 3.4.2.4 for a detailed explanation of the *DPUCZDX8G* and a list of all existing DPUs and respective characteristics. Next, Vitis-AI Runtime (VART) is responsible for providing developers with a high-level runtime API. Internally, the API is based on the Xilinx Runtime (XRT). It also uses the Xilinx Intermediate Representation (XIR) format to represent the neural network models.

VART provides both *C++* and *Python* implementations and exposes two main endpoints. The first is the *Vitis-AI* Library which provides more complex modules that implement varying levels of functionality. On one end, these functionalities can be complex classes and methods that implement classification and segmentation algorithms as well as entire demo applications. On the other end, they can be methods that expose the functionality of submitting and collecting inference jobs to and from the DPU. The second is the Vitis-AI profiler which allows the collection of information of the VART-based applications from data-processing *C++/Python* code that runs on the board CPU to the layer computation that runs on the DPU. This information includes the minimum/average/maximum run times of each neural network layer, the achieved frames per second (FPS) and memory read/write traffic. Finally, the *Petalinux* image also includes some additional libraries such as *OpenCV* [146] and *NumPy* [147] that further facilitate the creation of deep learning applications, particularly in the data loading and processing phase.

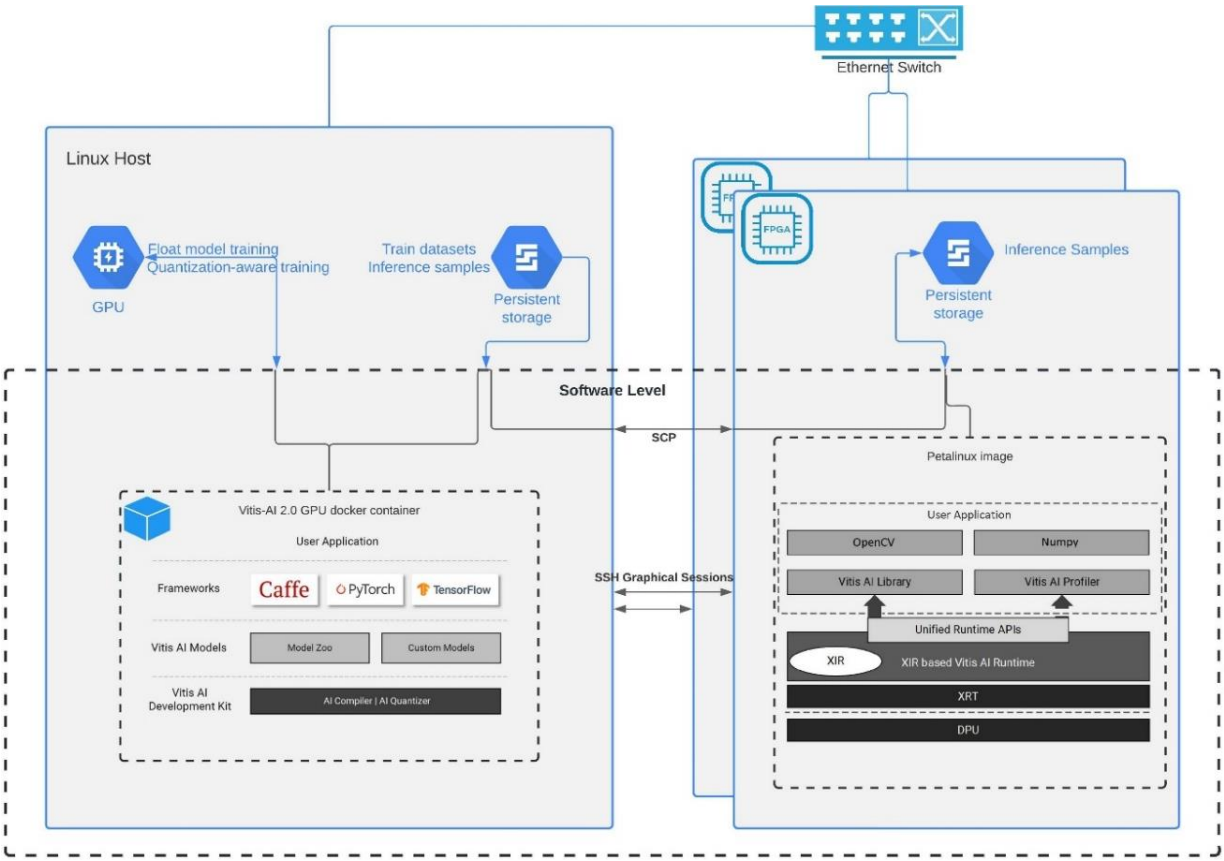

Figure 23. Experiment setup.

4.2.1 FLOAT MODEL TRAINING

The *ResNet-18* and *SqueezeNet* models were trained for 30 and 25 epochs respectively with a batch size of 128. The optimization algorithm used was the ADAM algorithm [148] with a cross-entropy loss function and a learning rate of 10^{-3}. The respective test and validation loss curves are plotted in Figure 24. ResNet-18 and SqueezeNet train plots.. As for data augmentation, random crops and random horizontal flips were used.

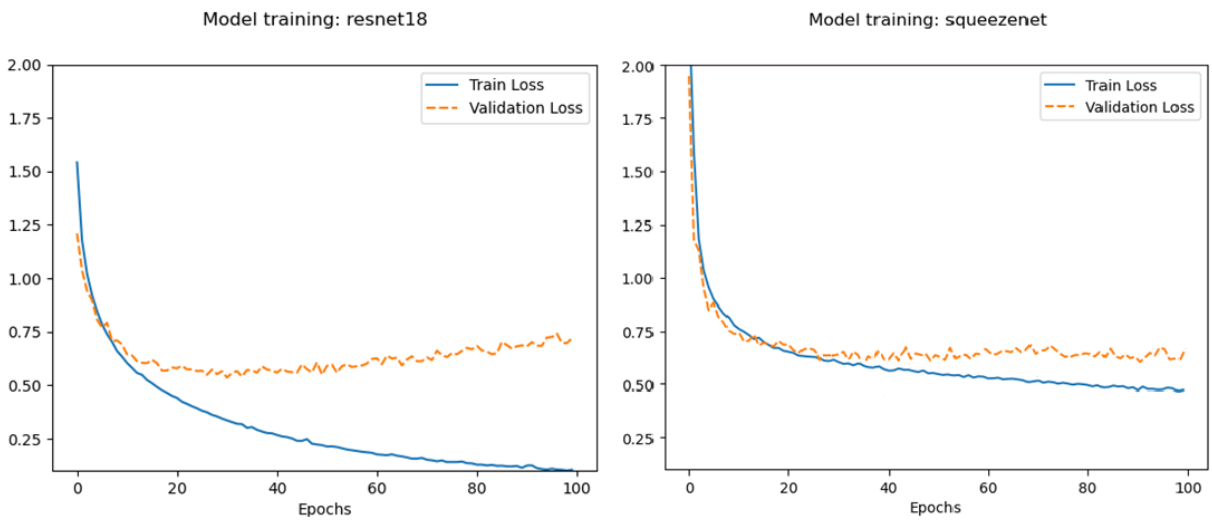

Figure 24. ResNet-18 and SqueezeNet train plots.

From the observation of the graphs above, one can see that the *ResNet-18* and *SqueezeNet*'s validation losses stop improving at around 30 and 25 epochs respectively. The two models' inability to achieve better validation loss can be explained by their architectures. The *ResNet-18* and *SqueezeNet* models from *Torchvision's Model Zoo* are designed for the *ImageNet* dataset, and so the feature maps become very low dimensional (down to 1x1 in the case of *ResNet*, Figure 25). However, because rather than maximizing the float model accuracy, this experiment's main goal is to explore the *Vitis-AI* capabilities as much as possible, the model architectures remained unchanged.

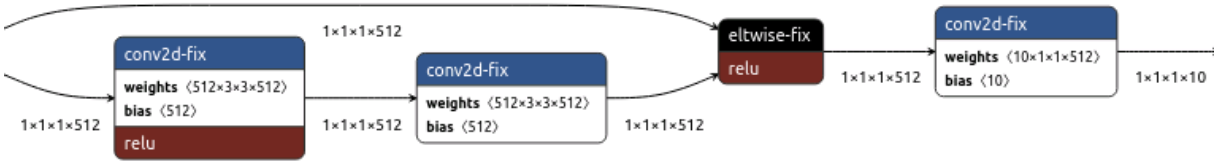

Figure 25. Visualization of ResNet-18 activation map shapes.

4.2.2 MODEL QUANTIZATION

There are 3 methods for quantizing a model in *Vitis-AI*. In increasing order of cost and accuracy performance, they are quantized calibration, fast finetuning, and QAT. This experiment contemplated all 3 methods by deploying and benchmarking quantized models obtained from all three quantization methods. 3.4.2.4 details which algorithms, and respective works, are used for each method.

Considering the *Pytorch* quantizer, and depending on the quantization method, a different set of requirements are needed to execute quantization successfully. The methods are made available through

a quantization API. This allowed the quantization of the two contemplated models to be in great part transparent. Table 8 summarizes the most basic requirements for each method, empirically evidenced through the quantization of both the *ResNet-18* and *SqueezeNet* models using the quantization API. The calibration dataset, used for the fast-finetuning quantization, consisted of an unlabeled subset of the original dataset. The length of this subset depends on the complexity of the data, but the *Vitis-AI* documentation refers to an interval between 100 and 1000 samples. In this experiment, a subset containing 1000 images was used. Another important point to note is that, although not strictly a requirement, the pre-processing and post-processing of the data, before and after model inference, should remain unmodified from the training of the float model to obtain comparable accuracy results.

Contrarily to the remaining methods, QAT also requires modifying the network description to perform the training. These modifications do not alter the architecture of the neural network. However, they must be performed manually and hence are prone to errors. A detailed list of the requirements, as well as a comparison between *ResNet-18's* original description and a modified description used for QAT, produced during this experiment, are available in figures 64 and 65 of Appendix II.

Lastly, all the quantization methods described quantized the models using an 8-bit representation of the parameters. This is because the quantization bit-width of the target DPUs is limited to 8 bits.

Table 8. Comparison of Vitis-AI quantization methods' requirements.

Method	Quantize Calibration	Fast Finetuning	QAT
Pre-trained float model	✓	✓	×
Python script w/ model description	✓	✓	✓
Calibration dataset	×	✓	×
Original dataset	×	×	✓

4.2.3 DEPLOYMENT ON TARGET HARDWARE

Before deploying a model, it was first needed to compile it for a target DPU. The compilation of both models targeted the *DPUCZDX8G* and *DPUCVDX8G* for the *Zynq UltraScale+ MPSoC ZCU104* and *Versal ACAP VCK190* board respectively. *Vitis-AI* compiler was used for this effect. The resulting compiled models were then sent to the respective target boards over secure copy protocol.

The deployment phase consisted in developing a *Python* application using the *Vitis-AI-Library* and *NumPy*. The core functionality of this application is to pre-process, infer and post-process 10000 CIFAR-10 samples measuring accuracy and inference speeds. The images are first normalized and then

converted to INT8 representation. Then, the inference is executed using a batch size of 1 to simulate an ADAS application. Lastly, the post-processing simply consists of the calculation of the index corresponding to the biggest probability in classification and the top-1 accuracy is calculated.

Since the target DPUs support pipelining, meaning that it is possible to execute different layers of networks of different images at the same time, a multi-threaded version of the application was also developed. The use of multiple threads shouldn't be confused with batch inferencing, where multiple images are forwarded through the same network layers at the same time. Figure 26 depicts the differences between sequential inferencing, pipelining, and batch inferencing in terms of single image inference latency and throughput.

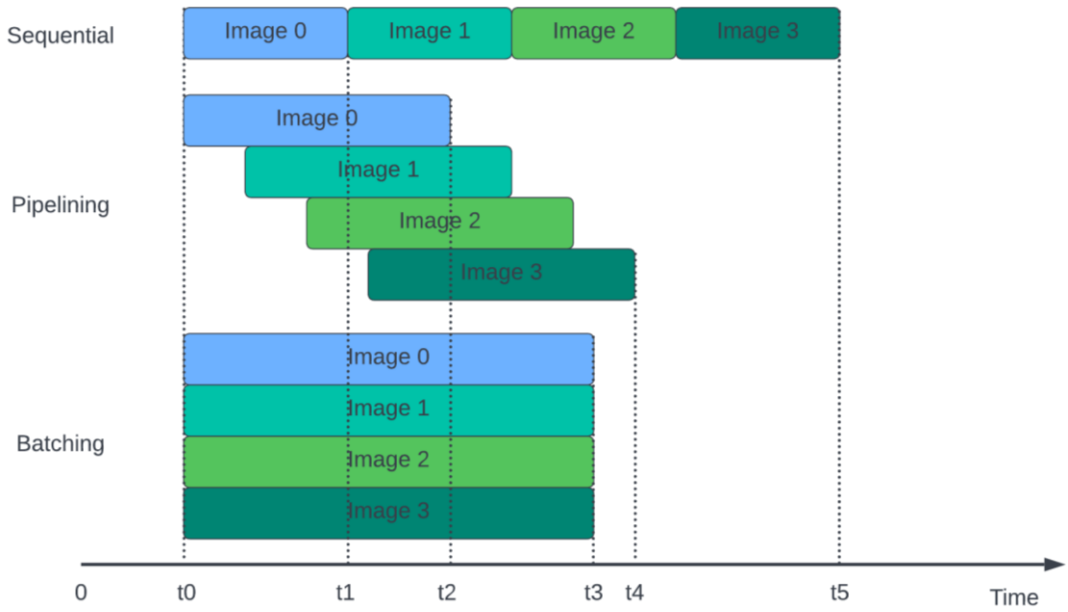

Figure 26. Sequential, Pipelined and Batched inference.

In the context of an ADAS application, one would think that inference latency should always be prioritized. While that is correct when a model's inference latency is smaller than the time it takes for the sensor to create a frame, sometimes it might not be possible to produce a fast enough inference. In this case, it might be useful to consider trading some throughput for inference latency. Consider Figure 27, where inference over a single image takes twice as long as the image generation. Here, when using a sequential inferencing approach, some images are discarded. For example, there is not a good reason to infer over image 1 when the more recent image 2 is already available. When using pipelining, all images become important at the expense of longer single image inference latency. This means that the results of inferencing over a single image will provide less relevant information because more time has passed

since the events that the results represent happened. On the other hand, the route planning module of the ADAS will have access to more images to produce accurate results. The tradeoff becomes quantity vs temporal relevance of the frames. The flexibility to be able to experiment with this tradeoff constituted a very valid reason to explore a pipelining approach.

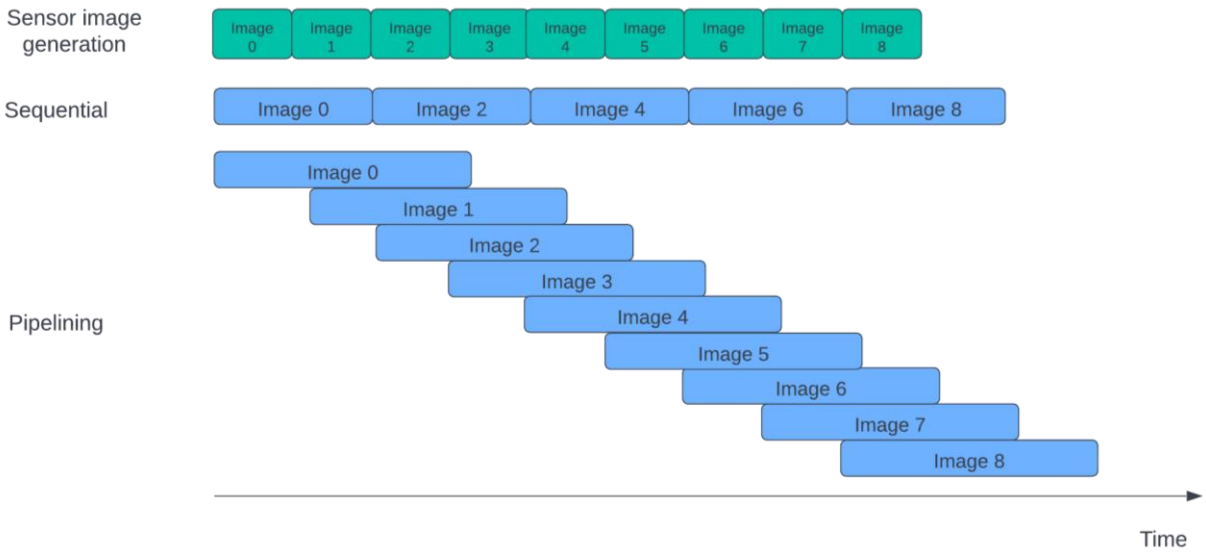

Figure 27. Inference latency vs temporal resolution trade-off.

In the developed multi-threaded application, as soon as the hardware responsible for computing a specific layer is available, a new image can start inferencing without having to wait for the previous image to be forwarded through all layers. This is similar to the operation pipelining that happens in CPUs.

To measure the theoretical frames that the developed application could process per second, the solution described in Figure 28 was used. In this solution, as soon as one CPU thread from the thread pool is available, it copies the data from an input image to a specific array that it sends to the DPU requesting inferencing. Then, it waits for the results and writes them to the results array. This means that the DPU will receive inference requests before it has finished previous requests. The management of the inferences given the DPU available resources is transparent to the application and made available by the *Vitis-AI Library API* through a *Graph Runner* object.

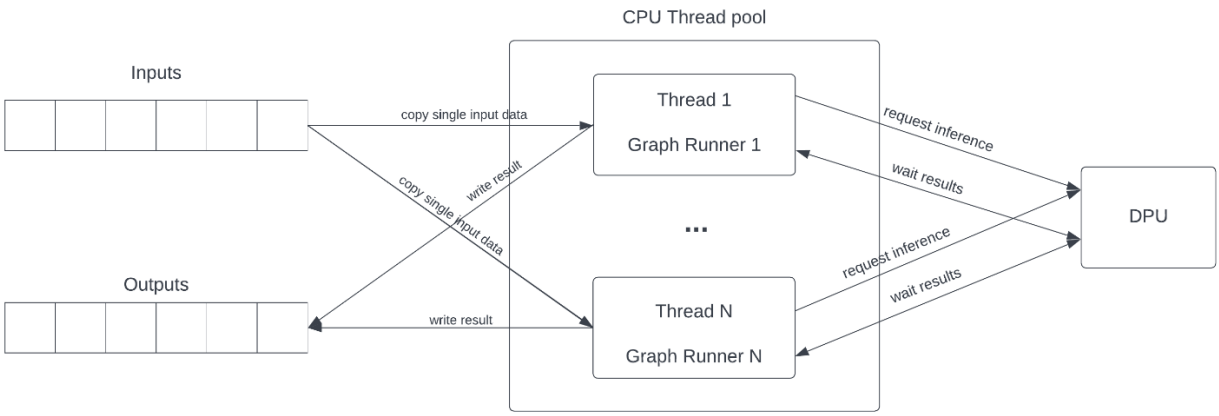

Figure 28. Multi-threaded application architecture.

4.3 RESULTS AND ANALYSIS

This section is divided into two subsections. The first presents the results relative to the quantization of the two targeted models. All quantization methods are contemplated. Model size and accuracy are the two metrics considered. QAT plots are also analyzed. The second focuses on the performance and efficiency results of the two targeted models on both target boards during inference by leveraging the results of the deployed application. More specifically, the average inference FPS and the peak power consumption. Comparisons are also drawn with the *NVIDIA RTX 3090 GPU*.

4.3.1 QUANTIZATION

All three quantization methods substantially reduced the model size while retaining the accuracy of the baseline float model. All methods achieved the same model size reduction ratio of 4.07 and 3.33 respectively for the *ResNet-18* and *SqueezeNet* models. Regarding accuracy, in the *ResNet-18* model, quantized calibration and fast finetuning achieved values withing 0.18% and 0.06% of the baseline accuracy. QAT surpassed the baseline accuracy by 1.11%. In the *SqueezeNet* model, the differences were 0.56% and 0.71% respectively for the first two methods and an increase of 0.42% with QAT. The results are listed in Table 9.

Table 9. Vitis-AI quantization accuracy and model size reduction.

Model	Quantization Method	Best Accuracy	Size (MB)
ResNet-18	None (Float)	83.67%	45.00
	Quantized Calibration	83.49%	11.04
	Fast Finetuning	83.61%	11.04
	Quantization-Aware Training	84.78%*	11.04
SqueezeNet	None (Float)	80.20%	2.90
	Quantized Calibration	79.64%	0.87
	Fast Finetuning	79.49%	0.87
	Quantization-Aware Training	80.62%*	0.87

* Trained for extra 5 epochs

The models were trained for 5 additional epochs during QAT. As an example, Figure 29 shows the training and validation loss curves of the *SqueezeNet* model during QAT. Contrary to the float model validation loss curves during the standard training (figure X), the validation loss keeps decreasing past epoch 25.

Figure 29. Quantization-aware training plot of SqueezeNet.

4.3.2 PERFORMANCE AND EFFICIENCY

The following results compare the FPS and average power consumption of the deployed application, specifically during inference. Refer to 4.1.5 for the complete list of hardware configurations tested. Pre-processing and post-processing were not considered. The FPS values result from dividing the number of frames in the test set (10000) by the total time it took to infer over all images. This procedure was

repeated 10 times and the results were averaged. A similar procedure was to calculate the peak power consumption. The maximum power consumed during all the 10000 frames' inferences was saved. The procedure was also repeated 10 times and the results were averaged.

For each configuration, the FPS and peak power consumption values were calculated for application runs with CPU threads varying in the interval $[1, 12]$. The complete list of the results is available in Table 29 of Appendix IV.

4.3.2.1 ZYNQ ULTRASCALE+ MPSOC ZCU104

Regarding the *ZCU104* architectures, and comparing framerates, the b4096 configurations outperformed the b1024 and b512 configurations for comparable DPU core counts and CPU thread numbers. Furthermore, configurations with a single DPU core only benefited from increasing the number of CPU threads up to 2. Configurations featuring 2 DPU cores benefited from an increase of up to 4 threads. No difference was seen between a hybrid and BRAM-only approach to on-chip memory (green and orange lines are approximately superimposed in both plots). The high ram usage in the single DPU core b4096 architecture resulted in residual improvement in FPS. These results can be observed by the plots in figures Figure 30 and Figure 31.

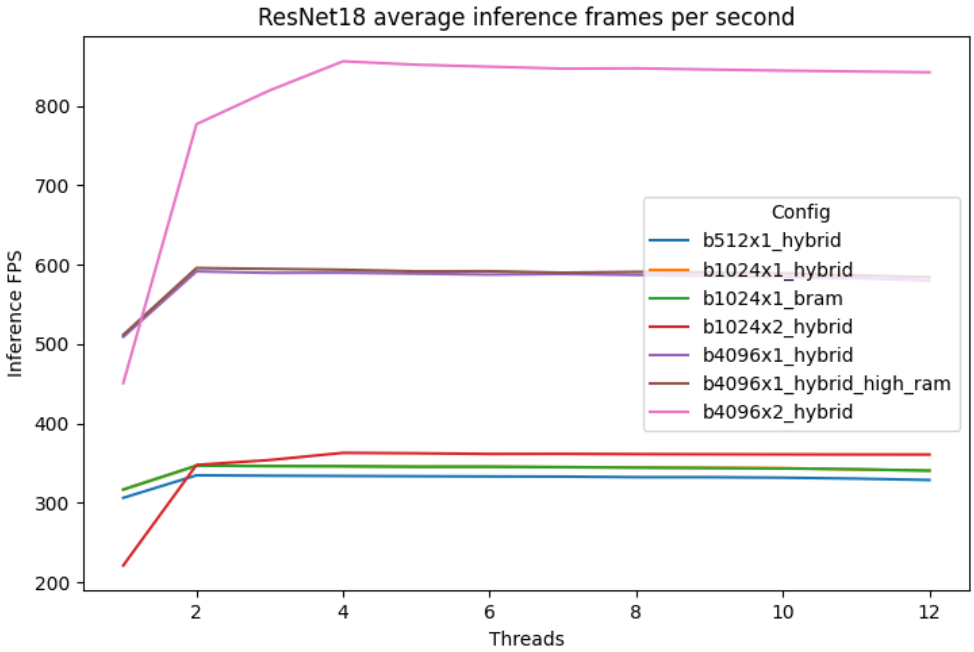

Figure 30. ResNet-18 average inference FPS on all ZCU104 configurations.

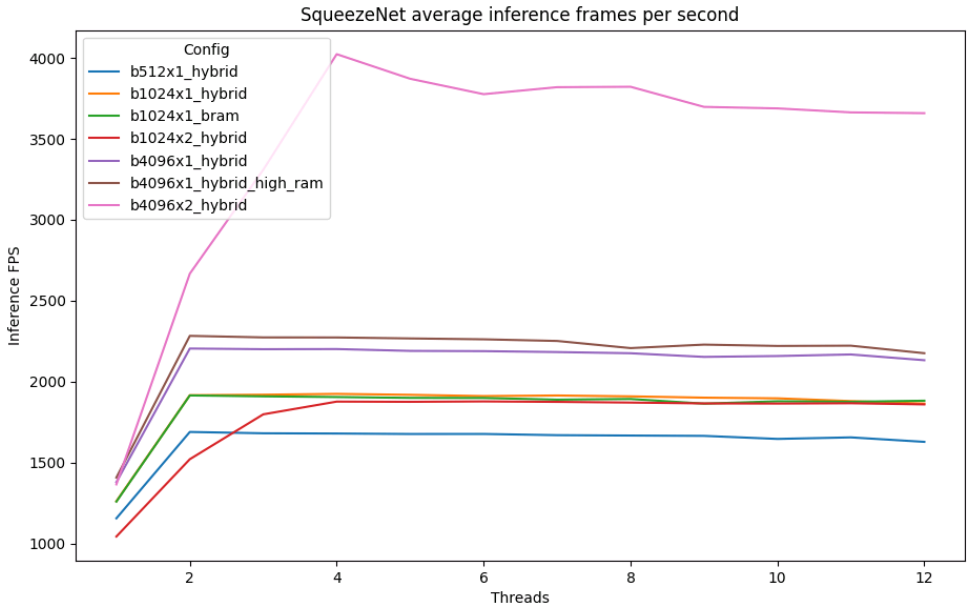

Figure 31. SqueezeNet average inference FPS on all ZCU104 configurations.

Figures Figure 32 and Figure 33 show that the power consumption of the configurations was consistent with the reported resource usage of each configuration (Table 6) with more resources translating to a higher peak power consumption during inference. Also, the peak power consumption increased when the number of threads increased. However, in most configurations, this increase was only significant until 2 or 4 threads.

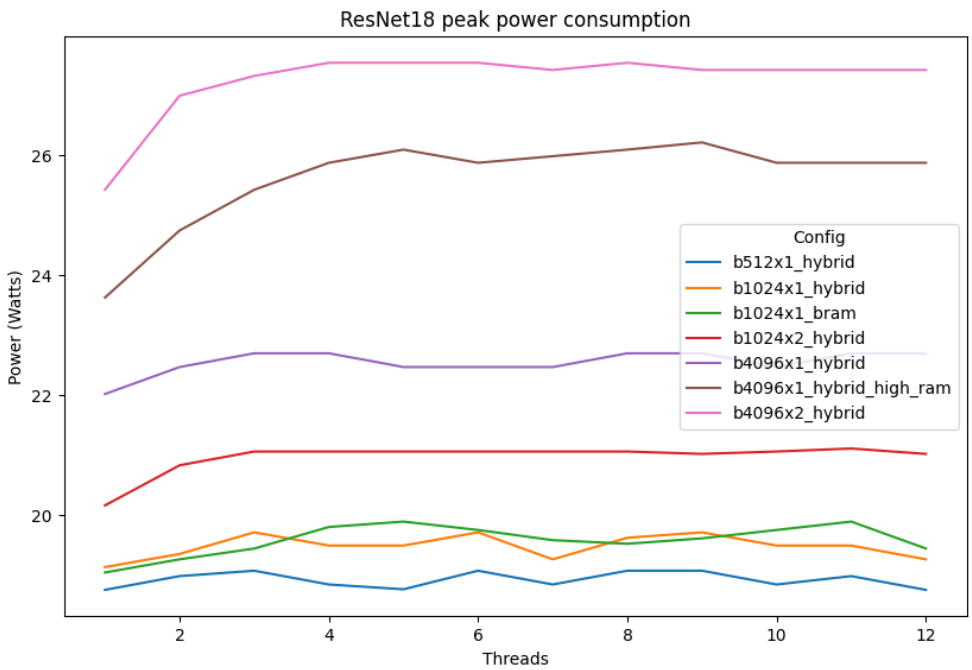

Figure 32. ResNet-18 peak power consumption on all ZCU104 configurations.

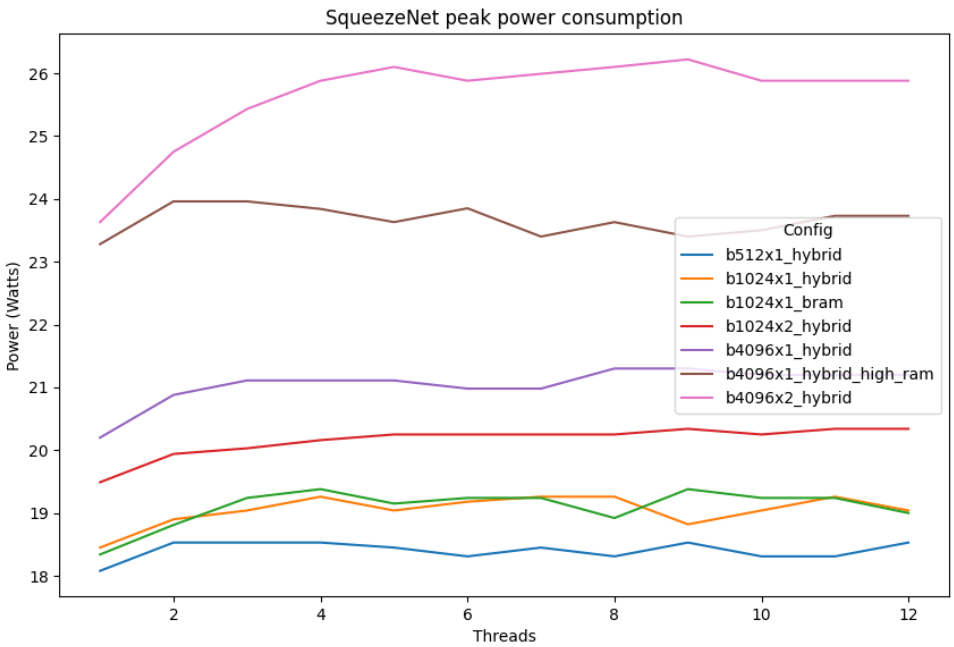

Figure 33. SqueezeNet peak power consumption on all ZCU104 configurations.

Combining the information of the two previous measurements, FPS and peak power consumption, it is possible to evaluate the performance per Watt of each configuration explored. See figures Figure 34 and Figure 35. The results closely mimic the FPS plots showing a well-marked superiority of the 2 DPU core b4096 configuration. These results also highlight the benefit of using multiple CPU threads which are a result of the greater improvements to FPS compared to power consumption when using more CPU threads. Again, the improvements were limited to 2 and 4 threads for single DPU core and 2 DPU cores configurations, as noted in the FPS results analysis.

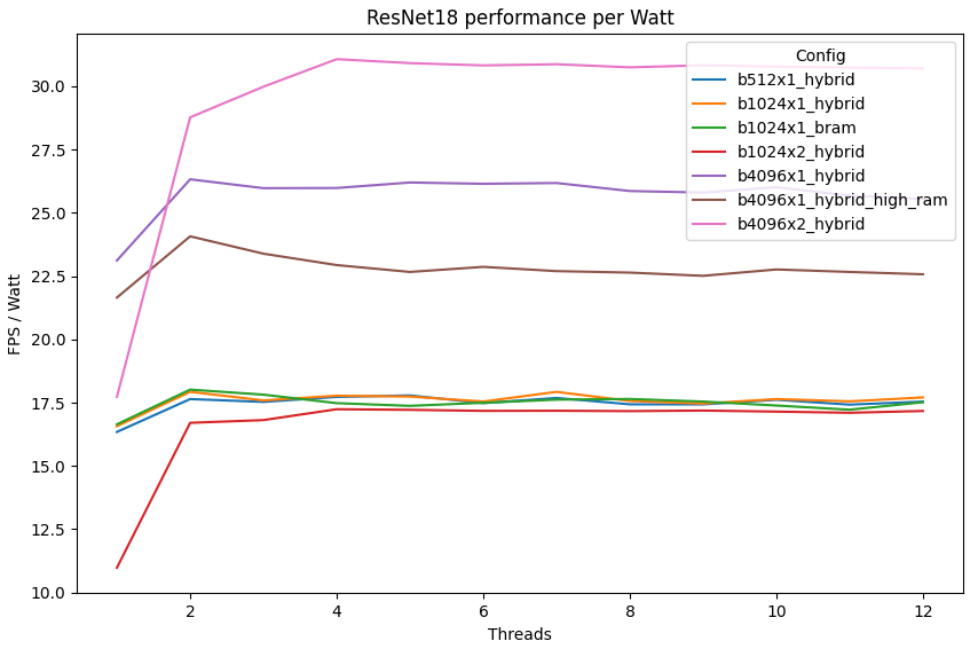

Figure 34. ResNet-18 performance per Watt on all ZCU104 configurations.

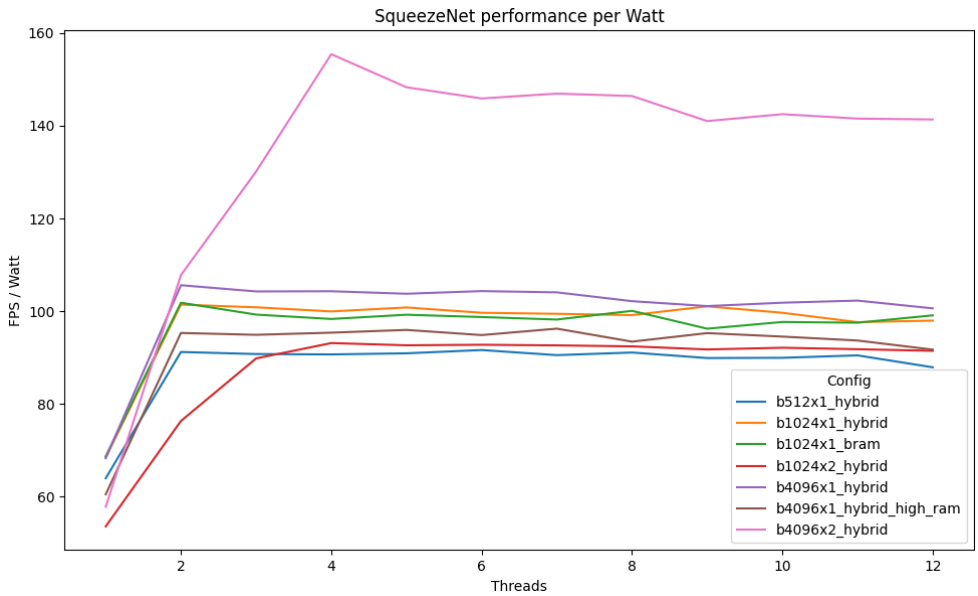

Figure 35. SqueezeNet performance per Watt on all ZCU104 configurations.

4.3.2.2 VERSAL ACAP VCK190

The same exact measurements were realized for the *VCK190* board. Regarding FPS, plots from figuresFigure 36 and Figure 37 indicate that the C64B1 configurations clearly improved results compared with the C32B1 configuration on the *ResNet-18* model. In the *SqueezeNet* model, the difference in FPS is not nearly as significant. Comparing the number of DPU cores of the C64B1 configurations, the advantage of using one additional CPU core in the heavier *ResNet-18* model is very clear. However, the

same is not true for the smaller *SqueezeNet* model. Similarly to the *ZCU104* configurations, additional CPU threads improve the performance up to 2 and 4 threads respectively for single DPU core and 2 DPU cores configurations.

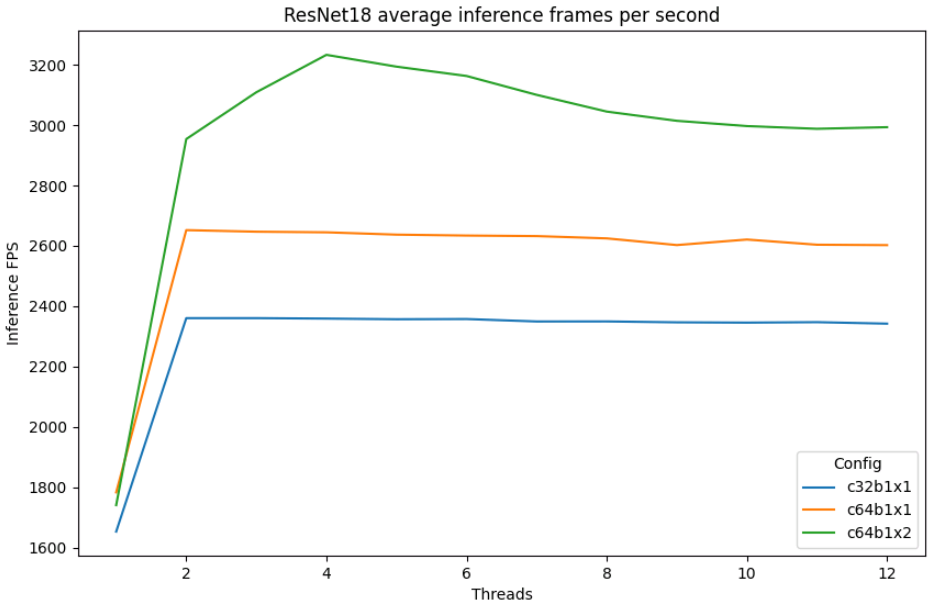

Figure 36. ResNet-18 average inference FPS on all VCK190 configurations.

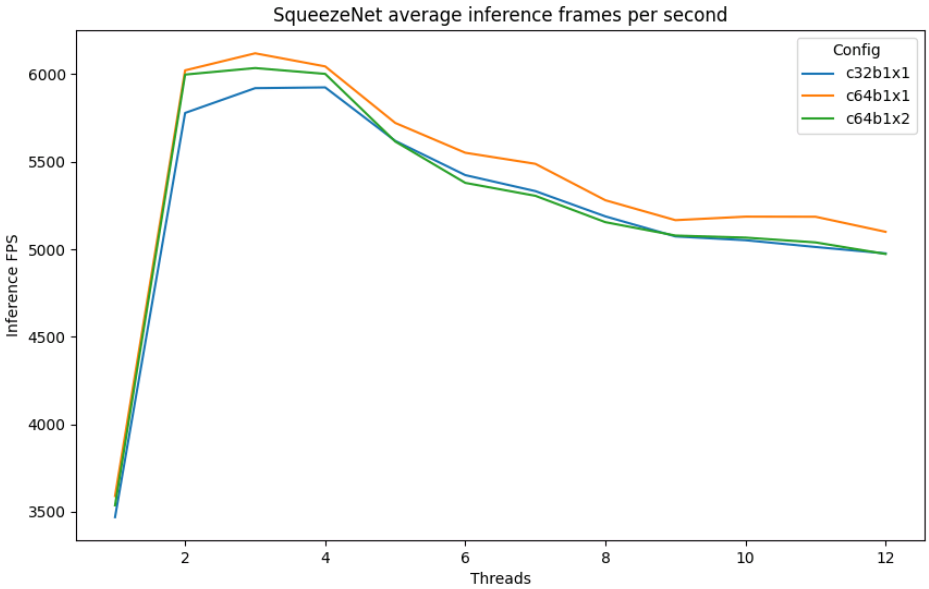

Figure 37. SqueezeNet average FPS on all VCK190 configurations.

Analogous to the *ZCU104* results, the peak power consumption of the *VCK190* configurations, visible in figures Figure 38 and Figure 39, also increases with the increase in resource usage and CPU threads.

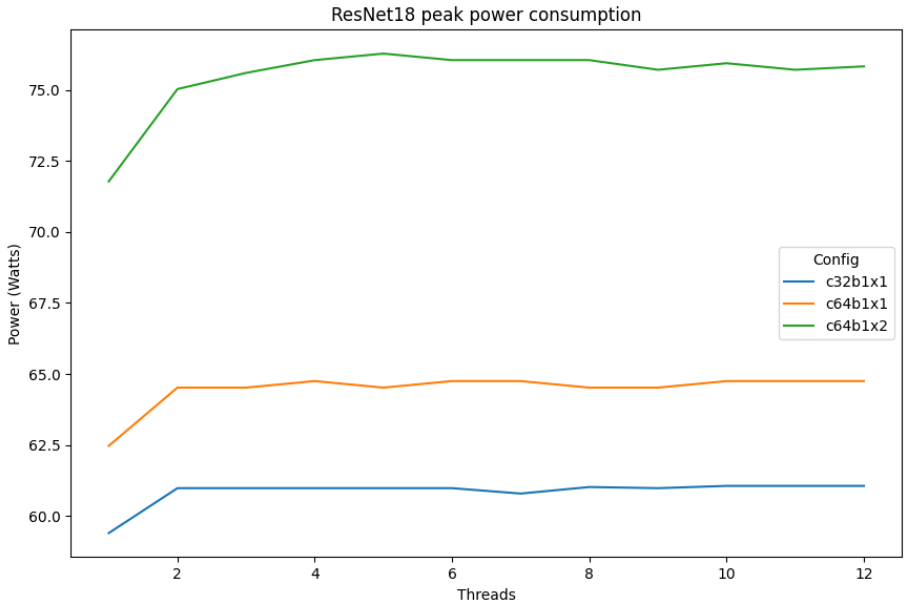

Figure 38. ResNet-18 peak power consumption on all VCK190 configurations.

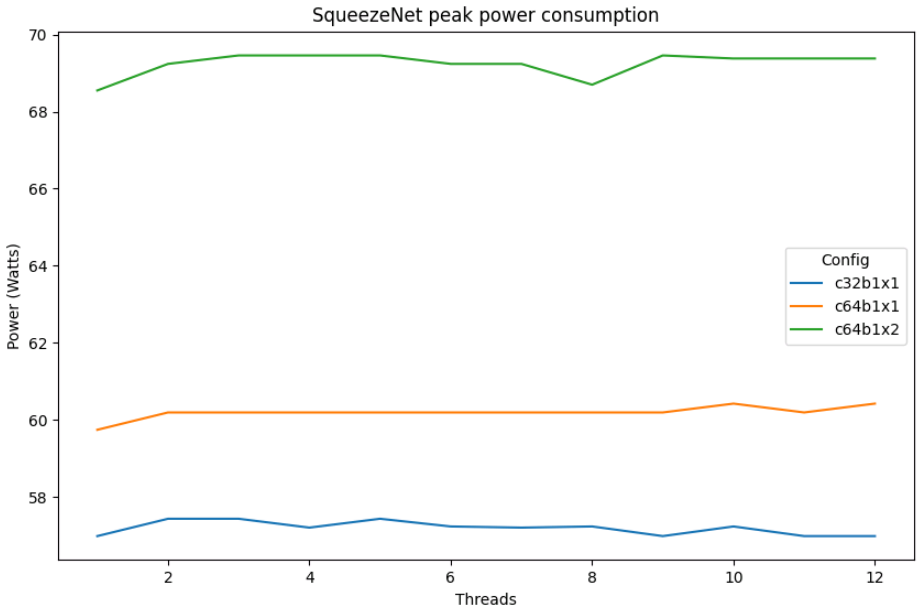

Figure 39. SqueezeNet peak power consumption on all VCK190 configurations.

The performance per Watt plot of the *ResNet-18* model shows that all three configurations have similar performance per Watt values. However, the situation is far from the same in the SqueezeNet model. The C64B1 with 2 DPU cores is significantly less efficient. Refer to figures Figure 40 and Figure 41 for the respective plots.

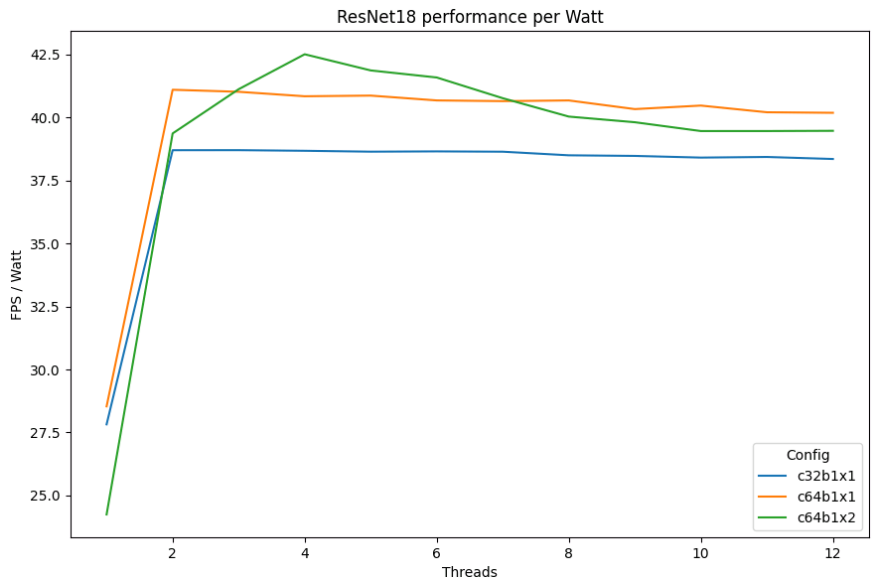

Figure 40. ResNet-18 performance per Watt on all VCK190 configurations.

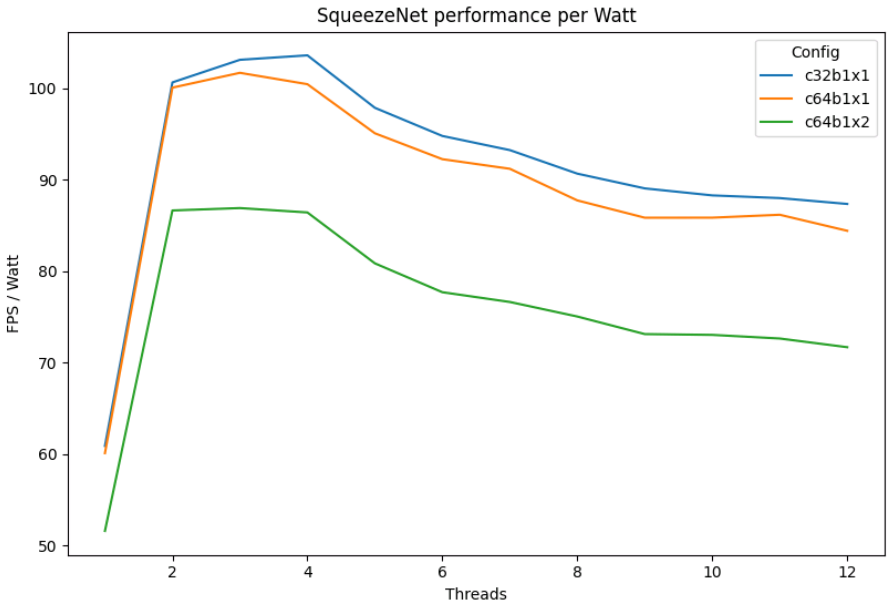

Figure 41. SqueezeNet performance per Watt on all VCK190 configurations.

4.3.2.3 ZCU104, VCK190, AND RTX 3090 COMPARISON

To further evidence the suitability of *Vitis-AI* to deploy CNN architectures on the two explored target boards, a comparison to the *NVIDIA RTX 3090* board was conducted. This comparison also helps to frame the achieved results in the literature since most published deep learning works target state-of-the-art GPUs. The GPU inference was performed in *Pytorch* with a batch size of 1. Clearly, the RTX3090 will be clearly underutilized with a batch size of 1 and low dimensional input. The inclusion of RTX3090 in this comparison is to preserve coherence with the next experiment but can also be used to show that

GPUs, especially GPUs such as RTX3090, are clearly not the appropriate hardware for low latency inferencing applications. As for the target boards, both the least resource intensive and most resource-intensive configurations and CPU thread number combinations of each board were considered.

Regarding FPS values, all but one of the tested configurations surpassed the FPS that the *RTX 3090* achieved. The difference in performance is very noticeable with the *ZCU104* reaching up to 2.47x and 11x the performance of the *RTX 3090* in the *ResNet-18* and *SqueezeNet* models respectively. Comparing the *RTX 3090* with the *VCK190*, the improvements are 9.34x and 16.44x. (Figure 42).

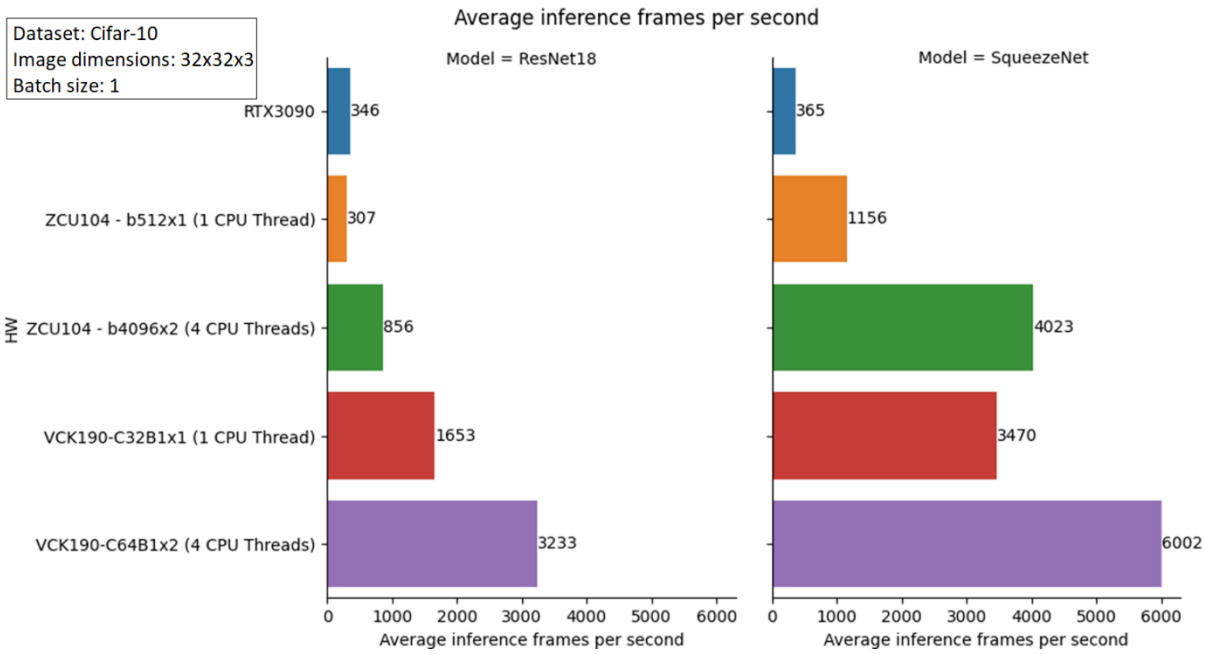

Figure 42. Avg inference FPS of RTX3090, ZCU104 and VCK190.

More interestingly is that the observed FPS improvements were achieved at much lower power consumption. Figure 43 shows that all configurations consumed substantially less power during inference when compared with the *RTX 3090*.

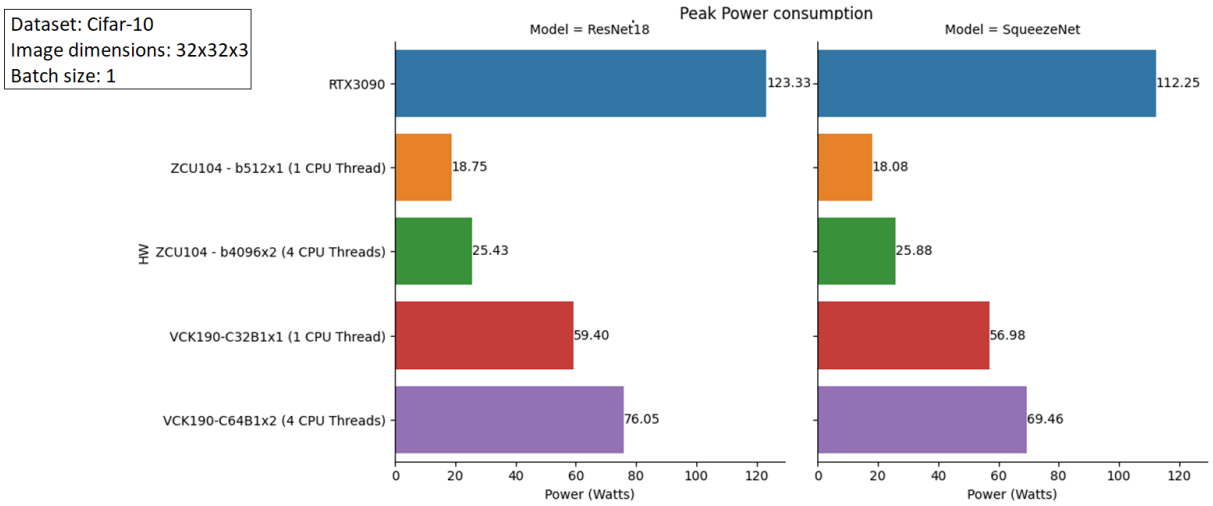

Figure 43. Peak power consumptions of RTX3090, ZCU104, and VCK190.

Consequently, the result is that the performance per watt of the *ZCU104* was up to 12x and 47.8x superior to the *RTX 3090* for the *ResNet-18* and *SqueezeNet* models. For the *VCK190*, the values were 15.1x and 26.6x superior respectively, as can be observed in Figure 44.

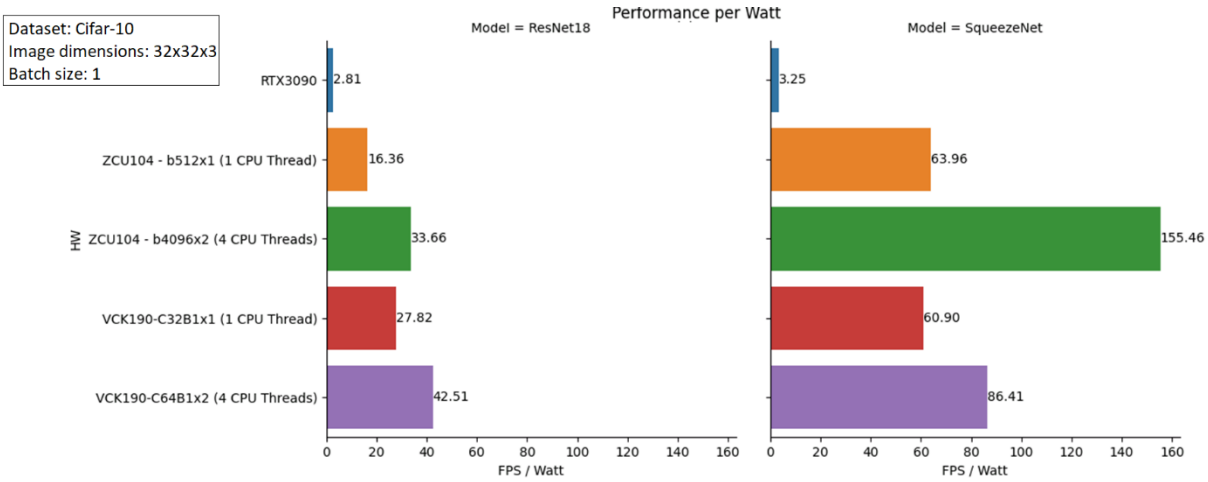

Figure 44. Performance per Watt of RTX3090, ZCU104, and VCK190.

Finally, it is possible to draw a comparison between the *ZCU104* and *VCK190* boards. Clearly, with more resources, the *VCK190* not only achieved higher FPS values but also consumed more power. In the *ResNet-18* model, the FPS difference between the configuration of the two boards is very large and compensates for the higher power consumption of the *VCK190* board. For this reason, the performance per Watt of the *VCK190* board was superior to the *ZCU104*. However, due to the smaller difference in FPS between the two boards' configurations in the smaller SqueezeNet model, the higher power

consumption of the *VCK190* translated into a lower performance per Watt of the *VCK190* board compared with the *ZCU104*.

4.4 DISCUSSION

This section aims to interpret the results and comment on the analysis made in the previous chapter. The questions it seeks to answer are related to the suitability of the *Vitis-AI* tool for the objectives of this work. To answer, an evaluation of the results is carried out through comparisons to theoretical values and other works highlighted in the literature review.

Once again, the discussion is divided into two topics, similar to the structure of the last section.

4.4.1 QUANTIZATION

As expected, the model size reduction was close to the ratio of reduction in bit-width, 4. It is important to note that to obtain the size of the quantized models, compilation for the target DPU must be performed first. As already discussed in 3.4.2.4, the compiler introduces some optimizations such as layer fusion that might cause the model size to increase or reduce. This explains why the reduction ratio was not exactly 4.

Quantization results were very satisfactory with close to no accuracy degradation in the first two quantization methods. Better accuracy results were expected from fast finetuning when compared to the simpler quantized calibration. This was indeed observed in the *ResNet-18* model, but not the *SqueezeNet* model. However, because the differences in accuracy were so small (<0.025%), they become irrelevant. It is expected that with more complex datasets which require more complex training and networks, the differences become more significant. Finally, QAT outperformed the baseline float model. Again, the difference in accuracy was not very large, especially for the *SqueezeNet* model.

Compared to the results found in the literature review, one can clearly be optimistic about the usage of *Vitis-AI* for quantizing deep neural networks. Not only was the ratio of model size reduction close to the theoretical value, 4, but the accuracy retention for the quantized calibration and fast finetuning methods also achieved results within less than 1% of the accuracy of the baseline float model. Both results are comparable to the ones advertised by Nagel, M., et al. work [102], responsible for the algorithm behind quantized calibration, and Hubara et al. work [103], responsible for the introduction of the post-training quantization algorithm. Regarding QAT, the results of this method even surpassed the model accuracy of both baseline float models. However, it is important to keep in mind that this experiment targeted a

simpler dataset than, for example, the ImageNet dataset. It might be the case that, even with the noise introduced by the quantization processes, the models were still able to retain accuracy because of the simplicity of the task they are solving. Hence, it is still necessary to evaluate the quantization methods on larger and more complex datasets. Chapter 5 explores a far more challenging dataset.

4.4.2 PERFORMANCE AND EFFICIENCY

The performance observed during inference in both target boards, represented by the FPS achieved during inference, was more than satisfactory and strengthens the argument for the suitability of *Vitis-AI* to the deployment of fully convolutional architectures. Even more so when allied with the benchmarks advertised on the *Vitis-AI GitHub* page [24]. Furthermore, the developed multi-threaded application was shown to increase the performance of all the configurations and can serve as a good starting point for future works that want to explore low latency and time resolution. Similarly, the power consumption results were also very satisfactory. But more importantly, the capability to easily trade performance with power consumption by changing DPU configurations, and the number of CPU threads, is probably the most important asset derived from developing such an application using Vitis-AI. Compared with the RTX 3090, the power consumption reduction of 6.58x and 6.21x of the *ResNet-18* and *SqueezeNet* models on the *ZCU104*, using the B512 single DPU core configuration with 1 CPU thread, which translates approximately into 85% and 84% reductions, compared with the results found in the work of Hashemi, S. et al. [112] described in 3.3.2.

Finally, there is an interesting observation that requires further attention when considering the performance per Watt of the *ZCU104* and *VCK190*. Figure 44 shows that depending on the model, different DPUs can become more efficient than others. For the *ResNet-18* model, the *VCK190* offers a better performance per watt on both configurations. For the *SqueezeNet* model, it is the *ZCU104* that offers better performance per watt. To understand why this happens it is important to consider the usage of the *Vitis-AI* profiler. By analyzing the load size of feature maps (LdFM), load size of weights and biases (LdWB), and store size of feature map (StFM), present in tables Table 10 and Table 11, it can be noted that the *ResNet-18* requires a lot more weights and biases loads. This is expected due to the larger size of the model, 11 MB to 0.87 MB. Because the average off-chip memory bandwidth is considerably lower in the *ZCU104*, the smaller amount of on-chip memory available in the *ZCU104* compared with the *VCK190* might explain the lower performance of the *ZCU104*. On the other hand, because the *SqueezeNet* model requires such a low amount of loads, the smaller off-chip memory of the *ZCU104* is not major a bottleneck on the performance.

Table 10. ResNet-18 inference DDR memory access information on ZCU104.

Model	Total LdWB (MB)	Total LdFM (MB)	Total StFM (MB)	Avg Bw (MB/s)
ResNet-18	10.650	0.054	0.030	6446.568
SqueezeNet	1.333	0.060	0.045	1672.493

Table 11. SqueezeNet inference DDR memory access information on ZCU104.

Model	Total LdWB (MB)	Total LdFM (MB)	Total StFM (MB)	Avg Bw (MB/s)
ResNet-18	10.662	0.048	0.030	26402.281
SqueezeNet	0.714	0.024	0.024	6106.555

The results discussed above are sufficient evidence for the usefulness of *Vitis-AI* to tackle the remaining objectives of this work. Furthermore, *Vitis-AI* proved useful in abstracting away most complex hardware architecture aspects of deployment while still providing a lot of control over resource utilization. This is evidenced by the tradeoff that was shown between performance and power consumption.

5 SqueezeSegV3 Deployment on an FPGA

5.1 EXPERIMENT DESCRIPTION

The previous experiment served as validation of the suitability of the *Vitis-AI* tool for the quantization and deployment of convolutional neural networks on *Xilinx*'s FPGAs. Based on the previously discussed results, the experiment here described aims to deploy a LiDAR-based deep neural network to perform semantic segmentation on point cloud data in real-time. The deep neural network should be deployed on a chosen target FPGA. Accuracy metrics, power consumption, model size, and FPS are the key performance indicators.

The data size of point clouds compared to images, as well as the more exotic architectures that are employed when processing 3D data, are expected to be a challenge for the real-time deployment of a deep learning model. For these reasons, this experiment aims to further validate the FPGAs' suitability for the deployment of 3D computer vision models in real-time applications.

5.1.1 OBJECTIVES

The choice of neural network architecture depends on the available hardware support within Vitis-AI. For this reason, the first objective is concerned with the choice of the neural network to be deployed. Then, because it is unlikely that all model layers are perfectly supported by the targeted DPU, alternatives to those layers should be identified, implemented, and benchmarked concerning accuracy metrics, model size, and FPS on the GPU. When a baseline supported architecture has been found, further changes in the neural network architecture should be benchmarked and evaluated as an accuracy/framerate tradeoff. This tradeoff should ensure that the final deployed model can surpass the 10 FPS to comply with the LiDAR frame generation frequency of 10 Hz identified in 3.1. Here, the inference application developed in the target FPGA is to be deployed similarly to the last experiment. Besides accuracy and inference, power consumption and model size should still be measured for every single change made in the baseline architecture to access its advantages and disadvantages.

5.1.2 DATASET

To evaluate the model accuracy, the *Semantic-KITTI* dataset [149] was used. The dataset is partitioned into 22 sequences containing between 200 and 5000 frames. Each sequence represents a portion of the circuit driven, and consequent frames represent consequent circuit points. The data collection was

performed across urban areas, rural areas, and highways. Up to 15 cars and 30 pedestrians are captured per frame. The frames were captured at a rate of 10 FPS.

Sequences 0 to 10 are intended for model training except for sequence 8 which is used for validation. All first 11 sequences are densely labeled. Concerning the task at hand, semantic segmentation, each point in the point cloud of the first eleven frames is labeled as one of 28 classes. However, only 19 are considered during training and evaluation. Figure 45 presents each class and the corresponding number of points across all frames. The remaining sequences are reserved for online test benchmarking of model submissions. Each point cloud is approximately 2.0MB in size.

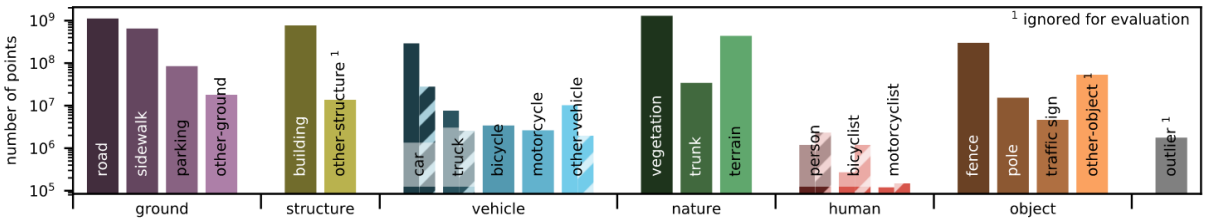

Figure 45. Semantic-KITTI dataset points class distribution. Retrieved from [150].

The choice of the *Semantic-KITTI* dataset is justified by the number of papers that currently use the dataset. This makes this work easier to compare with the current state-of-the-art. Because *Semantic-KITTI* used the mechanical *Velodyne HDL-64E* LiDAR, it also has a very good resolution compared with other 3D semantic segmentation capable datasets. Lastly, the dataset also provides the most data out of all the explored datasets. Table 12 summarizes all the dataset's relevant information.

Table 12. LiDAR-based 3D semantic segmentation capable datasets.

Dataset	Year	LiDAR resolution (V x H)	#Classes	#Points	#Papers (2019-2022) *
Oakland [151]	2009	? x 0.5° †	5	1.6 M	0
Paris-rue-Madame [152]	2014	1.33° x 0.1°- 0.4°	17	20 M	0
IQmulus [153]	2015	0.4° x 0.08°- 0.035°	8	200 M	0
Paris-Lille-3D [154]	2018	1.33° x 0.1°- 0.4°	50	143.1 M	9
Semantic-KITTI [150]	2019	0.4° x 0.08°- 0.035°	28	4548 M	183
Toronto-3D [155]	2020	1.33° x 0.1°- 0.4°	8	78.3 M	10

* Data from site Papers With Code

† Data collected using several 2D LiDAR scans

5.1.3 EVALUATION METRICS

To measure the deep learning model performance on the 3D semantic segmentation task, both accuracy and Intersection over Union (IoU) metrics are used (Equation 7).

Accuracy simply represents the ratio of correctly classified points over all points in the point cloud. However, accuracy alone can produce misleading results, especially when averaged over all classes, as less represented classes' results can become irrelevant. A pedestrian that represents only 1% of total points in a point cloud can be completely missed and yet have almost no influence on accuracy.

IoU quantifies the percent overlap between the predicted labels and the ground truth and is calculated per class. It does so by calculating the ratio between the intersection of the predictions with the ground truth over the union of the predictions with the ground truth. The fact that a union is used in the denominator discourages the models to ignore smaller classes. In the same example of the pedestrian, when predicting the pedestrian pixels as belonging to a more represented class, the IoU of the class will decrease. This wasn't the case with accuracy.

$$IoU = \frac{prediction \cap ground\ truth}{prediction \cup ground\ truth} \qquad \text{(Equation 7)}$$

IoU solves the problem of the average accuracy by allowing to discriminate which classes are being well identified. An average IoU can also be calculated by averaging the results of all classes. Figure 46 depicts what is the ground truth, prediction, and respective intersections and unions for a 2D image. The process for a 3D image is analogous.

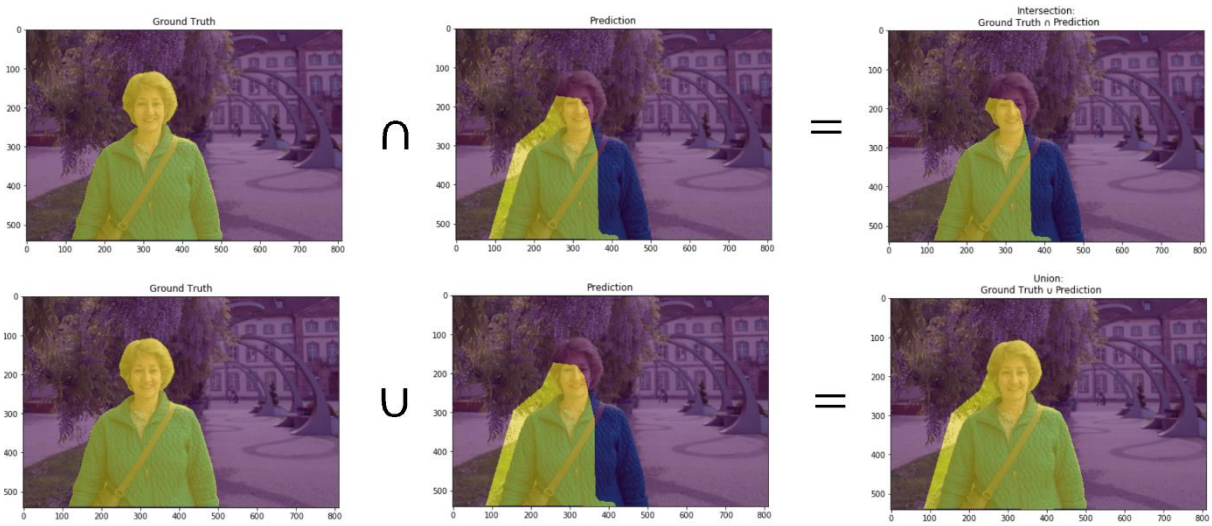

Figure 46. Intersection and union of ground truth and model predictions. Adapted from [156].

In the *Semantic-KITTI* dataset, it is common practice to measure the average accuracy, the per-class IoU, and the average IoU. These are the 3 performance metrics compared in this experiment.

5.1.4 DEEP LEARNING FRAMEWORK

In 4.1.3, the widespread adoption of *Pytorch* was highlighted. To no surprise, the original *SqueezeSegV3* implementation is also provided by the authors using the *Pytorch* framework. For this reason, *Pytorch* was a natural choice in the implementation of the neural network used in this experiment, as well as model training, quantization, and evaluation.

5.1.5 TARGETED DEEP NEURAL NETWORK

SqueezeSegV3-21 is the 21-layer architecture variant of the *SqueezeSegV3* model proposed by Xu, C. et al [157]. It is designed for efficient and real-time processing of large-scale point clouds such as in autonomous driving applications. It is a projection-based architecture. 2.3.3.1 presents a comprehensive description of these methods as well as their advantages and disadvantages.

5.1.5.1 SELECTION CRITERIA

Being a neural network specifically designed for real-time processing of large-scale point clouds makes *SqueezeSegV3* a very fitting choice for this work. Furthermore, its projection-based architecture requires fewer computations compared with architectures that rely on other representations. Despite these

favorable characteristics, the main aspect that drove the choice of this specific neural network was the limited support of *Vitis-AI* for other architectures featuring more exotic layers. Table 13 lists a set of neural networks for 3D semantic segmentation, the respective accuracy metrics, and the unsupported operations/layers in *Vitis-AI*. Some of these operations are replaceable by similar operations. However, in certain cases, some operations are either irreplaceable or are such a fundamental feature of the architecture that their replacement would cause the architecture to lose its identity, e.g., swap 3D convolution layers for 2D convolution layers. It is important to note that, depending on the implementation and targeted DPUs, there might be more operations that are unsupported in each model. This table only gives a broad overview. The only architecture studied with more detail was *SqueezeSegV3*. Table 30 of Appendix V contains a complete list of the *SqueezeSegV3-21* architecture layers of the *Pytorch* implementation with the corresponding *DPUCVDX8G* support for *Vitis-AI* version 2.0.

Table 13. Vitis-AI unsupported operations of 3D semantic segmentation deep learning models.

Model	**Semantic-KITTI Test-set mIOU**	**Vitis-AI Unsupported Operations**
Cylinder3D [158]	67.8	Conv3D, Deconv3D
SPVNAS [159]	66.4	Sparse Point-Voxel Convolution
JS3C-Net [160]	66.0	SparseConv
KPRNet [161]	63.1	KPConv
SalsaNext [140]	59.5	-
SqueezeSegV3* [157]	55.9	Unfold

*53-layer architecture with K-nearest neighbors post-processing

From the explored models, *SqueezeSegV3* was the option with the best IoU that did not contain any irreplaceable unsupported operation. 3D convolution layers, Sparse Convolution layers, and kernel-point convolution (KPConv) layers, all unsupported, are also the main operations of the listed architectures. To increase the relevancy of this work, *SalsaNext* was not chosen since *Xilinx* already provided two implementations of this model in *Vitis-AI*. Lastly, because the unfold operation of *SqueezeSegV3* is not fundamental to the architecture, it can be replaced.

5.1.5.2 MODEL ARCHITECTURE

As a preprocessing step, the 3D point cloud is first projected into a spherical surface creating a 2D grid representation of the LiDAR data. The 3D coordinates of each point before the projection are used as features of the same point when projected into a 2D pixel. The projection operation is realized using (Equation 8), where (p, q) are the resulting 2D grid coordinates or pixels, (h, w) are the height and width of the 2D grid, $f = f_{up} + f_{down}$ is the vertical field of view of the LiDAR sensor and $r = sqrt(x^2 + y^2 + z^2)$ is the range of each point in the point cloud.

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(1 - arctan(y, x)/\pi) \cdot w \\ (1 - (arcsin(z \cdot r^{-1}) + f_{up}) \cdot f^{-1}) \cdot h \end{bmatrix} \quad \text{(Equation 8)}$$

In the *SqueezeSegV3* implementation used in this work, the values for (h, w), f_{up} and f_{down} are (64, 2048), 3, and -25 respectively. If multiple points are projected to the same pixel on the 2D grid, the point with the highest range remains. The values of $x, y, z, r,$ and intensity are used as features of the resulting pixel, similar to the RGB values of 2D images.

The network architecture is similar to *RangeNet++* [162]. However, the standard convolution operations present in the *RangeNet++* architecture are replaced by Spatially Adaptive Convolution (SAC) blocks. This change aims to tackle the problem of spatially varying distribution caused by spherical projection. Unlike 2D RGB images, where the RGB feature distribution at different locations is rather similar when projecting the point cloud, the distribution at different locations is drastically different. For example, along the height dimension, points projected to the top of the 2D grid have higher z-values than the ones projected to the bottom. This spatially varying distribution can degrade the performance of convolution operations.

SqueezeSegV3 follows an encoder-decoder architecture. In the encoding phase, the 21-layer variant of the *SqueezeSegV3* network contains 5 encoder blocks, each containing two convolution layers. The first convolution layer is replaced by an SAC block. As depicted in Figure 47, the coordinate map, containing the $x, y, and\ z$ features of the original points, is processed by a 7x7 convolution layer. The input features tensor, containing all features (x, y, z, r and $intensity$), is unfolded. Both feature maps are then multiplied together and finally passed through two convolution layers with 1x1 and 3x3 kernels each. The resulting feature maps are then added to the original input features tensor. S simplifies (h, w) and I is 5.

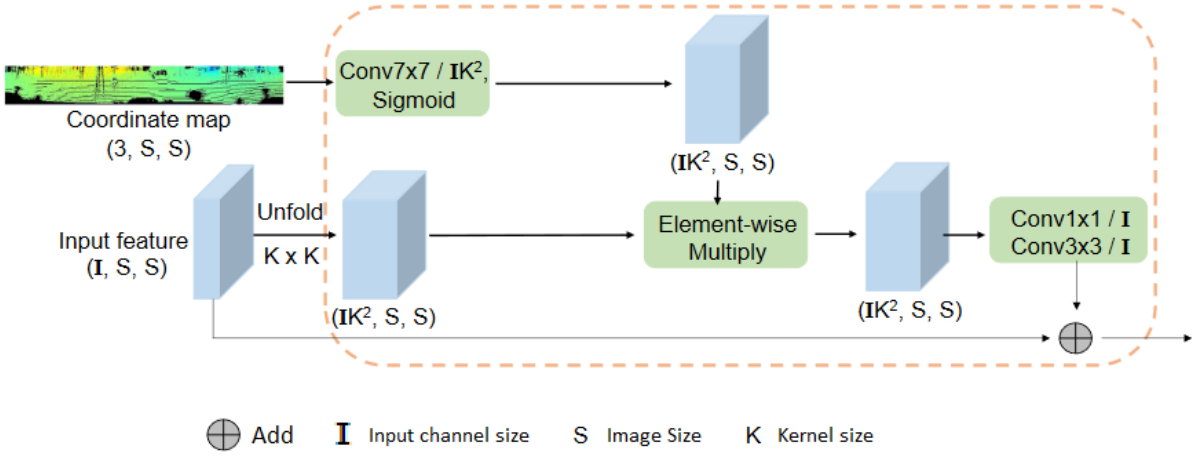

Figure 47. SqueezeSegV3 model's SAC block. Adapted from [157].

In the decoding phase, the feature maps are up sampled by transposed convolution layers. Standard 2D convolution layers are also used to refine the reconstruction of the projection. Residual connections are used between the feature maps of the encoding layers and the decoding layers. This allows the addition of the feature maps of the encoders to the feature maps of the decoders, recovering high-frequency edge information that gets lost during the down sampling process. Figure 48 illustrates the architecture and the data pipeline containing pre and post-processing of the data during the training of the *SqueezeSegV3* model. Lastly, the projected predictions can be restored by applying the inverse process of the projection to the spherical surface.

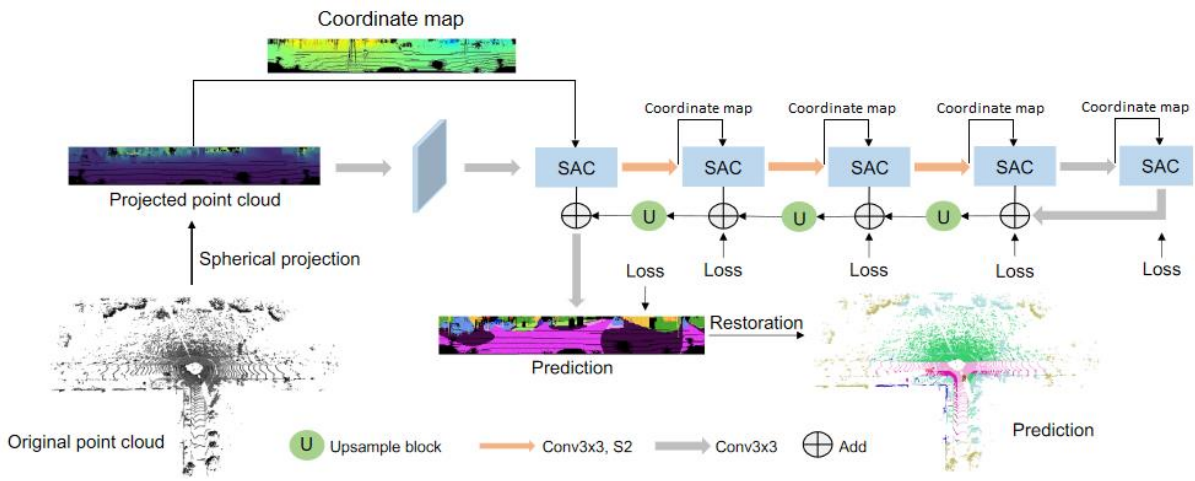

Figure 48. SqueezeSegV3 model architecture with pre and post-processing. Adapted from [157].

5.1.6 TARGETED HARDWARE

Unfortunately, the *DPUCZDX8G* is unable to support element-wise multiplication and addition of the feature maps in the SAC blocks. Because this operation is at the core of the *SqueezeSegV3-21* architecture, and there is no trivial substitute, the *DPUCZDX8G* was abandoned in this experiment. *Vitis-AI* does support offloading unsupported DPU operations to the target CPU. However, a benchmark realized in this work featuring another segmentation model, *PointPillars*, showed that the performance degradation of this approach was very significant. Table 14 briefly summarizes the results.

Table 14. PointPillars inference latency comparison between partial and complete DPU support.

Model	DPU support	ZCU 104 inference (ms)
PointPillars	Complete	5
	Partial	4593

The *DPUCVDX8G* provides support for element-wise multiplication, but not all layers of *SqueezeSegV3* are readily supported by this DPU. However, contrarily to the element-wise multiplication, the remaining unsupported operations can be replaced. A complete list of all layers, respective parameters, and DPU support of the *Pytorch* implementation of the *SqueezeSegV3-21* architecture used in this experiment can be consulted in Table 30 of Appendix V.

In summary, only the *DPUCVDX8G* was targeted in this experiment due to the limitations of the *DPUCZDX8G*. Another possible solution could have been to select another model. However, this would further enlarge the distance of the selected model to the current state-of-the-art models in the literature. *SqueezeSegV3* is a good compromise in terms of support in *Vitis-AI* - can be deployed in the *DPUCVDX8G* – and the proximity to the current best models in the literature for 3D semantic segmentation.

5.2 IMPLEMENTATION

Most work developed in the previous experiment applies to the implementation steps of this experiment. However, unlike in the *ResNet-18* and *SqueezeNet* cases, the model cannot be deployed as it is. Several architectural changes need to be performed first. Furthermore, the training and quantization process of the resulting model, already considerably more time-consuming due to the substantially larger dataset, becomes an iterative process due to the recurrent changes in the model architecture. In other

words, most of the developed work in this experiment focuses on the neural network architecture rather than the tools to deploy the resulting models.

5.2.1 ARCHITECTURAL CHANGES

Several architectural changes were made to the original *SqueezeSegV3* model. The reasons lie in the necessity to adapt the model for the target hardware and performance, namely size, frame rate, and power consumption.

5.2.1.1 DPU SUPPORT-DRIVEN CHANGES

The prebuilt DPUs made available by *Xilinx* support a wide range of neural network layers with a large set of parameter combinations. Furthermore, the suitability of both *DPUCZDX8G* and *DPUCVDX8G* for supporting convolutional architectures was evidenced by the previous experiment.

Although a convolutional architecture, *SqueezeSegV3* features layers that are not supported by the targeted DPUs. For this reason, it was necessary to perform architectural changes in the network to make the model deployable to the targeted hardware. A detailed list of all the supported operations of *DPUCZDX8G* and *DPUCVDX8G* for *Vitis-AI* version 2.0, as well as the corresponding *Pytorch* layers, can be consulted in Table 27 and Table 28 of Appendix III.

The first architectural change was the substitution of the *torch.nn.functional.unfold* layer by a 2D convolution layer. Because the unfold layer (commonly known as im2col) is lightweight when compared to a 2D convolution layer, the kernel size used was 1. Another important aspect to consider is that the convolution layer is a trainable layer, meaning that it contains parameters. Consequently, unlike the unfold operation, the 2D convolution layer also increases the model memory footprint during training and most importantly during inference. To reduce the computation overhead of this replacement, the kernel size chosen was also 1. Following, all occurrences of the sigmoid activation function had to be replaced by hard sigmoid activations. An advantage of using the hard sigmoid is that it can be computed more efficiently than the regular sigmoid since, by being a composition of linear functions, it avoids the calculation of the exponent. Figure 49 shows a plot of both activations.

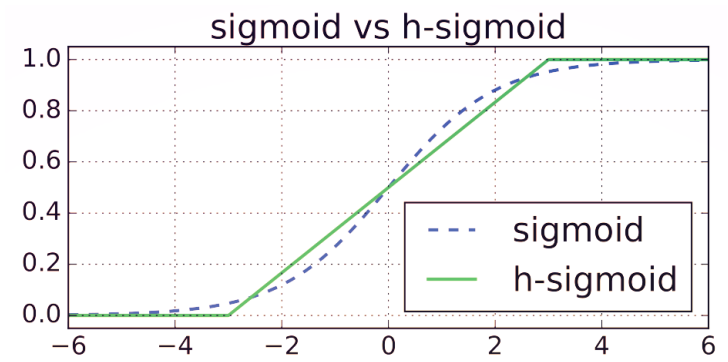

Figure 49. Sigmoid and hard-sigmoid activation functions.

The last change performed contemplated the *torch.nn.functional.upsample_bilinear* layer. This layer was responsible for reducing the width of the feature map by half. Again, the solution consisted in the usage of a 2D convolution layer, using double the stride in the width dimension. Because the number of input and output channels is 3 (x, y, z features), this convolution operation is very inexpensive and so the replacement presents no overhead in performance. In fact, the 3 instances of this *Conv2D* are the 3 least expensive layers by average time, with averages of 0.26 ms, 0.16 ms, and 0.12 ms. The complete list of layers and respective average times is present in Table 33 of Appendix VI. Table 33 contains alterations that are described in the next section. Similarly to the first 2D convolution layer, a kernel size of 1 was also adopted. Table 15 summarizes all the DPU support-driven architectural changes.

Table 15. SqueezeSegV3 support-driven architectural changes.

Original layer	**Parameters**	**Replacement layer**	**Parameters**
Unfold	Kernel size = 3	Conv2D	In channels = variable
	Padding = 1		Out channels = variable
			Kernel size = 1
			Padding = 0
Sigmoid	-	Hard sigmoid	-
Upsample_bilinear	Size = [cm_h, cm_w//2] †	Conv2D	In channels = 3
			Out channels = 3
			Kernel size = 1
			Stride = (1, 2)
			Padding = 0

*cm_[h/w] = coordinate map height/width

† // operator is floor division

5.2.1.2 PERFORMANCE-DRIVEN CHANGES

Table 16 lists the 4 most time-consuming layers of the model based on the average time it takes to compute each layer during inference. The results refer to an application running on the *VCK190 C64B1-2CU* configuration using 1 CPU thread, similar to the previous chapter. Again, Table 33 of Appendix VI contains the complete list of layers and respective average times.

All the layers in Table 16 are similar convolution layers that occur in the backbone, more specifically in the SAC Blocks of the encoder. They correspond to nearly a third of all the computation time during inference. Hence, these are all good candidate layers to be focused on in order to extract better performance. It is also relevant to note that, when considering the number of occurrences of each of the 4 layers, the total size of parameters is 7.8 MB, corresponding to almost 18% of the model size. This further increases the relevance of these layers.

Table 16. Top 4 most time-consuming layers during inference.

Location	Layer	Occurrences	Parameters	Parameter Size (MB)	Average Inference Time (ms)
Backbone/Encoder3/SACBlock	Conv2D	2	In channels = 3 Out channels = 1152 Kernel size = 7 Padding = 3	1.30	10.57
Backbone/Encoder2/SACBlock	Conv2D	1	In channels = 3 Out channels = 576 Kernel size = 7 Padding = 3	1.30	9.50
Backbone/Encoder5/SACBlock	Conv2D	1	In channels = 3 Out channels = 2304 Kernel size = 7 Padding = 3	1.30	8.63
Backbone/Encoder4/SACBlock	Conv2D	2	In channels = 3 Out channels = 2304 Kernel size = 7 Padding = 3	1.30	8.63

Table 17 shows all 3 variants of the abovementioned 4 *Conv2D* layers with their respective parameter and FLOPS count. Note that every SAC block contains one instance of this *Conv2D*. The data corresponds

to the 2D convolution layers of the SAC block of encoder 5. As can be observed in the table, there is a potential to reduce the model size by decreasing the listed convolution layers by 80% and 97% respectively using a kernel size of 3 and 1. The reduction in GFLOPS is 82% and 98% respectively.

All three variations were explored during this experiment.

Table 17. SAC block's convolution kernel size comparison.

Input Size (N, C, H, W)	In Channels	Out Channels	Kernel Size	Padding	# Parameters	Parameter Size (MB)	GFLOPS
			7	3	341k	1.36	5.55
1 x 3 x 64 x 256	3	2304	3	1	64.5K	0.26	1.02
			1	0	9K	0.04	0.11

Besides reducing a single layer's computation cost and size, it is also possible to reduce the number of layers altogether. One of the explored approaches consisted in removing the encoders 4 and 5, which correspond to 67% of the original model's total number of floating-point operations.

Lastly, a small architectural change that can be realized only at inference time is the removal of all but the 5th prediction head. Since the other 4 additional heads are used to compute a multi-layer loss and their outputs are ignored at inference time, they can be removed to save memory and computation. This was applied to all variants of models explored in this experiment. Table 18. SqueezeSegV3-21 model variants experimented. summarizes all models considered for deployment during this experiment, the respective combinations of support-driven changes, model size, and GFLOPS. It also contains the *SqueezeSegV3* model from the original paper for comparison.

Table 18. SqueezeSegV3-21 model variants experimented.

Model designation	SACBlock Conv2d kernel	Encoders 4, 5	Model Size (MB)	GFLOPS
Original	N/A	Yes	36	198
SSGV321-K7	7	Yes	44	241
SSGV321-K3	3	Yes	40	209
SSGV321-K1	1	Yes	39	202
SSGV321-K1N45	1	No	17.6	77.14

5.2.2 FLOAT MODEL TRAINING

All experimented models were trained for 72 epochs. The SGD optimizer was used with an initial learning rate of 10^{-3}, a momentum of 0.9 and a weight decay of 10^{-4}. A learning rate warmup was also performed for 1 epoch with a learning rate decay of 0.995. Due to memory limitations, the batch size used was 2. Similarly to the *SqueezeSegV3*, the loss function used was a multi-layer cross-entropy loss. Each of the five decoders of the model had a prediction head consisting of a dropout layer and a 2D convolution layer. The outputs of these prediction heads were used as multiple outputs of the network to calculate the loss. The authors of *SqueezeSegV3* defend that these "intermediate supervisions" guided the model to form features with more semantic meaning and helped mitigate vanishing gradients. The loss is described by the following equation:

$$Loss = \sum_{i=1}^{5} \frac{-\sum_{H_i,W_i} \sum_{c=1}^{C} w_c \cdot y_c \cdot log(\hat{y}_c)}{H_i \times W_i} \qquad \text{(Equation 9)}$$

where $w_c = \frac{1}{log(f_c+\varepsilon)}$ is a normalization factor, f_c represents the frequency of class c. H_i, W_i are the height and width of the output of the i-th prediction head, y_c is the prediction for the c-th class in each pixel and $\hat{y}_c$ is the corresponding label.

The training plots of the original and *SSGV321-K3* models are depicted in figures Figure 50 through Figure 53. Figure 50 shows the accuracy and IoU plots of the original paper's implementation of the SqueezeSegV3-21 model. Both the accuracy and IoU are evaluated in the train and validation sets.

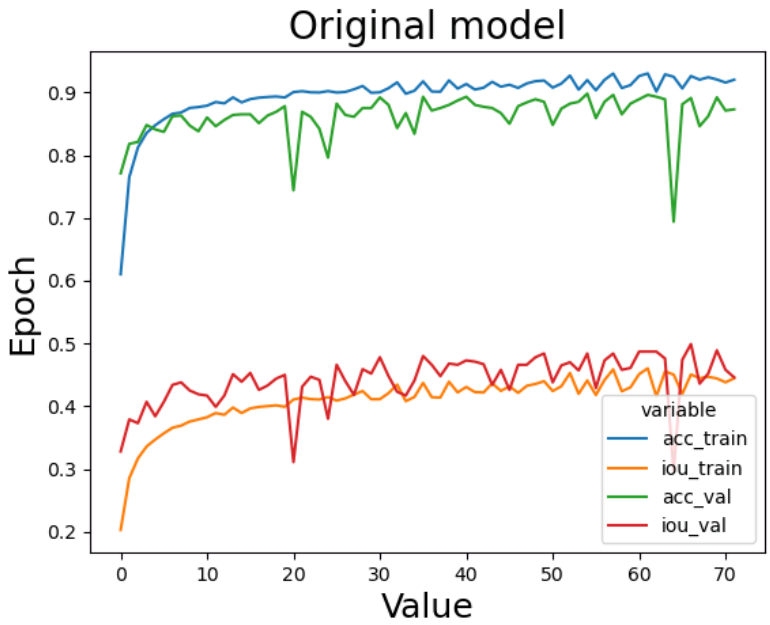

Figure 50. SqueezeSegV3-21 original model training: validation accuracies and IoUs.

Figure 51 shows the training loss of the original SqueezeSegV3-21 model.

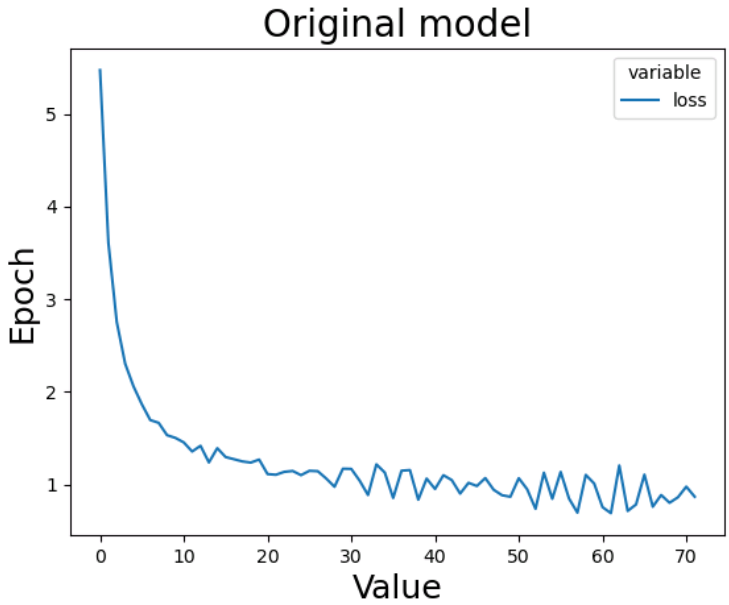

Figure 51. SqueezeSegV3-21 original model training: training set loss.

The next two figures are similar plots but for the *SSGV321-K3* model.

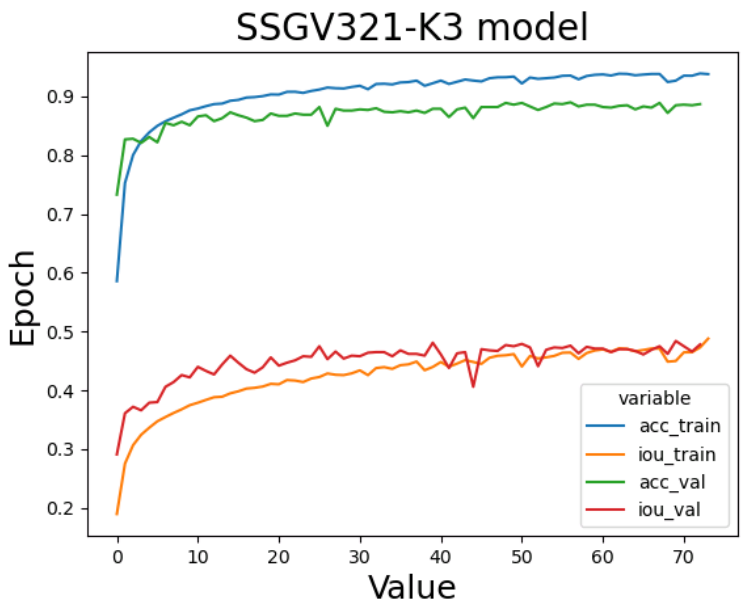

Figure 52. SSGV321-K3 model training: validation accuracies and IoUs.

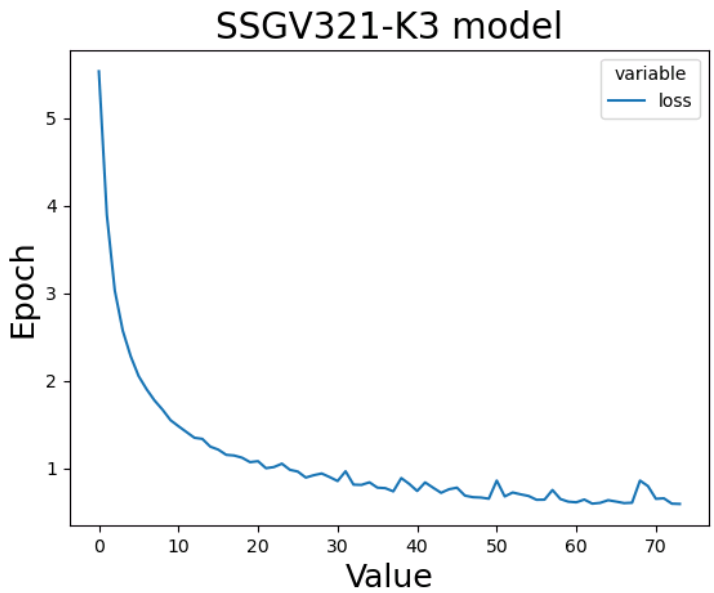

Figure 53. SSGV321-K3 model training: training set loss.

Similar training plots of the *SSGV321-K1* and *SSGV321-K1N45* models are available in Figure 66 through Figure 69 of Appendix VI.

One can see that, although models were trained for 72 epochs, there could still be room for improvement since the accuracy and IoU metrics seem to still be improving at the last epochs. From the results advertised by the original *SqueezeSegV3* paper, it is known that this is the case at least for the original model. To validate this hypothesis for *SSGV321-K3*, the model was also trained for 100 epochs.

The choice of training the *SSGV321-K3* model over *SSGV321-K7* becomes obvious later when the framerate results are presented. Figure 54 and Figure 55 show the accuracy and IoU as well as the training loss of the *SSGV321-K3* model trained for 100 epochs.

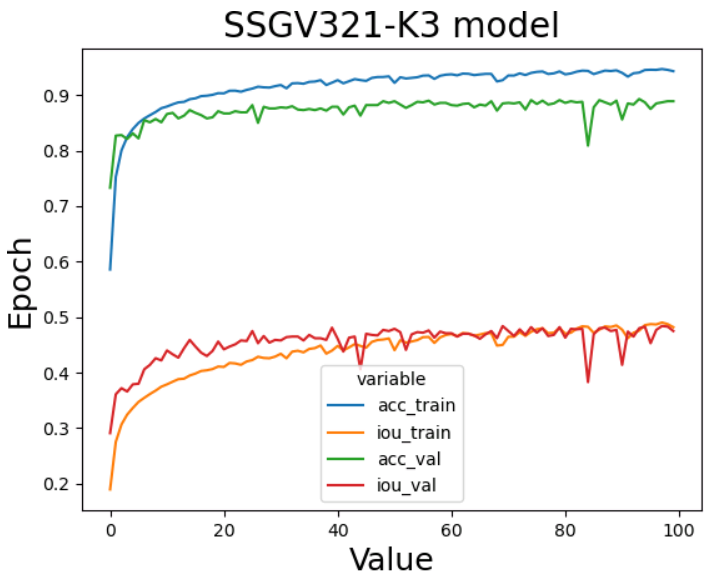

Figure 54. SSGV321-K3 model training: validation accuracies and IoUs (100 epochs training).

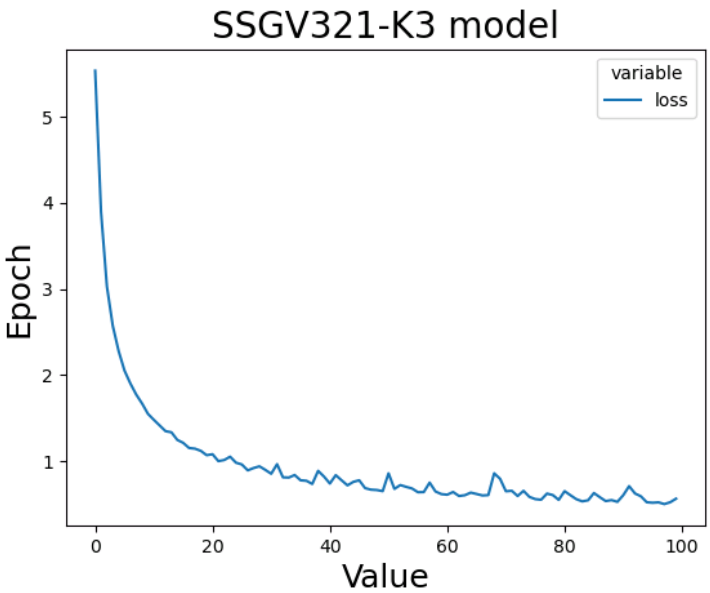

Figure 55. SSGV321-K3 model training: training set loss (100 epochs training).

As can be noted, there is a very slight improvement in the accuracy and IoU. However, for the rest of the experiment, and for comparison reasons, all models will be quantized using the float models trained for 72 epochs. The main reason is the lack of sufficient resources to train all models for 100 epochs. Nevertheless, this is still an interesting result that opens the possibility to improve the final results of this

work in terms of accuracy and IoU. For these reasons, the detailed results and comparison of the accuracy and IoU of the SSGV321-K3 model trained for 100 epochs with the same model trained for 72 epochs can be consulted in tables Table 34 and Table 35 of Appendix VI.

5.2.3 MODEL QUANTIZATION

The previous experiment compared all three available quantization methods in terms of model accuracy. The results of 4.3.1 allowed to conclude that QAT is the superior method with no drawbacks in terms of model size. However, it is also the costliest of all the methods. Unlike *ResNet-18* and *SqueezeNet* models using the *CIFAR-10* dataset, the training process of the *SqueezeSegV3* model using the *Semantic-KITTI* dataset was extremely time-consuming on the available hardware. For the above reasons, QAT was not performed in this experiment.

5.2.4 DEPLOYMENT ON TARGET HARDWARE

All the experimented models were compiled for the *DPUCVDX8G*. Again, the models were sent via SCP to the target *Versal ACAP VCK190* board. The deployment of the *SqueezeSegV3* model was realized with the same single-threaded and multi-threaded application used to deploy the *ResNet-18* and *SqueezeNet* models. The same application architecture, used in the last experiment, was adopted in this experiment. Figure 28 illustrates the application architecture. This means that the batch size used was 1 and that single inference and pipelined inference were both available. The only change to the previous experiment was that pre and post-processing of the data were not contemplated in the application.

To further illustrate the whole implementation process, Figure 56 presents a flowchart with all the steps from architectural changes to the model deployment.

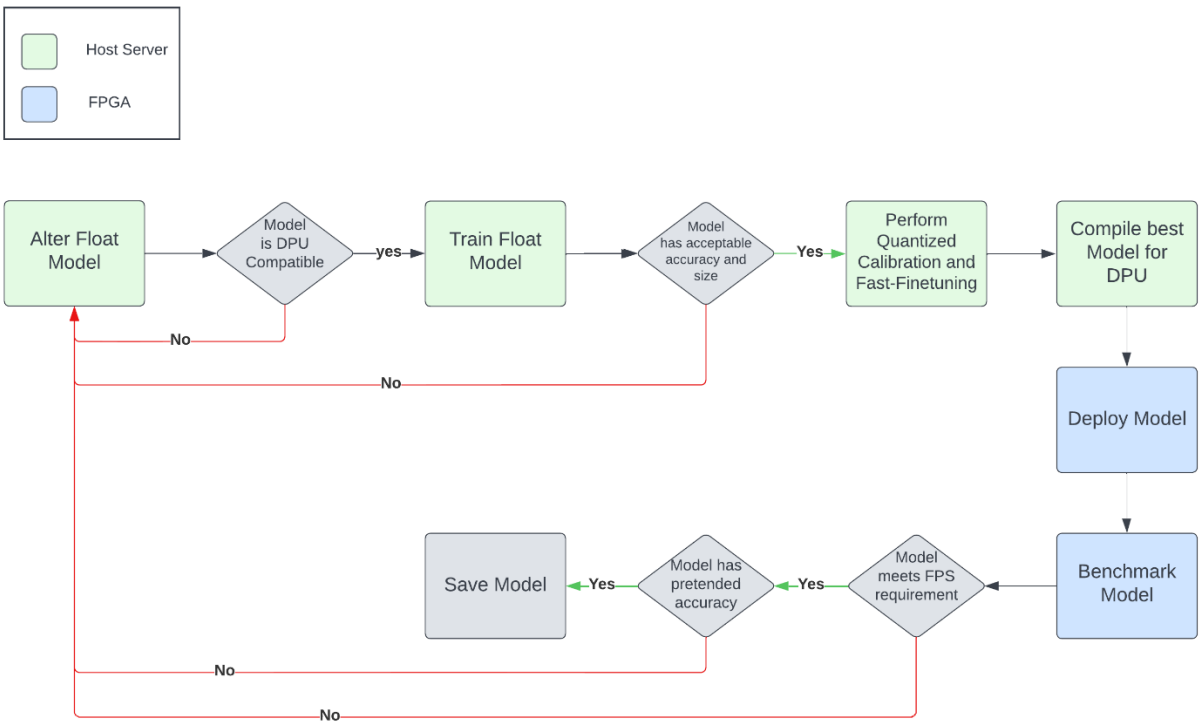

Figure 56. Model deployment flowchart.

5.3 RESULTS AND ANALYSIS

Again, this section is divided into two subsections. The first presents the results relative to the quantization of the *SqueezeSegV3-21* model variations. Accuracy, IoU, and model size are the 3 metrics considered. The second focuses on the performance and efficiency results of the targeted models during inference, assessing both the average inference FPS and the peak power consumption. For comparison, the *NVIDIA RTX 3090 GPU* is used.

5.3.1 QUANTIZATION

The quantization methods experimented were quantized calibration and fast finetuning. The average accuracy and average IoUs of all experiments are listed in Table 19. Per-class IoU of all the models can be consulted in Table 31 of Appendix VI. The "original" model was obtained from the original implementation of the *SqueezeSegV3* paper. Hence, it contains no modification related to DPU-support or performance. The results show an accuracy and IoU difference of 0.7 and 1.5 points respectively from the baseline model to the *SSGV321-K7* model. However, this loss does not correspond to accuracy since

it is related to the necessary DPU support-driven modifications. Quantization-wise, the results show that accuracy degradation is in the range of 2.8% to 4.4% when applying quantized calibration but falls to the 1% to 2.3% interval by leveraging fast-finetuning. Regarding IoU, a similar reduction is noticed. The intervals are of 1.4 to 3.1 and 1.1 to 2.6 points respectively.

Table 19. Quantization results of SqueezeSegV3-21 model variants.

Model	**Float model (72 epochs)**		**Quantized Calibration**		**Fast finetuning**	
	Avg Acc	**Avg IoU**	**Avg Acc**	**Avg IoU**	**Avg Acc**	**Avg IoU**
Original	0.870	0.460	N/A	N/A	N/A	N/A
SSGV321-K7	0.865	0.450	0.837	0.436	0.855	0.438
SSGV321-K3	0.862	0.439	0.834	0.421	0.851	0.428
SSGV321-K1	0.863	0.445	0.832	0.430	0.853	0.431
SSGV321-K7N45	0.859	0.427	0.815	0.396	0.836	0.401

*Model not supported by Vitis-AI v2.0

The reduction ratios of the models were within the 2.10 to 2.75 interval. The quantized models were subject to compilation and consequent compiler optimizations and other internal alterations that allow the model to be deployed on the *VCK190*. Hence, the sizes listed in Table 20 are subject to the file format of the compiled model, the ".xmodel" format.

Table 20. Model size reduction after quantization.

Model	**Model Size (MB)**		**Reduction Ratio**
	Float	**Quantized**	
SSGV321-K7	44	21	2.10
SSGV321-K3	40	14	2.86
SSGV321-K1	39	N/A	N/A
SSGV321-K1N45	17.6	6.4	2.75

5.3.2 PERFORMANCE AND EFFICIENCY

Similarly to chapter 4, the performance and efficiency results focused on the framerate and peak power consumption of the models. Both the framerate and peak power consumption measurement procedures are similar to the ones described in section 4.3.2.

Given the results from the last experiment, the thread number only varied in the $[1, 4]$ interval. Unfortunately, the *SSGV321-K1* model was not compatible with version 2.0 of *Vitis-AI*, and hence it was

discarded. The incompatibility is not trivial and is related to the internals of the compiler. Nevertheless, the variant without the encoders 4 and 5 was deployed without problems. The complete list of results for each model can be consulted in Table 32 of Appendix VI.

Figure 57 shows the FPS of the *SSGV321-K7* model during inference. Similarly to the results in chapter 4, the FPS improved with the number of threads up to 2 and 4 threads respectively when using a single DPU core and 2 DPU core configuration. This trend extends across all configurations and models. Albeit very close, the *SSGV321-K7* model did not achieve the desired 10 FPS.

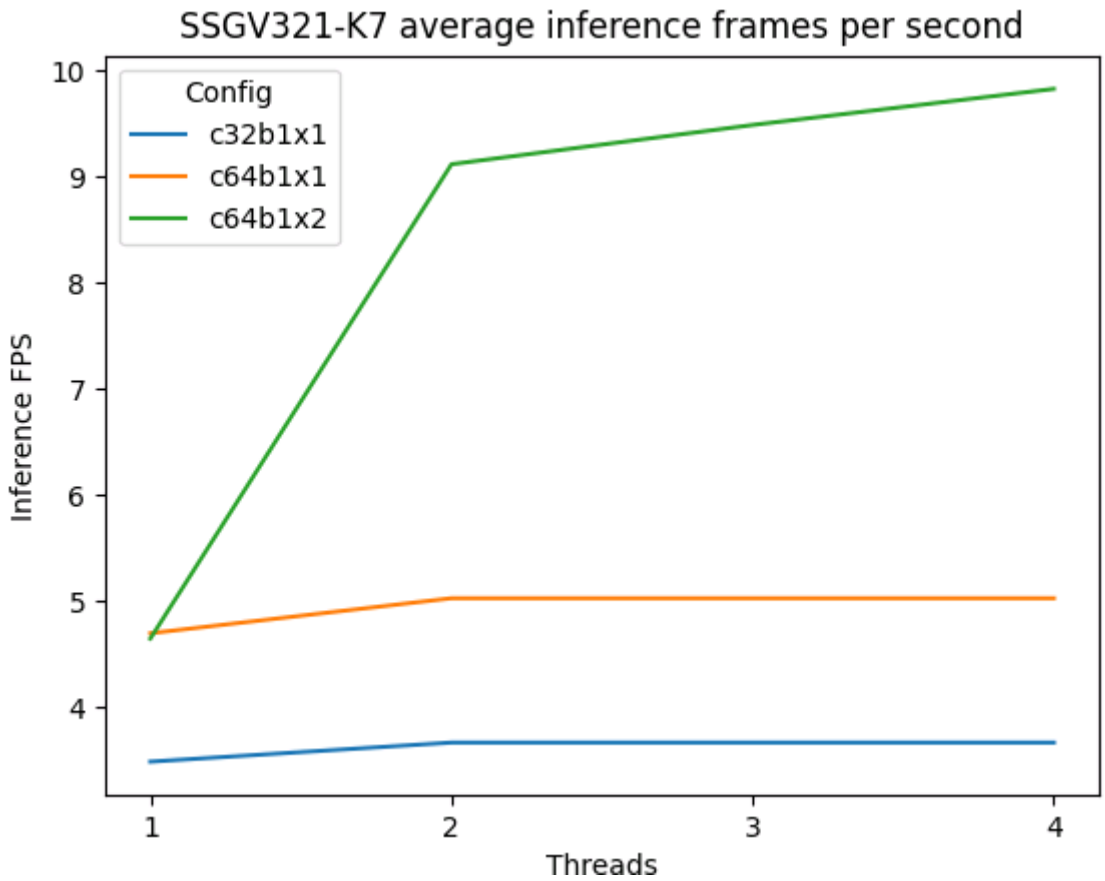

Figure 57. SSGV321-K7 average inference FPS on all VCK190 configurations.

Regarding threaded performance, the results for the *SSGV321-K3* model were similar to the previous model. However, this model did achieve a framerate of 11.17 when using the C64B1x2 configuration and 4 CPU threads (Figure 58).

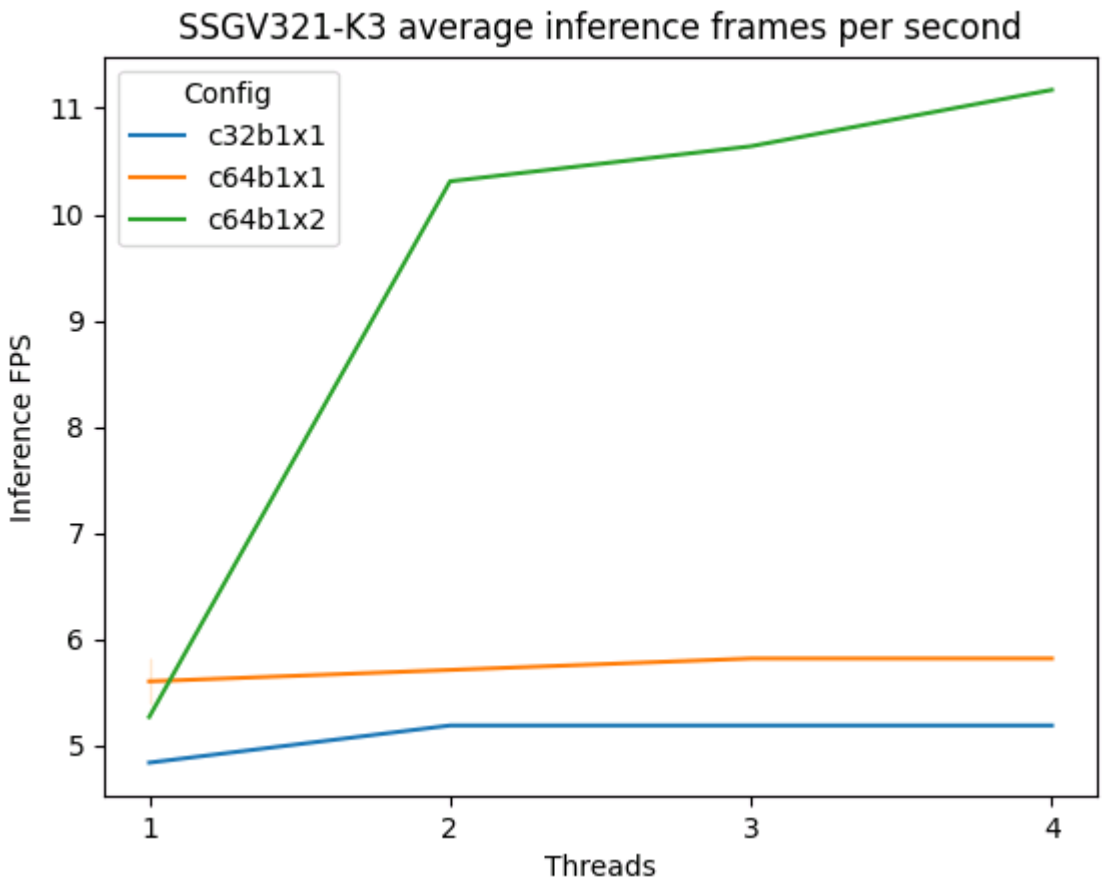

Figure 58. SSGV321-K3 average inference FPS on all VCK190 configurations.

The considerably smaller, and computationally less intensive, *SSGV321-K1N45* model exhibited the same patter when varying the number of threads. However, this time, the performance was substantially better, with a maximum of 18.92 FPS, as can be observed in Figure 59.

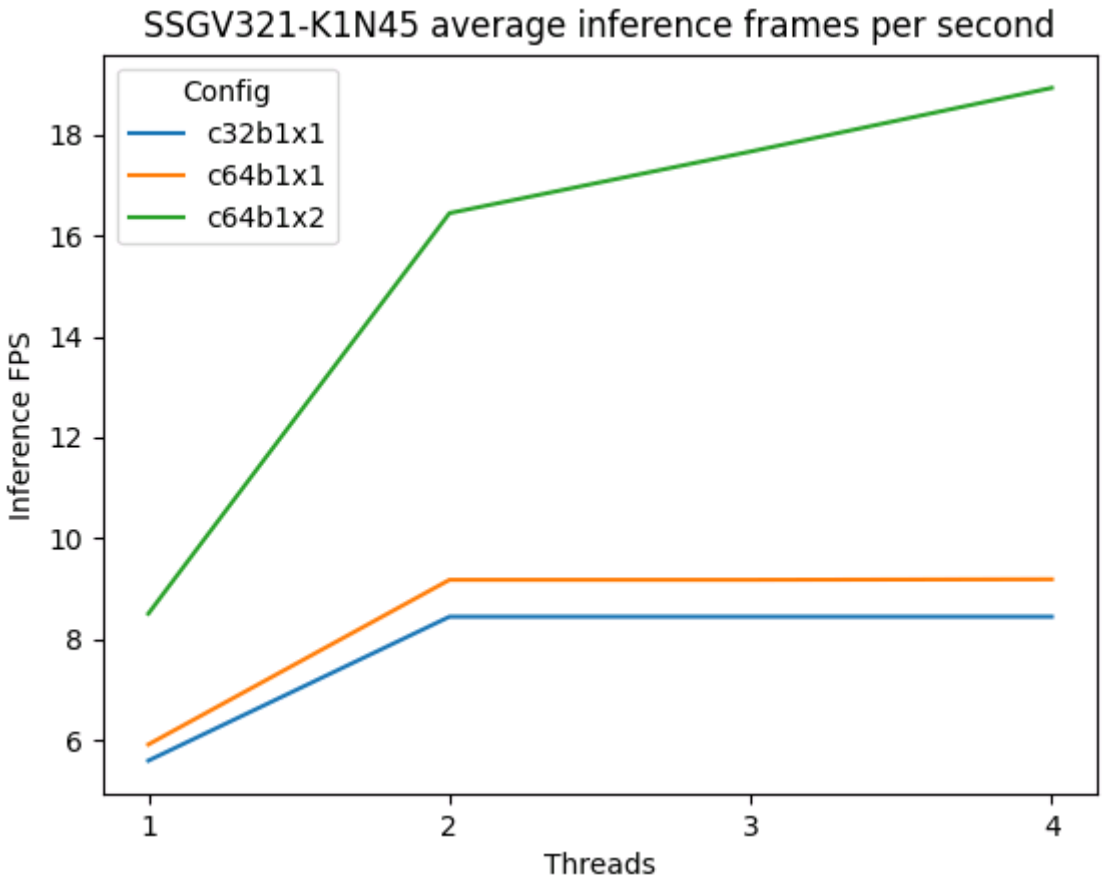

Figure 59. SSGV321-K1N45 average inference FPS on all VCK190 configurations.

FPS values were also substantially bigger in the GPU when compared with the C64B1x2 configuration across all 3 models. The results are listed below in Table 21. SSGV3-21 models framerate comparison between RTX3090 and C64B1x2.. GFLOPS and GOP/s are also included. Note that the GOP/s refers to the number of giga-operations each second computed in the FPGA and should not be confused with the GFLOPS of the float models, since operations on the FPGA are operating on quantized values. Also, the computation of each layer differs from the GPU to the FPGA and so the GOPS estimation depends on the hardware. The GOP/s values are extracted directly from *Vitis-AI profiler*.

Table 21. SSGV3-21 models framerate comparison between RTX3090 and C64B1x2.

Model	**FPS**	
	RTX 3090	**C64B1x2 - 4 CPU threads**
SSGV321-K7	19.93	9.83
SSGV321-K3	21.91	11.17
SSGV321-K1N45	36.50	18.92

The power consumption of the models was also proportional to their size, as expected. It was also proportional to the size of the configuration and the number of CPU threads. Figure 60 shows the peak power consumption of the *SSGV321-K7* model on the 3 targeted *VCK190* configurations.

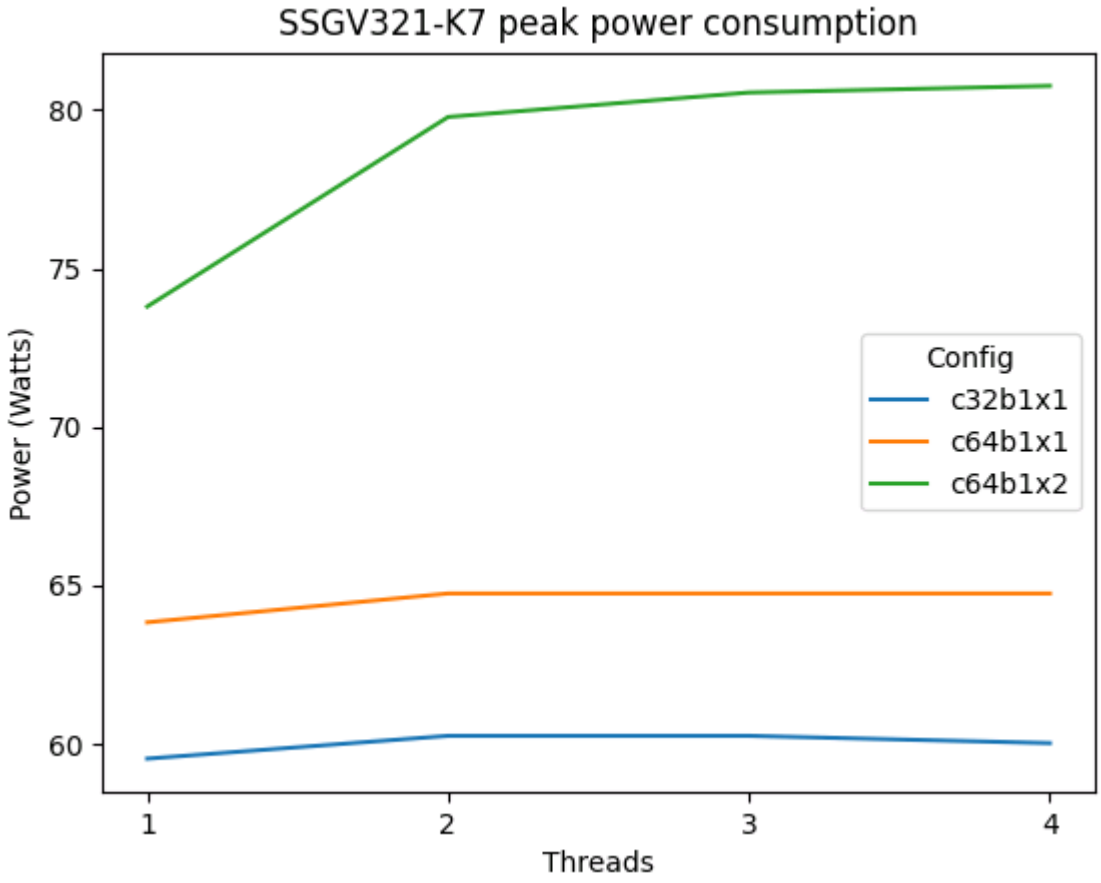

Figure 60. SSGV321-K7 peak power consumption on all VCK190 configurations.

Table 22 shows the power consumption of some configurations, namely the configurations that surpassed the 10 FPS performance. The smaller *SSGV321-K1N45* model consumed approximately less 2.5 Watts than the *SSGV321-K3* model, a negligible amount. Also, the peak power consumption did not increase substantially from 2 to 4 threads.

Table 22. C64B1x2 peak power consumption on all SqueezeSegV3-21 model variants.

Model	Configuration	# Threads	Peak Power Consumption (Watts)
SSGV321-K3	C64B1x2	2	78.08
SSGV321-K3	C64B1x2	4	78.30
SSGV321-K1N45	C64B1x2	2	75.57
SSGV321-K1N45	C64B1x2	4	76.15

Comparatively, the power consumption of the *RTX 3090 GPU* for the *SSGV321-K3* and *SSGV321-K1N45* models was 365.17 Watts and 228.13 Watts respectively.

The performance per watt of each of the 3 models for the set of the 7 most representative configurations, as well as in the *RTX 3090 GPU*, is present in Figure 61. The results show that the overall performance per Watt slightly increased from the *SSGV321-K7* to the *SSGV321-K3* model. A more noticeable increase was noted in the *SSGV321-K1N45* model with an approximately 1-7x to 1.8x increase in the C64B1x2 configuration compared with its single DPU core counterpart. The C64B1x2 configuration was the most efficient across all models. Nevertheless, the performance per Watt values are 2 orders of magnitude smaller than the values observed in the *ResNet-18* model of the previous chapter.

Regarding the *RTX 3090* performance, all VCK190 configurations achieved better efficiency except for the smaller model, where only the C64B1x2 configuration was superior when using 2 and 4 CPU threads.

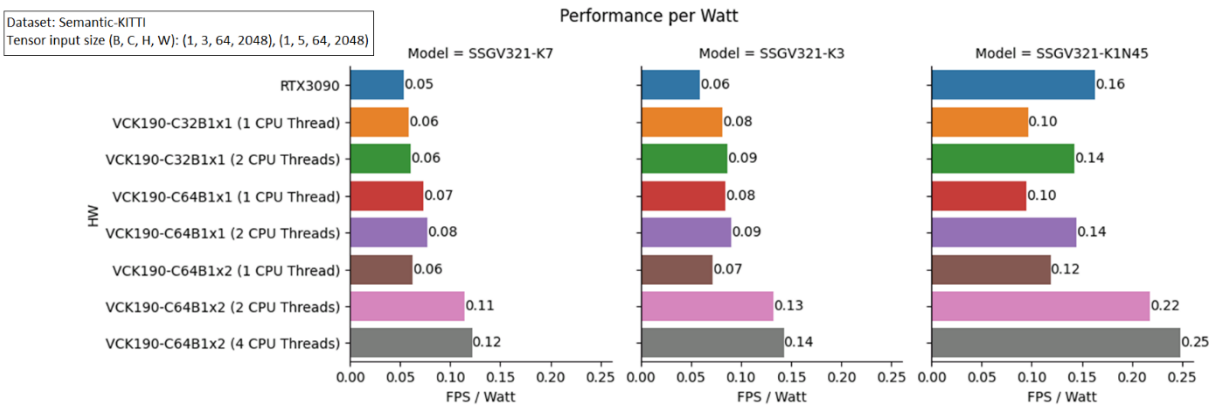

Figure 61. Performance per Watt of RTX3090 and VCK190.

5.3.3 QUALITATIVE

Qualitatively, it is possible to gain additional insights into the accuracy of the models. Figure 62 compares the semantic segmented point clouds of different models. The top left point cloud corresponds to the LiDAR point cloud labeled with the ground truth. On its right, the labeled point cloud corresponds

to the predictions of the original *SqueezeSegV3* paper model. The *SSGV321-K3* and *SSGV321-K1N45* predictions result from running the models on the *VCK190*. The *SSGV321-K7* model, with better results, was not considered because it did not achieve the necessary framerate to sustain the data rate from a 10 Hz LiDAR. From this figure, one can see that the labeling happens across all 360° horizontally. Some artifacts can be noted near the vegetation area in all the predictions, including the original paper model. The mislabeling appears to be more severe in the *SSGV321-K1N45* model (big red area), as expected because of the lower accuracy and IoU.

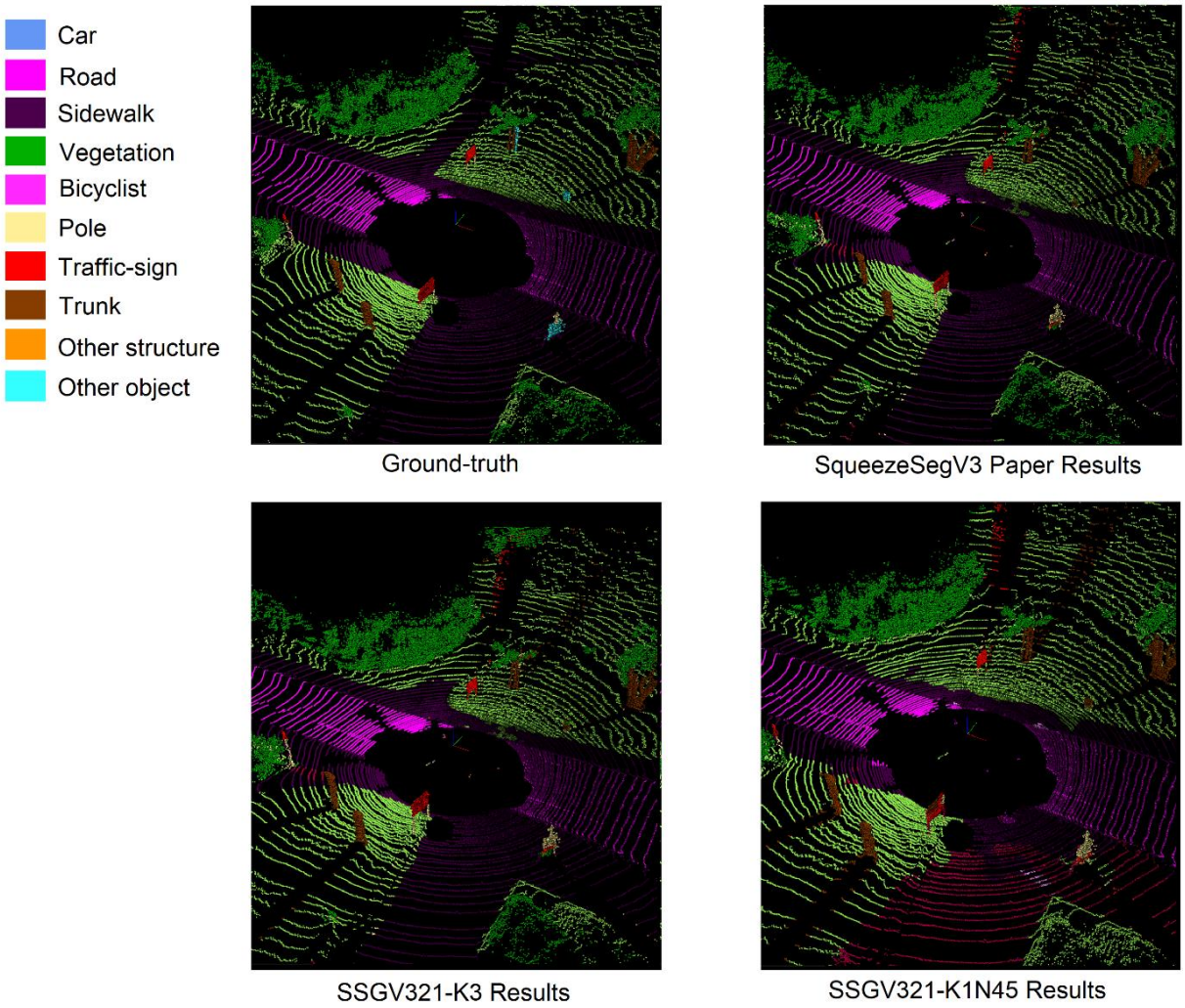

Figure 62. Semantic-KITTI semantic segmented point clouds. Ground-truth and predictions comparison.

Figure 63 allows closer inspection, as well as comparison with the corresponding camera image. The white bounding boxes delimit the zones where mislabeling happens. The number of mislabeling is higher in the *SSGV321-K1N45*, as expected. Nevertheless, all models mistake some areas. For example, some

trunks are mistaken for poles, the bicyclists' shadows are misrepresented as additional bicyclists, and a traffic sign is missed in all 3 models' predictions.

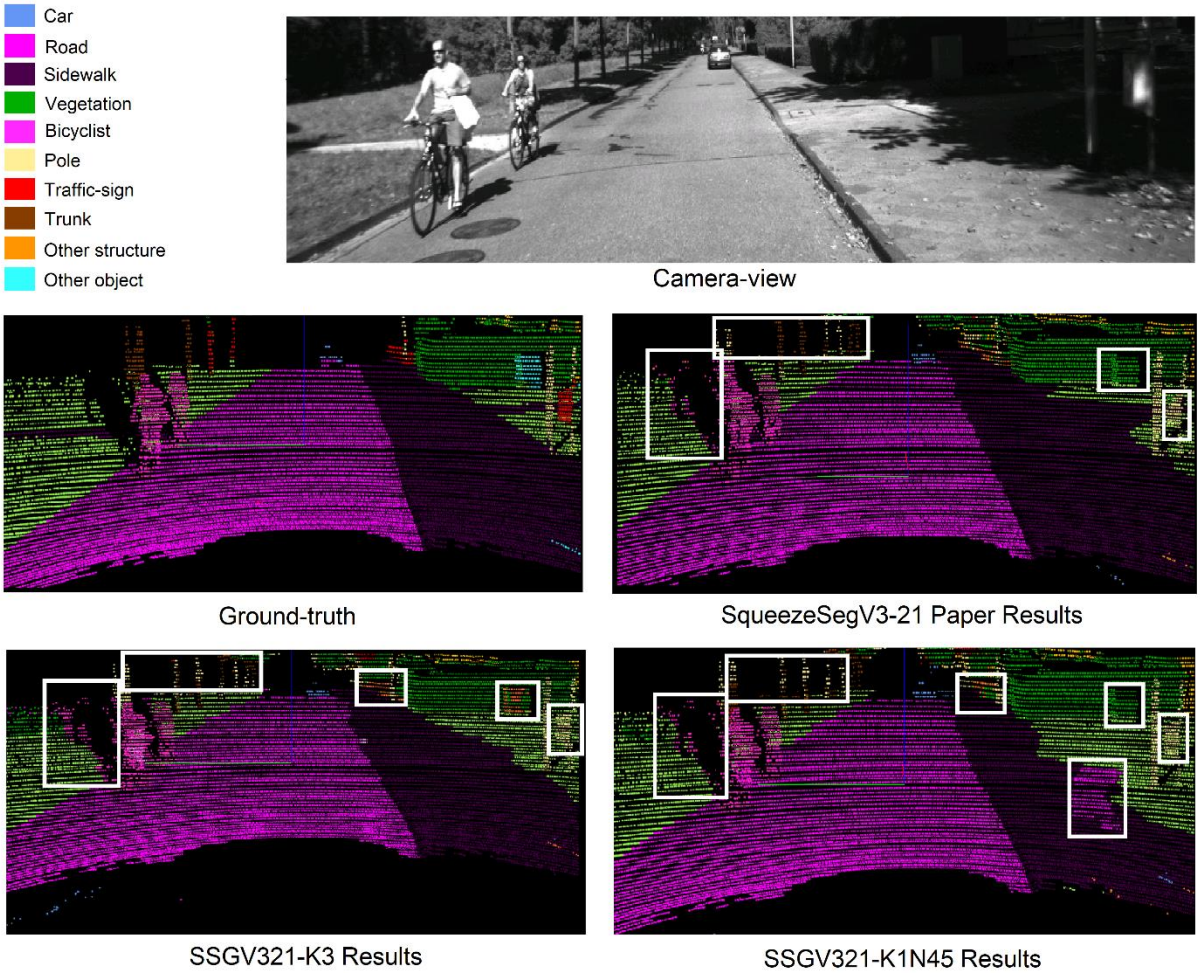

Figure 63. Detailed semantic segmented point clouds predictions. Comparison with ground-truth and camera-view.

5.4 DISCUSSION

Similarly to the previous chapter's discussion, here the aim is to interpret the findings and offer commentary on the previous section's results. More so, the contributions of this work are also put into perspective by being compared with similar works in the literature.

5.4.1 QUANTIZATION

The first important observation that can be made about the quantization results is that the fast-finetuning method outperformed quantized calibration. These results are consistent with the results in

the first experiment. It is also clear that the accuracy degradation increases slightly in comparison with the *ResNet-18* and *SqueezeNet* results. However, both the model and the dataset are vastly more complex. Nevertheless, leveraging the fast-finetuning method allows for accuracy degradations in the 1% to 2.3% interval. Although the literature on 3D computer vision models implemented on FPGAs is still in its infancy, it is possible to draw a comparison between two existing works. The *ChipNet* FPGA implementation exhibits degradation in the evaluation metrics (F1, Average Precision, Precision, Recall, FPR, and FNR) when quantizing to 12-bits. The quantization was not performed with a bit-width below 12. The degradation in average precision, for example, is 1.7%. All other metrics also suffered degradation due to quantization. The perception task being solved was also segmentation, but only of the drivable region. The *VoxelNet* implementation on FPGA also exhibits degradation in evaluation metrics such as F1 and average precision. The results show that for a 12-bit-width quantization, the average precision decreased by 5.8% and 9.05% in the F1 metric [126]. The other close work did not list the evaluation metrics degradation with quantization [21].

Regarding model size reduction, none of the above works mentions the ratio of reduction. Nevertheless, it is possible to draw a comparison with the results from the previous experiment. The model size reductions were smaller than the observed in chapter 4. The model size reduction ratios of *ResNet-18* and *SqueezeNet* were 4.07 and 3.33. These results already indicated that substantial differences in ratios were possible to occur with different models. The same happens with the *SSGV321* model variants. Nevertheless, it cannot be ignored that a reduction ratio equivalent to the theoretical value of 4 is not guaranteed for all models and is highly dependent on the model architecture and size. More so, even for very similar models, the reduction can differ substantially. Again, the difference in model size to the theoretically expected value could be further aggravated by the internal compiler optimizations. Also, quantization parameters are expected to also contribute to the size of the quantized models.

5.4.2 PERFORMANCE AND EFFICIENCY

The adopted approach of benchmarking different configurations showed once more the suitability of *Vitis-AI* to perform design space exploration. This was evidenced by the capability to perform changes to the *SSGV321* network layers, compile the model, and test the framerate and peak power consumption of the network on different target configurations and a variable number of CPU threads. Ultimately, this back-and-forth exploration for solutions resulted in 2 models that achieved the desired framerate of 10 Hz. The two models can be deployed to run in a real-time scenario for most of the LiDAR sensors in the market.

The performance per Watt of the application was once again superior in the *VCK190* when compared to the *RTX 3090 GPU*, on all the 3 models experimented with when considering the C64B1x2 configuration with 2 CPU threads. However, contrarily to chapter 4 where the FPS values were higher and the peak power consumption lower, this time the FPS values were lower in the *VCK190*. The superior performance per Watt is a consequence of the lower power consumption – 4.67x and 3.18x lower peak power consumption of the C64B1x2 configuration with 2CPU threads compared with the *RTX 3090*. There is not a single explanation for why the *RTX 3090* outperforms the *VCK190* in framerate since both memory accesses and computations of each layer can play a big role. Nevertheless, one of the possible factors might be related to the high quantity of data reads and stores in the DDR memory in the *VCK190*. Because the DDR memory is itself limited to 8 GB in the *VCK190*, main memory accesses might be a cause for the bottleneck. Evidently the same could be happening in the *RTX 3090 GPU*. However, the GPU contains a 24 GB DDR memory that withstands more of the weights, biases, and feature maps data reducing the access to the main memory. Table 23 summarizes the DDR memory accesses of both the *SGV321-K3* and *SGV321-K1N45* on the *VCK190*. LdWB, LdFM, Total StFM, and Avg Bw correspond respectively to the load size of feature maps, load size of weights and biases, store size of feature maps and the average bandwidth in the access to the DDR memory. Indeed, the amount of memory accesses is vastly superior in these 2 models when compared with the *ResNet-18* and *SqueezeNet* models from the last experiment.

Table 23. SSGV321-K3 and SSGV321-K1N45 inference DDR memory access information on VCK190.

Model	Total LdWB (MB)	Total LdFM (MB)	Total StFM (MB)	Avg Bw (MB/s)
SSGV321-K3	10.75	975.35	914.78	11122.57
SSGV321-K1N45	4.36	614.87	566.78	11062.45

5.4.3 QUALITATIVE

The visualizations show that there are indeed noticeable artifacts in the point clouds related to miss classification of points. This is visible across all models' predictions and is more noticeable for models with the worse accuracies and IoU. Nevertheless, one can confirm that the networks indeed learned to correctly classify harder classes such as cars, bicyclists, and traffic signs and not only the more abundant road, sidewalk, and vegetation. It also becomes clear that there is still room for improvement in the 3D computer vision model architectures that tackle semantic segmentation in LiDAR point clouds. However,

it is also important to remember that there are other models besides *SqueezeSegV3* that, although not currently supported in *Vitis-AI* (see 5.1.5.1), achieve better accuracy and IoU scores.

The visualizations show that the deployed models perform real-time semantic segmentation of the complete point-clouds. This is an important but often missed aspect of similar works. Table 24 lists similar works that focus on the FPGA implementation of 3D computer vision models using LiDAR point clouds. As evidenced by the below data, the other similar works identified either do not use all the point cloud data, do not implement the complete network in the FPGA, or use smaller frames. Having all 3 of these aspects and still achieving real-time performance is a strength of the achieved results.

Table 24. Comparison with similar works.

Model	**Complete Point Cloud Usage**	**# Points per Frame**	**Complete Network in FPGA**
ChipNet	×	Not specified	✓
VoxelNet	✓*	120k-125k	×
PointNet	✓	4k	✓
This work	✓	120k-125k	✓

*Depends on the class being detected

6 Conclusions

6.1 SYNOPSYS

This work proposed a hardware-software co-design approach for the deployment of the LiDAR-based 3D deep neural network, *SqueezeSegV3*, on *Xilinx's Versal ACAP VCK190*. Leveraging *Vitis-AI*, an inferencing application was developed allowing a real-time performance whose framerate surpassed 10 Hz, enabling the application to withstand the data rate of most commercially available LiDAR sensors, while solving the semantic segmentation task on the complete 360° point clouds of the *Semantic-KITTI* dataset. To achieve this, the first step consisted of the validation of the suitability of *Vitis-AI* for developing an inference application on an FPGA. A thorough exploration of *Vitis-AI* was first conducted to evaluate the capabilities of the tools available from float model quantization to the final deployment. To do so, the development of a multi-threaded application with real-time performance for the inference of both *ResNet-18* and *SqueezeNet* was executed. Then, a meticulous benchmarking of the developed application, focusing on framerate and power consumption, allowed to identify the possible accuracy/efficiency trade-off opportunities. The developed application was tested on FPGAs, namely the *Zynq Ultrascale+ MPSoC ZCU104* and the *Versal ACAP VCK190*. The results of this first experiment showed that the quantization tools allowed for significant model size reduction with little to no accuracy degradation. Regarding performance, the deployed application achieved real-time performance, with framerates in the thousands, while consuming very low power consumption. When comparing performance per Watt with the *RTX 3090*, both FPGAs produced overwhelmingly better results. These results proved enough to prove the suitability of *Vitis-AI*. Consequently, all the validated tools and the application developed in the first experiment were used very similarly to implement *SqueezeSegV3* on the *Versal ACAP VCK190*.

6.2 MAIN CONTRIBUTIONS

In no particular order, the main contributions of this work were:

- **Complete implementation of a LiDAR-based neural network implementation on an FPGA.** Some works only implement part of the layers of a neural network in the FPGA. The complete implementation allows accelerating an inference application completely on the FPGA avoiding data transfers that introduce latency and power consumption. Also, FPGA implementations of deep neural networks are still a relatively small, but very promising, research area, especially with 3D computer vison models.

- **Inference over large LiDAR point clouds with a 360° field of view in real-time.** LiDAR point clouds with higher resolutions result in better results but are also bigger. ADAS have to be capable of computing inference over such point clouds in real-time. The developed application serves as a proof of concept for such use cases and is capable of supporting all the 10 Hz LiDAR sensors.
- **Usage of the Versal ACAP VCK190.** The *VCK190* is underexplored in the literature. To the best of this thesis author's knowledge, this is the first implementation of a 3D computer vision model on the *VCK190* (except for *Xilinx*'s own *Model Zoo*) that is publicly available. In fact, very few works target this FPGA. For this reason, the results obtained serve as proof of the capabilities of the *VCK190* for the deployment of such 3D computer vision models and as a starting point of comparison of future similar works.
- **Validation of the Vitis-AI tool.** The successful real-time implementations of *ResNet-18, SqueezeNet,* and *SqueezeSegV3* on the *Zynq Ultrascale+ MPSoC ZCU104* and *Versal ACAP VCK190* used the entire toolset of *Vitis-AI.* Here, the main contribution was two-fold. Firstly it was shown that the Vitis-AI tools are successful in allowing the processes of quantization, profiling, and deployment of deep learning models on FPGAs. The deployment and profiling of the developed inference application on the two FPGAs required no FPGA expertise and produced the expected results. Secondly, the tools were shown to be useful in allowing a detailed exploration of a broad design space through the experimentation of the hardware configurations. From this design space exploration resulted different real-time capable models that sit along different points in the accuracy/efficiency trade-off.
- **Comparison with the RTX 3090 GPU.** The performance per Watt of the deployed models was compared with the *RTX 3090*. This comparison allowed to frame the results in the broader applied deep learning literature, which usually focuses on GPU implementations, further highlighting the potential of exploring different hardware solutions.

6.3 RESEARCH OPPORTUNITIES

Despite requiring hardware expertise, *Vitis-AI* allows for the design of customized DPUs. This would solve the main limitation of the Vitis-AI tool encountered in this work. Because only a limited number of instructions are natively supported by each DPU, it is not possible to deploy some models completely. This results in a heavy dependence of the developer on the supported operations, hampering the possibility to explore new more exotic operations that constitute the base of most state-of-the-art

architectures that outperform *SqueezeSegV3*. Custom DPUs would, in theory, solve this dependence on the supported operations of the prebuilt available DPUs used in this work.

Another interesting direction that would provide more insights into the deep neural network's performance would be to more closely and thoroughly examine the memory hierarchy usage during the execution of the models. This would help answer questions regarding specific layer bottlenecks that have been identified in this work but not completely explained. FPGA expertise would also be required. On the same note, it would also be interesting to quantify the inference latency-frame time resolution trade-off resulting from the developed application. Empirically the increase in throughput was very evident, but the penalty in single frame inference was not quantified. This would allow for an easier comparison of solutions with requirements defined by any developer or team that whishes to implement a similar application using Vitis-AI.

REFERENCES

[1] "Waymo Driver – Waymo." https://waymo.com/waymo-driver/ (accessed Dec. 13, 2021).

[2] "Yandex Self-Driving Cars." https://sdg.yandex.com/ (accessed Jul. 28, 2022).

[3] "Luminar to be Standardized on Next Generation Electric Volvo | Luminar Technologies, Inc." https://investors.luminartech.com/news-releases/news-release-details/luminar-be-standardized-next-generation-electric-volvo (accessed Jul. 28, 2022).

[4] "Sensor setup | a2d2.audi." https://www.a2d2.audi/a2d2/en/sensor-setup.html (accessed Sep. 04, 2022).

[5] "NVIDIA Drive Hyperion | NVIDIA Developer." https://developer.nvidia.com/drive/drive-hyperion (accessed Dec. 13, 2021).

[6] "Apollo | Robotaxi Autonomous Driving Solution," 2020. https://apollo.auto/robotaxi/index.html (accessed Dec. 14, 2021).

[7] "Autonomous Vehicle Technology | Driverless Cars | Cruise." https://getcruise.com/technology/ (accessed Sep. 04, 2022).

[8] "Autonomous Car Market Size to Grow Worth USD 11.03 Billion." https://www.globenewswire.com/en/news-release/2022/06/08/2458427/0/en/Autonomous-Car-Market-Size-to-Grow-Worth-USD-11-03-Billion-at-a-CAGR-of-31-3-for-2021-2029-Fortune-Business-Insights.html (accessed Sep. 04, 2022).

[9] "Global Autonomous Vehicles Market (2022-2030) - Projected." https://www.globenewswire.com/en/news-release/2022/06/01/2454154/28124/en/Global-Autonomous-Vehicles-Market-2022-2030-Projected-CAGR-of-53-6-During-the-Forecast-Period.html (accessed Sep. 04, 2022).

[10] "Projected sales of autonomous vehicles worldwide | Statista." https://www.statista.com/statistics/1230733/projected-sales-autonomous-vehicles-worldwide/ (accessed Sep. 04, 2022).

[11] J. Kocic, N. Jovicic, and V. Drndarevic, "Sensors and Sensor Fusion in Autonomous Vehicles," *2018 26th Telecommunications Forum, TELFOR 2018 - Proceedings*, 2018, doi: 10.1109/TELFOR.2018.8612054.

[12] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," *Adv Neural Inf Process Syst*, vol. 2020-December, May 2020, doi: 10.48550/arxiv.2005.14165.

[13] J. Yu *et al.*, "CoCa: Contrastive Captioners are Image-Text Foundation Models," May 2022, doi: 10.48550/arxiv.2205.01917.

[14] Q. Li, Q. Xiao, and Y. Liang, "Enabling high performance deep learning networks on embedded systems," *Proceedings IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, vol. 2017-January, pp. 8405–8410, Dec. 2017, doi: 10.1109/IECON.2017.8217476.

[15] E. H. C. Tourad and M. Eleuldj, "Survey of Deep Learning Neural Networks Implementation on FPGAs," *Proceedings of 2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications, CloudTech 2020*, Nov. 2020, doi: 10.1109/CLOUDTECH49835.2020.9365911.

[16] G. Lacey, G. W. Taylor, and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," Feb. 2016, doi: 10.48550/arxiv.1602.04283.

[17] N. Ghielmetti *et al.*, "Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml," May 2022, doi: 10.48550/arxiv.2205.07690.

[18] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo, "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA," *IEEE Trans Very Large Scale Integr VLSI Syst*, vol. 26, no. 7, pp. 1354–1367, Jul. 2018, doi: 10.1109/TVLSI.2018.2815603.

[19] J. Faraone, G. Gambardella, N. Fraser, M. Blott, P. Leong, and D. Boland, "Customizing low-precision deep neural networks for FPGAs," *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, pp. 97–100, Nov. 2018, doi: 10.1109/FPL.2018.00025.

[20] Y. Lyu, 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, Aug. 2018, doi: 10.1109/TCSI.2018.2881162.

[21] L. Bai, Y. Lyu, X. Xu, and X. Huang, "Pointnet on FPGA for real-time LiDAR point cloud processing," *Proceedings - IEEE International Symposium on Circuits and Systems*, vol. 2020-October, 2020, doi: 10.1109/iscas45731.2020.9180841.

[22] B. Li, T. Zhang, and T. Xia, "Vehicle Detection from 3D Lidar Using Fully Convolutional Network," *Robotics: Science and Systems*, vol. 12, Aug. 2016, doi: 10.15607/rss.2016.xii.042.

[23] M. Blott *et al.*, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Trans Reconfigurable Technol Syst*, vol. 11, no. 3, Sep. 2018, doi: 10.48550/arxiv.1809.04570.

[24] "GitHub - Xilinx/Vitis-AI: Vitis AI is Xilinx's development stack for AI inference on Xilinx hardware platforms, including both edge devices and Alveo cards." https://github.com/Xilinx/Vitis-AI (accessed Jul. 28, 2022).

[25] "Effective range and the high resolution advantage | Ouster." https://ouster.com/blog/effective-range-and-resolution/ (accessed Sep. 04, 2022).

[26] N. O. Mahony *et al.*, "Deep Learning vs. Traditional Computer Vision," vol. 943, Oct. 2019, doi: 10.1007/978-3-030-17795-9.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dec. 2015, doi: 10.1109/CVPR.2016.90.

[28] Z. Dai, H. Liu, Q. v. Le, and M. Tan, "CoAtNet: Marrying Convolution and Attention for All Data Sizes," Jun. 2021, Accessed: Dec. 14, 2021. [Online]. Available: https://arxiv.org/abs/2106.04803v2

[29] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection", Accessed: Dec. 14, 2021. [Online]. Available: https://github.com/AlexeyAB/darknet.

[30] H. Qiu, Y. Ma, Z. Li, S. Liu, and J. Sun, "BorderDet: Border Feature for Dense Object Detection," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12346 LNCS, pp. 549–564, Jul. 2020, doi: 10.1007/978-3-030-58452-8_32.

[31] M. Tan and Q. v. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 10691–10700, May 2019, Accessed: Dec. 14, 2021. [Online]. Available: https://arxiv.org/abs/1905.11946v5

[32] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *Int J Comput Vis*, vol. 115, no. 3, pp. 211–252, Sep. 2014, doi: 10.1007/s11263-015-0816-y.

[33] K. Minemura, H. Liau, A. Monrroy, and S. Kato, "LMNet: Real-time Multiclass Object Detection on CPU using 3D LiDAR," *Proceedings of 2018 3rd Asia-Pacific Conference on Intelligent Robot Systems, ACIRS 2018*, pp. 28–34, May 2018, doi: 10.1109/ACIRS.2018.8467245.

[34] M. Simon, S. Milz, K. Amende, and H.-M. Gross, "Complex-YOLO: Real-time 3D Object Detection on Point Clouds," Mar. 2018, Accessed: Dec. 14, 2021. [Online]. Available: https://arxiv.org/abs/1803.06199v2

[35] J. Beltrán, C. Guindel, F. M. Moreno, D. Cruzado, F. García, and A. de La Escalera, "BirdNet: a 3D Object Detection Framework from LiDAR information," *IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC*, vol. 2018-November, pp. 3517–3523, May 2018, doi: 10.1109/ITSC.2018.8569311.

[36] B. Yang, W. Luo, and R. Urtasun, "PIXOR: Real-time 3D Object Detection from Point Clouds," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 7652–7660, Dec. 2018, doi: 10.1109/CVPR.2018.00798.

[37] G. Zamanakos, L. Tsochatzidis, A. Amanatiadis, and I. Pratikakis, "A comprehensive survey of LIDAR-based 3D object detection methods with deep learning for autonomous driving," *Comput Graph*, vol. 99, pp. 153–181, Oct. 2021, doi: 10.1016/J.CAG.2021.07.003.

[38] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "PointPillars: Fast Encoders for Object Detection from Point Clouds," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 12689–12697, Dec. 2018, doi: 10.1109/CVPR.2019.01298.

[39] G. Riegler, A. O. Ulusoy, and A. Geiger, "OctNet: Learning Deep 3D Representations at High Resolutions," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 6620–6629, Nov. 2016, doi: 10.1109/CVPR.2017.701.

[40] S. A. Bello, S. Yu, C. Wang, J. M. Adam, and J. Li, "Review: deep learning on 3D point clouds," *Remote Sens (Basel)*, vol. 12, no. 11, Jan. 2020, doi: 10.3390/rs12111729.

[41] S. Shi *et al.*, "PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 10526–10535, Dec. 2019, doi: 10.1109/CVPR42600.2020.01054.

[42] J. Deng, S. Shi, P. Li, W. Zhou, Y. Zhang, and H. Li, "Voxel R-CNN: Towards High Performance Voxel-based 3D Object Detection," Dec. 2020, Accessed: Jan. 21, 2022. [Online]. Available: https://arxiv.org/abs/2012.15712v2

[43] Y. Li *et al.*, "Deep Learning for LiDAR Point Clouds in Autonomous Driving: A Review".

[44] R. Klokov and V. Lempitsky, "Escape from Cells: Deep Kd-Networks for the Recognition of 3D Point Cloud Models," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-October, pp. 863–872, Apr. 2017, doi: 10.1109/ICCV.2017.99.

[45] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic Graph CNN for Learning on Point Clouds," *ACM Trans Graph*, vol. 38, no. 5, p. 13, Jan. 2018, doi: 10.1145/3326362.

[46] C. Wang, B. Samari, and K. Siddiqi, "Local Spectral Graph Convolution for Point Set Feature Learning," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11208 LNCS, pp. 56–71, Sep. 2018, doi: 10.1007/978-3-030-01225-0_4.

[47] W. Han, C. Wen, C. Wang, X. Li, and Q. Li, "Point2Node: Correlation Learning of Dynamic-Node for Point Cloud Feature Modeling," *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence*, pp. 10925–10932, Dec. 2019, doi: 10.1609/aaai.v34i07.6725.

[48] W. Shi and R. Rajkumar, "Point-GNN: Graph Neural Network for 3D Object Detection in a Point Cloud," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1708–1716, Mar. 2020, doi: 10.1109/CVPR42600.2020.00178.

[49] I. Alonso, L. Riazuelo, L. Montesano, and A. C. Murillo, "3D-MiniNet: Learning a 2D Representation from Point Clouds for Fast and Efficient 3D LIDAR Semantic Segmentation," *IEEE Robot Autom Lett*, vol. 5, no. 4, pp. 5432–5439, Feb. 2020, doi: 10.48550/arxiv.2002.10893.

[50] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," *NIPS 2011*, 2011, Accessed: Dec. 15, 2021. [Online]. Available: https://research.google/pubs/pub37631/

[51] Q. Chen, C. Xin, C. Zou, X. Wang, and B. Wang, "A low bit-width parameter representation method for hardware-oriented convolution neural networks," *Proceedings of International Conference on ASIC*, vol. 2017-October, pp. 148–151, Jul. 2017, doi: 10.1109/ASICON.2017.8252433.

[52] C. Murphy and Y. Fu, "Xilinx All Programmable Devices: A Superior Platform for Compute-Intensive Systems (WP492)," 2017, Accessed: Dec. 15, 2021. [Online]. Available: www.xilinx.com1

[53] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation," Apr. 2020, doi: 10.48550/arxiv.2004.09602.

[54] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation," Apr. 2020, doi: 10.48550/arxiv.2004.09602.

[55] T. Gale, E. Elsen, and S. Hooker, "The State of Sparsity in Deep Neural Networks," Feb. 2019, doi: 10.48550/arxiv.1902.09574.

[56] S. Swaminathan, D. Garg, R. Kannan, and F. Andres, "Sparse low rank factorization for deep neural network compression," *Neurocomputing*, vol. 398, pp. 185–196, Jul. 2020, doi: 10.1016/J.NEUCOM.2020.02.035.

[57] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," Mar. 2015, doi: 10.48550/arxiv.1503.02531.

[58] I. Lang, A. Manor, and S. Avidan, "SampleNet: Differentiable Point Cloud Sampling," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 7575–7585, Dec. 2019, doi: 10.1109/CVPR42600.2020.00760.

[59] "The HDL-64E Lidar Sensor Retires | Velodyne Lidar." https://velodynelidar.com/blog/hdl-64e-lidar-sensor-retires/ (accessed Sep. 04, 2022).

[60] "Alpha Prime | Velodyne Lidar." https://velodynelidar.com/products/alpha-prime/ (accessed Dec. 13, 2021).

[61] "The Ibeo Lux LiDAR sensor - Homepage." https://www.ibeo-as.com/en/products/sensors/IbeoLUX (accessed Sep. 04, 2022).

[62] "OS2 Long-range lidar sensor for autonomous vehicles, trucking, and drones | Ouster." https://ouster.com/products/scanning-lidar/os2-sensor/ (accessed Sep. 04, 2022).

[63] "Continental Automotive - High Resolution 3D Flash LiDAR™." https://www.continental-automotive.com/en-gl/Passenger-Cars/Autonomous-Mobility/Enablers/Lidars/3D-Flash-Lidar (accessed Sep. 04, 2022).

[64] "Iris | Luminar (Nasdaq: LAZR)." https://www.luminartech.com/iris/ (accessed Sep. 04, 2022).

[65] "InnovizTwo | 2nd-Generation Automotive Lidar." https://innoviz.tech/innoviztwo (accessed Dec. 14, 2021).

[66] "InnovizOne | Automotive Lidar System." https://innoviz.tech/innoviz360 (accessed Sep. 04, 2022).

[67] "Spectrum HD25 - Baraja." https://www.baraja.com/en/spectrum-hd25 (accessed Sep. 04, 2022).

[68] "Continental Automotive - HRL131 Long Range LiDAR." https://www.continental-automotive.com/en-gl/Passenger-Cars/Autonomous-Mobility/Enablers/Lidars/HRL131 (accessed Sep. 04, 2022).

[69] "LiDAR sensor | Autonomous vehicle sensors | Valeo." https://www.valeo.com/en/valeo-scala-lidar/ (accessed Sep. 04, 2022).

[70] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges," *Proceedings of the IEEE*, 2019, doi: 10.1109/JPROC.2019.2915983.

[71] "Artificial Intelligence :: Omdia." https://omdia.tech.informa.com/topic-pages/artificial-intelligence (accessed Jul. 28, 2022).

[72] "OpenBLAS : An optimized BLAS library." https://www.openblas.net/ (accessed Jul. 28, 2022).

[73] "Intel oneAPI Math Kernel Library (oneMKL)." https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl.html (accessed Jul. 28, 2022).

[74] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, Jul. 2015, doi: 10.1145/2788396.

[75] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," *ACM International Conference Proceeding Series*, vol. 382, 2009, doi: 10.1145/1553374.1553486.

[76] J. Yin and X. Wang, "Measurement of machine learning performance with different condition and hyperparameter," 2020.

[77] "CUDA Toolkit Documentation." https://docs.nvidia.com/cuda/ (accessed Jul. 28, 2022).

[78] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Comput Sci Eng*, vol. 12, no. 3, pp. 66–72, May 2010, doi: 10.1109/MCSE.2010.69.

[79] P. Kharya, "TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x | NVIDIA Blog," May 14, 2020. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/ (accessed Dec. 21, 2021).

[80] N. Corporation, "NVIDIA A100 TENSOR CORE GPU Unprecedented Acceleration at Every Scale." 2020. Accessed: Dec. 21, 2021. [Online]. Available: www.nvidia.com/a100

[81] J. H. Gawron, G. A. Keoleian, R. D. de Kleine, T. J. Wallington, and H. C. Kim, "Life Cycle Assessment of Connected and Automated Vehicles: Sensing and Computing Subsystem and Vehicle Level Effects," *Environ Sci Technol*, vol. 52, no. 5, pp. 3249–3256, Mar. 2018, doi: 10.1021/ACS.EST.7B04576/SUPPL_FILE/ES7B04576_SI_001.PDF.

[82] S.-C. Lin *et al.*, "The Architectural Implications of Autonomous Driving: Constraints and Acceleration," *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 18, 2018, doi: 10.1145/3173162.

[83] "NVIDIA RTX Embedded GPU Solutions | NVIDIA." https://www.nvidia.com/en-us/design-visualization/resources/rtx-embedded/ (accessed Jul. 28, 2022).

[84] "Cloud Tensor Processing Units (TPUs) | Google Cloud." https://cloud.google.com/tpu/docs/tpus (accessed Dec. 21, 2021).

[85] F. N. Iandola, 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," Feb. 2016, Accessed: Dec. 21, 2021. [Online]. Available: https://arxiv.org/abs/1602.07360v4

[86] Y. Wang, G.-Y. Wei, D. Brooks, and J. A. Paulson, "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning," Jul. 2019, Accessed: Dec. 21, 2021. [Online]. Available: https://arxiv.org/abs/1907.10701v4

[87] "Artificial Intelligence & Autopilot | Tesla." https://www.tesla.com/AI (accessed Dec. 21, 2021).

[88] "EyeQ5 - Mobileye - WikiChip." https://en.wikichip.org/wiki/mobileye/eyeq/eyeq5 (accessed Dec. 21, 2021).

[89] A. Boutros, S. Yazdanshenas, and V. Betz, "You Cannot Improve What You Do not Measure," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, Dec. 2018, doi: 10.1145/3242898.

[90] "ASIC Prototyping with Stratix Series FPGAs | Intel." https://www.intel.com/content/www/us/en/programmable/products/general/fpga/stratix-fpgas/about/stx-asic-prototyping.html (accessed Dec. 21, 2021).

[91] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL™ deep learning accelerator on Arria 10," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 55–64, Feb. 2017, doi: 10.1145/3020078.3021738.

[92] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pp. 77–84, May 2017, doi: 10.1109/FPT.2016.7929192.

[93] "Quantization — PyTorch 1.12 documentation." https://pytorch.org/docs/stable/quantization.html (accessed Jul. 28, 2022).

[94] "TensorFlow Lite | ML for Mobile and Edge Devices." https://www.tensorflow.org/lite (accessed Jul. 28, 2022).

[95] "GitHub - pytorch/QNNPACK: Quantized Neural Network PACKage - mobile-optimized implementation of quantized neural network operators." https://github.com/pytorch/QNNPACK (accessed Jul. 28, 2022).

[96] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," *Low-Power Computer Vision*, pp. 291–326, Mar. 2021, doi: 10.48550/arxiv.2103.13630.

[97] S. R. Jain, A. Gural, M. Wu, and C. H. Dick, "Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks," Mar. 2019, doi: 10.48550/arxiv.1903.08066.

[98] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned Step Size Quantization," Feb. 2019, doi: 10.48550/arxiv.1902.08153.

[99] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks With Pruning, Trained Quantization And HUFFMAN Coding," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Oct. 2015, Accessed: Dec. 15, 2021. [Online]. Available: https://arxiv.org/abs/1510.00149v5

[100] M. Nagel *et al.*, "A White Paper on Neural Network Quantization," Jun. 2021, doi: 10.48550/arxiv.2106.08295.

[101] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, Dec. 2017, doi: 10.1109/CVPR.2018.00286.

[102] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization Through Weight Equalization and Bias Correction," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 1325–1334, Jun. 2019, doi: 10.48550/arxiv.1906.04721.

[103] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming," Jun. 2020, doi: 10.48550/arxiv.2006.10518.

[104] M. Courbariaux, Y. Bengio, and J. P. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *Adv Neural Inf Process Syst*, vol. 2015-January, pp. 3123–3131, Nov. 2015, Accessed: Dec. 21, 2021. [Online]. Available: https://arxiv.org/abs/1511.00363v3

[105] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. Bengio, and Y. U. Com, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1," Feb. 2016, Accessed: Dec. 21, 2021. [Online]. Available: https://arxiv.org/abs/1602.02830v3

[106] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *Lecture Notes in Computer Science (including subseries*

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9908 LNCS, pp. 525–542, Mar. 2016, doi: 10.1007/978-3-319-46493-0_32.

[107] F. Li, B. Zhang, and B. Liu, "Ternary Weight Networks," May 2016, Accessed: Dec. 21, 2021. [Online]. Available: https://arxiv.org/abs/1605.04711v2

[108] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating Deep Convolutional Networks using low-precision and sparsity," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 2861–2865, Oct. 2016, doi: 10.1109/ICASSP.2017.7952679.

[109] E. Nurvitadhi *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?," *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14, Feb. 2017, doi: 10.1145/3020078.3021740.

[110] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," *Low-Power Computer Vision*, pp. 291–326, Mar. 2021, doi: 10.48550/arxiv.2103.13630.

[111] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," *Dig Tech Pap IEEE Int Solid State Circuits Conf*, vol. 57, pp. 10–14, 2014, doi: 10.1109/ISSCC.2014.6757323.

[112] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, "Understanding the Impact of Precision Quantization on the Accuracy and Energy of Neural Networks," *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pp. 1474–1479, Dec. 2016, doi: 10.48550/arxiv.1612.03940.

[113] S. Kim, G. Park, and Y. Yi, "Performance Evaluation of INT8 Quantized Inference on Mobile GPUs," *IEEE Access*, vol. 9, pp. 164245–164255, 2021, doi: 10.1109/ACCESS.2021.3133100.

[114] Z. Jin and H. Finkel, "Analyzing deep learning model inferences for image classification using OpenVINO," *Proceedings - 2020 IEEE 34th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020*, pp. 908–911, May 2020, doi: 10.1109/IPDPSW50202.2020.00152.

[115] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," *Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017*, pp. 152–159, Jun. 2017, doi: 10.1109/FCCM.2017.25.

[116] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," *FPGA 2015 - 2015 ACM/SIGDA International*

Symposium on Field-Programmable Gate Arrays, pp. 161–170, Feb. 2015, doi: 10.1145/2684746.2689060.

[117] "FINN | finn." https://xilinx.github.io/finn/ (accessed Jul. 28, 2022).

[118] F. Fahim *et al.*, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," *TinyML Research Symposium*, vol. 21, Mar. 2021, doi: 10.48550/arxiv.2103.05579.

[119] "MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges." http://yann.lecun.com/exdb/mnist/ (accessed Jul. 28, 2022).

[120] H. Qu, C. Li, and S. Qian, "Particle Transformer for Jet Tagging," Feb. 2022, doi: 10.48550/arxiv.2202.03772.

[121] G. di Guglielmo *et al.*, "Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML," *Mach Learn Sci Technol*, vol. 2, no. 1, p. 015001, Mar. 2020, doi: 10.1088/2632-2153/aba042.

[122] "GitHub - google/qkeras: QKeras: a quantization deep learning library for Tensorflow Keras." https://github.com/google/qkeras (accessed Jul. 28, 2022).

[123] T. Aarrestad *et al.*, "Fast convolutional neural networks on FPGAs with hls4ml," *Mach Learn Sci Technol*, vol. 2, no. 4, Jan. 2021, doi: 10.1088/2632-2153/ac0ea1.

[124] "LegUp 4.0 Documentation — LegUp 4.0 documentation." http://legup.eecg.utoronto.ca/docs/4.0/index.html (accessed Jul. 28, 2022).

[125] "Intel® FPGA Simulation - ModelSim*-Intel® FPGA." https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html (accessed Jul. 28, 2022).

[126] J. G. López, A. Agudo, and F. Moreno-Noguer, "3D vehicle detection on an FPGA from LIDAR point clouds," *ACM International Conference Proceeding Series*, pp. 21–26, Sep. 2019, doi: 10.1145/3369973.3369984.

[127] "Azure Machine Learning - ML as a Service | Microsoft Azure." https://azure.microsoft.com/en-us/services/machine-learning/ (accessed Jul. 28, 2022).

[128] "GitHub - Xilinx/brevitas: Brevitas: quantization-aware training in PyTorch." https://github.com/Xilinx/brevitas (accessed Jul. 28, 2022).

[129] "PYNQ - Python productivity for Zynq - Home." http://www.pynq.io/ (accessed Jul. 28, 2022).

[130] "Ultra96 - 96Boards." https://www.96boards.org/product/ultra96/ (accessed Jul. 28, 2022).

[131] "Amazon EC2 F1 Instances." https://aws.amazon.com/ec2/instance-types/f1/ (accessed Jul. 28, 2022).

[132] "GitHub - Xilinx/finn: Dataflow compiler for QNN inference on FPGAs." https://github.com/Xilinx/finn (accessed Sep. 04, 2022).

[133] W. Jung, D. Jung, and B. Kim, S. Lee, W. Rhee, and J. H. Ahn, "Restructuring Batch Normalization to Accelerate CNN Training," Jul. 2018, doi: 10.48550/arxiv.1807.01702.

[134] "OpenVINO™ Documentation — OpenVINO™ documentation — Version(latest)." https://docs.openvino.ai/latest/index.html (accessed Jul. 28, 2022).

[135] "GitHub - PaddlePaddle/Paddle: PArallel Distributed Deep LEarning: Machine Learning Framework from Industrial Practice ." https://github.com/PaddlePaddle/Paddle (accessed Jul. 28, 2022).

[136] "Apache MXNet | A flexible and efficient library for deep learning." https://mxnet.apache.org/versions/1.9.1/ (accessed Jul. 28, 2022).

[137] "FPGA Plugin - OpenVINO™ Toolkit." https://docs.openvino.ai/2020.4/openvino_docs_IE_DG_supported_plugins_FPGA.html (accessed Jul. 28, 2022).

[138] "Hardware Architecture • DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338) • Reader • Documentation Portal." https://docs.xilinx.com/r/en-US/pg338-dpu/Hardware-Architecture (accessed Jul. 28, 2022).

[139] "Vitis AI Overview • Vitis AI User Guide (UG1414) • Reader • Documentation Portal." https://docs.xilinx.com/r/2.0-English/ug1414-vitis-ai/Vitis-AI-Overview (accessed Jul. 28, 2022).

[140] T. Cortinhal, G. Tzelepis, and E. E. Aksoy, "SalsaNext: Fast, Uncertainty-aware Semantic Segmentation of LiDAR Point Clouds for Autonomous Driving," *NVIDIA Technical Report NVR-2016-002*, pp. 1–9, Mar. 2020, doi: 10.48550/arxiv.2003.03653.

[141] "CIFAR-10 and CIFAR-100 datasets." https://www.cs.toronto.edu/~kriz/cifar.html (accessed Jul. 28, 2022).

[142] "PyTorch vs TensorFlow in 2022." https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/ (accessed Jul. 28, 2022).

[143] "The latest in Machine Learning | Papers With Code." https://paperswithcode.com/ (accessed Jul. 28, 2022).

[144] F. Ramzan *et al.*, "A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks," *J Med Syst*, vol. 44, no. 2, Feb. 2019, doi: 10.1007/S10916-019-1475-2.

[145] "PetaLinux Tools." https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html (accessed Sep. 04, 2022).

[146] "Home - OpenCV." https://opencv.org/ (accessed Jul. 28, 2022).

[147] "NumPy." https://numpy.org/ (accessed Jul. 28, 2022).

[148] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization," *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Dec. 2014, doi: 10.48550/arxiv.1412.6980.

[149] J. Behley *et al.*, "Towards 3D LiDAR-based semantic scene understanding of 3D point cloud sequences: The SemanticKITTI Dataset:," *https://doi.org/10.1177/02783649211006735*, vol. 40, no. 8–9, pp. 959–967, Apr. 2021, doi: 10.1177/02783649211006735.

[150] "SemanticKITTI - A Dataset for LiDAR-based Semantic Scene Understanding." http://www.semantic-kitti.org/dataset.html#overview (accessed Sep. 04, 2022).

[151] "Oakland 3-D Point Cloud Dataset - CVPR 2009 subset." https://www.cs.cmu.edu/~vmr/datasets/oakland_3d/cvpr09/doc/ (accessed Sep. 04, 2022).

[152] A. Serna, B. Marcotegui, F. Goulette, and J. E. Deschaud, "Paris-rue-madame database: A 3D mobile laser scanner dataset for benchmarking urban detection, segmentation and classification methods," *ICPRAM 2014 - Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods*, pp. 819–824, 2014, doi: 10.5220/0004934808190824.

[153] "iQmulus & TerraMobilita 3D urban analysis contest." http://data.ign.fr/benchmarks/UrbanAnalysis/ (accessed Sep. 04, 2022).

[154] X. Roynard, J. E. Deschaud, and F. Goulette, "Paris-Lille-3D: a large and high-quality ground truth urban point cloud dataset for automatic segmentation and classification," *International Journal of Robotics Research*, vol. 37, no. 6, pp. 545–557, Nov. 2017, doi: 10.48550/arxiv.1712.00032.

[155] W. Tan *et al.*, "Toronto-3D: A Large-scale Mobile LiDAR Dataset for Semantic Segmentation of Urban Roadways," *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2020-June, pp. 797–806, Mar. 2020, doi: 10.1109/CVPRW50498.2020.00109.

[156] "Evaluating image segmentation models." https://www.jeremyjordan.me/evaluating-image-segmentation-models/ (accessed Sep. 04, 2022).

[157] C. Xu *et al.*, "SqueezeSegV3: Spatially-Adaptive Convolution for Efficient Point-Cloud Segmentation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12373 LNCS, pp. 1–19, Apr. 2020, doi: 10.48550/arxiv.2004.01803.

[158] H. Zhou *et al.*, "Cylinder3D: An Effective 3D Framework for Driving-scene LiDAR Semantic Segmentation," Aug. 2020, doi: 10.48550/arxiv.2008.01550.

[159] H. Tang *et al.*, "Searching Efficient 3D Architectures with Sparse Point-Voxel Convolution," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12373 LNCS, pp. 685–702, Jul. 2020, doi: 10.48550/arxiv.2007.16100.

[160] X. Yan *et al.*, "Sparse Single Sweep LiDAR Point Cloud Segmentation via Learning Contextual Shape Priors from Scene Completion," *35th AAAI Conference on Artificial Intelligence, AAAI 2021*, vol. 4A, pp. 3101–3109, Dec. 2020, doi: 10.48550/arxiv.2012.03762.

[161] D. Kochanov, F. K. Nejadasl, and O. Booij, "KPRNet: Improving projection-based LiDAR semantic segmentation," Jul. 2020, doi: 10.48550/arxiv.2007.12668.

[162] A. Milioto, I. Vizzo, J. Behley, and C. Stachniss, "RangeNet ++: Fast and Accurate LiDAR Semantic Segmentation," *IEEE International Conference on Intelligent Robots and Systems*, pp. 4213–4220, Nov. 2019, doi: 10.1109/IROS40897.2019.8967762.

APPENDIX I – SCALE AND ZERO-POINT DERIVATION

Recall that $x \in [\alpha, \beta]$ are the floating-point values and $x_q = [\alpha_q, \beta_q]$ are the quantized values.

$$\begin{cases} \alpha = \alpha_q \\ \beta = \beta_q \end{cases}$$

$$\Leftrightarrow$$

(Equations 1 and 2)

$$\Leftrightarrow$$

$$\begin{cases} \alpha = S \cdot (\alpha_q + Z) \\ \beta = S \cdot (\beta_q + Z) \end{cases}$$

$$\Leftrightarrow$$

$$\begin{cases} S = \dfrac{\beta - \alpha}{\beta_q - \alpha_q} \\ S = \dfrac{\alpha \cdot \beta_q - \beta \cdot \alpha_q}{\beta - \alpha} \end{cases}$$

APPENDIX II – VITIS-AI QAT REQUIREMENTS

The QAT APIs have some requirements for the model to be trained. The following list details all the requirements as of version 2.0 of Vitis-AI.

1. All operations to be quantized must be instances of the *torch.nn.Module* object, rather than Torch functions or Python operators. Operations that need replacement are listed in the following table.

Table 25. QAT mandatory operation replacement.

Operation	Replacement
+	pytorch_nndct.nn.modules.functional.Add
-	pytorch_nndct.nn.modules.functional.Sub
torch.add	pytorch_nndct.nn.modules.functional.Add
torch.sub	pytorch_nndct.nn.modules.functional.Sub

2. It is advised to call modules only once. For example, if a model architecture uses several ReLU activations, for each call, a different torch.nn.ReLu module should be used.
3. QuantStub should be used to quantize the inputs of the network and DeQuantStub to de-quantize the outputs of the network. Any sub-network from QuantStub to DeQuantStub in a forward pass will be quantized. Multiple QuantStub-DeQuantStub pairs are allowed.

The following code corresponds to parts of the original Torchvision implementation of the ResNet-18 model.

```
class BasicBlock(nn.Module):
        (...)
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x: Tensor) -> Tensor:
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out

class ResNet(nn.Module):
        (...)
    def _forward_impl(self, x: Tensor) -> Tensor:
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

Figure 64. Excerpt of ResNet-18's torchvision implementation.

This next code block corresponds to the ResNet-18 model modified to fit Vitis-AI's QAT requirements. Note the additions of several *torch.nn.ReLU* modules, the replacement of the '+' operation by the *torch.functional.Add()*, and the definition of *pytorch_nndct.nn.QuantStub()* and *pytorch_nndct.nn.DeQuantStub()* used in the *forward_impl* method.

```
class BasicBlock(nn.Module):
 (…)
self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu1 = nn.ReLU(inplace=True)         #added
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
       # additional relu
       self.relu2 = nn.ReLU(inplace=True)          #added
       # add functional Add
       self.skip_add = functional.Add()            #added

def forward(self, x: Tensor) -> Tensor:
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu1(out)                      #added
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out = self.skip_add(out, identity)         #added
        out = self.relu2(out)                      #added
        return out

class ResNet(nn.Module):
   self.quant_stub = pytorch_nndct.nn.QuantStub()        #added
   self.dequant_stub = pytorch_nndct.nn.DeQuantStub()    #added
   (…)
   def _forward_impl(self, x: Tensor) -> Tensor:
        x = self.quant_stub(x)                           #added
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        x = torch.reshape(x, (x.shape[0], x.shape[1]))   #added
        x = self.fc(x)
        x = self.dequant_stub(x)                         #added
        return x
```

Figure 65. Excerpt of ResNet-18's QAT compatible implementation.

APPENDIX III – DPUCZDX8G AND DPUCVDX8G SUPPORTED OPERATORS

The following information is a summary of the Vitis-AI documentation and refers to version 2.0 of the tool.

Table 26. DPUCZDX8G and DPUCVDX8G channel parallel and bank depth possible values – Vitis-AI 2.0.

Intrinsic Parameters	DPUCZDX8G	DPUCVDX8G
Channel Parallel	16	16
Bank Depth	2048	16384

Table 27. DPUCZDX8G and DPUCVDX8G XIR operations and parameters support – Vitis-AI 2.0.

CNN Operation	Parameters	DPUCZDX8G	DPUCVDX8G
Conv2d	Kernel size	w, h: [1, 16]	w, h: [1, 16] w * h <= 64
	Strides	w, h: [1, 8]	w, h: [1, 8]
	Dilation	dilation * input_channel <= 256 * channel_parallel	
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	
	In Size	kernel_w * kernel_h * ceil (input_channel / channel_parallel) <= bank_depth	
	Out Size	output_channel <= 256 * channel_parallel	
	Activation	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
Depthwise-conv2d	Kernel size	w, h: [1, 16]	w, h: [1, 256]
	Strides	w, h: [1, 8]	w, h: [1, 8]
	Dilation	dilation * input_channel <= 256 * channel_parallel	
	Paddings	pad_left, pad_right: [0, (kernel_w - 1) * dilation_w] pad_top, pad_bottom: [0, (kernel_h - 1) * dilation_h]	pad_left, pad_right: [0, 15 * dilation_w] pad_top, pad_bottom: [0, 15 * dilation_h]
	In Size	kernel_w * kernel_h * ceil (input_channel / channel_parallel) <= bank_depth	
	Out Size	output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6	

Transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]	
	Strides		
	Paddings	pad_left, pad_right: [1, kernel_w-1] pad_top, pad_bottom: [1, kernel_h-1]	
	Out Size	output_channel <= 256 * channel_parallel	
	Activation	ReLU, LeakyReLU, ReLU6	ReLU, LeakyReLU, ReLU6, Hard-Swish, Hard-Sigmoid
Depthwise-transposed-conv2d	Kernel size	kernel_w/stride_w, kernel_h/stride_h: [1, 16]	
	Strides	kernel_w/stride_w, kernel_h/stride_h: [1, 256]	
	Paddings	pad_left, pad_right: [1, kernel_w-1] pad_top, pad_bottom: [1, kernel_h-1]	pad_left, pad_right: [1, 15] pad_top, pad_bottom: [1, 15]
	Out Size	output_channel <= 256 * channel_parallel	
	Activation	ReLU, ReLU6	
Max-pooling	Kernel size	w, h: [2, 8]	w, h: [1, 256]
	Strides	w, h: [1, 8]	
	Paddings	pad_left, pad_right: [1, kernel_w-1] pad_top, pad_bottom: [1, kernel_h-1]	pad_left, pad_right: [1, 15] pad_top, pad_bottom: [1, 15]
	Activation	ReLU	ReLU, ReLU6
Average-pooling	Kernel size	w, h: [2, 8] w==h	w, h: [1, 256]
	Strides	w, h: [1, 8]	
	Paddings	pad_left, pad_right: [1, kernel_w-1] pad_top, pad_bottom: [1, kernel_h-1]	pad_left, pad_right: [1, 15] pad_top, pad_bottom: [1, 15]
	Activation	ReLU	ReLU, ReLU6
Eltwise	Type	sum	sum, prod
	Input Channel	input_channel <= 256 * channel_parallel	
	Activation	ReLU	
Concat	Network-specific limitation, which relates to the size of feature maps, quantization results and compiler optimizations		
Reorg	Strides	reverse==false : stride ^ 2 * input_channel <= 256 * channel_parallel reverse==true : input_channel <= 256 * channel_parallel	
Pad	In Size	input_channel <= 256 * channel_parallel	
	Mode	"SYMMETRIC" ("CONSTANT" pad (value=0) would be fused into adjacent operators during compiler optimization process)	
Global pooling	Global pooling will be processed as general pooling with kernel size equal to input tensor size		

InnerProduct, Fully Connected, Matmul	These operations will be transformed into conv2d op

Table 28. Pytorch operations to XIR operations translation.

Pytorch	**XIR**	**DPU implementation notes**
Conv2d	Conv2d	-
ConvTranspose2d	transposed-conv2d	-
Matmul	Conv2d	The matmul would be transformed to conv2d and compiled to Convolution Engine. If the matmul fails to be transformed, it would be implemented by CPU.
MaxPool2d / AdaptiveMaxPool2d	Maxpool2d	Pooling Engine
AvgPool2d / AdaptiveAvgPool2d	Avgpool2d	Pooling Engine
ReLU	ReLU	Activations would be fused to adjacent operations such as convolution and add
LeakyReLU	LeakyReLU	
ReLU6	ReLU6	
Hardsigmoid	Hardsigmoid	
Hardswish	Hardswish	
ConstantPad2d / ZeroPad2d	pad	"CONSTANT" padding would be fused adjacent operations.
Add	Add	If the add is an element-wise add, the add would be mapped to DPU Element-wise Add Engine. If the add is a channel-wise add, search for opportunities to fuse the add with adjacent operations such as convolutions. If they are shape-related operations, they would be removed during compilation. If they are components of a coarse-grained operation, they would be fused with adjacent operations. Otherwise, they
Sub / Rsub	Sub	
Mul	Mul	
Neg	Neg	
Sum	Reduction Sum	
Max	Reduction Max	
Mean	Reduction Mean	

		would be compiled into CPU implementations.
Interpolate / Upsample / Upsample_bilinear / Upsample_nearest	Resize	If the mode of the resize is 'BILINEAR', align_corner=false, half_pixel_centers = false, size = 2, 4, 8; align_corner=false, half_pixel_centers = true, size = 2, 4 can be transformed to DPU implementations (pad+depthwise-transposed conv2d). If the mode of the resize is 'NEAREST' and the size are integers, the resize would be mapped to DPU implementations.
Transpose	Transpose	These operations would be transformed to the reshape operation in some cases. Additionally, search for opportunities to fuse the dimension transformation operations into special load or save instructions of adjacent operations to reduce the overhead. Otherwise, they would be mapped to CPU.
Permute	Transpose	
View/Reshape	Reshape	
Flatten	Reshape/Flatten	
Squeeze	Reshape/Squeeze	
Cat	Concat	-
Aten::slice	Strided_slice	If the strided_slice is shape-related or is the component of a coarse-grained operation, it would be removed. Otherwise, the strided_slice would be compiled into CPU implementations.
BatchNorm2d	Depthwise-conv2d / scale	If the batch_norm is quantized and can be transformed to a depthwise-conv2d equivalently, it would be transformed to depthwise-conv2d, and the compiler would search for compilation opportunities to map the batch_norm into DPU implementations. Otherwise, the batch_norm would be executed by CPU.

Softmax	Softmax	They would only be compiled into CPU implementations.
Tanh	Tanh	
Sigmoid	Sigmoid	
PixelShuffle	Pixel_Shuffle	They would be transformed to tile if there's convolution as its input.
PixelUnshuffle	Pixel_Shuffle	

APPENDIX IV – RESNET-18 AND SQUEEZENET MODELS COMPLETE RESULTS

The following table lists all the inference FPS and peak power consumption in Watts measured during the inference of the ResNet-18 and SqueezeNet models on the Cifar-10 test set. The number of CPU threads used by the application that enables inference varies in the interval $[1, 12]$.

Table 29. ResNet-18 and SqueezeNet average inference FPS and peak power consumption across all ZCU104 and VCK190 configurations.

Model	Hardware/Config	# Threads	Avg Inference FPS	Peak Power Consumption (Watts)
ResNet-18	ZCU104 – B512 Hybrid	1	306.72	18.75
		2	334.98	18.98
		3	334.32	19.07
		4	334.01	18.84
		5	333.68	18.76
		6	333.40	19.07
		7	333.22	18.84
		8	332.50	19.07
		9	332.45	19.07
		10	331.87	18.84
		11	330.70	18.98
		12	328.92	18.75
	ZCU104 – B1024 BRAM-only	1	316.91	19.04
		2	346.99	19.26
		3	346.37	19.44
		4	346.14	19.80
		5	345.59	19.89
		6	345.56	19.75
		7	345.02	19.58
		8	344.47	19.52
		9	344.03	19.61

		10	343.42	19.75
		11	342.56	19.89
		12	340.57	19.44
	ZCU104 – B1024 Hybrid	1	316.99	19.13
		2	346.92	19.35
		3	346.71	19.71
		4	346.45	19.49
		5	345.79	19.49
		6	345.84	19.71
		7	345.29	19.26
		8	344.68	19.62
		9	344.23	19.71
		10	343.89	19.49
		11	342.11	19.49
		12	341.04	19.26
	ZCU104 – 2x B1024 Hybrid	1	221.33	20.16
		2	347.95	20.83
		3	354.08	21.06
		4	363.11	21.06
		5	362.60	21.06
		6	361.73	21.06
		7	361.84	21.06
		8	361.53	21.06
		9	361.31	21.02
		10	361.13	21.06
		11	361.02	21.11
		12	360.97	21.02
	ZCU104 – B4096 Hybrid	1	509.03	22.02
		2	591.54	22.47
		3	589.65	22.70
		4	589.81	22.70
		5	588.71	22.47
		6	587.56	22.47
		7	588.26	22.47
		8	587.10	22.70
		9	585.71	22.70
		10	584.50	22.47

		11	582.99	22.70
		12	579.99	22.70
	ZCU104 – 2x B4096 Hybrid	1	450.86	25.43
		2	776.93	27.00
		3	819.52	27.33
		4	856.06	27.55
		5	851.75	27.55
		6	849.27	27.55
		7	846.82	27.43
		8	847.18	27.55
		9	845.70	27.43
		10	844.32	27.43
		11	843.23	27.43
		12	842.22	27.43
	ZCU104 – B4096 Hybrid + High RAM usage	1	511.63	21.50
		2	595.81	22.18
		3	594.74	22.37
		4	593.61	22.37
		5	591.58	22.37
		6	591.80	22.15
		7	589.93	22.15
		8	590.91	21.92
		9	590.27	22.37
		10	589.14	21.92
		11	586.59	22.15
		12	584.22	22.37
	VCK190 – C32B1	1	1652.69	59.40
		2	2360.24	60.98
		3	2360.32	60.98
		4	2358.81	60.98
		5	2356.67	60.98
		6	2357.35	60.98
		7	2349.25	60.79
		8	2349.42	61.02
		9	2346.41	60.98
		10	2345.42	61.06
		11	2346.94	61.06

		12	2342.03	61.06
	VCK190 – C64B1	1	1782.73	62.47
		2	2652.05	64.52
		3	2646.83	64.52
		4	2644.76	64.75
		5	2637.08	64.52
		6	2634.07	64.75
		7	2632.27	64.75
		8	2624.67	64.52
		9	2602.50	64.52
		10	2620.86	64.75
		11	2603.64	64.75
		12	2602.29	64.75
	VCK190 – 2x C64B1	1	1740.46	71.78
		2	2953.95	75.03
		3	3109.08	75.60
		4	3233.10	76.05
		5	3193.95	76.28
		6	3163.00	76.05
		7	3100.44	76.05
		8	3044.98	76.05
		9	3014.39	75.71
		10	2996.97	75.94
		11	2987.94	75.71
		12	2993.33	75.83
SqueezeNet	ZCU104 – B512 Hybrid	1	1156.31	18.08
		2	1689.90	18.53
		3	1681.65	18.53
		4	1680.18	18.53
		5	1677.29	18.45
		6	1677.51	18.31
		7	1670.03	18.45
		8	1667.70	18.31
		9	1665.64	18.53
		10	1646.84	18.31
		11	1656.37	18.31
		12	1628.70	18.53

	ZCU104 – B1024 BRAM-only	1	1259.68	18.34
		2	1915.40	18.81
		3	1909.82	19.24
		4	1905.46	19.38
		5	1900.35	19.15
		6	1899.72	19.24
		7	1889.33	19.24
		8	1893.44	18.92
		9	1864.79	19.38
		10	1879.17	19.24
		11	1876.17	19.24
		12	1882.83	19.00
	ZCU104 – B1024 Hybrid	1	1262.26	18.45
		2	1916.81	18.90
		3	1919.94	19.04
		4	1925.07	19.26
		5	1919.28	19.04
		6	1911.44	19.18
		7	1915.10	19.26
		8	1909.77	19.26
		9	1901.69	18.82
		10	1897.50	19.04
		11	1880.49	19.26
		12	1865.17	19.04
	ZCU104 – 2x B1024 Hybrid	1	1043.95	19.49
		2	1521.51	19.94
		3	1798.77	20.03
		4	1877.30	20.16
		5	1876.00	20.25
		6	1878.36	20.25
		7	1875.58	20.25
		8	1871.34	20.25
		9	1866.03	20.34
		10	1865.23	20.25
		11	1867.11	20.34
		12	1860.00	20.34
	ZCU104 – B4096 Hybrid	1	1379.30	20.20

		2	2205.12	20.88
		3	2201.07	21.11
		4	2201.91	21.11
		5	2190.55	21.11
		6	2189.03	20.98
		7	2183.35	20.98
		8	2176.03	21.30
		9	2153.46	21.30
		10	2158.81	21.20
		11	2168.50	21.20
		12	2132.96	21.20
	ZCU104 – 2x B4096 Hybrid	1	1365.87	23.63
		2	2667.46	24.75
		3	3309.21	25.43
		4	4023.24	25.88
		5	3871.23	26.10
		6	3776.04	25.88
		7	3819.29	25.99
		8	3821.84	26.10
		9	3697.68	26.22
		10	3688.21	25.88
		11	3663.61	25.88
		12	3658.74	25.88
	ZCU104 – B4096 Hybrid + High RAM Usage	1	1408.05	23.28
		2	2283.38	23.96
		3	2273.84	23.96
		4	2273.53	23.84
		5	2267.49	23.63
		6	2262.06	23.85
		7	2251.87	23.40
		8	2207.76	23.63
		9	2229.32	23.40
		10	2221.19	23.50
		11	2222.77	23.73
		12	2176.24	23.73
	VCK190 – C32B1	1	3469.86	56.98
		2	5779.07	57.43

		3	5920.78	57.43
		4	5925.12	57.20
		5	5619.54	57.43
		6	5423.93	57.23
		7	5332.58	57.20
		8	5188.17	57.23
		9	5073.66	56.98
		10	5052.10	57.23
		11	5013.37	56.98
		12	4976.70	56.98
	VCK190 – C64B1	1	3590.69	59.74
		2	6022.36	60.19
		3	6119.82	60.19
		4	6045.03	60.19
		5	5722.04	60.42
		6	5551.71	60.19
		7	5488.51	60.19
		8	5280.35	60.19
		9	5166.57	60.19
		10	5186.66	60 42
		11	5186.07	60.19
		12	5100.02	60 42
	VCK190 – 2x C64B1	1	3537.25	68.55
		2	5998.34	69.24
		3	6035.55	69.46
		4	6002.28	69.46
		5	5615.90	69.46
		6	5379.63	69.24
		7	5305.77	69.24
		8	5155.14	68.70
		9	5078.76	69.46
		10	5066.98	69.38
		11	5039.50	69.38
		12	4973.46	69.38

APPENDIX V – SQUEEZESEGV3-21 PYTORCH ARCHITECTURE DESCRIPTION

The following table contains the original SqueezeSegV3-21 architecture as implemented by the authors. The rightmost column details whether the DPUCVDX8G supports the layer with the specified parameters, considering Vitis-AI version 2.0.

Table 30. SqueezeSegV3-21 original implementation's list of pytorch operations and respective parameters.

Architecture Section	Operation	Parameters	Support
Backbone	nn.Conv2d	In Channels = 5 Out Channels = 32 Kernel Size = (3, 3) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 32 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
Backbone/encoder 1	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 288 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 288 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 288 Out Channels = 32	DPU

		Kernel Size = (1, 1) Stride = (1, 1)	
	nn.BatchNorm2d	Num Features = 32 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	nn.Conv2d	In channels = 32 Out Channels = 32 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 32 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Add	Input tensors	DPU
	nn.Conv2d	In channels = 32 Out Channels = 64 Kernel Size = (3, 3) Stride = (1, 2) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Functional.upsample_bilinear	Input = torch.Tensor Size = [torch.Tensor.size()[2], torch.Tensor.size()[3]//2]	None
	Tensor.Detach	-	None*
	nn.Dropout2d	P = 0.01	None*
Backbone/encoder 2	Functional.unfold	Kernel Size = 3	None

		Padding = 1	
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 576 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 576 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 576 Out Channels = 64 Kernel Size = (1, 1) Stride = (1, 1)	
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 64 Out Channels = 64 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	nn.Conv2d	In Channels = 64 Out Channels = 128 Kernel Size = (3, 3) Stride = (1, 2) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Functional.upsample_bilinear	Input = torch.Tensor	None

		Size = [torch.Tensor.size()[2], torch.Tensor.size()[3]//2	
	Tensor.detach	-	None*
	nn.Dropout2d	P = 0.5	None*
Backbone/encoder 3	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 1152 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 1152 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 1152 Out Channels = 128 Kernel Size = (1, 1) Stride = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 128 Out Channels = 128 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3	DPU

		Out Channels = 1152 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	
	nn.BatchNorm2d	Num Features = 1152 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 1152 Out Channels = 128 Kernel Size = (1, 1) Stride = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 128 Out Channels = 128 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	nn.Conv2d	In Channels = 128 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 2) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Functional.upsample_bilinear	Input = torch.Tensor Size = [torch.Tensor.size()[2], torch.Tensor.size()[3]//2	None
	Tensor.detach	-	None*
	nn.Dropout2d	P = 0.5	None*

Backbone/encoder 4	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 2304 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 2304 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 2304 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1)	
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 2304 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 2304 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU

	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 2304 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	nn.Dropout2d	P = 0.5	None*
Backbone/encoder 5	Functional.unfold	Kernel Size = 3 Padding = 1	None
	Tensor.view	Shape	DPU
	nn.Conv2d	In Channels = 3 Out Channels = 2304 Kernel Size = (7, 7) Stride = (1, 1) Padding = (3, 3)	DPU
	nn.BatchNorm2d	Num Features = 2304 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	Functional.sigmoid	-	CPU
	Mul	Input tensors	DPU
	nn.Conv2d	In Channels = 2304 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU

	nn.ReLu	Inplace = True	DPU
	Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.1 Affine = True Track Running Stats = True	DPU
	nn.ReLu	inplace=True	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	nn.Dropout2d	P = 0.5	None*
Decoder/decoder 5	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*

	Add	Input tensors	DPU
Decoder/decoder 4	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 256 Out Channels = 256 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	Add	Input tensors	DPU
Decoder/decoder 3	nn.ConvTranspose2d	In Channels = 256 Out Channels = 128 Kernel Size = (1, 4) Stride = (1, 2) Padding = (0, 1)	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 128 Out Channels = 256 Kernel Size = (1, 1) Stride = (1, 1)	DPU

		Bias = False	
	nn.BatchNorm2d	Num Features = 256 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 256 Out Channels = 128 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	Add	Input tensors	DPU
Decoder/decoder 2	nn.ConvTranspose2d	In Channels = 128 Out Channels = 64 Kernel Size = (1, 4) Stride = (1, 2) Padding = (0, 1)	DPU
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 64 Out Channels = 128 Kernel Size = (1, 1) Stride = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 128 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 128 Out Channels = 64 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.01 Affine = True	DPU

		Track Running Stats = True	
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	Add	Input tensors	DPU
Decoder/decoder 2	nn.ConvTranspose2d	In Channels = 64 Out Channels = 32 Kernel Size = (1, 4) Stride = (1, 2) Padding = (0, 1)	DPU
	nn.BatchNorm2d	Num Features = 32 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 32 Out Channels = 64 Kernel Size = (1, 1) Stride = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 64 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	nn.Conv2d	In Channels = 64 Out Channels = 32 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1) Bias = False	DPU
	nn.BatchNorm2d	Num Features = 32 Eps = 1e-05 Momentum = 0.01 Affine = True Track Running Stats = True	DPU
	nn.LeakyReLu	Negative Slope = 0.1	DPU
	Add	Input tensors	DPU
	Tensor.detach	-	None*
	Add	Input tensors	DPU
Decoder	nn.Dropout2d	P = 0.01 Inplace = False	None*
Head 5	nn.Dropout2d	P = 0.01 Inplace = False	None*
	nn.Conv2d	In Channels = 32 Out Channels = 20 Kernel Size = (3, 3) Stride = (1, 1) Padding = (1, 1)	DPU

	Functional.softmax	Dim=1	CPU

*Discarded after training

APPENDIX VI – SQUEEZESEGV3-21 COMPLETE RESULTS

Figure 66 through Figure 69 show the accuracy and IoU, as well as the loss plot of the SSGV321-K1 and SSGV321-K1N45 models.

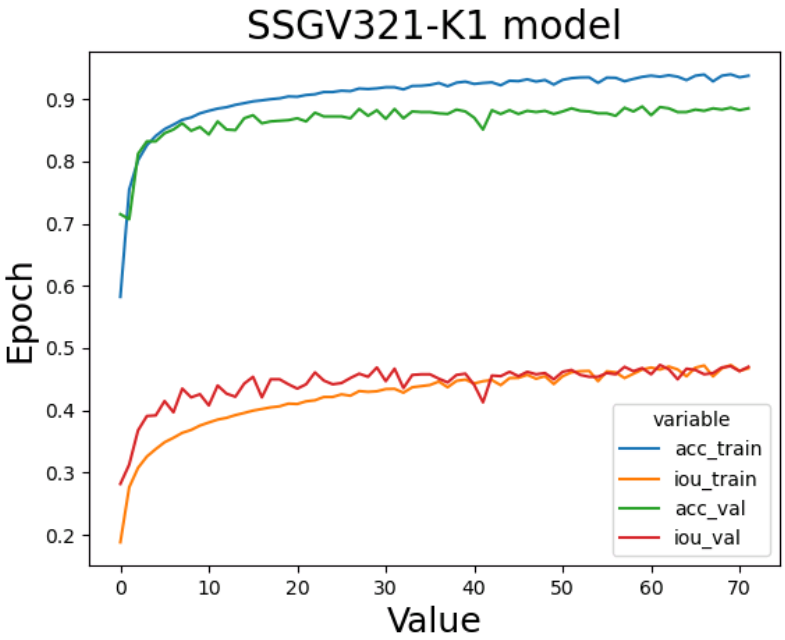

Figure 66. SSGV321-K1 model training: validation accuracies and IoUs.

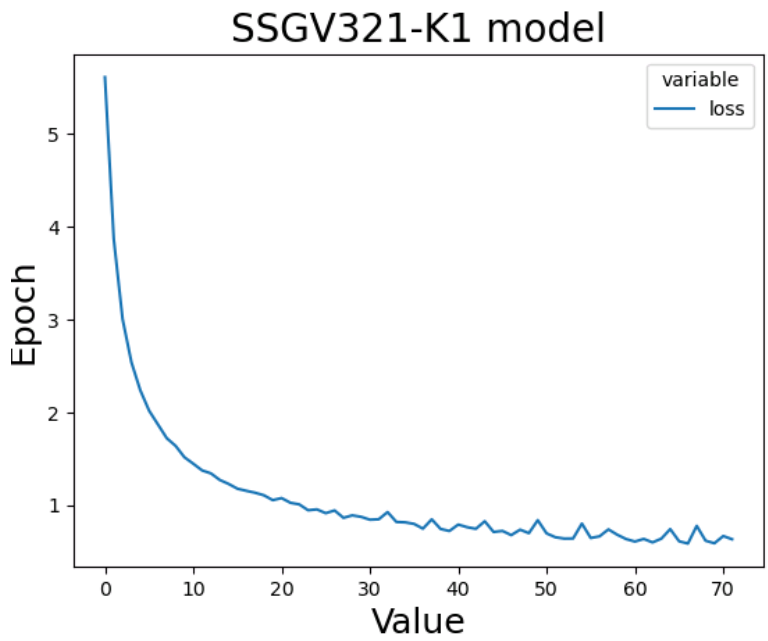

Figure 67. SSGV321-K1 model training: training set loss.

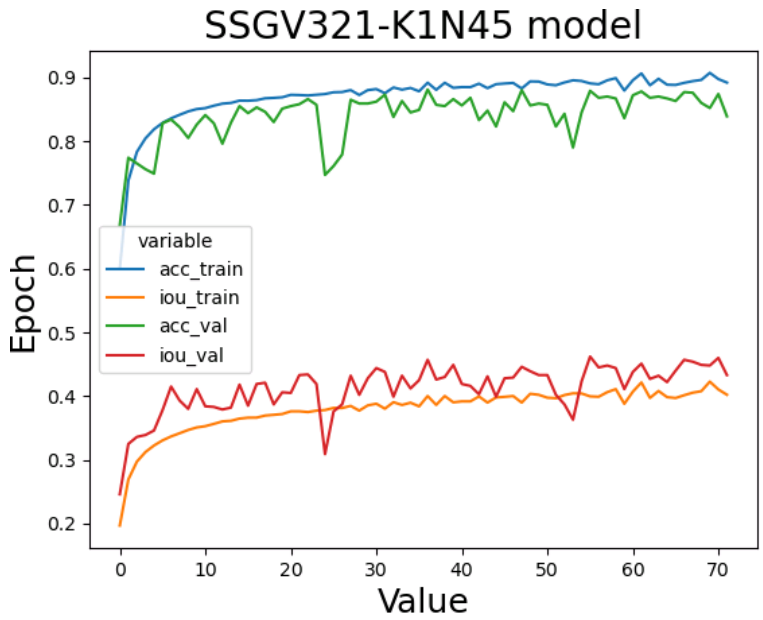

Figure 68. SSGV321-K1N45 model training: validation accuracies and IoUs.

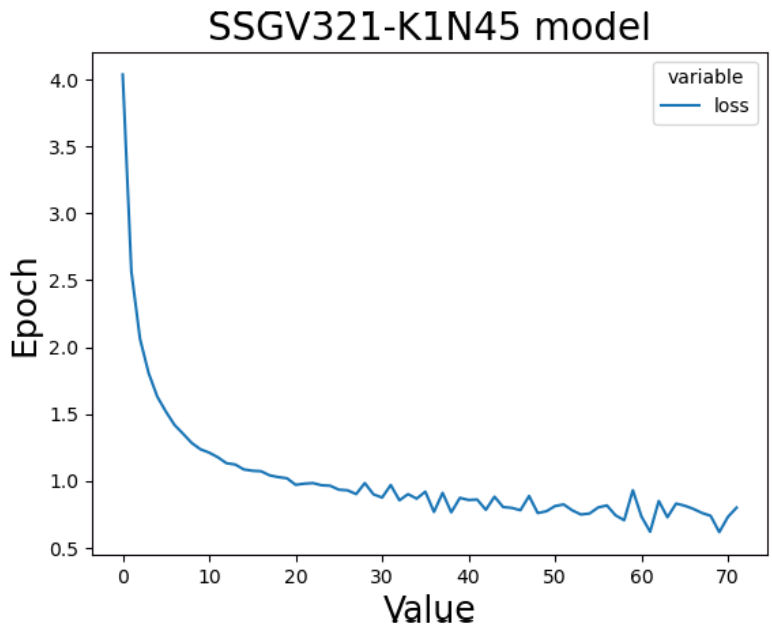

Figure 69. SSGV321-K1 model training: training set loss.

The following table lists the per-class IoU results of the 3 SSGV3-21 model variants on the validation set of the Semantic-KITTI dataset (sequence 08). All models have been trained for 72 epochs.

Table 31. Per-class IoU of the 3 SSGV3-21 variants on the validation set of Semantic-KITTI.

Class	**SSGV321-K7**	**SSGV321-K3**	**SSGV321-K1N45**
Car	**0.745**	0.744	0.729
Bicycle	0.223	**0.225**	0.181
Motorcycle	**0.409**	0.371	0.367
Truck	**0.231**	0.190	0.104
Other vehicle	0.163	**0.229**	0.181
Person	**0.397**	0.386	0.391
Bicyclist	0.498	0.430	**0.508**
Road	0.924	**0.926**	0.918
Parking	0.397	**0.385**	0.366
Sidewalk	0.790	**0.791**	0.756
Other ground	0.004	0.003	**0.007**
Building	**0.785**	0.784	0.771
Fence	0.312	**0.317**	0.280
Vegetation	0.802	0.790	**0.804**
Trunk	**0.463**	0.441	0.438
Terrain	0.719	0.710	**0.723**

Pole	**0.348**	0.332	0.280
Traffic sign	**0.333**	0.293	0.317

*Bold values represent the best IoU of all 3 models.

The following table lists all the inference FPS and peak power consumption in Watts measured during the inference of the SSGV321-K7, SSGV321-K3, and SSGV321-K1N45 models on Semantic-KITTI dataset samples. The number of CPU threads used by the application that enables inference varies in the interval $[1, 4]$.

Table 32. SSGV321-K7, SSGV321-K3, and SSGV321-K1N45 average inference FPS and peak power consumption across all VCK190 configurations.

Model	**Hardware/Configuration**	**# Threads**	**Avg Inference FPS**	**Peak Power Consumption (Watts)**
SSGV321-K7	VCK190 - C32B1	1	3.49	59.54
		2	3.67	60.26
		3	3.67	60.26
		4	3.67	60.03
	VCK190 – C64B1	1	4.70	63.84
		2	5.03	64.75
		3	5.03	64.75
		4	5.03	64.75
	VCK190 - 2x C64B1	1	4.65	73.80
		2	9.12	79.78
		3	9.49	80.55
		4	9.83	80.76
SSGV321-K3	VCK190 – C32B1	1	4.84	59.11
		2	5.19	59.80
		3	5.19	59.80
		4	5.19	59.80
	VCK190 - C64B1	1	5.39	63.56
		2	5.82	64.07
		3	5.82	63.84
		4	5.82	64.07
	VCK190 - 2x C64B1	1	5.27	73.13
		2	10.31	78.08
		3	10.64	78.30

		4	11.17	78.30
SSGV321-K1N45	VCK190 - C32B1	1	5.60	58.19
		2	8.45	59.31
		3	8.45	59.31
		4	8.45	59.02
	VCK190 - C64B1	1	5.92	62.24
		2	9.18	63.43
		3	9.18	63.20
		4	9.19	63.43
	VCK190 - 2x C64B1	1	8.51	71.68
		2	16.44	75.57
		3	17.66	76.26
		4	18.92	76.15

The following table lists all SSGV321-K7 layers, respective parameters, and average computation time during inference on the VCK190 C64B1x2 configuration and 1 CPU thread.

Table 33. SSGV321-K7 layer-by-layer average computation time during inference on C64B1x2 with 1 CPU thread.

Location	**Layer**	**Parameters**	**Occurrences**	**Average Computation Time (ms)**
Backbone/Before Encoder1	Conv2D + LeakyReLU	In channels = 5 Out channels = 32 Kernel size = 3 Padding = 1	1	0.611
Backbone/Encoder1/SACBlock	Conv2D	In channels = 32 Out channels = 288 Kernel size = 1 Padding = 0	1	4.817
Backbone/Encoder1/SACBlock	Conv2D + ReLU	In channels = 288 Out channels = 32 Kernel size = 1 Padding = 0	1	3.643
Backbone/Encoder1/SACBlock	Conv2D + ReLU	In channels = 32 Out channels = 32 Kernel size = 3 Padding = 1	1	1.031

Backbone/Encoder1/SACBlock	Conv2D	In channels = 3 Out channels = 288 Kernel size = 7 Padding = 3	1	4.891
Backbone/Encoder1/SACBlock	Mul	-	1	7.275
Backbone/Encoder1	Conv2D + LeakyReLU	In channels = 32 Out channels = 64 Kernel size = 3 Padding = 1	1	0.612
Backbone/After Encoder1	Conv2D	In channels = 3 Out channels = 3 Kernel size = 1 Padding = 0	1	0.259
Backbone/Encoder2/SACBlock	Conv2D	In channels = 64 Out channels = 576 Kernel size = 1 Padding = 0	1	4.818
Backbone/Encoder2/SACBlock	Conv2D	In channels = 3 Out channels = 576 Kernel size = 7 Padding = 3	1	9.496
Backbone/Encoder2/SACBlock	Conv2D + ReLU	In channels = 576 Out channels = 64 Kernel size = 1 Padding = 0	1	3.640
Backbone/Encoder2/SACBlock	Conv2D + ReLU	In channels = 64 Out channels = 64 Kernel size = 3 Padding = 1	1	4.844
Backbone/Encoder2/SACBlock	Mul	-	1	8.143
Backbone/Encoder2/SACBlock	Conv2D + LeakyReLU	In channels = 64 Out channels = 128 Kernel size = 3 Padding = 1	1	0.616
Backbone/After Encoder2	Conv2D	In channels = 3 Out channels = 3 Kernel size = 1 Padding = 0	1	0.156

Backbone/Encoder3/SACBlock	Conv2D	In channels = 128 Out channels = 1152 Kernel size = 1 Padding = 0	2	4.818
Backbone/Encoder3/SACBlock	Conv2D	In channels = 3 Out channels = 1152 Kernel size = 7 Padding = 3	2	10.573
Backbone/Encoder3/SACBlock	Mul	-	2	8.143
Backbone/Encoder3/SACBlock	Conv2D + ReLU	In channels = 1152 Out channels = 128 Kernel size = 1 Padding = 0	2	3.642
Backbone/Encoder3/SACBlock	Conv2D + ReLU	In channels = 128 Out channels = 128 Kernel size = 3 Padding = 1	2	0.998
Backbone/Encoder3	Conv2D + LeakyReLU	In channels = 128 Out channels = 256 Kernel size = 3 Padding = 1	1	0.925
Backbone/After Encoder 3	Conv2D	In channels = 3 Out channels = 3 Kernel size = 1 Padding = 0	1	0.123
Backbone/Encoder4/SACBlock	Conv2D	In channels = 3 Out channels = 2304 Kernel size = 7 Padding = 3	2	8.634
Backbone/Encoder4/SACBlock	Conv2D	In channels = 256 Out channels = 2304 Kernel size = 1 Padding = 0	2	4.819
Backbone/Encoder4/SACBlock	Mul	-	1	8.118
Backbone/Encoder4/SACBlock	Conv2D + ReLU	In channels = 2304 Out channels = 256 Kernel size = 1 Padding = 0	2	3.656

Backbone/Encoder4/SACBlock	Conv2D + ReLU	In channels = 256 Out channels = 256 Kernel size = 3 Padding = 1	2	1.406
Backbone/Encoder5/SACBlock	Conv2D	In channels = 3 Out channels = 2304 Kernel size = 7 Padding = 3	1	8.634
Backbone/Encoder5/SACBlock	Conv2D	In channels = 256 Out channels = 2304 Kernel size = 1 Padding = 0	1	4.820
Backbone/Encoder5/SACBlock	Mul	-	1	7.250
Backbone/Encoder5/SACBlock	Conv2D	In channels = 2304 Out channels = 256 Kernel size = 1 Padding = 0	1	3.655
Backbone/Encoder5/SACBlock	Conv2D	In channels = 256 Out channels = 256 Kernel size = 3 Padding = 1	1	1.406
Decoder5	Conv2D + LeakyReLU	In channels = 256 Out channels = 256 Kernel size = 3 Padding = 1	2	1.573
Decoder5	Conv2D + LeakyReLU	In channels = 256 Out channels = 256 Kernel size = 1 Padding = 0	1	0.652
Decoder4	Conv2D + LeakyReLU	In channels = 256 Out channels = 256 Kernel size = 3 Padding = 1	2	1.573
Decoder4	Conv2D + LeakyReLU	In channels = 256 Out channels = 256 Kernel size = 1 Padding = 0	1	0.652

Decoder3	Transposed Conv2D + LeakyReLU	Dilation = 1 Kernel = (4, 1) Pad = (1, 1, 0, 0) Stride = (2, 1)	2	0.612
Decoder3	Conv2D + LeakyReLU	In channels = 128 Out channels = 256 Kernel size = 1 Padding = 0	1	1.145
Decoder3	Add	-	1	1.512
Decoder3	Add	-	1	0.882
Decoder3	Conv2D + LeakyReLU	In channels = 64 Out channels = 128 Kernel size = 1 Padding = 0	1	1.146
Decoder2	Add	-	1	1.431
Decoder2	Add	-	1	0.882
Decoder1	Transposed Conv2D + LeakyReLU	Dilation = 1 Kernel = (4, 1) Pad = (1, 1, 0, 0) Stride = (2, 1)	1	0.620
Decoder1	Conv2D + LeakyReLU	In channels = 32 Out channels = 64 Kernel size = 1 Padding = 0	1	1.139
Decoder1	Add	-	1	1.506
Decoder1	Add	-	1	0.962
Head5	Conv2D	In channels = 32 Out channels = 20 Kernel size = 3 Padding = 1	1	0.546

The following two tables contain the detailed results of training the SSGV321-K3 for 100 epochs and a comparison with the same model trained for 72 epochs.

Table 34. Average accuracy and average IoU of SSGV321-K3. 72 epochs vs 100 epochs training.

Model	**Average Accuracy**	**Average IoU**
SSGV321-K3 (72 epochs)	0.862	0.439

SSGV321-K3 (100 epochs)	**0.865**	**0.448**

Table 35. Per class IoU of SSGV321-K3. 72 epochs vs 100 epochs training.

Class	SSGV321-K3 (72 epochs)	SSGV321-K3 (100 epochs)
Car	0.744	**0.770**
Bicycle	0.225	**0.252**
Motorcycle	0.371	**0.379**
Truck	**0.190**	0.165
Other vehicle	**0.229**	0.164
Person	0.386	**0.415**
Bicyclist	0.430	**0.490**
Road	**0.926**	0.919
Parking	**0.385**	0.378
Sidewalk	**0.791**	0.788
Other ground	**0.003**	0.000
Building	0.784	**0.792**
Fence	0.317	**0.325**
Vegetation	0.790	**0.798**
Trunk	0.441	**0.468**
Terrain	**0.710**	0.702
Pole	0.332	**0.370**
Traffic sign	0.293	**0.339**