TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Project Report

On

FPGA based Accelerator Co-design using Neural Architectural Search

Submitted By:

Amit Raj Pant (41754)

Bishal Rijal (41763)

Kshitiz Poudel (41767)

Pilot Khadka (41774)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

April, 2024

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

THAPATHALI CAMPUS

A Project Report

On

FPGA based Accelerator Co-design using Neural Architectural Search

Submitted By:

Amit Raj Pant	(41754)
Bishal Rijal	(41763)
Kshitiz Poudel	(41767)
Pilot Khadka	(41774)

Submitted To:

Department of Electronics and Computer Engineering

Thapathali Campus

Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor's Degree in Computer Engineering

Under the Supervision of

Associate Professor Er. Shanta Maharjan

April, 2024

DECLARATION

We hereby declare that the report of the project entitled **"FPGA based Accelerator Co-design using Neural Architectural Search"** which is being submitted to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, in the partial fulfillment of the requirements for the award of the Degree of Bachelor of Engineering in **Computer Engineering**, is a bonafide report of the work carried out by us. The materials contained in this report have not been submitted to any University or Institution for the award of any degree and we are the only author of this complete work and no sources other than the listed here have been used in this work.

Amit Raj Pant (THA076BCT005) _____

Bishal Rijal (THA076BCT014) _____

Kshitiz Poudel (THA076BCT018) _____

Pilot Khadka (THA076BCT025) _____

Date: April, 2024

CERTIFICATE OF APPROVAL

The undersigned certify that they have read and recommended to the **Department of Electronics and Computer Engineering, IOE, Thapathali Campus**, a major project work entitled "**FPGA based Accelerator Co-design using Neural Architectural Search**" submitted by **Amit Raj Pant, Bishal Rijal, Kshitz Poudel** and **Pilot Khadka** in partial fulfillment for the award of Bachelor's Degree in Computer Engineering. The Project was carried out under special supervision and within the time frame prescribed by the syllabus.

We found the students to be hardworking, skilled and ready to undertake any related work to their field of study and hence we recommend the award of partial fulfillment of Bachelor's degree of Computer Engineering.

_____

Project Supervisor
Associate Professor Er. Shanta Maharjan
Department of Electronics and Computer Engineering, Thapathali Campus

_____

External Examiner

_____

Project Co-ordinator
Mr. Umesh Kanta Ghimire
Department of Electronics and Computer Engineering, Thapathali Campus

_____

Mr. Kiran Chandra Dahal
Head of the Department,
Department of Electronics and Computer Engineering, Thapathali Campus

March, 2024

COPYRIGHT

ACKNOWLEDGEMENT

We are grateful to the Department of Electronics and Computer Engineering, Thapathali Campus, for equipping us with the necessary knowledge and resources to accomplish our project goals. The constant feedback and suggestions from the department's esteemed faculty members played a crucial role in the successful initiation of this project.

We would like to express our deep gratitude to Er. Krishna Gaihre and LogicTronix Technologies Pvt. Ltd. for their invaluable guidance to our project. Their support in providing the FPGA board and their expertise during the implementation phase were vital in bringing our ideas to fruition.

We would also like to acknowledge Er. Dinesh Baniya Kshatri for his invaluable contribution in the conception of this project. His insightful ideas and suggestions were instrumental in shaping the direction and scope of our project.

We would like to thank our supervisor, Associate Professor Er. Shanta Maharjan, for the invaluable guidance and support provided throughout our project.

Amit Raj Pant (THA076BCT005)

Bishal Rijal (THA076BCT014)

Kshitiz Poudel (THA076BCT018)

Pilot Khadka (THA076BCT025)

ABSTRACT

Neural Architecture Search (NAS) is an automated approach to designing task-specific Neural Networks. Previous efforts have primarily focused on optimizing the model itself, often overlooking the potential for optimizing the hardware on which the model is deployed. Our approach achieves design of the model and hardware, leveraging FPGA technology as the foundation. This study contributes to the field of image classification by highlighting the benefits of HW-NAS and FPGA-based acceleration in achieving improved efficiency and real-time performance. We explored more than 40 million architectures within the MobileNet-V3 search space using evolutionary search process to find the set of best performing architectures in terms of latency at inference, accuracy, and model size targeting the Ultra96-V2 MPSoC board. A comprehensive latency table targeting the search space was developed to support latency estimation and an accuracy predictor was utilized to filter out best performing networks. Seven architectures of varying size and latency were discovered and compared with state-of-the-art architectures. Among them, the FPGANet_L25 was able to achieve an impressive top-1 accuracy of 79.12% on ILSVRC2012 with 25ms of inference time. Additionally, our searched architectures outperformed many well established networks like EfficientNet-B0, ResNet-50, and MobileNet-V3 both in terms of inference latency (measured on FPGA) as well as accuracy.

Keywords: Evolutionary search, FPGA, HW-NAS, MobileNet-V3, Neural Architecture Search,

TABLE OF CONTENTS

DECLARATION ..... **i**

CERTIFICATE OF APPROVAL ..... **ii**

COPYRIGHT ..... **iii**

ACKNOWLEDGEMENT ..... **iv**

ABSTRACT ..... **v**

List of Figures ..... **x**

List of Tables ..... **xiii**

List of Abbreviations ..... **xiv**

1. INTRODUCTION **1**

1.1 Background 1

1.2 Motivation 2

1.3 Problem Definition 3

1.4 Objectives 3

1.5 Scope and Applications 3

2. LITERATURE REVIEW **5**

2.1 Neural Architecture Search 5

2.2 Hardware Aware Neural Architecture Search and Co-design-NAS 7

2.3 Pruning 9

2.4 Quantization 9

3. REQUIREMENT ANALYSIS **10**

3.1 Project Requirements 10

3.1.1. Software Requirements 10

3.1.2. Hardware Requirements 11

3.2 Feasibility Analysis 13

3.2.1. Technical Feasibility 13

3.2.2. Economic Feasibility 13

3.2.3. Schedule Feasibility 13

4. SYSTEM ARCHITECTURE AND METHODOLOGY **14**

4.1 System Block Diagram/Architecture 14
4.2 CNNs and Efficientnets 16
4.3 Datasets 20
4.3.1. Ultra96v2 Latency Dataset 20
4.3.2. ImageNet 1K Dataset 20
4.4 Zynq MPSOC 21
4.4.1. Processing system (PS) 21
4.4.2. Programmable Logic (PL) 23
4.4.3. AXI interface: 25
4.5 Model Compression 26
4.5.1. Pruning 27
4.5.2. Quantization 30
4.6 Xilinx DPU 37
4.7 Hardware Software Co-search for Efficient Architecture 38
4.7.1. Generation of Convolutional Search space 38
4.7.2. Search Space Exploration: Evolutionary Search Algorithm 40
4.7.3. Estimating Accuracy 41
4.7.4. OFA networks 41
4.7.5. Evaluator Design 44
4.8 Flowchart: HW-aware evolutionary search with accuracy and efficiency constraints 46

5. IMPLEMENTATION DETAILS 47

5.1. Model Compression: EfficientNet-V2 47
5.1.1. Fine tuning Efficient-Net on CIFAR-10 48
5.1.2. Pruning the Refined Model 48
5.1.3. Replacing SiLU with Hard Swish Activation 50
5.1.4. Channel Pruning 51
5.1.5. Quantization 51
5.2 Implementation on FPGA 55
5.2.1. Zynq MPSoC IP 56
5.2.2. DPUCZDX8G 59
5.2.3. Clocking Wizard IP 60
5.2.4. Processor System Reset 61

5.2.5. Implemented DPU design 61
5.2.6. Quantization and Compilation of Model 63
5.2.7. Running CNN model in the board 65
5.2.8. Implementation of Design using PetaLinux 67
5.2.9. Latency profiling of the CNN model 68
5.3 Hardware Aware Neural Architectural Search 71
5.3.1. Accuracy Predictor 71
5.3.2. Efficiency predictor 72
5.3.3. Latency Estimation using Latency Lookup Table 73
5.3.4. Evolutionary Algorithm 73
5.4 Different designs for Hardware 74
5.4.1. Comparison between the two designs: 75
6. RESULT AND ANALYSIS 79
6.1 EfficientNet-V2 on CIFAR-10 79
6.2 Unstructured Pruning 80
6.3 Channel Pruning 82
6.4 Quantization 84
6.5 Implementation of DPU on FPGA 85
6.5.1. Resource Utilization on B1024 DPU 85
6.5.2. Power analysis 85
6.5.3. Timing report 86
6.6 Deploying CNN models 87
6.7 Latency Profiling of Model 88
6.8 Search Results of Evolutionary Algorithm 91
6.8.1. Searching with Arithmetic Intensity 91
6.8.2. Searching with Latency Constraint 93
6.9 Comparison with other Architectures 97
6.10 Error Analysis on Latency 99
7. Future Enhancements 100
8. Conclusion 101
9. APPENDICES 102
Appendix A: Project Timeline 102

Appendix B: Project Budget 103
Appendix C: Implementation of Design using PetaLinux 104
Appendix D: Ultra96-v2 board 110
Appendix E: Overall Schematic diagram 111
Appendix F: Clock management tile (CMT) 112
Appendix G: Processor system reset 113
Appendix H: Used Linux Commands 114
Appendix I: Snapshot of Quantized Efficientnet-V2 116

LIST OF FIGURES

Figure 2-1: Number of papers describing HW-NAS by Dec 2020 [14] 7

Figure 3-1: Ultra96-v2 block diagram 12

Figure 4-1: Block diagram for Model compression and evaluation on FPGA 14

Figure 4-2: Block Diagram of Latency aware Neural architectural search for FPGAs 15

Figure 4-3: Inspiration behind CNNs 16

Figure 4-4: Convolution operation in CNNs 17

Figure 4-5: MBConv and Fused-MBConv block 18

Figure 4-6: MPSOC block diagram 21

Figure 4-7: Simplified Diagram of Zynq MPSoC device architecture [22] 22

Figure 4-8: Logic Cells in FPGA 23

Figure 4-9: Block diagram of function X=A'B'C+ABC' 24

Figure 4-10: AXI read transaction uses read address and read data channels. 25

Figure 4-11: AXI write transaction uses write address, data, and response channels. 26

Figure 4-12. Model Pruning 27

Figure 4-13: Structured and unstructured Pruning 28

Figure 4-14: Channel pruning and filter pruning 28

Figure 4-15: Pruning and Refining 29

Figure 4-16: Example of weight distribution of convolution layer [24] 30

Figure 4-17: Layer wise and Channel wise quantization 34

Figure 4-18: Uniform and non-uniform quantization 36

Figure 4-19: Quantization process 37

Figure 4-20: Mobilenetv3 block 38

Figure 4-21: OFA Supernet 42

Figure 4-22: Kernel weight sharing 43

Figure 4-23: Progressive Shrinking of OFA 43

Figure 4-24: Flowchart Evolutionary Search....46
Figure 5-1: Sample images from CIFAR-10 47
Figure 5-2: Parameter distribution in EfficientNetV2 49
Figure 5-3: Comparison of HardSwish, SiLU and ReLU....50
Figure 5-4: Differentiability of Swish and Hardswish....51
Figure 5-5: Quantization example 54
Figure 5-6: Straight through Estimator example 55
Figure 5-7: MPSoC Top level block diagram....56
Figure 5-8: MPSoC IP Block Diagram....57
Figure 5-9: DPU IP block 59
Figure 5-10: Clocking Wizard IP Block 60
Figure 5-11: Processor System Reset IP block....61
Figure 5-12: Implemented DPU Design 62
Figure 5-13: Quantization and compilation for deployment in FPGA 63
Figure 5-14: Running CNN model in the MPSoC....66
Figure 5-15: Overall latency profiling of the CNN model 69
Figure 5-16: Layer-wise latency profiling of model using Vitis Profiler....70
Figure 5-17: Cross over and Mutation....74
Figure 5-18: Resource utilization of design-1 76
Figure 5-19: Resource utilization of design-2 76
Figure 5-20: Power consumption of design-1....77
Figure 5-21: Power consumption of design-2....77
Figure 6-1: Training and validation loss across epoch 79
Figure 6-2: Training and validation accuracy evolution over epochs....79
Figure 6-3: Pruning the parameters and recovering the accuracy....80
Figure 6-4: Parameter distribution of 81% sparse model 81
Figure 6-5: Illustration of reduction in channels after pruning....82

Figure 6-6: Efficiency Improvements after channel pruning.....83

Figure 6-7: Accuracy comparison of original and pruned model.....83

Figure 6-8: Power analysis of implemented design.....86

Figure 6-9: Timing report of implemented design.....87

Figure 6-10: CNN model for layer-wise latency profiling.....89

Figure 6-11: Searching with arithmetic intensity constraint.....92

Figure 6-12: Our Search architectures vs existing models.....93

Figure 6-13: Networks with (25ms, 20ms, 15ms, 13ms, 12ms, 11ms, 10ms).....95

Figure 6-14: Accuracy vs MACs and latency of different models discovered through search process.....96

Figure 6-15: Comparison of our searched model with exiting work in terms of accuracy and number of parameters.....97

Figure 6-16: Top-1 and top-5 accuracy of our searched models.....97

Figure 6-17: Top-1 Accuracy vs Latency measured on FPGA.....98

Figure 9-1: Boot Mode Location in Ultra96v2.....108

Figure 9-2: UART connection in Ultra96v2.....109

LIST OF TABLES

Table 3-1: Ultra96-v2 attributes.....13

Table 4-1: Summary of EfficientNetV2 architecture.....19

Table 4-2: Implementation of X = A'B'C+ABC' 24

Table 4-3: Model size and accuracy tradeoff for sparse-InceptionV3.....29

Table 4-5: Example Resource used by ZCU102 in different architectures 38

Table 5-1: PS-PL interface used in MPSoC IP.....58

Table 5-2: PS-PL interface used in DPU IP.....60

Table 6-1: Sparsity and decrease in accuracy.....81

Table 6-2: Parameters pruned in different superblocks of the network.....82

Table 6-3: Quantization result 84

Table 6-4: Implemented design Resource Utilization 85

Table 6-5: Inference time of different Models.....88

Table 6-6: Layer-wise Profiling of Blocks 89

Table 6-7: Extracted Information Summarization.....90

Table 6-8: FPGA aware NAS search results with varying parameters.....96

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
APU	Application Processor Unit
ASIC	Application Specific Integrated circuit
AXI	Advanced extensible Interface
BRAM	Block RAM
CLB	Configurable Logic Blocks
CNN	Convolutional Neural Network
DDN	Deep Neural Network
DPU	Deep Processing Unit
DSP	Digital Signal Processing
EA	Evolutionary Algorithm
EcoNAS	Economical evolutionary-based NAS
FF	Flip-Flop
FIFO	First In First Out
FLOPS	Floating Point Operations per second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HW-NAS	Hardware Neural Architecture Search
IOB	Input/Output Blocks
IOT	Internet of Things
IP	Intellectual Property
LUT	Look up Table
MAC	Multiply Accumulate
MLP	Multi-Layer Perceptron
MPSoC	MultiProcessor System On Chip

NAS	Neural Architecture Search
OFA	Once For All
PL	Programmable Logic
PS	Processing System
PTQ	Post Training Quantization
QAT	Quantization Aware Training
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RPU	Real Time Processor Unit
STE	Straight Through Estimator
UART	Universal Asynchronous Receiver-Transmitter
VHDL	Very High-Speed Integrated Circuit Hardware Description Language

1. INTRODUCTION

1.1 Background

The traditional approach to designing neural network architectures presents a significant challenge due to the need for specialized expertise and requires a significant amount of time. Creating an efficient architecture involves a deep understanding of the problem domain, network architecture principles, and the intricacies of different layers and connections within the network. It involves many hits and trials and requires designers to test multiple configurations such as number of layers, regularization technique, and activation functions to achieve optimal performance.

In recent years, there has been extensive research conducted on NAS (Neural Architecture Search). Its profound influence can be primarily observed in the domains of image classification and object detection tasks, where it has yielded state-of-the-art results. By automating the architecture search process, NAS can save significant time and effort, while also enabling the exploration of unconventional or complex architectures that might have been overlooked by human designers.

Previous research efforts have illustrated that NAS possesses the capability to produce deep neural networks (DNNs) that exhibit comparable, and sometimes superior, accuracy compared to manually designed models like AlexNet, VGGNet, GoogleNet, and ResNet. Current NAS methods can automatically find optimized neural architectures by improving their software-related hyper parameters such as the number of channels, the number of layers and topological connections. A lot of developments have occurred in NAS recently. Google has just announced LayerNAS which reformulates the multi-objective NAS problem within the framework of combinatorial optimization to reduce the complexity and improve performance. But most of them, including LayerNAS are focused on GPU or CPU platforms.

Despite the commendable progress achieved so far, the practical application of NAS to real-world problems continues to present substantial challenges and remains limited in its widespread feasibility. In general, CNN architectures are considered too complex for resource-limited platforms like IoT, mobile, and embedded systems. Their complexity presents challenges when it comes to deploying them efficiently on such devices.

NAS has indeed demonstrated its effectiveness by introducing cutting-edge models for Object Detection and Image Classification. However, these models usually contain millions and billions of parameters (FLOPs) which makes it challenging to meet the memory requirements and computational burden in resource limited environments. Additionally, deploying these models in a timely manner or for real-time applications often necessitates specialized hardware such as GPUs or TPUs.

Field-programmable gate arrays (FPGAs), due to their configurability and high energy efficiency, are becoming increasingly popular in the accelerating CNNs. The underlying hardware architecture plays an important role in designing cost-effective Deep Neural Networks (DNNs).

Codesign-NAS has emerged as one of the most promising techniques to automatically generate efficient CNN models accomplishing acceptable accuracy-performance trade-offs. The traditional approach to NAS involves searching for the optimal neural network architecture through techniques like reinforcement learning, evolutionary algorithms, or Bayesian optimization. These methods consist of vast search space of candidate architectures, evaluating each candidate's performance on predefined task or dataset. However, they usually neglect the hardware search space and are computationally very expensive. Co-design-NAS algorithms explore the search space of a CNN by jointly optimizing architecture as well as hardware search space. By considering these constraints, co-design can prominently identify architectures that are not only accurate but also optimized for a particular hardware configuration of devices like FPGA.

1.2 Motivation

Traditional approaches to designing CNN model for image classification involve manual exploration of architectural configurations, which can be time-consuming, resource-intensive, expensive, and may not lead to optimal designs. Since image classification is the base line of computer vision which can be differentiate into many application domains like Object detection, Facial recognition, Medical imaging and so on. Our project aims to leverage the power of Hardware aware NAS techniques that allow us to automate the design process considering not only the accuracy but also consider the limitations of edge device like FPGAs. Through different NAS techniques, this project can efficiently explore a vast design space, considering various architectural configurations and optimizing for performance, accuracy, and resource utilization.

1.3 Problem Definition

The architecture of the neural network depends on the application. So, the architecture developed for one application may not work on other application. The architecture needs to be designed for every application. Furthermore, deep knowledge about neural networks is required to develop effective Networks. This type of knowledge may not be with everyone. Hence, Neural Architecture search is a technique to design architecture automatically.

Another problem is the resource intensiveness of Neural Networks such as Convolutional Neural Networks (CNNs) in resource limited devices such as mobile and embedded systems. By considering the latency and available memory of the hardware device, NAS can be used to develop efficient and effective architecture tailored for that hardware.

1.4 Objectives

The main objectives of our project are listed below:

- To optimize and implement EfficientNetV2 for inferencing on FPGA.
- To find the best Pareto frontier model-accelerator pair on FPGA through NAS and evaluate its performance in terms of latency, storage and accuracy.

1.5 Scope and Applications

This project's scope includes developing an image classification system that leverages hardware acceleration through FPGA and neural architecture search (NAS) techniques. The project involves implementing a Co-design NAS framework based on FPGA which includes optimization of neural network architectures and hardware itself. The focus is on achieving high-performance and efficient image classification while considering real-time constraints.

Applications of the project are as follows:

- System can be deployed in surveillance cameras or security systems to detect and classify objects in real-time, enabling treat detection.
- Establishes a systematic workflow using FPGA hardware acceleration and Neural Architecture Search (NAS) for the development of efficient image

classification systems tailored for edge devices.

- Can be used to deploy efficient architectures optimized for FPGA on the fly.
- The generated latency table serves as a valuable resource for future researchers and students in the Hardware-Neural Architecture Search (Hw-NAS) sector, offering insights for further exploration.

2. LITERATURE REVIEW

Deep learning techniques have achieved success in many domains such as object detection and natural language processing [1]. The performance of the DNNs depends on their architecture and weights. They have parameters in millions and have succeeded in solving many problems from image classification, to solving dynamic analysis, power flow calculation, anomaly detection, and natural language processing. [2]. But the search process for finding an excellent neural architecture that works well in the domain is hard, time consuming and requires extensive knowledge.

2.1 Neural Architecture Search

The goal of Neural Architecture Search is to find architectures that achieve high performance on a given task while minimizing the need for manual intervention, and tedious task of selecting among hundreds of parameters and checking which could possibly perform best. NAS algorithms typically operate by searching through a large space of architectures, evaluating their performance, and iteratively improving them. By leveraging automated techniques, NAS has already surpassed the performance of even the most sophisticated human-designed architectures across various tasks.

The early use of NAS has been mostly on image classification but now it is being widely researched on to use for object detection, image segmentation, partial differential equation solving, protein folding, and NLP. [3] Many benchmark datasets have also been created to evaluate the efficiency of the searched architectures and to make NAS research more robust and scientific. [4] [5]

Historically NAS dates to the late 1980s, when for the first time neurogenetic learning algorithm was introduced which provided an integrated method to design and train the neural networks [6]. However, it wasn't until the publication of the influential paper "NAS with Reinforcement Learning" [7] by Zoph and Le in 2017 that NAS gained significant recognition and widespread attention.

They employed recurrent networks to generate descriptions of neural network models, and reinforcement learning to train the network. They explored almost 12,800 architectures until they succeeded in finding the best validation accuracy. They successfully demonstrated NAS can be used to design novel neural architectures that

outperform the hand designed architectures on CIFAR-10 dataset. The model had a test error rate of 3.65, and it was 0.09 percent better and 1.05x faster than the previous state of the art model at that time outperforming the previous state-of-the-art models.

Later in 2018, a modified evolutionary algorithm called AmoebaNet-A [8]was able to surpass hand-designed classifiers for the first time. By introducing an age property to favor younger genotypes, AmoebaNet-A achieved comparable accuracy to state-of-the-art ImageNet models. When scaled up, it achieved an impressive 83.9% top-1 / 96.6% top-5 ImageNet accuracy, setting a new state-of-the-art benchmark. In comparison to a reinforcement learning algorithm, the evolutionary approach demonstrated faster results with the same hardware, making it an effective method for discovering high-quality architectures, especially when computing resources are limited.

At this point, most of the past work in NAS was focused on the use of Evolutionary Algorithm (EA) [9] and Reinforcement Learning (RL). The problem is NAS by reinforcement leaning and evolutionary search can take a lot of time ranging from hundreds of GPU days to thousands of days. To overcome this researcher started to explore new techniques and came up with solutions such as Bayesian optimization, and NAS-specific techniques based on weight sharing, and one-shot learning.

In an attempt to reduce the time taken by NAS, Pham et al. introduced ENAS [10] which used weight sharing paradigm in order to reduce the computational cost. They introduced the concept of supernet which involves parameter sharing among the child architectures during the search process. Notably it was 1000x faster than all the existing works in terms of GPU hours.

Later came concepts like one-shot NAS. The original one-shot NAS technique had some issues. A recent work on one-shot NAS introduced the concept of PGONAS [11]. The main improvement of PGONAS over one-shot are architecture enhancement, improved sampling strategies, and prior-based regularization.

A family of models called EfficientNets were developed using the NAS technique [12].The architecture uses mobile inverted bottleneck convolution like MobileNet V2. They used scaling method that scales all dimensions using compound coefficient. The EfficientNet-B7, among other EfficientNets, achieved 84.3% top-1 accuracy on ImageNet. The first EfficientNets had some drawbacks, the authors noticed them, and

later in 2021 used training-aware neural architecture search and scaling technique to develop more optimized family models and named them EfficientNetV2 [13]. They are optimized for faster training and inferencing. In Imagenet21K their model achieved remarkable accuracy of 87.3%.

2.2 Hardware Aware Neural Architecture Search and Co-design-NAS

By now, considerable progress has been made in NAS, with techniques introduced to make search and evaluation effective. However, most work only focused on the software aspect of NAS, neglecting the vital considerations of hardware constraints that will be used to implement the architecture. Notably very few papers considered this aspect.

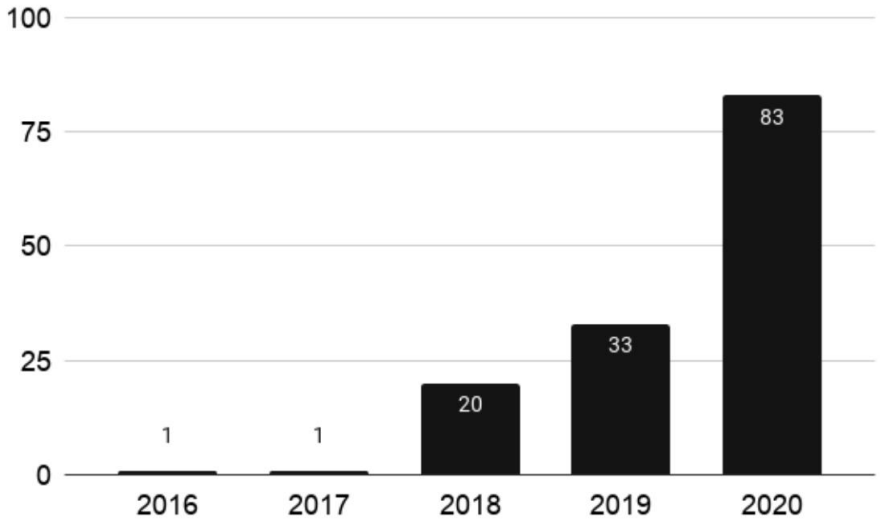


Figure 2-1: Number of papers describing HW-NAS by Dec 2020 [14]

HW-NAS emerged in 2017 and has since achieved state-of-the-art (SOTA) results in resource-constrained environments. Cai et al. introduced Once-for-all (OFA) [15], a method to train network once and deploy across many devices and resource constraints, especially on edge devices. As per the paper OFA can acquire an unexpectedly high quantity of sub-networks (over 10^19) that can adapt to various hardware platforms and latency limitations, all while maintaining the same level of accuracy as when trained individually.

DPP-Net [16] was the first device-aware neural architecture search approach

outperforming state-of-the-art handcrafted mobile CNNs. They succeeded in finding Pareto-optimal networks for CIFAR-10. In comparison to CondenseNet and NASNet (Mobile), DPP-Net demonstrated superior performance in terms of both accuracy and inference time across a range of devices. Furthermore, DPP-Net was able to achieve state-of-the-art performance on the ImageNet dataset.

In 2019, Jiang et al. introduced an innovative hardware-aware Neural Architecture Search (HW-NAS) framework called FNAS [17]. This work made a notable contribution to the field by utilizing Field Programmable Gate Arrays (FPGAs) to explore and optimize neural architectures that adhere to specific hardware constraints. FNAS represented a significant milestone as it was the first to implement NAS on FPGA-based platforms. By incorporating hardware awareness into the search process, the researchers effectively reduced the search space and focused on identifying architectures that met desired latency specifications. The hardware and software co-exploration framework of FNAS facilitated an efficient search process by concurrently considering both aspects. By pruning architectures that failed to meet latency constraints, the search was refined, leading to the discovery of superior architectures. The pioneering work of FNAS exemplifies the significance of considering hardware constraints in the quest for optimal neural architectures. They were able to impressive results, generating architectures that met latency specifications with less than 1% accuracy loss, despite being 7.81 times faster than state-of-the-art methods. By using hardware constraints, they accelerated the search process up to 11.13 times. Though they only considered latency as hardware specification, the work laid the foundation for future exploration.

Later, Jahanshahi came up TinyCNN [18], a CNN accelerator on FPGA designed specifically for image classification. He addressed the computational and resource limitations faced when integrating CNN-based methods into embedded systems. Though his work did not include HW-NAS, he managed to implement CNN in FPGA with a slight 3% accuracy loss and significant speedup of up to 15.75×.

In 2021 in paper [14] Benmeziane et al. presented a novel contribution in the form of a comprehensive review focused on HW-NAS. They categorize existing HW-NAS research based on four key dimensions: search space, search strategy, the acceleration technique, and the hardware cost estimation strategies.

2.3 Pruning

The early work on pruning was by Han et al. (2015) [19] when he introduces the innovative concept of magnitude-based pruning. Their approach involved utilizing L1-norm regularization to identify and eliminate unimportant weights in a neural network. Later techniques like structured pruning emerged as powerful methods for eliminating redundant structures in neural networks, offering significant model size reduction without compromising performance. Instead of pruning individual weights or neurons, structured pruning aims to remove entire filters, channels, or layers from the network while preserving its original structure. This approach offers better interpretability and allows for more efficient hardware implementation. For instance, He et al. (2018) proposed a method called "ThiNet" [20] that leverages group sparsity regularization to prune filters in CNNs. Their approach achieved remarkable model compression with a minimal impact on accuracy.

2.4 Quantization

Network quantization is another effective technique for compressing large networks. In a paper titled "A White Paper on Neural Network Quantization" [21] Nagel et al. introduced state-of-the-art algorithms, namely Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT), to address the accuracy degradation caused by quantization noise. They introduced new cutting-edge algorithms aim to mitigate the impact of quantization noise on a network's performance while maintaining low-bit weights and activations. Their work presents tested pipelines that were based on a thorough review of existing literature and extensive experimentation, establishing their work as state-of-the-art in the field.

3. REQUIREMENT ANALYSIS

3.1 Project Requirements

We will need both software components and hardware components to complete our project. Software components will include tools to build the project and hardware components will be needed to deploy the project.

3.1.1. Software Requirements

Verilog

Verilog is another popular hardware description language used for designing and describing digital circuits, including those implemented on FPGAs. It might be needed to implement HLS unsupported function in FPGA.

PyTorch

PyTorch is a popular deep-learning framework that provides a Python-based interface for building and training neural networks. PyTorch is required for implementing deep learning models, visualizing them and for quantization and pruning.

ONNX Tools

ONNX (Open Neural Network Exchange), an open standard format, was used to compare the inference time of quantized and non-quantized variants of EfficientNetV2. Both models were converted to ONNX format for a framework-agnostic representation, ensuring seamless interoperability. The ONNX runtime was used to execute the models on identical input data, allowing a direct comparison of their inference efficiency. This use of ONNX enabled the exploration of model optimization techniques, evaluating the trade-offs between model size reduction through quantization and potential inference speed improvements. ONNX tools were also used to measure the Multiply-Accumulate Operations (MACs) of both models.

FPGA Development Tools

Xilinx Vivado is a popular development tool for FPGA design and implementation. They offer a complete development environment for designing, simulating, and synthesizing FPGA circuits. It will be required for timing simulation which allows to

validate architecture's functionality and performance. It enable the conversion of the optimized neural network architecture into an FPGA-compatible hardware representation. It also supports integrating Xilinx's Intellectual Property (IP) cores into our design.

Xilinx Vitis AI

Vitis AI is a comprehensive development platform provided by Xilinx for accelerating and deploying AI inference on Xilinx FPGA and SoC (System-on-chip) devices. It simplifies the development and deployment of AI models by providing a unified framework that integrates with deep learning frameworks. It will help to convert trained models like EfficientNetV2 into an optimized format for deployment on Xilinx FPGA. It offers tools for quantizing and optimizing EfficientNetv2 or NAS final output model to achieve better performance and resource optimization on Ultra96-V2. It also seamlessly integrates with the Xilinx Deep Learning Processing Unit (DPU), which is a dedicated hardware accelerator designed to accelerate deep learning inference tasks on FPGAs. The DPU IP cores can be easily integrated into the FPGA design using Vitis AI tools, enabling efficient deployment of EfficientNetV2 and NAS final output model on FPGA.

PetaLinux

PetaLinux is an essential tool for implementing deep learning models in FPGA-based systems. It empowers developers to create custom Linux distributions tailored to their FPGA hardware platforms. We will utilize PetaLinux to write bitstream to the FPGA board, enabling us to utilize the full potential of our FPGA-based system and efficiently without having to deal with huge burden to flash FPGA.

3.1.2. Hardware Requirements

USB Camera

A USB camera will be required to capture the real time image and then feed it to the model implemented on FPGA for inferencing. The camera will play a crucial role by capturing the visual data, which will be then fed into the model implemented on an FPGA for inferencing. This process allows for the analysis and interpretation of the captured images, facilitating the project's objectives and desired outcomes. For interfacing the camera, Xilinx AXI streaming module will be used.

Ultra96-V2 FPGA

Field-Programmable Gate Arrays (FPGAs) are highly versatile integrated circuits that provide developers with the ability to create custom digital circuits tailored to their specific application requirements. Unlike traditional Application-Specific Integrated Circuits (ASICs), FPGAs can be reprogrammed and reconfigured, making them a popular choice in a wide range of industries, including telecommunications, automotive, aerospace, and many others.

The Ultra96-V2 MPSoC (Multi-Processor System on Chip) is a powerful development board created by Avnet in collaboration with Xilinx. It serves as a versatile platform for developing and prototyping applications in fields such as artificial intelligence, machine learning, robotics, and more.

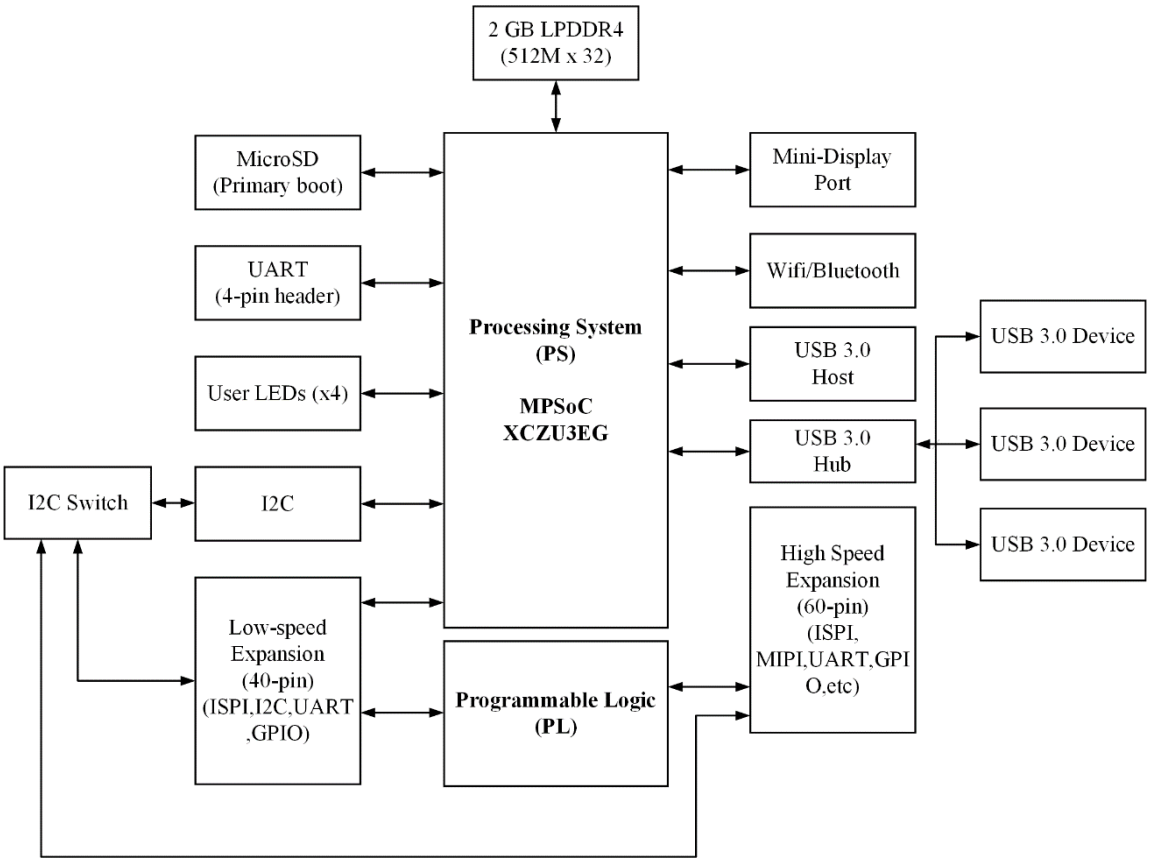


Figure 3-1: Ultra96-v2 block diagram

The datasheet can be used to learn about the parameters of such as the number and types of logic elements, including Logic Cells and DSP slices. Information about the

embedded memory resources, such as Block RAM and distributed RAM, is also provided, enabling designers to assess the memory capacity for their designs.

Table 3-1: Ultra96-v2 attributes

Ultra96-v2 (XCZU3EG)	Resources
System Logic Cells	154,350
Equivalent logic cells	70,560
Distributed RAM (Mb)	1.8
Block RAM	216
DSP slices	360

3.2 Feasibility Analysis

3.2.1. Technical Feasibility

The project's technical feasibility depends on the availability of FPGA development boards, FPGA design tools, and software frameworks for training and optimization. The necessary knowledge and skills for this project are FPGA programming, understanding of ML and DL, and NAS techniques. The success of our project is crucial in access to computational resources for training and experimentation.

3.2.2. Economic Feasibility

The cost includes the cost of acquiring FPGA development boards and computational resources. Additionally, any necessary hardware peripherals, such as memory interfaces or image input/output modules, could also increase the cost. Assessing the budget constraints and identifying potential sources of funding or cost-saving measures may enhance the economic feasibility of our project.

3.2.3. Schedule Feasibility

This project can be completed in pre-scheduled time if the project schedule is followed strictly. So, this project is still feasible considering the time factor. Detailed scheduling of project is placed in appendix in project timeline.

4. SYSTEM ARCHITECTURE AND METHODOLOGY

4.1 System Block Diagram/Architecture

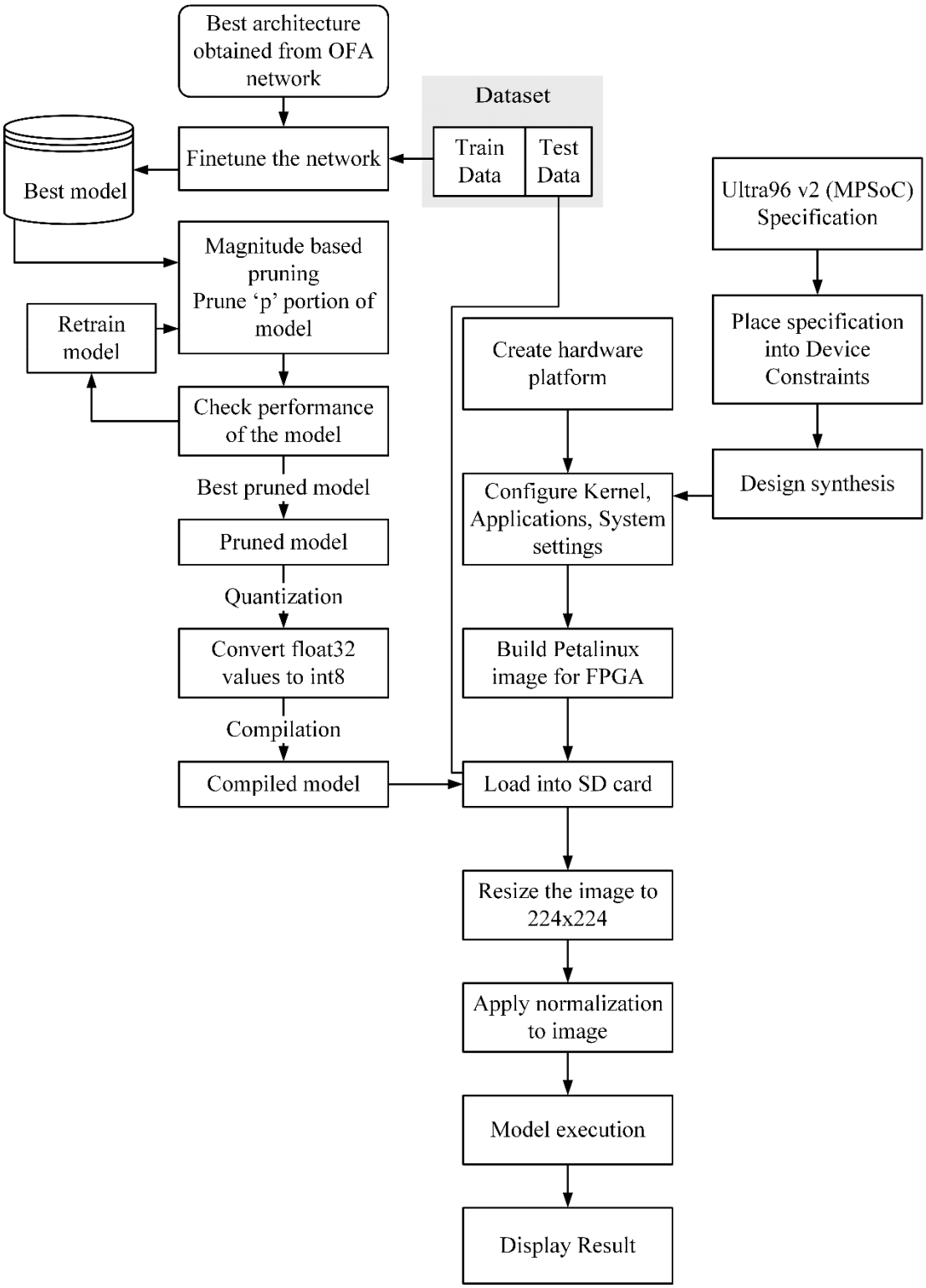


Figure 4-1: Block diagram for Model compression and evaluation on FPGA

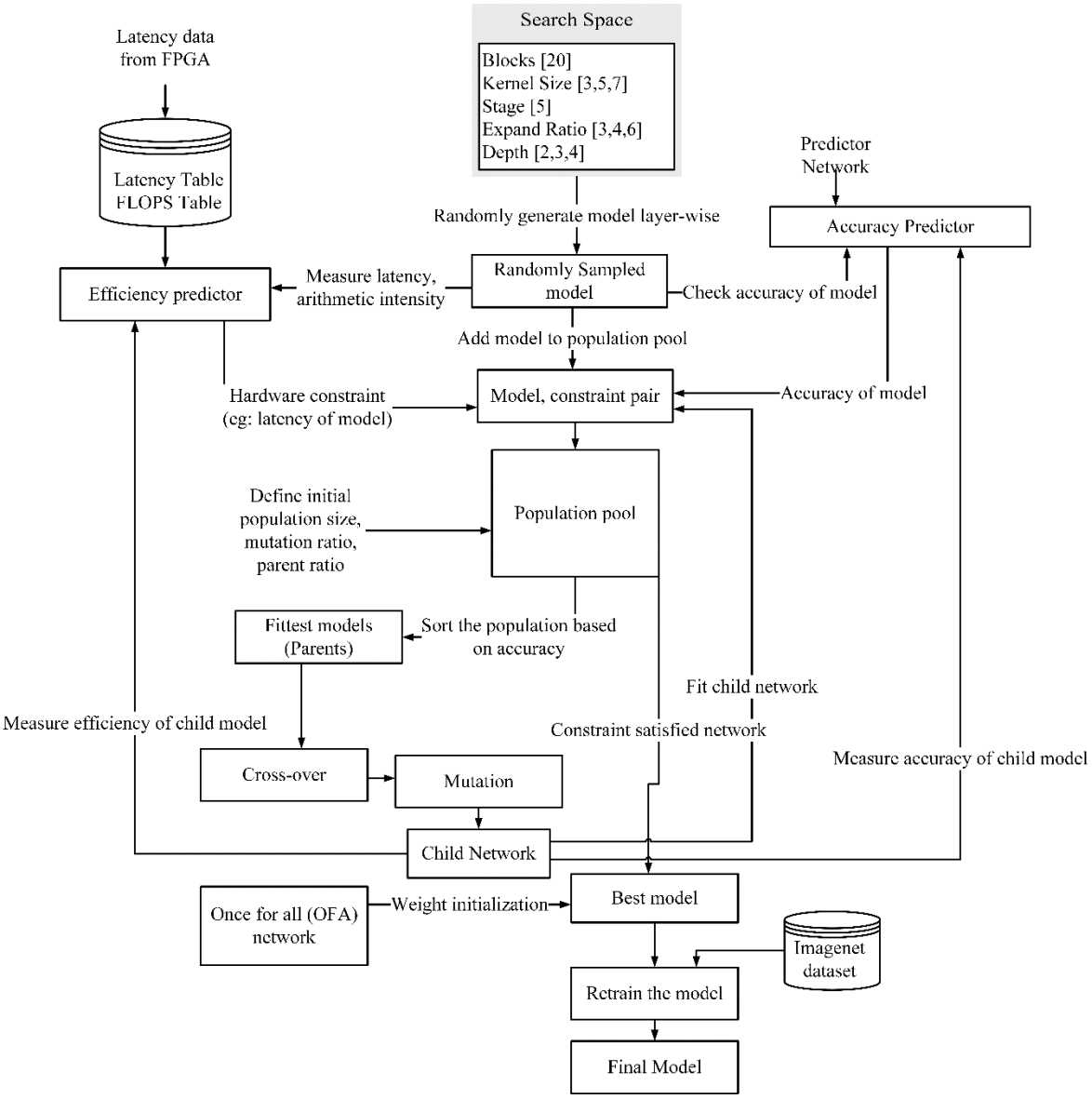


Figure 4-2: Block Diagram of Latency aware Neural architectural search for FPGAs

4.2 CNNs and Efficientnets

Convolutional neural networks (CNNs) and related architectures are the go-to architecture for computer vision tasks. They are heavily inspired by our own visual cortex. Computer scientists once studied the human brain and then transferred that knowledge to computers. Around the 1960s, two researchers, Hubel and Wiesel, performed a series of experiments on the brains of half-awake cats. They showed different shapes to the cats and noticed that one neuron, now known as the "Hubel and Wiesel," was active for specific cells. During their research, they also discovered two types of cells in the visual cortex: simple cells, which detect features, and complex cells, which have a larger receptive field and can summarize the output of simple cells. This groundbreaking work laid the foundation for the development of Convolutional Neural Networks, computer scientist took inspiration from this work revolutionizing computer vision and pattern recognition tasks.

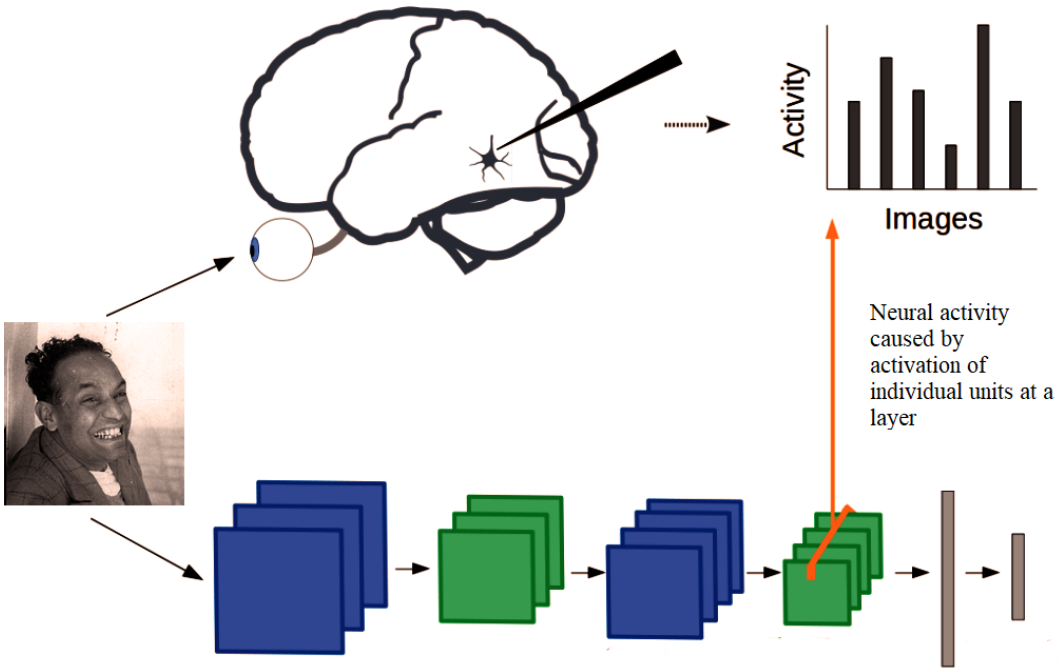


Figure 4-3: Inspiration behind CNNs

The main useful features of CNNs are shift invariance and parameter sharing which make them useful for tasks like image classification, object detection, segmentation and many more.

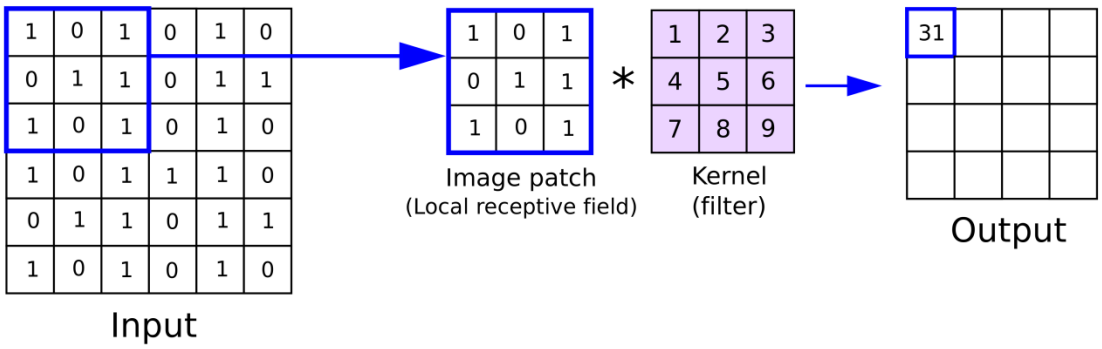


Figure 4-4: Convolution operation in CNNs

Typical CNN architecture consists of a stack of convolution, pooling, and activation layers, and finally fully connected dense layers. Different permutations and combinations of these layers, kernel sizes, pooling operations, regularization, normalization, and a few other techniques have resulted in different architectures.

EfficientNets

EfficientNets is a family of similar architectures. These architectures have comparatively better parameter efficiency and training speed than existing models. The idea of EfficientNet is to create a well-designed optimal baseline architecture block and scale it multiple times to create a bigger architecture. The baseline architecture was obtained from Neural Architecture Search (NAS). EfficientNet also introduced the concept of compound scaling, which means scaling depth, width, and resolution of input images together in a standard way.

α, β, and γ are the scaling constant factors for depth, width, and image resolution.

$$depth: d = \alpha^{\varphi}$$
$$width: w = \beta^{\varphi}$$
$$resolution: r = \gamma^{\varphi}$$
$$\text{where } \alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

Increasing depth increases FLOPS by double the amount, while increasing width and resolution makes FLOPS increase by four times. Therefore, compound scaling has an effect of $\alpha.\beta^2.\gamma^2$ on the FLOPS. By changing the ∅ parameter according to available

resources and performing grid search with constraints, the optimal value of scaling constants is determined.

EfficientNet-v2 further adds training-aware NAS-based search, including smaller models for faster training and better accuracy. It also fuses depthwise convolution and 1x1 convolution from MbConv to create a fused layer. This was done as the GPU utilization of MBConv layer used in EfficientNetV1 was very less.

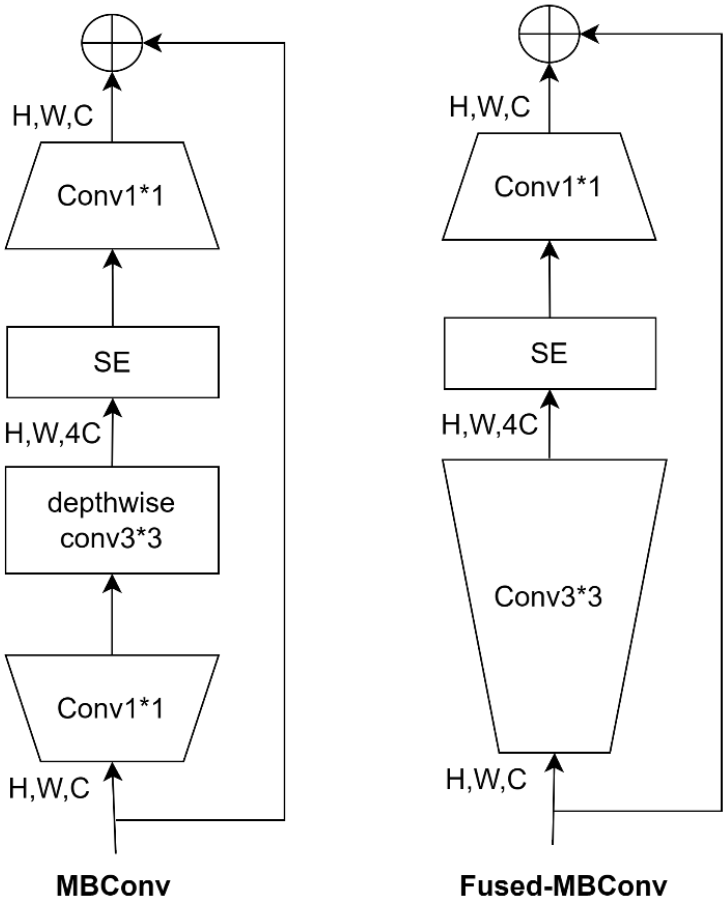


Figure 4-5: MBConv and Fused-MBConv block

The MVconv and FusedMVconv have elements of previous architectures like the squeeze and excitation form SEnet and skip connections from ResNet. The correct combination of these layers makes the EfficientNetV2 architecture.

Table 4-1: Summary of EfficientNetV2 architecture

Stage	Operator	Stride	Channels	Layers
0	Conv3x3	2	24	1
1	Fused - MBConv1, k3x3	1	24	2
2	Fused - MBConv4, k3x3	2	48	24
3	Fused - MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Calculating MAC Operations for 2D Convolutional Layers

In convolutional neural networks (CNNs), understanding the computational cost of convolutional layers is crucial for optimizing model performance. MAC operations (Multiply-Accumulate) are used to quantify this computational cost.

The total MAC operations for a 2D convolutional layer can be computed using the following formula:

$$MAC\_ops = C\_in * C\_out * K\_h * K\_w * H\_out * W\_out \quad (4.1)$$

Where,

C_in: Number of input channels (input feature maps).
C_out: Number of output channels (output feature maps).
K_h: Height of the convolutional kernel/filter.
K_w: Width of the convolutional kernel/filter.
H_out: Height of the output feature map.
W_out: Width of the output feature map

4.3 Datasets

4.3.1. Ultra96v2 Latency Dataset

Ultra96v2 Latency Dataset consists of latency of each constituent block within the MobilenetV3 search space. To obtain latency information for all possible parameter combinations, inference was executed on each block. The dataset consists of metrics, including number of runs, minimum time, maximum time, and average time for each layer during execution. Thus the dataset give the latency of each individual blocks in MobilenetV3 search space which is utilized on our NAS process.

Dataset Sample

'Kernel Name': ResidualBlock-in_96x14x14-out_136-k_7-e_3-s_1-act_h_swish-use_se_True

'Number Of Runs': 500

'CU Full Name': 'DPUCZDX8G_1:batch-1'

'Minimum Time (ms)': 0.708

'Average Time (ms)': 0.719

'Maximum Time (ms)': 0.868

Workload(GOP): 0.032

'DPU Performance(GOP': {s): 10.596}

'Mem IO(MB)': 0.162

'Mem Bandwidth(MB': {s): 244.496}

4.3.2. ImageNet 1K Dataset

The ImageNet dataset is a large-scale visual database designed for use in visual object recognition software research. ImageNet was created by researchers at Stanford University and Princeton University and has been popularly used in the development and evaluation of deep learning models for image classification, object detection, and other computer-vision tasks. The Imagenet1k consists of images from 1000 different classes along with their labels. This dataset is utilized to train the searched network that was discovered using search process.

4.4 Zynq MPSOC

Traditional FPGA consists of only programmable logic. However, the Zynq architecture has two parts: PS (Processing System) and PL (Programmable Logic). The two are connected by AXI (Advanced eXtensible Interface) standard, an on chip communication standard developed by Arm.

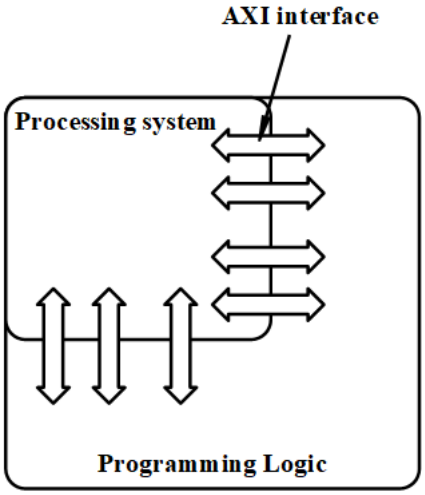


Figure 4-6: MPSOC block diagram

4.4.1. Processing system (PS)

The processing system consists of a powerful Arm-based processor, in addition to several other essential components, which are outlined briefly below. The Ultra96-v2 board incorporates a Quad-core Arm-based processor and a dual-core Real-Time Processing Unit (RPU). Moreover, it includes a dedicated GPU to provide graphics support. The processing system supports operating systems, such as Linux stack, which offers a wide range of OS services, enabling developers to focus on their tasks without having to worry about intricate implementation details.

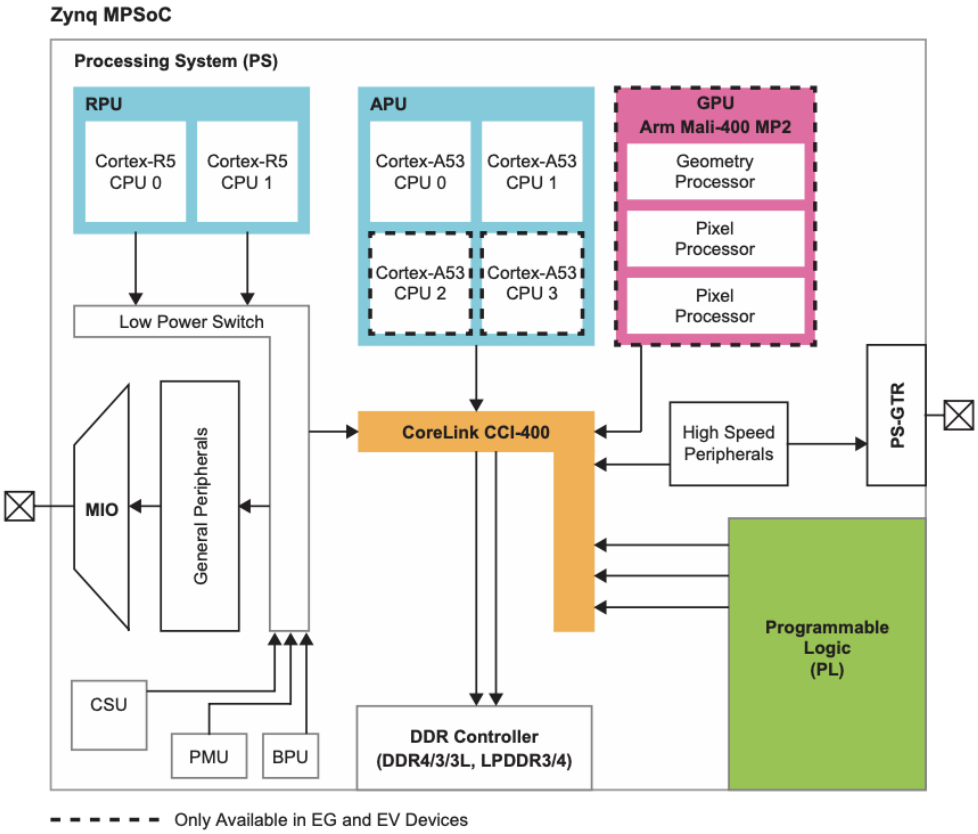


Figure 4-7: Simplified Diagram of Zynq MPSoC device architecture [22]

The Processing system part of the MPSoC consists of:

Application Processor Unit (APU): It consists of Quad core Arm cortex processor along with 256 kb of on chip cache memory. Each processor is equipped with a Floating-Point Unit (FPU), Media Engine, etc, and a dedicated L1 cache per core. The L1 cache serves to store frequently accessed data efficiently. It also incorporates an L2 cache that provides additional storage capacity.

Real Time Processing Unit (RPU): The RPU is composed of dual core Arm processors specifically designed for real-time applications. These processing cores support floating-point arithmetic operations in both single and double precision.

Interconnects and memory interface - This component facilitates communication between the Processing System (PS) and Programmable Logic (PL) sections of the MPSoC.

I/O peripherals - The MPSoC is equipped with a range of I/O peripherals, including

standard interfaces like USB and UART, among others, enabling efficient connectivity with external devices.

4.4.2. Programmable Logic (PL)

The PL part consists of Slices and Configurable Logic Blocks (CLBs). The CLBs are placed in a two dimensional array in the FPGA fabric. The CLBs are placed in a switch matrix that connect the CLBs.

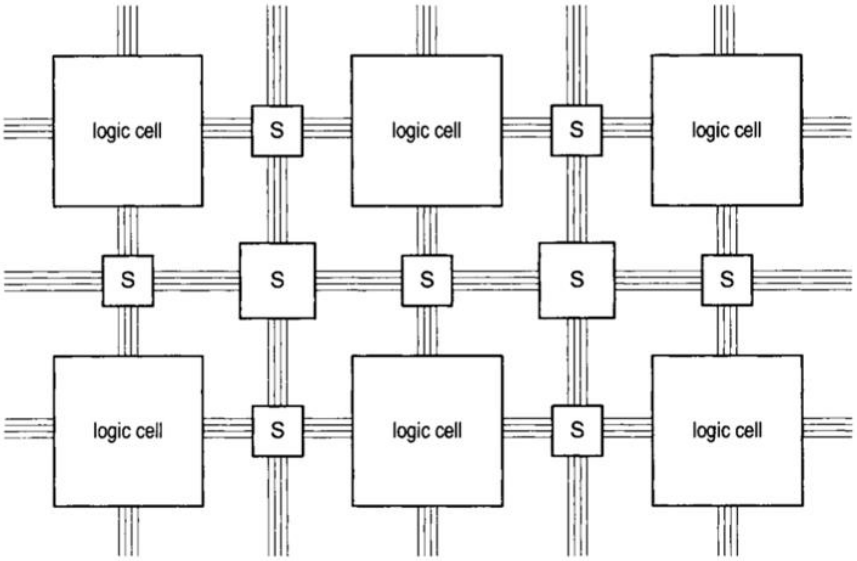


Figure 4-8: Logic Cells in FPGA

For the MPSoC architecture, a Configurable Logic Block (CLB) consists of a single slice, which includes eight 6-input Look-Up Tables (LUTs), 16 Flip-Flops (FFs), and routing logic. In addition, to implement larger arithmetic circuits, adjacent CLBs can be connected using carry logic. The number of memory locations accessed by the LUT table is 2^N, where N represents the number of inputs. This property allows the LUT to implement an impressive range of functions, specifically $2^{2^{\wedge N}}$.

If we were to implement a function X = A'B'C + ABC' using 4 input LUT, the input variables A, B, and C correspond to the data inputs data1, data2, and data3, respectively. The fourth input, data4, is considered a don't care, meaning it does not affect the function's output.

Table 4-2: Implementation of X = A'B'C+ABC'

Data 1(A)	Data 2(B)	Data 3(C)	Data 4	LUT output
0	0	0	x	0
0	0	1	x	1
0	1	0	x	0
0	1	1	x	0
1	0	0	x	0
1	0	1	x	0
1	1	0	x	1
1	1	1	x	0

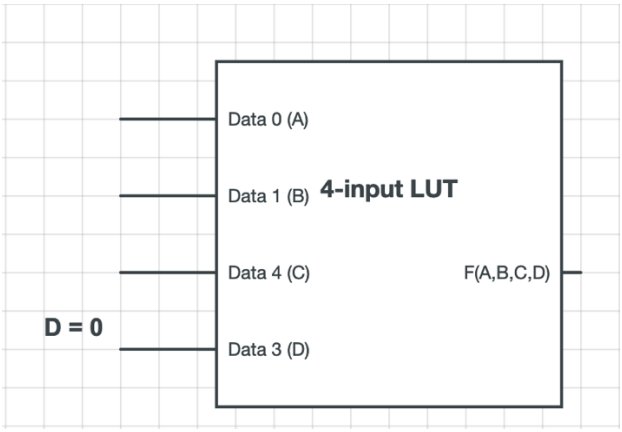


Figure 4-9: Block diagram of function X=A'B'C+ABC'

DSP48 Block

DSP is the most complex block available in the Xilinx FPGA. The 48 in the DSP48 represents the 48-Bit Logic. The block consists of three stages: Pre-adder, Multiplier and the Logic unit depending on the device.The pre-adder is a 25-bit adder/subtractor that takes the lower 25 bits of input A and D as input. The result is then multiplied with input B. The logic unit works on the input C, multiplier output and concatenated result of A and D.

Block RAM (BRAM):

Block RAM in MPSoC can be used as RAM, ROM, and FIFO buffers. BRAM provides chip storage of up to 36 KB. Multiple vertical BRAM can be paired to create a memory array.

4.4.3. AXI interface:

AXI specification describes an interface between a single AXI master and AXI slave that can exchange information within each other. It is used to connect PE and IP blocks. Data can move between the master and slave simultaneously. The burst transaction of up to limit of 256 data transfers exists in AXI4. It provides separate data and address connection for reads and writes which allows for bidirectional connection. AXI interface has been extensively used for communication between the PS and the PL.

The AXI protocol incorporates five distinct channels, each serving a specific purpose. Among them, two channels facilitate Read transactions, namely the "read address" channel and the "read data" channel. Additionally, the AXI protocol employs three channels to support Write transactions. These channels include the "write address" channel, which specifies the location where data should be written, the "write data" channel, which transmits the actual data to be written, and the "write response" channel, which provides feedback regarding the status of the write operation.

AXI read transaction

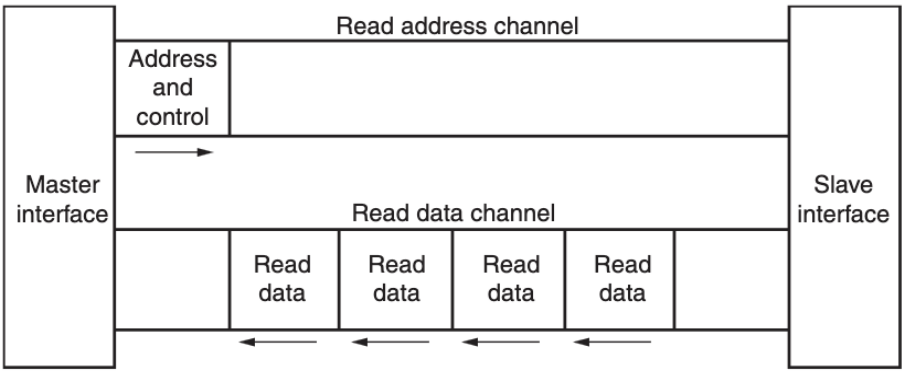


Figure 4-10: AXI read transaction uses read address and read data channels.

First, read signal sent from the master interface to the slave interface that sets the address and control signal. Upon receiving the signal, the Slave interface sends data through the read data channel.

AXI write transaction

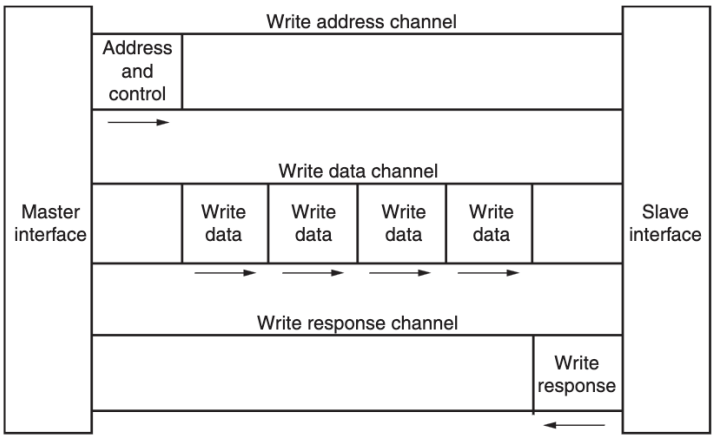


Figure 4-11: AXI write transaction uses write address, data, and response channels.

First, a signal is sent from the master to set the address and control signal. Then the master sends the data through the write data channel toward the slave interface. After receiving the data, the slave interface sends a response through the write response channel.

4.5 Model Compression

FPGA, a resource-constrained device, cannot directly implement models trained using libraries like TensorFlow and PyTorch. These models need to be compressed to reduce their size as much as possible without affecting accuracy, using various techniques.

Model compression, which reduces the size and complexity of a deep learning model without significantly sacrificing performance and accuracy, is necessary for running ML applications on low-resource edge devices. Different model compression techniques include Quantization, Pruning, Knowledge Distillation, Matrix and Tensor Decomposition, etc. Among these, Quantization and Pruning are the most practical and effective because methods like Knowledge Distillation and Tensor Decomposition are more complex and do not yield as good results.

Model compression methods like Quantization and pruning are possible in neural networks due to the unique characteristics of the problems they solve and the desired metrics for evaluation. Unlike traditional research, where well-posed and well-

conditioned problems are typically addressed with specific solutions, neural network applications focus on forward error metrics such as classification quality and perplexity. Over parameterization of neural networks allows for numerous models that optimize these metrics, resulting in the potential for high error or distance between quantized and non-quantized models while still achieving excellent generalization performance.

Furthermore, the layered structure of neural network models introduces an additional dimension to consider. Different layers within a neural network contribute differently to the overall loss function, which motivates the adoption of a mixed-precision approach to quantization.

4.5.1. Pruning

Pruning is an effective technique for reducing the size of our model while minimizing the impact on its performance. By selectively removing connections in the network that do not contribute significantly to the final model's output, pruning can significantly decrease the model's size with minimal or zero loss.

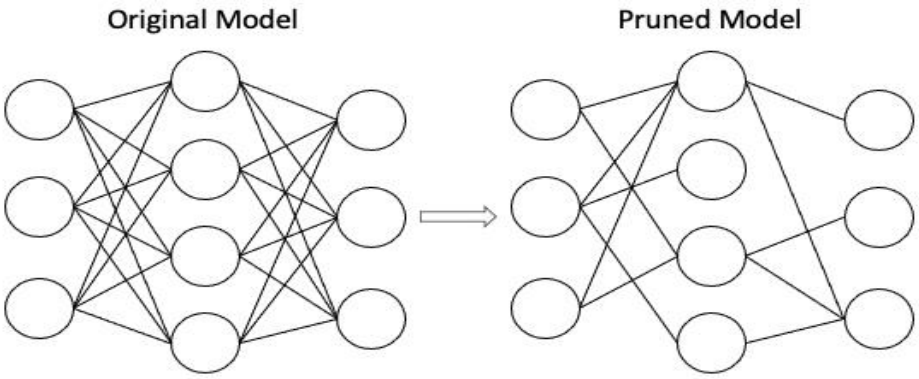


Figure 4-12. Model Pruning

This process involves eliminating weights or neurons with low saliency, meaning those weights that have little or no effect on the output or loss function. Pruning methods can be broadly categorized into two types: structured and unstructured pruning.

Unstructured pruning removes neurons with small saliency wherever they occur, removing most NN parameters without impacting model generalization performance. However, this approach leads to sparse matrix operations, which are hard to accelerate, and memory bound.

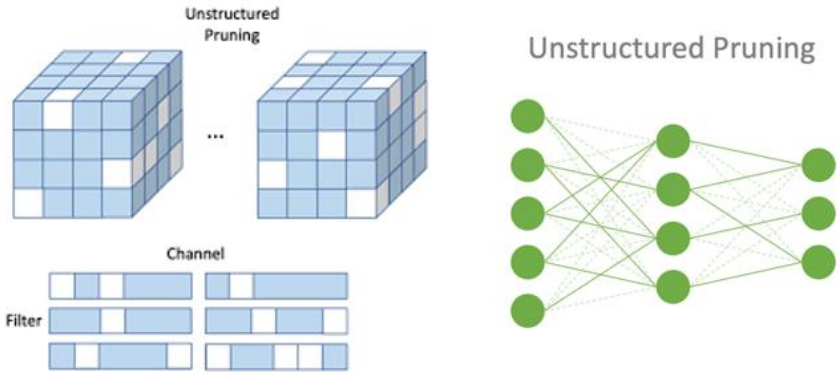


Figure 4-13: Structured and unstructured Pruning

Structured pruning removes a group of parameters, such as convolution filters, allowing density matrix operations but often causing significant accuracy degradation. This changes the shape of input and output layers and weight matrices, thus permitting dense matrix operations.

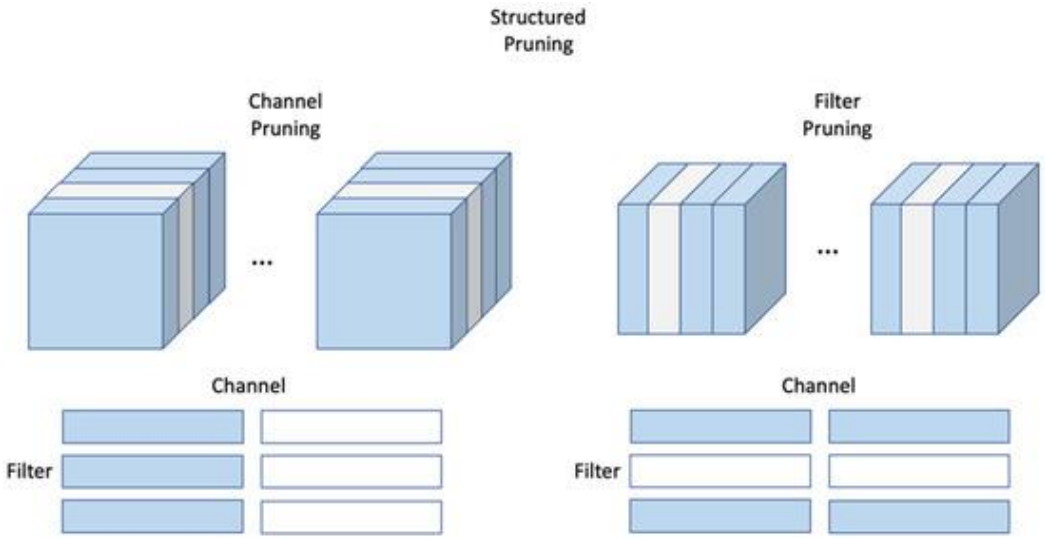


Figure 4-14: Channel pruning and filter pruning

In this paper [23], the authors reduced the size of Inception V3 by almost half by increasing the sparsity (reducing number of zero weights connections) with a loss of only 0.1% in model accuracy.

Table 4-3: Model size and accuracy tradeoff for sparse-InceptionV3

Sparsity	Params	Top – 1 Acc	Top - 5 Acc .
0 %	27.1M	78.1 %	94.3 %
50 %	13.6M	78.0 %	94.2 %
75 %	6.8M	76.1 %	93.2 %
87.5 %	3.3M	74.6 %	92.5 %

So, we can implement this to prune our EfficientNetV2 too, we can selectively prune the connections that have less significant effect on output accuracy. Then, we can retrain our model again with a slightly lower learning rate and keep repeating the process up to a certain limit. One simple way to achieve this is to list sort out all the weights of network in descending manner and then prune the weights based on the sparsity level.

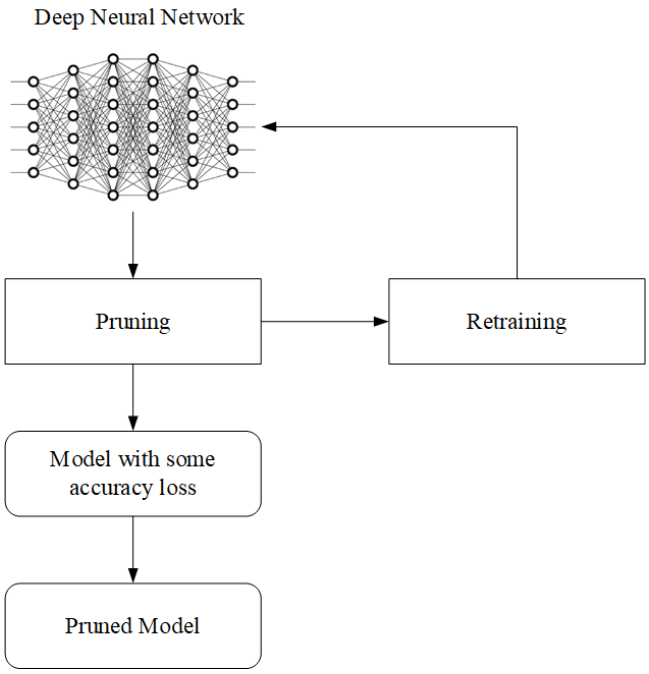


Figure 4-15: Pruning and Refining

4.5.2. Quantization

Another way to reduce the model size is to reduce the bits used to represent the weights of the network. The idea behind quantization is that most of the weights and biases in the network are confined in certain regions. The goal of quantization is to reduce the precision of both the parameters (θ), as well as the intermediate activation maps (h and a) to low-precision values while minimizing impact on the generalization power/accuracy of the model.

Normally the weights in the network are represented in Float32, by using techniques of quantization these weights can be represented in Int8 or Int16. This not only reduces the size but also speeds of the computation process as it is easier to do integer calculations rather than float.

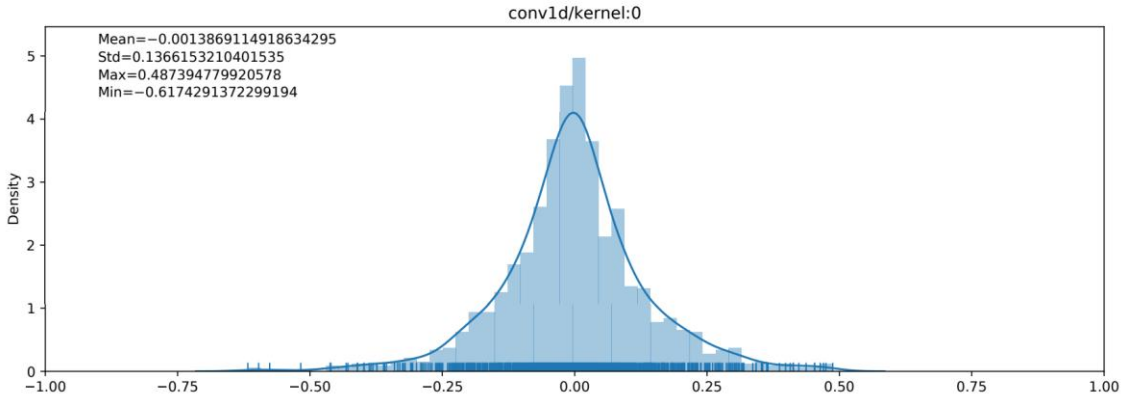


Figure 4-16: Example of weight distribution of convolution layer [24]

Now, the weights in the layer can be mapped into the integers and could be represented by int8 with slight quantization error. Most of the weight above are confined in the region [-0.5, 0.5], we can use ‘0’ to represent -0.5 and ‘255’ to represent +0.5.

In order to mathematically formulate the quantization process, it is essential to define a precise quantization operator that effectively maps floating point values to quantized equivalents. Our objective is to develop a function capable of quantizing neural network (NN) weights and activations into a finite set of values.

One popular choice for a quantization function is as follows, which exhibits uniform quantization and evenly spaces the resulting integers:

$$Q(r) = Int(r/S) - Z \quad (4.2)$$

Where:

Q represents the quantization function,

r denotes the real-valued weight or activation to be quantized,

S stands for the real-valued scaling factor, and

Z represents the integer zero point.

The Int operator in the formula returns the nearest integer value. By utilizing this quantization function, we can effectively convert continuous values into discrete integers.

For the de-quantization process and to recover the real values from their quantized counterparts, the following operation, known as De-quantization, can be applied:

$$r' = S * (Q(r) + Z \quad (4.3)$$

By employing this inverse operation, we can approximate the original real values, r, from the quantized values, Q(r). Although recovered real values, r', may not be identical to the original values of r due to the inherent rounding performed during quantization.

Now, let's delve into the determination of the scaling factor, S, which plays a vital role in the quantization process. If we consider (α, β) as the range of real numbers or weights and 'b' as the desired number of bits or bit width for quantizing the weights, the scaling factor can be calculated as:

$$S = (\beta - \alpha) / (2^b - 1) \quad (4.4)$$

This equation ensures that the range (α, β) is adequately divided into 2^b - 1 intervals, allowing for the representation of a finite set of quantized values.

In this case, the quantization process is symmetric, implying that the positive and negative ranges of the real values are equally represented by the quantized values. The

selection of the symmetric or asymmetric quantization approach largely depends on the characteristics and requirements of the specific application.

For a symmetric quantization scenario, where α and β are unknown, the following equation can be utilized to determine the values:

$$-\alpha = \beta = \max(|r\text{-}max|, |r\text{-}min|) \quad (4.5)$$

By employing this equation, the maximum absolute value among |r-max| and |r-min| is assigned to both α and β, ensuring symmetric quantization around the zero point.

Hence in this way, after training our network we will plot the weight distribution of network and then choose appropriate representation while minimizing the loss accuracy.

Quantization granularity:

In CNN architectures, the activation input to a layer is convolved with many different convolutional filters. Each of these convolutional filters can have a different range of values. So, one differentiator for quantization methods is the granularity of how the clipping ranges [α, β] is calculated for the weights. We can categorize these methods based on how these ranges are calculated, and each approach has its own strengths and limitations.

a) **Layer-wise quantization:** In layer-wise quantization, a single range is set for all channels within a specific layer. However, this approach often performs poorly due to the diverse value ranges exhibited across different channels within the same layer. Consequently, the quantization resolution tends to be compromised, impacting accuracy and precision.
b) **Channel-wise quantization:** Channel-wise quantization, currently regarded as the standard technique, assigns unique ranges to each individual channel. By tailoring the clipping ranges according to the characteristics of each channel, this method enhances the quantization resolution and preserves accuracy. The individualized

treatment of channels allows for a more fine-grained and precise quantization process.

c) **Group-wise quantization:** Group-wise quantization takes an intermediate approach by grouping together a subset of channels into layers and calculating the clipping range [-α, β] for that specific group. This technique strikes a balance between layer-wise and channel-wise quantization methods. By grouping related channels, the quantization process can capture the specific range of values within the group more effectively. Group-wise quantization offers improved granularity compared to layer-wise quantization while reducing complexity and computational overhead.

d) **Sub-channel-wise quantization:** Sub-channel-wise quantization is a more intricate approach that involves determining the clipping range with respect to any selected group of parameters within a channel. However, this method introduces considerable complexity and computational overhead, making it less practical for most implementations. Therefore, it is not commonly employed in quantization scenarios.

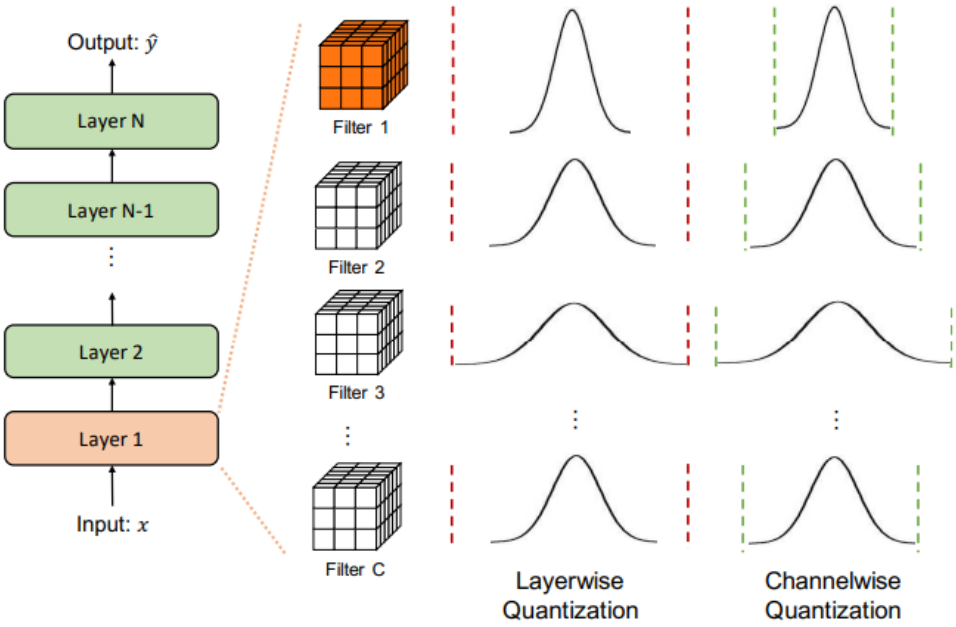


Figure 4-17: Layer wise and Channel wise quantization [25]

Dealing with weight distributions that contain outliers

When dealing with weight distributions that contain outliers, applying uniform quantization directly may result in low resolution and compromised accuracy. To address this issue while maintaining uniform quantization, alternative approaches can be considered.

One approach is to use percentiles instead of the entire min/max range of the signal. Rather than considering the largest and smallest values, the k-th largest and smallest values can be utilized as β and α, respectively. This technique allows for a more robust quantization process that considers the distribution of values, thereby enhancing resolution and preserving accuracy.

Another method involves selecting α and β to minimize the Kullback-Leibler (KL) divergence, which measures the information loss between the real values and the quantized values. By optimizing the clipping range based on KL divergence, it is possible to achieve quantization that minimizes the loss of information, thus maintaining higher fidelity to the original data.

Static vs. dynamic quantization

In the context of a neural network (NN) model, the weights remain the same for every input, allowing them to be pre-quantized. However, the activation values vary across different inputs, necessitating the dynamic definition of the range, denoted as $[-\alpha, \beta]$. This dynamic approach requires the computation of signal statistics, such as minimum, maximum, percentiles, etc., in real-time. However, this real-time computation can introduce a high computational overhead.

While static quantization can be applied to activations as well, it often results in lower accuracy compared to dynamic quantization. To determine the appropriate range, a popular method involves running a series of calibration inputs, typically selected from training examples, to compute the typical range of activations.

Several metrics have been proposed to identify the best range for quantization. One common approach is to minimize the Mean Squared Error (MSE) between the original un-quantized weight distribution and the corresponding quantized values. Additionally, other metrics such as entropy can also be considered, although MSE remains the most widely used method. Another approach involves learning or imposing the clipping range during NN training, allowing the model to adapt and optimize the quantization process based on specific objectives.

By considering the trade-offs between static and dynamic quantization and selecting appropriate metrics to determine the clipping range, we can achieve the desired balance between accuracy and computational efficiency in the quantization process.

Uniform vs. Non-uniform quantization:

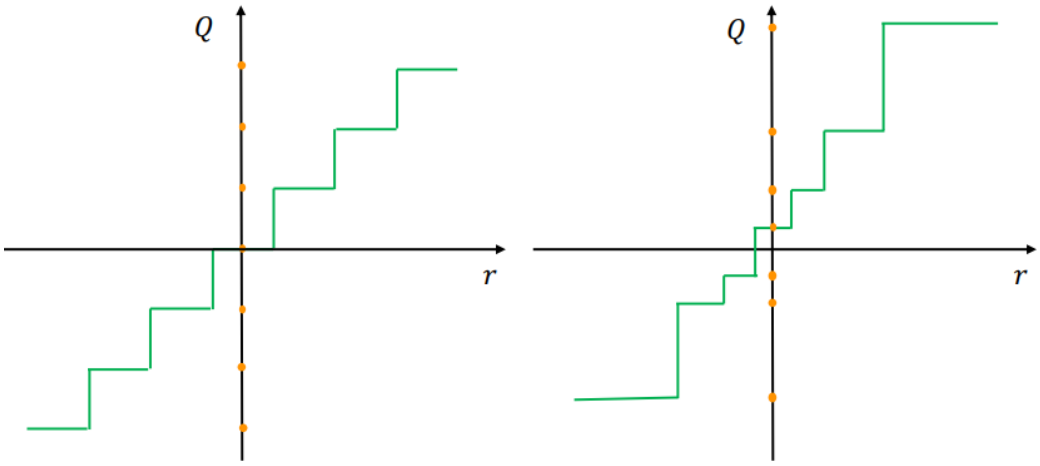


Figure 4-18: Uniform and non-uniform quantization [25]

Each level in above figure (total 7 levels) can be assigned a binary number as an example 1001, 1010, 1011. In non-uniform quantization, a real number 'r' is mapped to a discrete value if it falls within the interval Δ(i) and Δ(i+1), where the intervals are not uniform.

For fixed bit length, non-uniform quantization helps achieve better accuracy by allowing the representation of more important areas with higher resolution (using more levels) within a specific range. Non-uniform quantization can be rule-based or optimization-based.

One example of a rule-based approach is to use a logarithmic distribution, where quantization steps and levels increase exponentially instead of linearly. Optimization-based quantization methods, such as K-means and Hessian weighted K-means, can be applied to identify important areas and determine the quantization steps and levels.

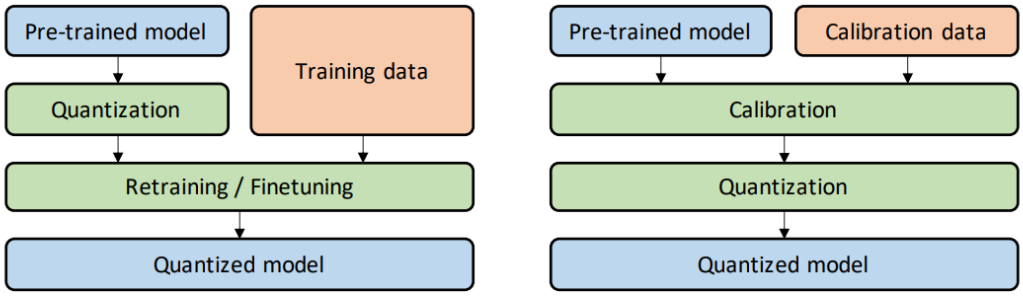


Figure 4-19: Quantization process [25]

4.6 Xilinx DPU

The DPUCZDX8G is a specialized Deep Learning Processing Unit (DPU) designed for MPSoC (Multi-Processor System-on-Chip) architectures. This powerful computation engine is specifically optimized for Convolutional Neural Networks (CNNs). The IP comes in various variants, including B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096, which indicate the total number of Multiply-Accumulate (MAC) operations per DPU clock cycle.

In the case of B512 architecture, the DPU exhibits 4-fold pixel parallelism, 8-fold input channel parallelism, and output channel parallelism. Consequently, the total number of Multiply-Accumulate (MAC) operations can be calculated as follows: 4 * 8 * 8 = 256. However, the convolution array performs two operations, namely multiplication and accumulation, in each clock cycle. Therefore, the peak number of operations per cycle is counted as 128 * 2 = 512, doubling the previous calculation.

DSP slices are used for accumulation of convolution modules. According to selection, if low DSP usage is selected, the DPU IP will use slices for multiplication only. In the high DSP mode, DSP slices are used for both multiplication and accumulation. An example data tested on the ZCU102 platform with low RAM usage shows the resources used for different architectures under high and low DSP usage.

Table 4-4: Example Resource used by ZCU102 in different architectures

DPU Architecture	LUT	Register	BRAM	DSP
B512	26922	34543	72	118
B800	29721	41147	90	166
B1024	34074	48057	104	230

4.7 Hardware Software Co-search for Efficient Architecture

4.7.1. Generation of Convolutional Search space

The search space we used is based on of Mobilenetv3 blocks with different configurations.

Mobilenetv3 was designed for very high throughput on mobile devices. So, it was designed to be deployed efficiently on mobile and edge devices. Search space can be designed in several methods, but this search space uses block-based design. Which means that the Mobilenetv3 blocks will be reused throughout the entire network by changing parameters like kernel size, number of channels, choice of whether to keep squeeze excitation layers or not and depth. Thus, we do not have to design search space in the lower levels. Also there only exist few blocks which makes it easier for latency profiling.

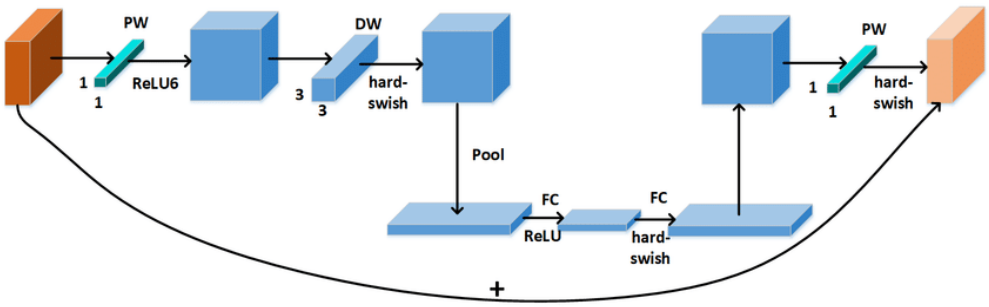


Figure 4-20: Mobilenetv3 block

A single Mobile net V3 block starts with normal convolution followed by 1*1 pointwise convolution to increase the number of channels. Then depth-wise separable convolution

with Hard-swish activation function and kernel size sampled from search space is applied which does not change the number of channels. Then squeeze and excitation layer is applied.

The squeeze and excitation layer consists of the global average pooling of each channel, followed by dense layer with ReLU activation and again dense layer with Hard sigmoid activation. The search space has a Boolean choice of whether to apply squeeze excite or not. If not, the layer is skipped. Finally, the inverted bottleneck structure has a 1*1 point-wise convolution to reduce the number of channels to the original size and thus also a residual skip connection. The search space governs the number of channels in the depth-wise layer and squeeze-excite layer at the same time.

Search space parameters:

Number of blocks = 20
Number of stages = 5
kernel sizes = [3, 5, 7]
expand ratios = [3, 4, 6]
depths = [2, 3, 4]
resolutions = [224]

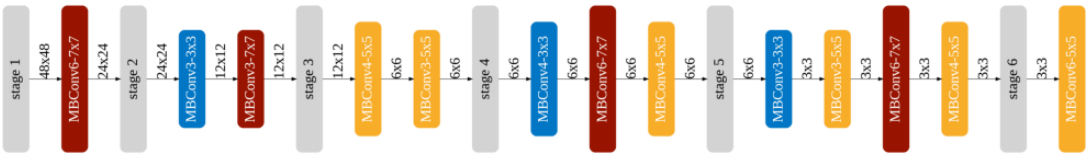
Each stage is marked by a change in the number of channels to double and both width and height reduction by half. All blocks in a stage have either squeeze excitation layer or direct connection without it.

Let us consider sample search space as following:

- Elastic resolution=input images can have different size (ex 128*128,32*32,224*224)
- Elastic kernel size= Each block can have different kernel sizes (3*3,5*5,7*7)
- Elastic width/expand ratio= Each block can have a different number of channels in each layer.
- Elastic depth= Each unit/stage can have arbitrary number of blocks

Each unit consists of a sequence of layers where only the first layer has stride 2 if the feature map size decreases, all the other layers in the units have stride 1.

The input image size ranges from 128 to 224 with a stride 4; the depth of each unit is chosen from {2, 3, 4}; the width expansion ratio in each layer is chosen from {3, 4, 6}; the kernel size is chosen from {3, 5, 7}. Therefore, with 5 units, we have roughly ((3 × 3) ^2 + (3 × 3) ^3 + (3 × 3) ^4) ^ 5 ≈ 2 × 10^19



If at a stage/unit, depth of 2 is chosen there are (3*3) choices of kernel size and expansion ratio respectively for each layer/block. So, (3*3) ^2 choices for the stage. But the depth can be 3 or 4. So (3 × 3) ^2 + (3 × 3) ^3 + (3 × 3) ^4. Finally, since there are 5 stages for the network, we raise to power by 5 to account choices for each stage.

So, with only a few choices in the search space we can have an extremely large search space.

4.7.2. Search Space Exploration: Evolutionary Search Algorithm

Once the search space is generated, we need to guide the architectural exploration.

Early Neural Architecture Search (NAS) methods, such as NAS Net and MNAS-Net, exhaustively train networks in the design space using RNN-based controllers with reinforcement learning. These methods are computationally expensive as each network is trained from scratch.

Later, differentiable NAS methods like DARTS, Proxyless-NAS, and FB-Net were developed, significantly reducing training costs. DARTS models each layer's output as a weighted average of different operations' outputs, while Proxyless-NAS reduces DARTS's memory cost by keeping only two paths in memory. One-shot methods like Single Path One Shot further optimize this by keeping only one path during training.

Despite being more efficient, differentiable NAS and one-shot NAS still require running the entire pipeline for each new network design. This incurs significant costs

(200-300 GPU hours for ImageNet), especially considering the vast number of IoT devices.

The Aging Evolution has emerged as a fast efficient and feasible search strategy. RL based methods require a lot of training time. Unlike traditional methods that only preserve the best architectures, Aging Evolution assigns an ‘age’ to each genotype in the population. This age is factored into the selection process, giving preference to younger genotypes. This approach allows for a more extensive exploration of the search space, preventing premature focus on seemingly good models.

The search will be guided by the optimization objective of 3 parameters namely arithmetic intensity, accuracy and latency.

4.7.3. Estimating Accuracy

During the search process, generated network architectures need to be trained for few epochs, then validation accuracy needs to be calculated. Not only that, but the architecture also needs to be implemented on the FPGA. This takes a lot of time and is difficult to integrate in the search process. The Once for all network many of these problems. Once the OFA network is trained there is no need to train any further. It decouples the architecture search process from network training. During the architecture search process, sample architectures in Evolutionary algorithm can be looked upon in the OFA and latency and accuracy can be calculated without training. To further reduce the time, a separate network is trained to predict the accuracy of sampled architecture.

4.7.4. OFA networks

All possible architectures in the above search space are present in OFA (Once for all). [15] Once the large super-network is trained, we can directly sample a subnet and inference without further training.

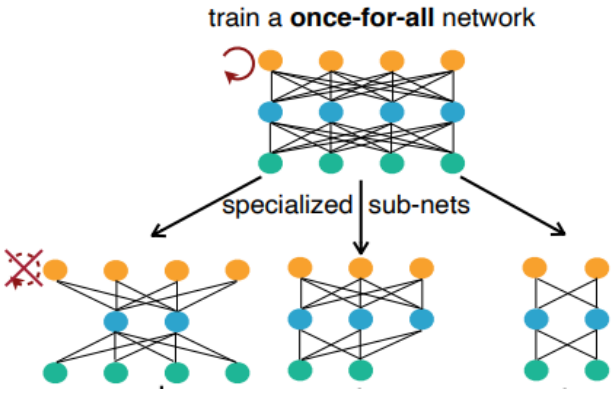


Figure 4-21: OFA Supernet [15]

The training objective of the once for all network is:

$$\min_{W_o} \sum_{arch_i} L_{val}(C(W_o, arch_i)) \tag{4.6}$$

Where,

W_o represents weights of the entire OFA network.

$C(W_o, arch_i)$ is a selection scheme that samples a subnetwork $arch_i$

By minimizing the average validation accuracy of all subnetworks, we can get fairly equal amount of accuracy for all subnetwork architectures $arch_i$

But training OFA is nontrivial because it consists of lot of subnetworks. The authors used a technique called progressive shrinking to efficiently train the large network at once. It generalizes the concept of network pruning by first training a larger subnetwork and then pruning not only channels like channel pruning in Efficient-Net compression but also other dimensions (by decreasing kernel size, expand ratio, and depth). The pruned network can share its weight with parent network and thus can converge faster.

Pruning channels or decreasing the width is like channel pruning in efficient net. L-1 norm is used to prune less important channels until the desired number of channels are pruned.

Reducing the kernel size is more complex. To reduce kernel size and still employ weight sharing for efficient training, kernel transformation matrices can be used to derive a smaller kernel size (say 3*3) from larger size (5*5) by doing some matrix multiplications.

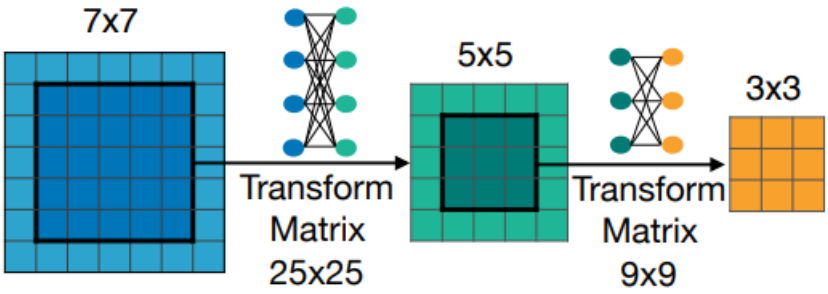


Figure 4-22: Kernel weight sharing [15]

The middle portion of a larger kernel size is clipped and then transformed by a matrix to get new kernel weights for smaller kernel size. The weights of the matrix are also learned during training. The kernel transformation matrix is the same for all channels in a single block. So only a few extra parameters (25*25 +9*9 =706) need to be stored for each block.

In case of depth shrinking the final block/s are entirely removed and then fine-tuned to get smaller architecture. Doing so instead of just randomly removing any other block allows to share the weights of first few unremoved blocks.

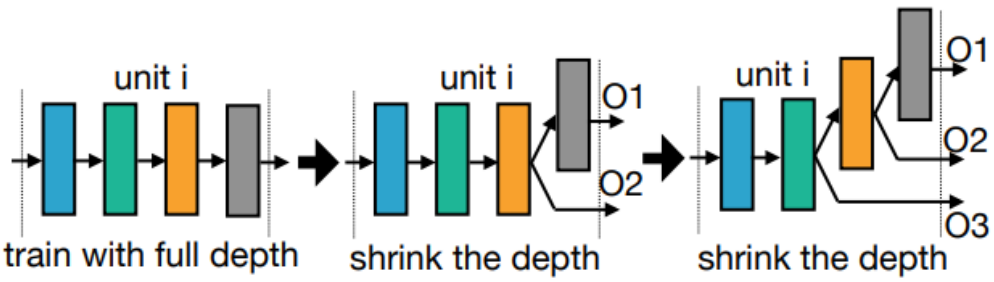


Figure 4-23: Progressive Shrinking of OFA [15]

The sequence of pruning is Kernel size, Depth, Width.

Thus, a single subnetwork consists of multiple other subnetworks. Once a larger subnetwork with largest kernel size, largest number of channels and maximum depth is trained, the network is pruned and fine-tuned in the above sequence.

Once the OFA network is trained there is no need to train any further. It decouples the architecture search process from network training. During the architecture search process, sample architectures in Evolutionary algorithm can be looked upon in the OFA and latency and accuracy can be calculated without training. To further reduce the time, a separate network is trained to predict the accuracy of sampled architecture. The accuracy predictor network takes input the architecture configuration of sampled network and estimates its accuracy. The training dataset for this network are few of architectures from OFA and their corresponding validation accuracy.

Like accuracy prediction, a Hardware aware NAS requires latency estimation. For this, latencies of different block configuration are measured in our FPGA. The overall latency is estimated by just summing the latencies of individual blocks.

4.7.5. **Evaluator Design**

Evaluation involves training and testing the architectures that were generated to determine their accuracy, latency, efficiency, and other relevant metrics. Evaluation process is carried out iteratively along with other processes. Multiple architectures are trained, tested, and evaluated during the workflow to find best architecture.

Since prediction-based search is fast and efficient, we predict the validation accuracy, latency and calculate the arithmetic intensity which encourages more computation on less size.

Training the network architecture obtained by sampling from search space on the fly is a very time-consuming process even for obtaining the validation accuracy. Therefore, a dense Multi-Layer-Perceptron is used as accuracy predictor which is trained on the accuracy dataset consisting of architecture and accuracy pairs sampled from the OFA

network. The network takes as input a one hot vector representing the architecture encoding of the network and outputs accuracy.

Arithmetic intensity

Another metric is arithmetic intensity, defined as the number of MAC operation per size of byte. One basic principle on most devices and on FPGA is that computation is cheaper than data movement. Data movement to and from memory is significantly slower than computation. A higher arithmetic intensity means that more operations can be performed with the same amount of data movement. A network that does more operation can have better accuracy and less parameters, which means that there is less data movement from memory, thus faster inference.

Latency

The third constraint is the network's latency, estimated using a latency table. Latencies of individual blocks are measured on the FPGA using latency profiler and a latency lookup table is made. During the evolutionary search, the latency of network is roughly estimated by the sum of latencies of individual blocks.

The search process first checks if the efficiency constraints are satisfied or not. If they are satisfied, then only the network gets selected. Then the networks are sorted based on accuracy to select parents for the next generation.

4.8 Flowchart: HW-aware evolutionary search with accuracy and efficiency constraints

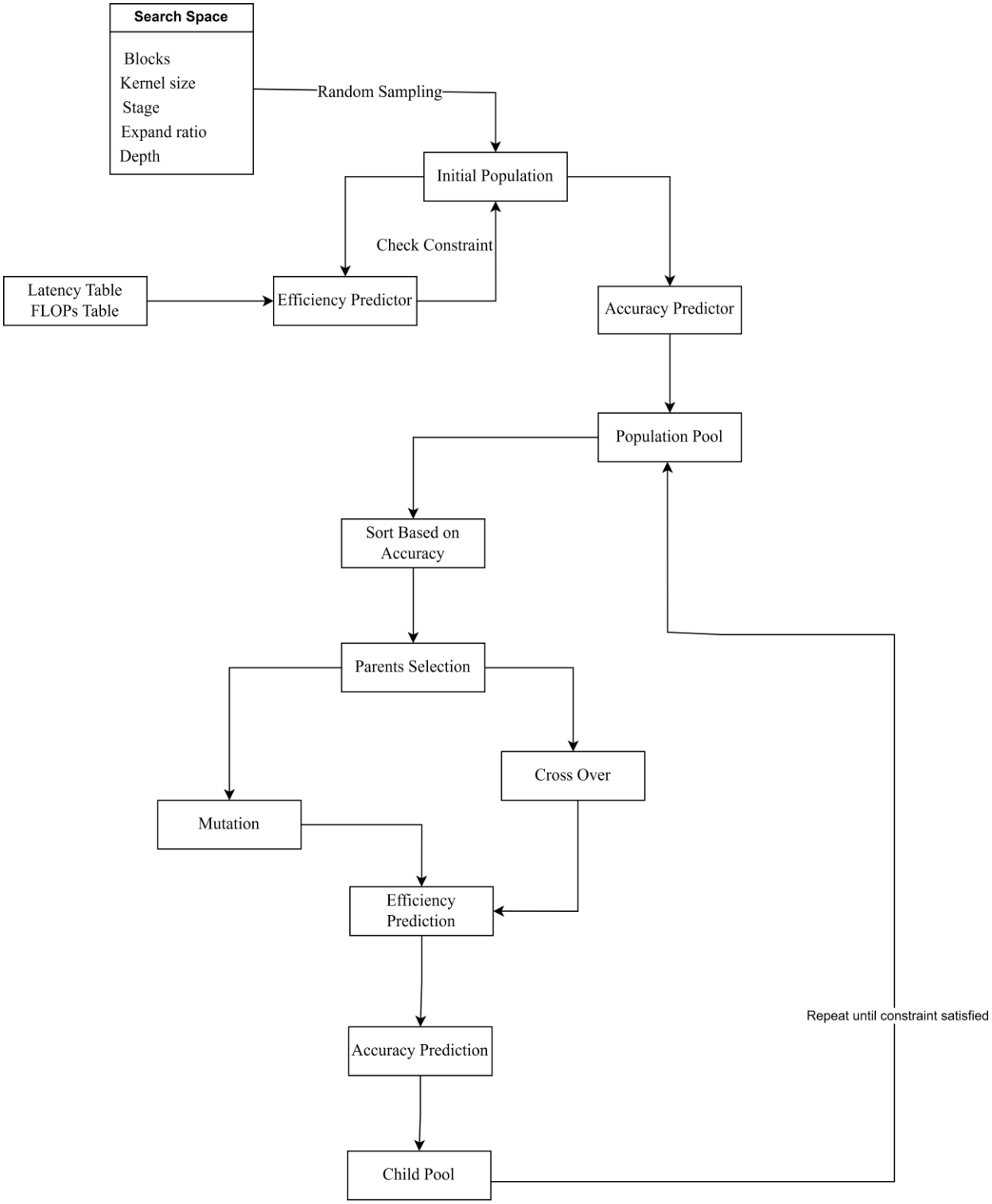


Figure 4-24: Flowchart Evolutionary Search

5. IMPLEMENTATION DETAILS

5.1. Model Compression: EfficientNet-V2

EfficientNetV2-S achieved 83.9% accuracy on ImageNet with 22M parameters and 8.8B FLOPs, making it fast with a 24ms inference time. However, its size poses FPGA challenges. Hence, we're retraining it on CIFAR-10 before applying compression techniques like pruning and quantization. We initially tried Tiny ImageNet, but its 200 classes complicated fine-tuning and testing compression methods. CIFAR-10 Dataset

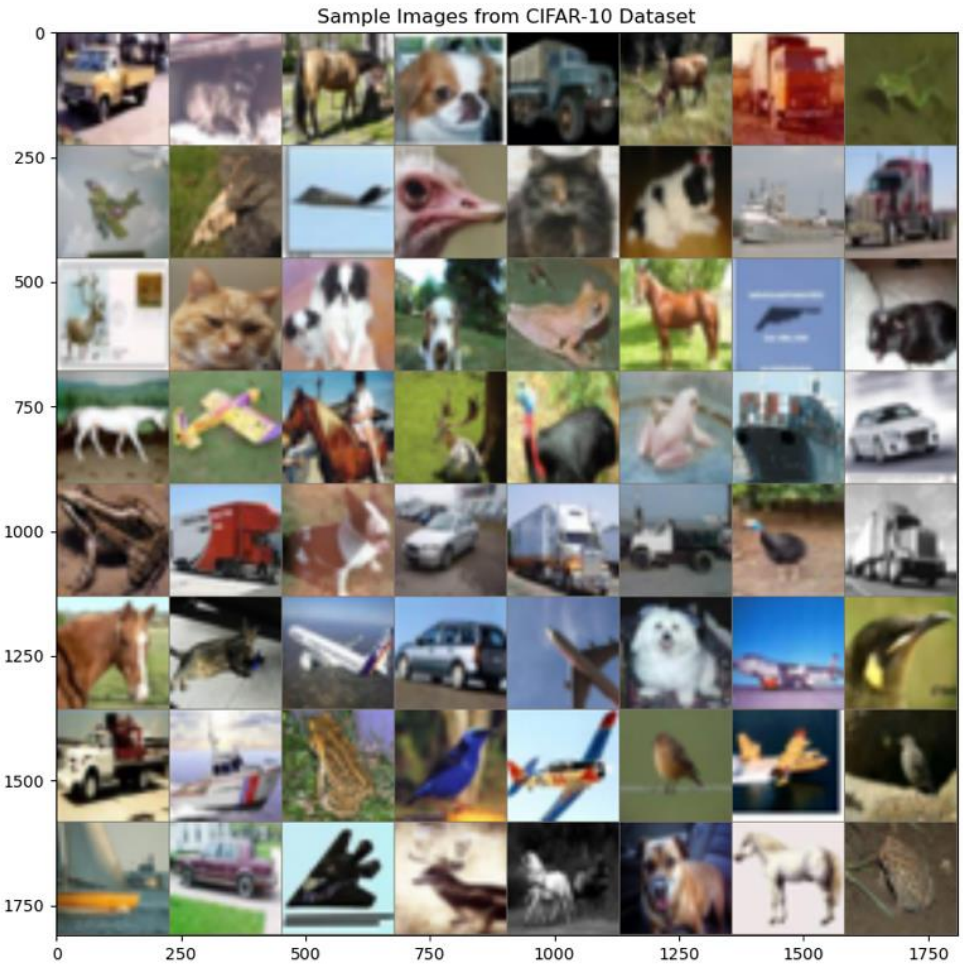


Figure 5-1: Sample images from CIFAR-10

The CIFAR-10 dataset, a benchmark in computer vision, contains 60,000 labeled 32x32 color images of common objects. Its split into 50,000 training and 10,000 test images, evenly distributed across classes. Its characteristics make it ideal for our efficient model retraining and evaluation.

5.1.1. Fine tuning Efficient-Net on CIFAR-10

We adapted EfficientNetV2 for CIFAR-10 by replacing its 1000-class layer with a 10-neuron layer. We resized CIFAR-10's 32x32 images to 224x224 to match EfficientNetV2's input size. To prevent overfitting, we randomly flipped images horizontally. The best model was saved for later pruning techniques.

5.1.2. Pruning the Refined Model

Pruning involves selectively removing a *p%* percentage of parameters and subsequently retraining the model until the desired level of sparsity is achieved.

We employed magnitude-based unstructured pruning to eliminate less valuable parameters from the network. Specifically, we implemented global unstructured pruning, which entails calculating the L1 norm of each parameter in the network and then pruning the p% of weights with the lowest L1 norm.

The L1 norm of a weight parameter can be calculated using the formula:

$$L1\_norm(x) = abs(x) \tag{5.1}$$

-0.1	0.027	-0.91
1.24	-3	2.1
-0.57	2.5	0.5

After pruning, the resulting filter becomes sparse. In our case, the weights 0.027, 0.5, and -0.1 would be pruned since they have the lowest L1 norm. These weights were eliminated because they possessed the lowest magnitude. As a result, the resulting kernel would appear as follows.

0	0	-0.91
1.24	-3	2.1
-0.57	2.5	0

Parameter distribution in EfficientNetV2

EfficienynNetV2 network can be logically grouped into eight blocks. These groupings were created based on the number of output channels, each layer inside the same block has the equal number of output channels. The composition of each block varies in terms of the number of MbConv and FusedMBconv blocks. The first and last blocks consist of Conv2DNormActivation layers and do not include any MBconv or FusedMBconv blocks.

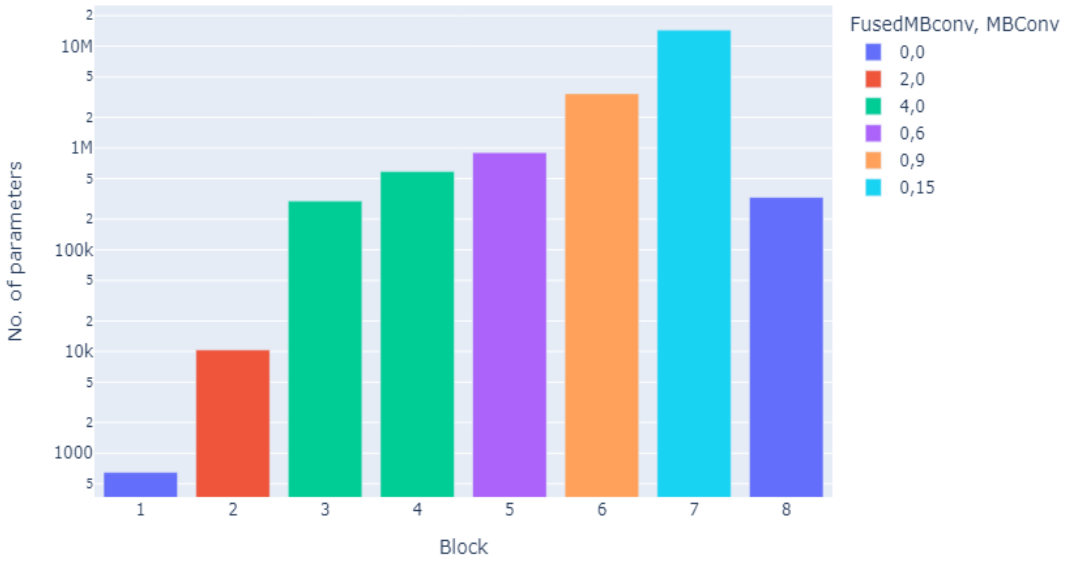


Figure 5-2: Parameter distribution in EfficientNetV2

Iteratively pruning the network

Initially we pruned 40% of EfficientNetV2's parameters, dropping accuracy to 92.29%, but 3 epochs of retraining improved it to 97.53%. Further pruning and retraining cycles maintained high accuracy levels, even with a model 70% sparser. Despite accuracy

drops after each pruning, retraining consistently recovered it, demonstrating our approach's effectiveness.

5.1.3. Replacing SiLU with Hard Swish Activation

Efficient-Net used SiLU activation function. SiLU activation function is a special case of Swish activation when β parameter is 1. We replaced SiLU with HardSwish activation function and then retrained the pruned model. HardSwish activation can be defined as.

$$\text{Hardswish}(x) = \begin{cases} 0 & \&\text{if } x \leq -3 \\ x & \text{if } x \geq +3 \\ x \cdot \frac{x+3}{6} & \text{otherwise} \end{cases} \quad (5.2)$$

HardSwish replaced SiLU due to its computational efficiency, using simple arithmetic instead of expensive exponentials. It's more memory-efficient, requiring basic arithmetic and no intermediate value storage. Despite these changes, HardSwish closely approximates SiLU's performance, making it suitable for resource-constrained devices.

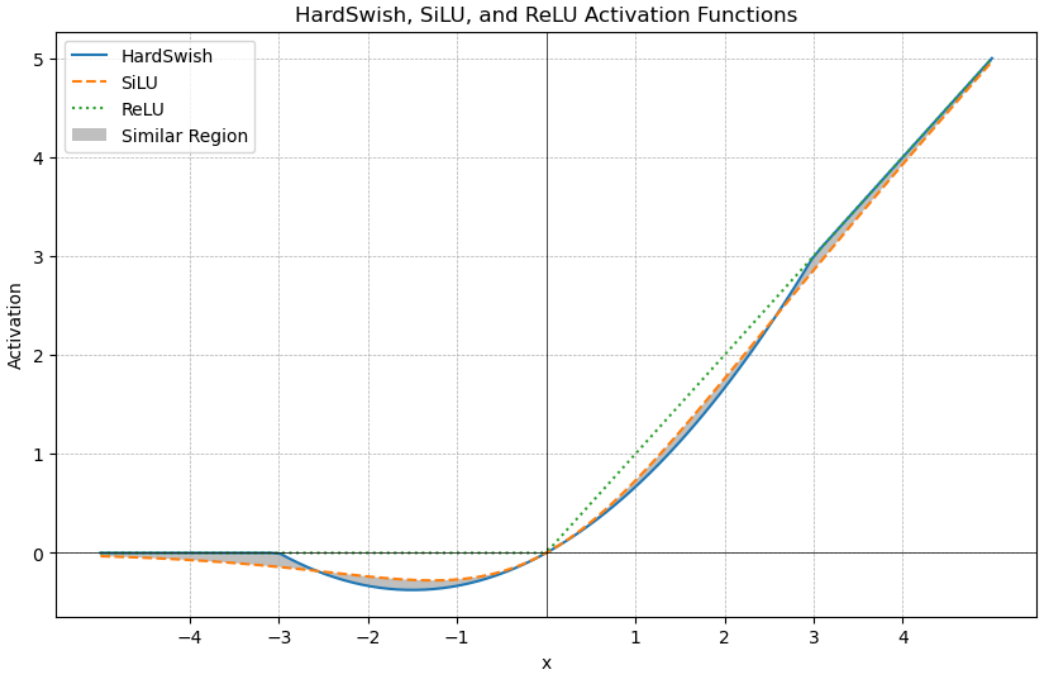


Figure 5-3: Comparison of HardSwish, SiLU and ReLU

Although very similar to Swish, the derivative of this activation is simple and piecewise linear. So, it becomes easy to calculate or compute. The derivative of constant and linear term is zero while the quadratic term has linear derivative as shown in the figure below.

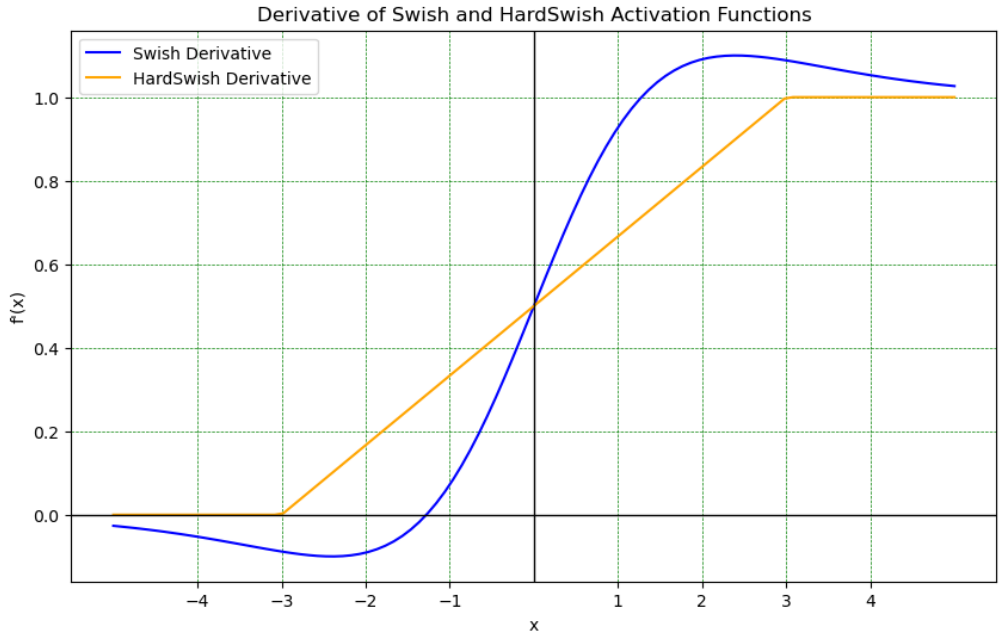


Figure 5-4: Differentiability of Swish and Hardswish

5.1.4. Channel Pruning

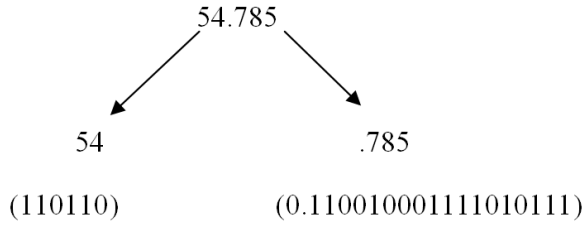
Unstructured pruning didn't result in reduction in size and computational needs but gave us insights on the redundancy of the model. So, Channel pruning was implemented and tested. The first 3 layers and final classification layer were frozen, and the rest of the layers were iteratively pruned for 11 iterations resulting in 87% pruned model. The criterion was magnitude, the group of channels with lowest L2 norm were pruned globally. The method we implemented was Layer-adaptive sparsity for the Magnitude-based Pruning (LAMP).

LAMP involves pruning the least important channels and their dependencies from the network iteratively so that the accuracy losses can be minimized. At first the global importance score for each channel is determined. The importance of each channel is determined using the L2 Norm, a measure that calculates the square root of the sum of the squares of the elements in a vector.

5.1.5. Quantization

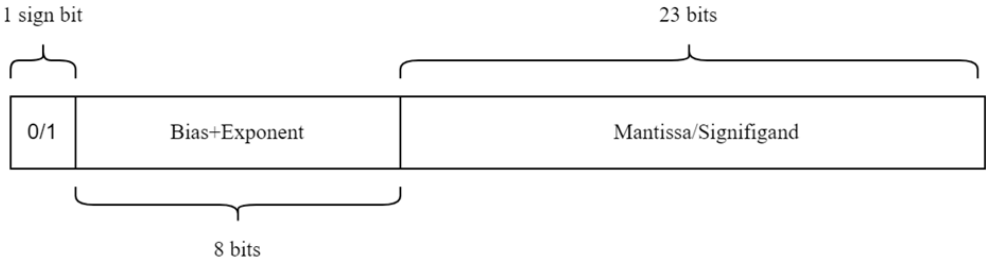
Weights or activation values are initially stored as floating numbers in binary. More specifically in 32-bit format. Let us clarify this with an example. Suppose we have

Decimal number = 54.785. To convert the number to binary we can separate it into integer and fractional part.



The integer part is repeatedly divided by 2 and remainders are taken while the fraction part is repeatedly multiplied by 2 and integer parts after each multiplication are accumulated. We need to convert this into 32-bit floating point representation format with 1 sign bit 8 exponent bits and 23 bits for mantissa which can be expressed as

$$(-1)^S * 0.M * 2^E \qquad (5.3)$$



54.785 = 110110. 110100011110101 11

$= 0.11011011001000111101011 1 * 2^6$

But since 23 bits are available for the representation of Mantissa, the binary digits at the right are truncated. This still captures the original number with minimal error.

$0.11011011001000111101011 1 * 2^6 = 54.784999847412109375$

If the mantissa is not long enough or less than 23 bits, we append zeros in the right side as the decimal is in the beginning or left most side. Nevertheless, to represent the number in float 32 format we must write exponent in binary

6=110=00000110 (for exponent we append zeroes in left side). Hence the decimal number 54.785 can be represented in float 32 as:

0	00000110	11011011001000111101011 1

If the exponent is negative, then one way to write negative numbers in 8 bit is in the form of 2's complement.

For instance, if the exponent is -2 then

2=0000 0010

-2=1111 1110 (2's complement)

Another way we can use biasing and write floating numbers as

$-1^{S} * 0.1M * 2^{(E-B)}$.

Where Bias (B)= 2^7 for 8-bit representation

So, to represent -2, the process looks like

Exponent= -2+128=126

= (0111 1110)

Similarly, the original number 54.785= 0.11011011001000111101011* 2^6 which had exponent 6 will be represented as:

Exponent (E) =6+128= 134= 1000 0110

0	1000 0110	11011011001000111101011 1

Quantization of floating points to integer

Only 256 values can be represented in int8, while float32 can represent a very wide range of values. The idea is to find the best way to project our range [a, b] of float32 values to the int8 space. When inferencing the int8 weights are multiplied with the scale factor as part of de-quantization formula.

0.5279287	1.61614345	-1.38775427
0.41121489	0.33652957	-0.70687819
-0.6734712	-0.91871875	-0.29793607

Before Quantization

35	127	-128
25	18	-70
-67	-88	-35

After Quantization

Figure 5-5: Quantization example

For the above example clipping range was calculated as

$[\alpha, \beta]$ = [Min(weights), Max(weights)]

= [-1.38775427,1.61614345]

Scaling factor(S) = $(\beta-\alpha)/ (2^8-1)$ =0.01177999106

Zero point(Z)=0

Quantized weight (say 35) = [{0.5279287- (-1.38775427)}/0.01177999106]-128

Post Training static quantization

This includes quantizing both the weights and activations values. To estimate the typical clipping range, 1000 images that represent the dataset were taken as calibration input. With the help of these images, activation values were recorded by observer modules in the PyTorch library. During inference activation values were quantized with the recalculated parameters for quantization function.

Quantization Aware training

Quantization aware training QAT) was done for 20 epochs. QAT involves forward propagating with quantized weights then calculating the cost and gradients with respect to those weights. While propagating backwards the gradients are floating points and not int. Performing the backward pass with floating point is important, as accumulating the gradients in quantized precision can result in zero gradient or gradients that have high error, especially in low precision.

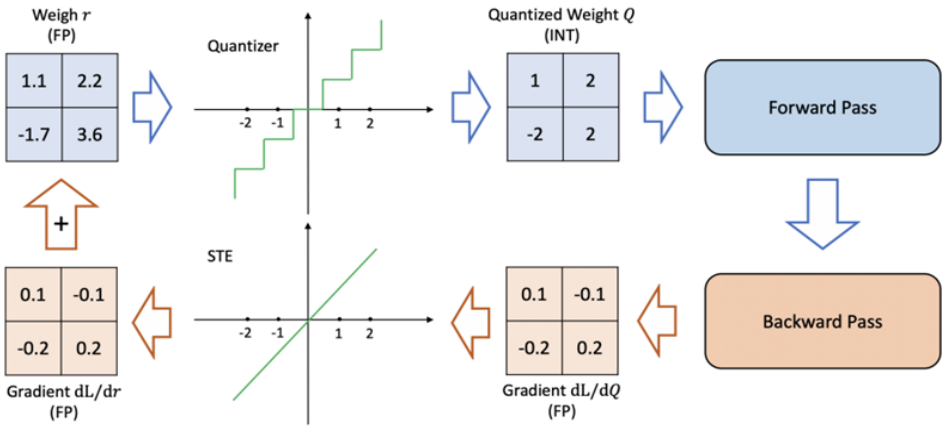


Figure 5-6: Straight through Estimator example [25]

The STE (straight through Estimator) derivative approximation plays a very important role because the rounding operation for getting quantized weights has zero gradients as it is a piecewise flat operator. So, the STE treats the quantization and de-quantization function as if it were identity function in the clipping range [α, β] and constant function outside the clipping range [α, β]. Therefore, the resulting derivatives are 1 in the clippings range [α, β] and 0 outside the clipping range [α, β]. Thus, the gradients for the quantization and dequantization operations are calculated.

5.2 Implementation on FPGA

Xilinx provides Intellectual property (IP) cores that provides functionality such as AXI infrastructure, allowing us to focus on the Convolutional Neural Network of the project. We implemented these IP cores to implement DNN in FPGAs.

5.2.1. Zynq MPSoC IP

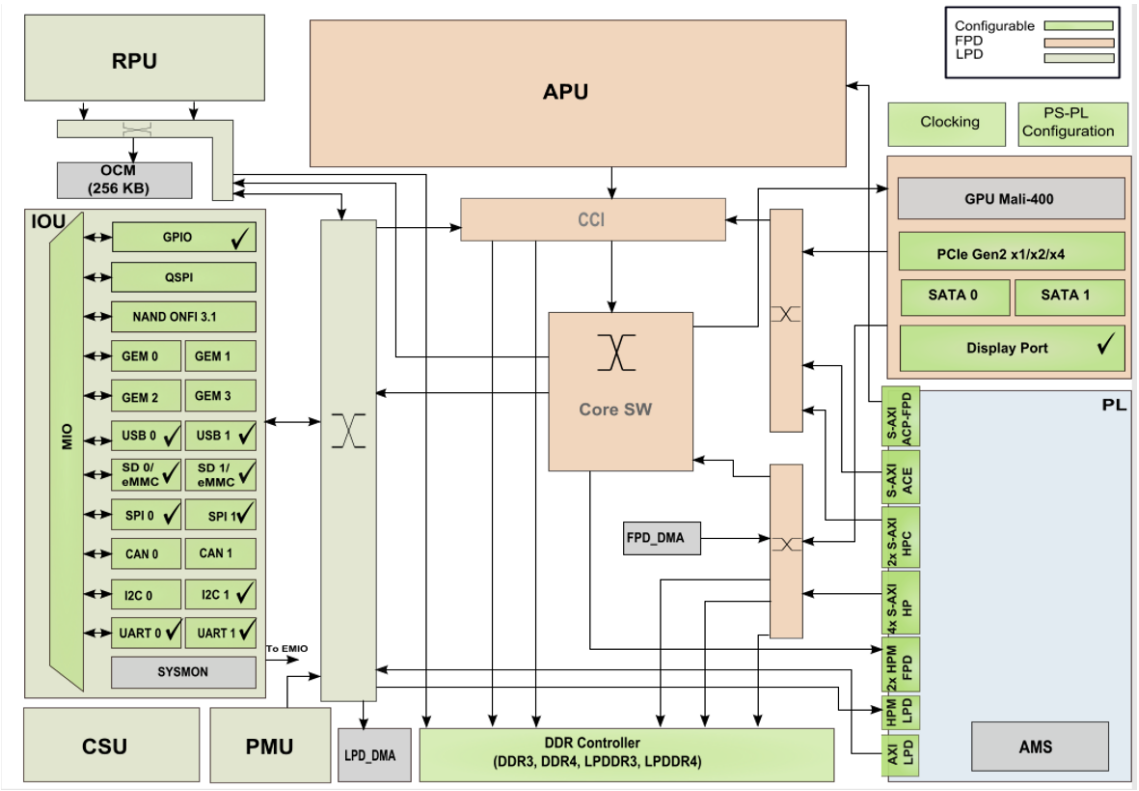


Figure 5-7: MPSoC Top level block diagram

Zynq MPSoC IP is a software interface for Ultrascale+ MPSoC processing system and provides logic connection between the PS and Pl. The IP wrapper instantiates the PS part of device. It provides functionalities such as PS internal clocking, IOP interfacing, AXI interfacing, PL clocks and interrupts, etc. When the board preset is applied to the MPSoC IP, several changes are made to configure the system to work with the Ultra96v2 board. These changes typically include:

Pin Mapping: The board preset configures the MPSoC IP to utilize the correct pins and interfaces available on the Ultra96v2 board.

Clock Configuration: The board preset sets up the clocking scheme for the MPSoC IP based on the Ultra96v2 board's requirements. This includes configuring the clock sources, divisor, and PLLs to provide the necessary clock signals to the IP.

Peripheral Enable: The board preset enables and configures various peripherals available on the Ultra96v2 board. This includes ports such as GPO (General Purpose Outputs), USB (Universal Serial Bus), SPI (Serial Peripheral Interface), UART (Universal Asynchronous Receiver-Transmitter), and the display port. By enabling these peripherals, the MPSoC IP can interact with external devices or interfaces connected to the board.

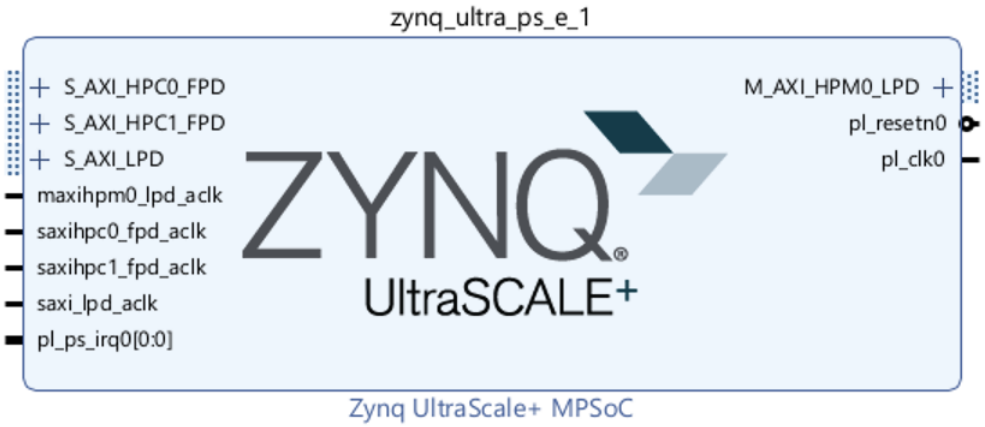


Figure 5-8: MPSoC IP Block Diagram

The MPSoC utilizes the AXI4 interface for instruction and data transfer between the PS (Processing System) and the PL (Programmable Logic).

For instruction transfer between the PS and PL, the M_AXI_HPM0_LPD interface is utilized. This AXI interface is memory-mapped and offers configurability for various data widths. As a 32-bit data width was sufficient for the transfer of instructions, a 32-bit data width was chosen instead of 128 or 64-bit data widths. To guarantee proper clock domain synchronization for the Master port, the signal maxihpm0_lpd_aclk is configured to be clock domain synchronous.

For data transfer between the PL and DDR memory, S_AXI_HPC {0,1} _FPD interfaces are employed. These interfaces have a 128-bit data width, which enables high-bandwidth data communication. The PL serves as the master in this configuration, reading and/or writing data to the DDR memory of the device.

Additionally, for instruction transfer between the DPU and the PL, the S_AXI_LPD interface is utilized. It has a 32-bit data width and allows communication between the DPU and PL.

- pl_ps_irq[0:0] serves as an Interrupt Bus Lane, conveying interrupts from the Programmable Logic to the Processing System.
- pl_resetn0 is a reset signal transmitted from the Processing System to the Programmable Logic, and it is active low.

To maintain synchronization, pl_clk0 is the clock signal with a frequency of 100MHz, transmitted from the Processing System to the Programmable Logic.

Table 5-1: PS-PL interface used in MPSoC IP

Port name	Description	Master	Slave	Data Width
M_AXI_HPM0_LPD	For data transfer between PS-DDR and PL by FP-DMA.	PS FPD	PL	128/64/32
S_AXI_HPC0_FPD	High performance coherence port connected CCI and Memory management unit (MMU)	PL	PS FPD	128
S_AXI_HPC0_FPD		PL	PS FPD	128
S_AXI_LPD	Path from PL to PS	PL	PS FPD	128/64/32

5.2.2. DPUCZDX8G

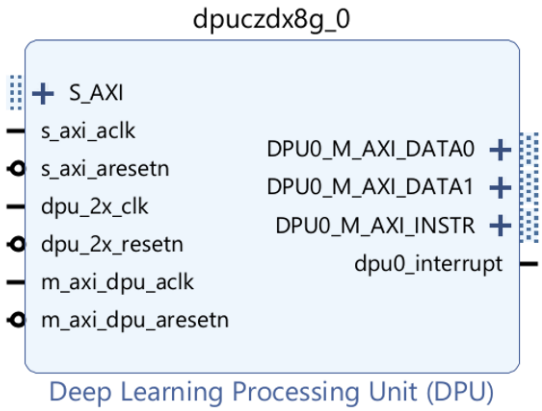


Figure 5-9: DPU IP block

The DPUCZDX8G IP comes with a range of architectures, each denoted by a specific letter followed by a number representing the amount of parallelism implemented in the design. The available architectures are B512, B800, B1024, B1152, B1600, B2304, B3136, and B4096. After consideration, we used B1024 architecture which includes 8 pixel parallelism, 8 input and output channel parallelism with low RAM usage and low DSP usage.

The DPU operates by receiving instructions from the Processing System (PS) through the S_AXI interface. To read or write data, it utilizes DPU0_M_AXI_DATA{0,1} interfaces. While multiple DPU cores can be used, our design utilizes a single DPU core, which utilizes two master AXI data interfaces. Clock and reset signals accompany the interfaces to ensure proper functionality. The DPU employs two clocks:

- **1x Clock:** This clock is dedicated to general logic operations and functions within the DPUCZDX8G.
- **2x Clock:** The DSP slices require a clock operating at twice the frequency of the 1x clock.

Table 5-2: PS-PL interface used in DPU IP

Port name	Description	Master	Slave	Data Width
DPU_M_AXI_DATA_INSTR	Memory mapped AXI interface for instruction fetch	PL	PS	32
DPU_M_AXI_DATA_0	Memory mapped AXI interface	PL	PS	128
DPU_M_AXI_DATA_0		PL	PS	128
S_AXI	Interface for registers	PS	PL	32

5.2.3. Clocking Wizard IP

Figure 5-10: Clocking Wizard IP Block

The clocking wizard was configured with following ports:

clk_in1 (I): This is a Single-ended primary input clock port. It becomes available when a single-ended primary clock source is selected. This clock serves as the primary input for the clocking network.

clk_out[n] (O): These are user-specified optional output clocks of the clocking network. The index 'n' can range in value from 2 to 7. These output clocks provide flexibility for users to access different clock frequencies as needed for various components within the design.

5.2.4. Processor System Reset

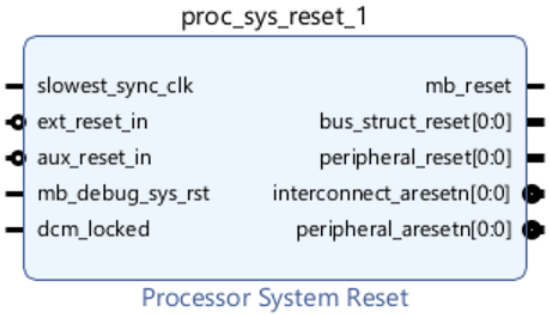


Figure 5-11: Processor System Reset IP block

To ensure proper operation of the DPU, we must implement reset that is synchronized with the clock source. Given the presence of three clocks (100MHz, 200MHz, 400MHz), it's essential to have each reset synchronized with its corresponding clock. Therefore, the Processor system reset serves the critical purpose of guaranteeing reliable and consistent operation of the DPU.

5.2.5. Implemented DPU design

The design presented utilizes the mentioned IPs to create a system with a connection between the DPU and the MPSoC. To achieve this, the DPU is implemented on the PL side.

The clocking wizard is a crucial component in the system, responsible for taking in a 100MHz clock signal and generating two output signals: one at 200MHz and another at 400MHz. The DPU is clocked at 200MHz and with a 2x clock frequency of 400MHz. To ensure proper synchronization, processor system reset modules are instantiated for each clock frequency. The resulting reset signals are then fed to their respective clock frequencies.

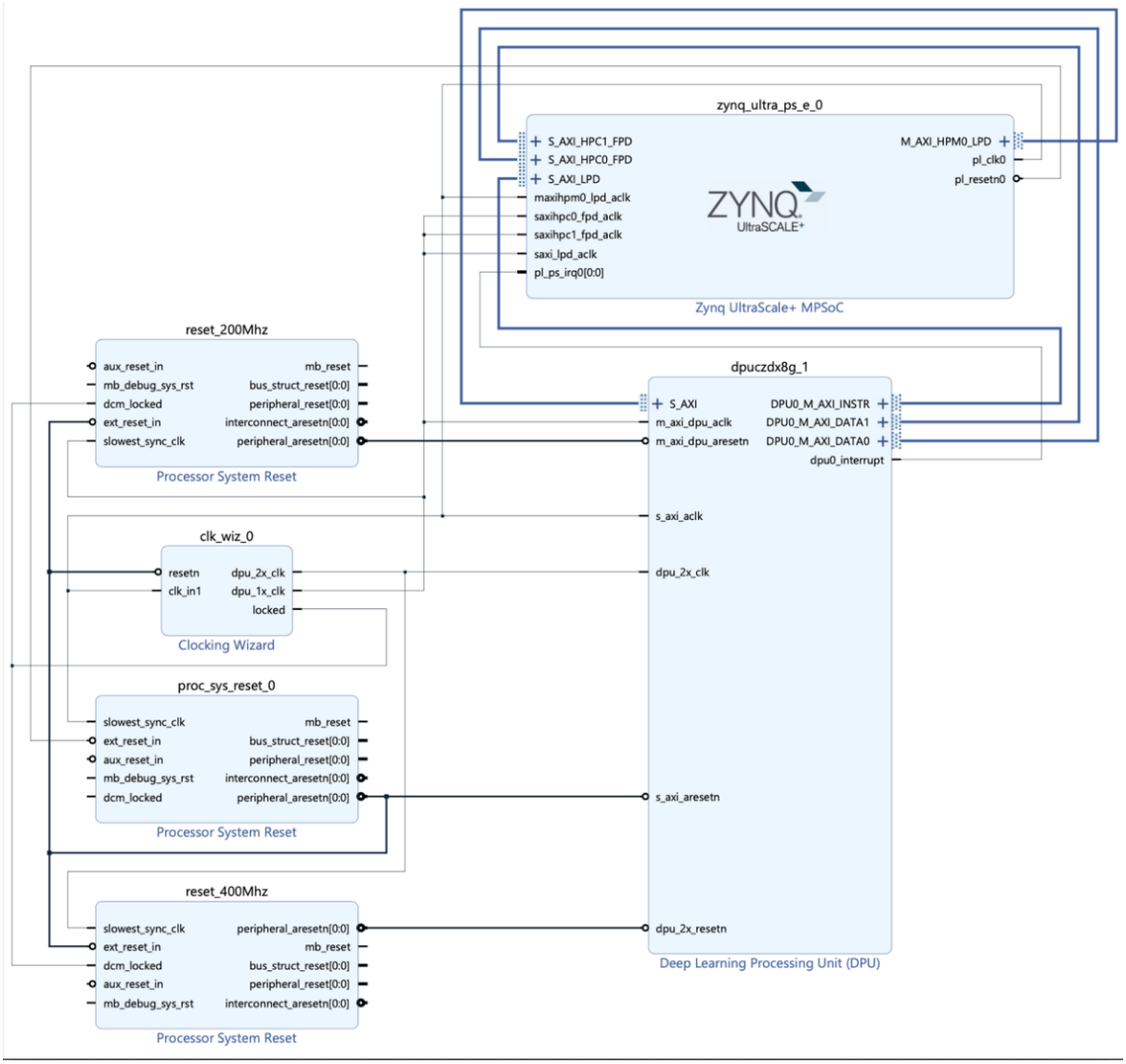


Figure 5-12: Implemented DPU Design

5.2.6. Quantization and Compilation of Model

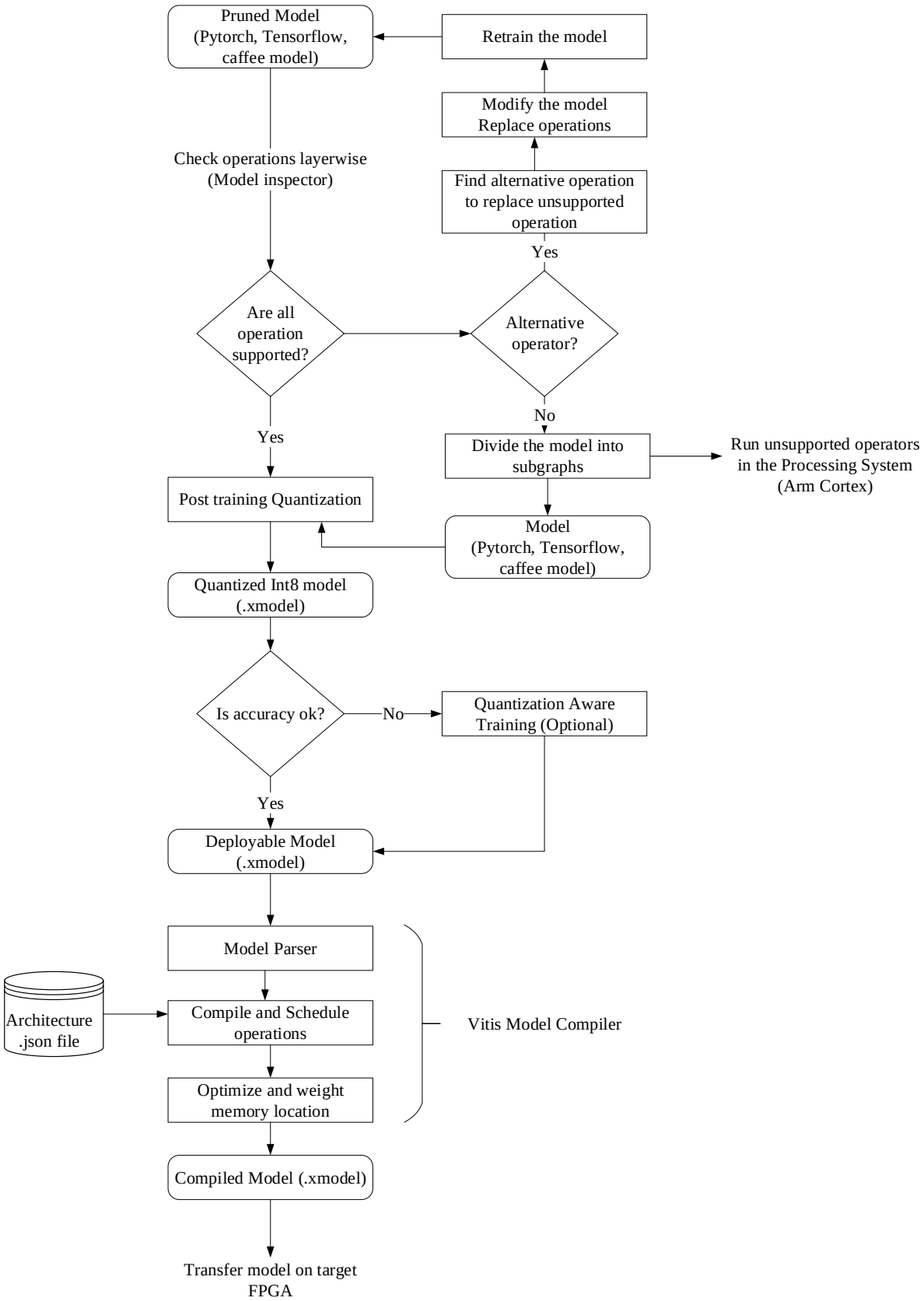


Figure 5-13: Quantization and compilation for deployment in FPGA

Pruning is a crucial step in model optimization, which is achieved through deep learning frameworks such as PyTorch or TensorFlow. Once pruned, the model is prepared for quantization and compilation, intended for deployment on FPGA.

The first phase involves a layer wise examination of the model to verify if all its operations align with the FPGA's implemented design. If any operations are unsupported, two strategies are available for resolution.

The first method involves implementing these unsupported operations in the Processing system (PS), which allows the compilation of high-level languages like Python and C/C++. The model is subdivided into subgraphs, with the FPGA executing a subgraph and returning an intermediate result. The Processing system then handles the unsupported operation. The result obtained is then passed to the next subgraph in similar fashion until the input is fully executed. However, due to the Processing system's constraints, latency increases when managing unsupported operations.

Alternatively, the second method involves replacing the unsupported operation within the CNN model itself. This process identifies a suitable operation to replace the unsupported one, carries out the replacement, evaluates the CNN model's performance, and retrains if necessary. Post-operator replacement, the model is then again checked layer wise for its unsupported operations. Upon passing this test, the quantization of the model proceeds.

The CNN model, initially operating in float32, is converted to int8 during quantization. The resulting quantized model (.xmodel) undergoes a performance test with a test dataset. If the model performs satisfactorily post-quantization, it is deemed ready for deployment. Optionally, users dissatisfied with the quantized model's performance may choose to utilize Quantization Aware training for fine-tuning.

The quantized model undergoes deployment preparations using the Vitis library vai_ic_xir, which compiles the model into a deployable format. This compilation involves an architecture JSON file specifying the implemented design's architecture, ensuring compatibility. The Vitis model compiler performs this compilation, producing an encrypted and deployable model. Subsequently, the model is transferred to the

FPGA board through an SSH connection. Once transferred, the model is available for inference through scripts.

5.2.7. Running CNN model in the board

The CNN inference model is compiled into the (.xmodel) format and is subsequently stored in the directory structure of the Embedded Linux environment, specifically Petalinux. Petalinux is a Linux-based operating system that executes on the Processing System of the MPSoC board. The Processing system that is in the Ultra96v2 board is Quad-Core Arm Cortex-A53. This processing system and the use of Embedded Linux allows communication between the FPGA (Programming logic) and the Processing System using AXI interface which was defined during the design phase.

The execution of compiled model is using high-level scripts like Python or C++/C within the Processing System. First the model is loaded and deserialized, converting the compiled model from byte form into a constructible object representing the CNN model. The object is then queried from the number of subgraphs, with a single subgraph indicating that the entire model runs on FPGA. Multiple subgraphs imply that the operations within the CNN model are not handled by the FPGA and those operations are executed on the Processing System. This, however, introduces a bottleneck due to the Processing System's inherent limitations in handling computationally intensive tasks like computer vision, leading to a deviation from the desired latency. To address this, a warning is triggered if multiple subgraphs are detected.

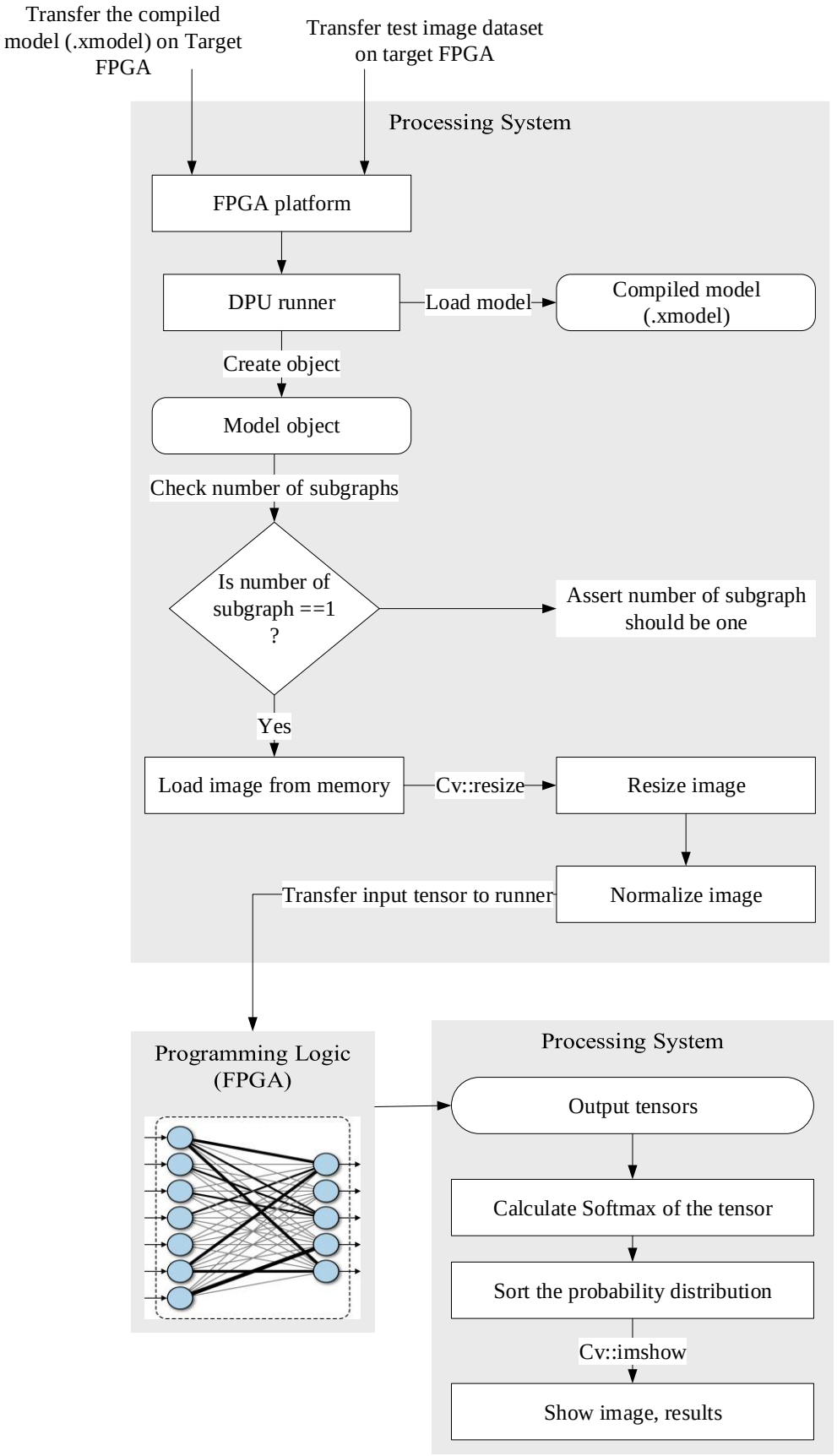


Figure 5-14: Running CNN model in the MPSoC

If there is a single subgraph of the model, then a runner class is instantiated for the subgraph of the model. Subsequently, the preprocessing of the input image for the CNN model starts. The preprocessing steps, involving loading the image from the SD card, resizing, and normalization, are executed within the processing system of the MPSoC board. The board image, compiled with the OpenCV library, utilizes the OpenCV cv::imread function to read images from the loaded data into memory. The loaded image is then normalized by segregating it into red, green, and blue channels, followed by reintegration of color information.

The preprocessed image is then forwarded to the DPU runner class. The runner class serves as an API that allows communication between the Processing system and the Programming Logic. Vitis AI provides a runner class used to submit input tensor for execution and receive output tensors for result storage. This function returns a job id along with the status of the function call.

Post-inference, the output tensor undergoes processing in the Processing System. The tensor goes through the SoftMax function that generates the probability distribution of the output tensor, which can be sorted to identify the result with the highest likelihood.

5.2.8. Implementation of Design using PetaLinux

Peta Linux is an embedded Linux Software development kit (SDK) targeting FPGA-based system-on-a-chip (SOC) designs or FPGA designs. To build the PetaLinux project, we transfer our entire Vivado project to a Linux machine including all the PetaLinux tools installed. The implementation process will begin with the creation of Petalinux project.

The project can be created anywhere, but it should be created in the Vivado project directory created earlier so that the corresponding hardware is clear. When simplifying the hardware design for hardware emulation, it’s recommended to keep all the peripherals that need device tree and drivers so that the auto-generated device tree can be reused. If the two design has different address-able peripherals, we will need to create two sets of device trees for hardware running if we use BSP including custom design of the hardware.

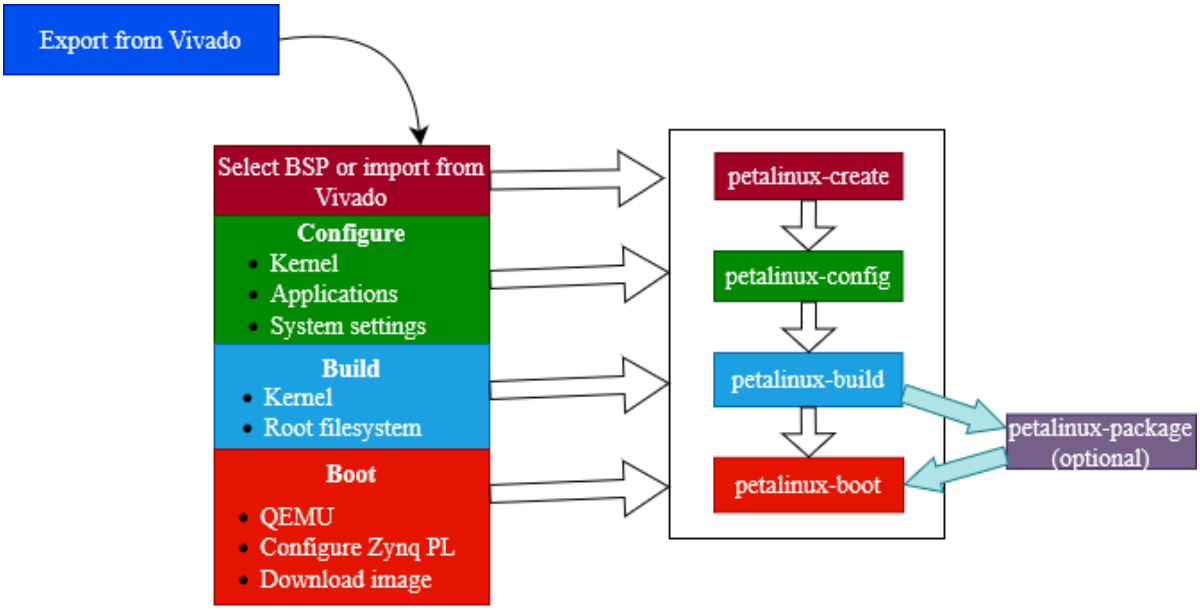


Figure 5-14: PetaLinux Tools Flow

The summary of the development of custom Linux for Ultra96v2 is shown in figure. The complete flow along with code with its description is visualized in **Appendix C.**

5.2.9. Latency profiling of the CNN model

Profiling the latency of the deployed model involves assessing the time required for model inference and the time taken for image loading, resizing, and normalization.

Two methods were tested to calculate the latency of the model. The first method utilizes Python's time library, implemented within the Processing System of the board. In this approach, the current time is recorded using time.time() just before transmitting the input tensor to the Programming Logic (FPGA). Subsequently, the system waits until the output tensor is obtained. Upon receiving the result, the time is recorded once again recorded using time.time(). The difference between the two-time measurements yields the latency of the model's inference. Similarly, the latency information for preprocessing steps, including image loading, resizing, and normalization, can also be determined using this method.

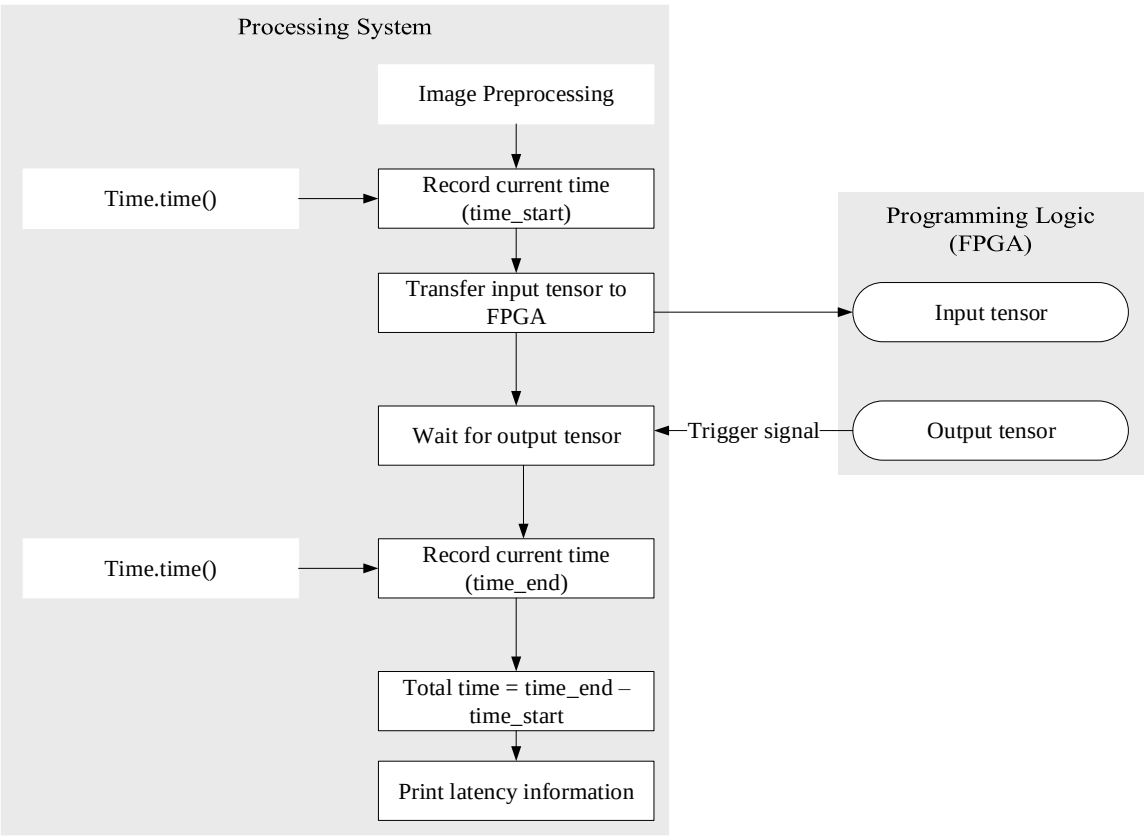


Figure 5-15: Overall latency profiling of the CNN model

The second method involves the use of the vaitrace library, provided by the Vitis AI library. To implement this library, a configuration file must be defined, specifying the type of profiling required: normal (for overall latency profiling of the CNN model) or fine-grained (for layer wise latency information of the CNN model). The configuration file also includes commands specifying the script file, image location, compiled model, libraries to profile (e.g., OpenCV), and custom functions defined in the script, such as calculating SoftMax or obtaining TopK results from the SoftMax output.

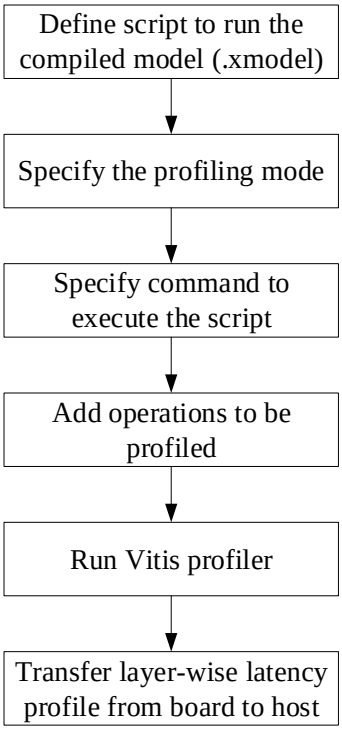


Figure 5-16: Layer-wise latency profiling of model using Vitis Profiler

The profiler generates a CSV file that can be transferred to another system through SSH connection. Once the generated files are transferred to the host computer, the results can be analyzed.

5.3 Hardware Aware Neural Architectural Search

To search for efficient architectures for hardware efficiency on FPGAs we have incorporated two constraints, arithmetic intensity and latency in evolutionary search algorithm. For this, efficiency estimator was implemented which returns latency and arithmetic intensity given a sample architecture.

Another step in the search process is to accurately estimate how an architectural would perform on the given dataset. For this accuracy predictor is used which takes an architecture configuration as input and predicts the accuracy of network.

Once efficiency predictor and accuracy predictor are ready, the evolutionary search process begins. The algorithm uses these two as parameters and guides the search process.

5.3.1. Accuracy Predictor

The accuracy predictor was trained on a dataset of 50k (architecture, accuracy) pairs constructed by randomly sampling subnets from OFA and then evaluating them on an ImageNet validation set of 10k images. Architecture encoding is used in order to represent each network and then this encoding is used as input for the prediction.

Architecture encoding for accuracy predictor

At first the architecture is encoded in the form of dictionary then this encoding is converted in to one hot encoding.

{'Ks': [3, 7, 3, 5, 3, 5, 5, 5, 7, 5, 3, 3, 5, 3, 5, 5, 7, 3, 7, 5],

'e': [4, 6, 3, 6, 3, 4, 3, 3, 3, 6, 6, 3, 4, 6, 3, 4, 6, 3, 6, 3],

'd': [3, 3, 4, 2, 3]}

This is architecture encoding of a random subnetwork in our search space.

One hot Encoding Representation

a) **Kernel size: 3= [**1,0,0], 5 = [0,1,0], 7 = [0,0,1]

b) **Expand ratio:** 3= [1,0,0], 5 = [0,1,0], 7 = [0,0,1], embedding is same for overall width multiplier of the network.
c) **Skipped blocks:** represented by [0,0,0] for each kernel size and expand ratio
d) **Depth** is encoded in the vector itself. Less depth than the base depth is represented by skipped blocks.

Sample one hot Network Embedding

[0 0 0 0 1 | 1 0 0 | 1 0 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 1 0 | 0 0 1 | 0 0 1 | 0 0 1 | 0 0 1 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 1 0 0 | 0 0 1 | 0 1 0 | 0 1 0 | 0 0 1 | 0 0 1 | 0 1 0 | 0 0 0 | 0 0 0 | 0 1 0 | 0 1 0 | 0 0 1 | 0 0 1 | 0 0 0 | 0 0 0 | 0 0 0 | 0 0 0 | 0 1 0 | 0 1 0 | 1 0 0 | 1 0 0 | 0 0 1 | 0 1 0 | 0 0 0 | 0 0 0 | 0 0 1 | 0 0 1]

MLP network summary:

(0): Linear (in_features=128, out_features=400, bias=True)

(1): Linear (in_features=400, out_features=400, bias=True)

(2): Linear (in_features=400, out_features=400, bias=True)

(3): Linear (in_features=400, out_features=400, bias=True)

(4): Linear (in_features=400, out_features=1, bias=False)

Each dense layer used ReLU activation function. The binary one hot vector representing the encoding of the architecture is 128 dimensions. The network contains a few hidden layers and finally an output neuron to predict percentage accuracy.

5.3.2. Efficiency predictor

Arithmetic Intensity Calculator

To calculate the arithmetic intensity of a network we need MACs and size in bytes of the entire network. For this total number of MACs and size in bytes of each possible layer are stored in the lookup table. Arithmetic intensity calculator receives the encoded configuration of model. Based on this all the data (MACs and size in bytes) of each block are fetched from the stored table. Thereafter the total number of macs and size of parameters in the byte is calculated by summing up the values from individual layers. After this arithmetic intensity is calculated and is returned to the evolutionary algorithm

5.3.3. Latency Estimation using Latency Lookup Table

The latency table contains the latency of each block. Where latency predictor is utilized to calculate the latency of the network. This works since most FPGAs including Xilinx DPU, run the network layer by layer. Thus, if we get the latency for each possible configuration of each layer, the latency of the network can be calculated by simply adding them up.

As there are 95 unique blocks in our search space, we need to calculate the latency of each block and then store it in to our table. In order to do that we first quantized and compiled each block in a similar fashion to implementation of EficientnetV2 on FPGA.

Thereby we quantize and compile all 95 blocks was using Vitis AI and then it is deployed and inferenced on FPGA. Thereafter we utilized latency profiling of blocks layer by layer as mentioned in **Latency profiling of CNN model** on Board part. Once the latency of each block is obtained, a table is constructed that contained latency of each block and its configuration.

The table is then utilized to predict the latency of each block and is implemented as latency estimator. This latency estimator is used in evolutionary search process to search for latency aware networks.

5.3.4. Evolutionary Algorithm

At first population of a size P is seeded by randomly sampling the architectures from the search space. Each sampled architecture must satisfy efficiency constraint. While sampling if the sample satisfies constraint it is added to initial population otherwise is rejected.

Once the seeding completes, we move towards generating the next generation of population. For this best performing samples from the population are selected based of parent size in order to produce child. Certain percentage of samples from parents are mutated (say x) and are evaluated for efficiency and they are checked to satisfy constraint, if they satisfy then are append to child pool.

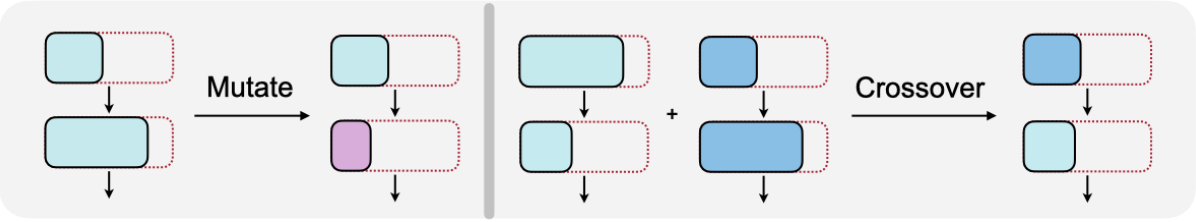


Figure 5-17: Cross over and Mutation

Mutation involves changing the attributes or parameters within the search space of the network in the current generation. Mutation is used to diversify the population of solutions and enhance algorithm's exploration of the search space. Randomly changing the size of kernel, number of channels, and depth of blocks.

Crossover selects attributes for each block from parent's parameters. Crossover generates new offspring that potentially capture better traits from the parents.

Once the child population is spawned, the population is then evaluated for accuracy. The best performing samples from the pool are now selected as parent for next generation and the process is repeated till the predefined number of generations is reached.

5.4 Different designs for Hardware

The hardware design relied on the Tensil library, which allowed customization options for Convolutional Neural Networks (CNNs). This library allowed for the specification of key architectural parameters for board configuration. These parameters included:

- Data type: This denotes the numerical format used for hardware calculations, such as FP16BP8 (Fixed-point format with a 16-bit width and an 8-bit binary point).
- Array size: The size of the systolic array, e.g., 8.
- DRAM0, DRAM1 depth: The number of vectors allocated in DRAM for bank 0 and bank 1.
- Load depth: The number of vectors allocated in the logic fabric.
- Accumulator depth: The number of vectors allocated in fabric memory for accumulation.

In the experimentation phase, the focus was on varying the systolic array size and accumulator depth. For the initial run, an array size of 16 and an accumulator depth of 4096 were chosen. In subsequent runs, the array size was reduced to 8, with an accumulator depth of 2048, while keeping other parameters constant.

The architectural parameters between the two runs have been tabulated below.

Table 5-3: Architectural parameters between different runs

Parameter	Design-1	Design-2
Data type	FP16BP8	FP16BP8
Array Size	16	8
DRAM0 depth	2097152	2097152
DRAM1 depth	2097152	2097152
Local depth	20480	20480
Accumulator depth	4096	2048
SIMD registers depth	1	1
Stride0 depth	8	8
Stride 1 depth	8	8

5.4.1. Comparison between the two designs:

The comparison between the two designs centered on the resource consumption of the FPGA board, which includes factors like LUTs, BRAM, and DSP. Additionally, power consumption was analyzed to gauge the energy usage of each design.

Resource consumption:

In the first run, using a larger array size of 16 and an accumulator depth of 4096, the consumption of resources was as follows: LUTs were at 35%, BRAM at 83%, and DSP at 76%.

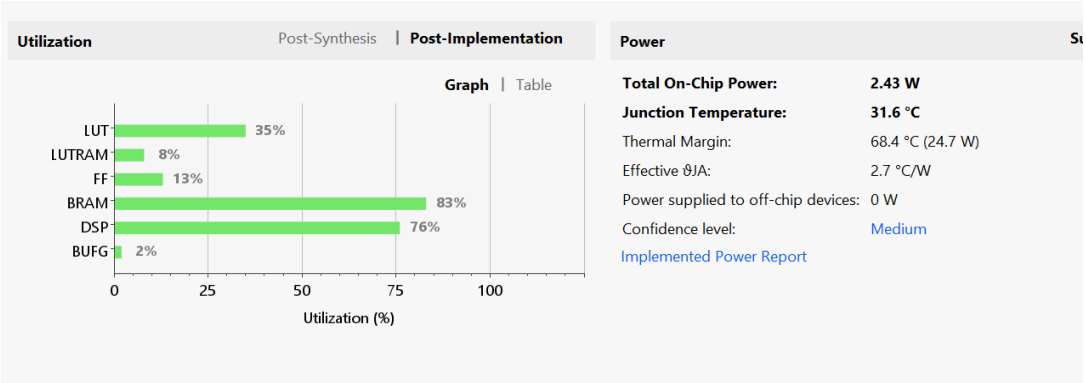


Figure 5-18: Resource utilization of design-1

For the second run, implementing a smaller array size of 8 and an accumulator depth of 2048, the resource consumption decreased notably: LUTs were at 20%, BRAM at 41%, and DSP at 20%.

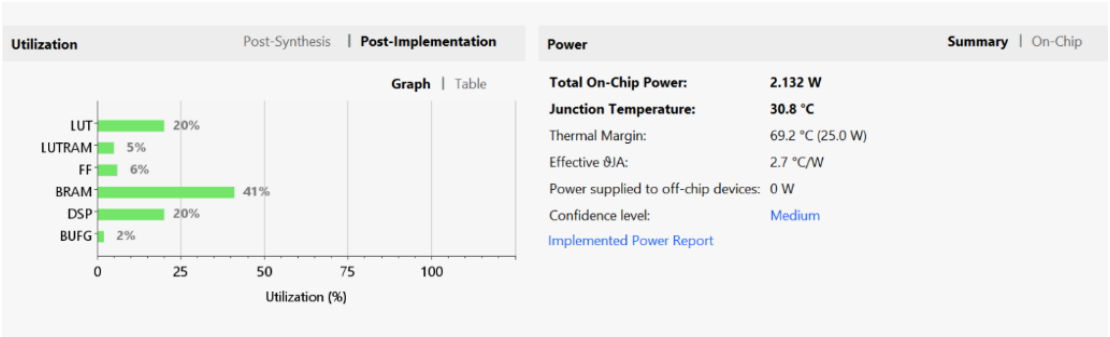


Figure 5-19: Resource utilization of design-2

Power comparison:

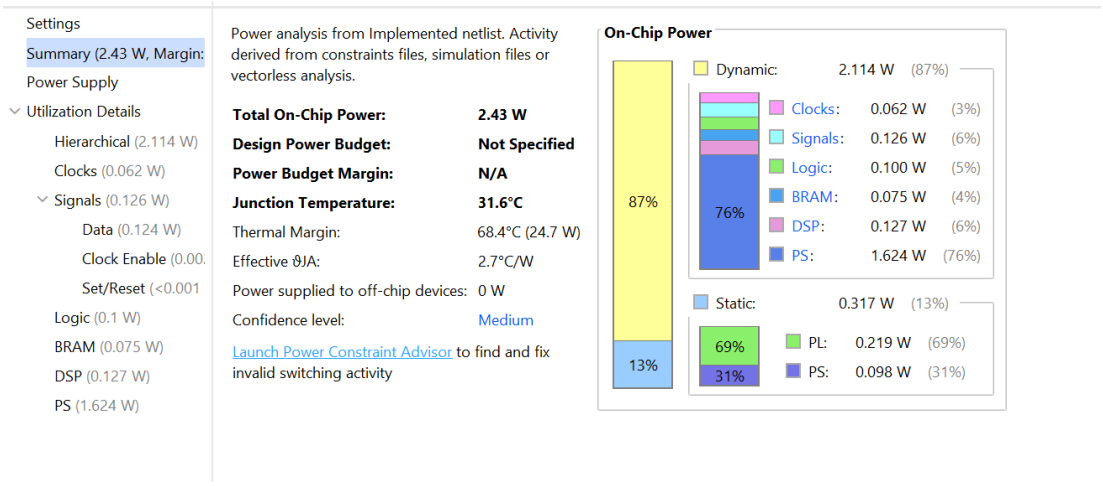


Figure 5-20: Power consumption of design-1

In design-1, the total power consumption observed was 2.43 Watts. This figure consists of two components: dynamic power and static power. Dynamic power refers to the power consumed by the device when it is active, performing computations or tasks. In this case, design-1 consumed 2.114W of dynamic power. Static power, on the other hand, represents the power consumed by the device when it is in an idle or inactive state, where there is no activity or computation being performed. In design-1, the static power consumption was measured at 0.317W.

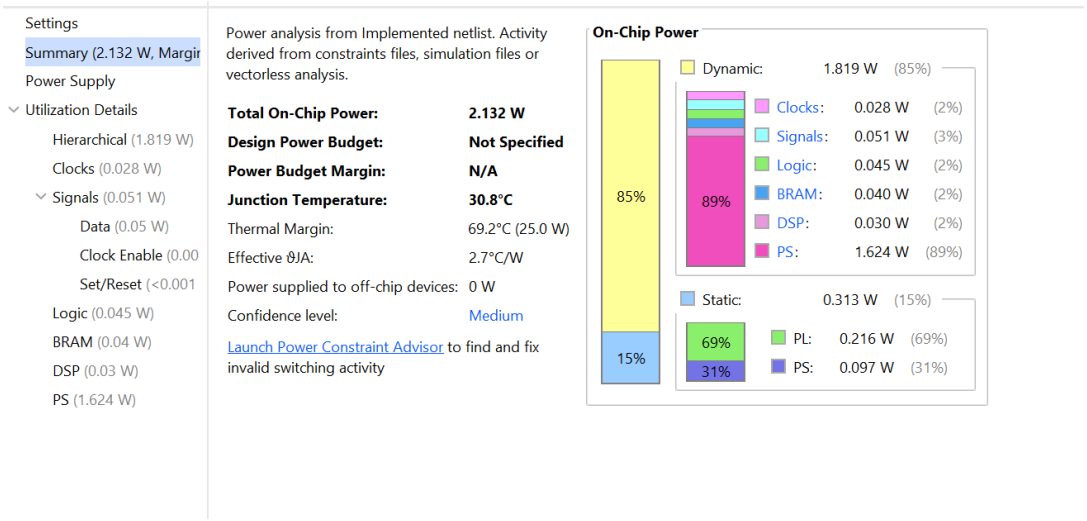


Figure 5-21: Power consumption of design-2

Similarly, for design-2, the total power consumption was 1.819 Watts. This also comprises dynamic and static power components. Design-2 consumed 1.819W of

dynamic power, indicating the power consumed when the device is actively processing tasks. Additionally, the static power consumption for design-2 was measured at 0.313 Watts, denoting the power consumed when the device is idle or not actively performing any computations.

Timing Requirement

Worst negative slack in both the designs being positive means both the design meets the timing requirement.

Table 5-4: Timing requirement of two designs

	Worst Negative Slack	**Worst Hold slack**	**Worst Pulse width slack**
Design-1	0.436ns	0.010ns	3.500ns
Design-2	0.239ns	0.011ns	3.500ns

6. RESULT AND ANALYSIS

6.1 EfficientNet-V2 on CIFAR-10

The retrained model on CIFAR-10 dataset was able to accurately classify 97.92% of images in test set. The model was only trained for 20 epochs.

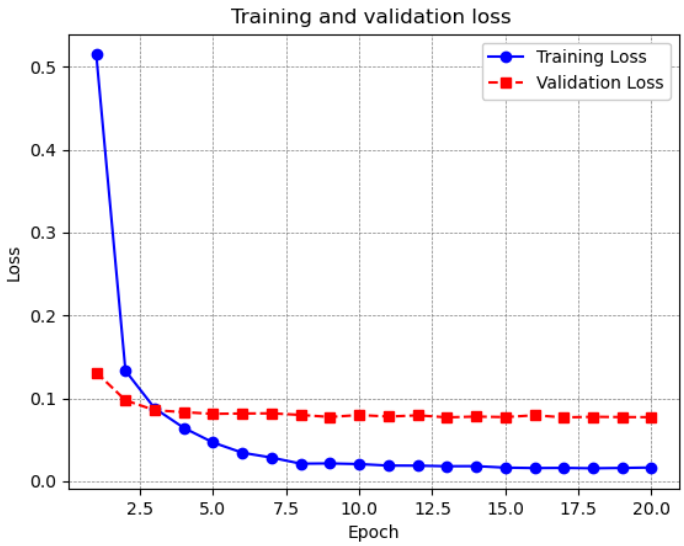


Figure 6-1: Training and validation loss across epoch

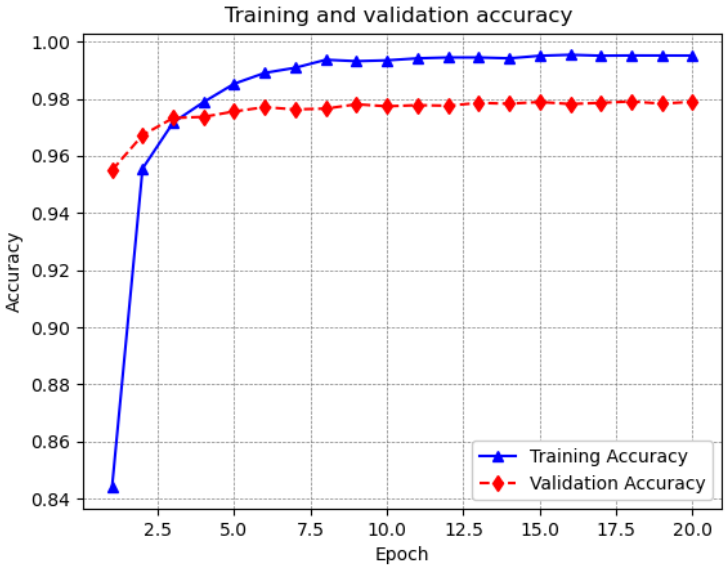


Figure 6-2: Training and validation accuracy evolution over epochs

The SOTA accuracy of EfficientNet-v2(s) on CIFAR-10 is 98.7%, we couldn't achieve this as the model was only re-trained for 20 epochs.

6.2 Unstructured Pruning

EfficientNetV2 was iteratively pruned and retrained, reducing parameters by 81% and maintaining 97.6% accuracy. Despite initial accuracy drops, retraining helped recover performance.

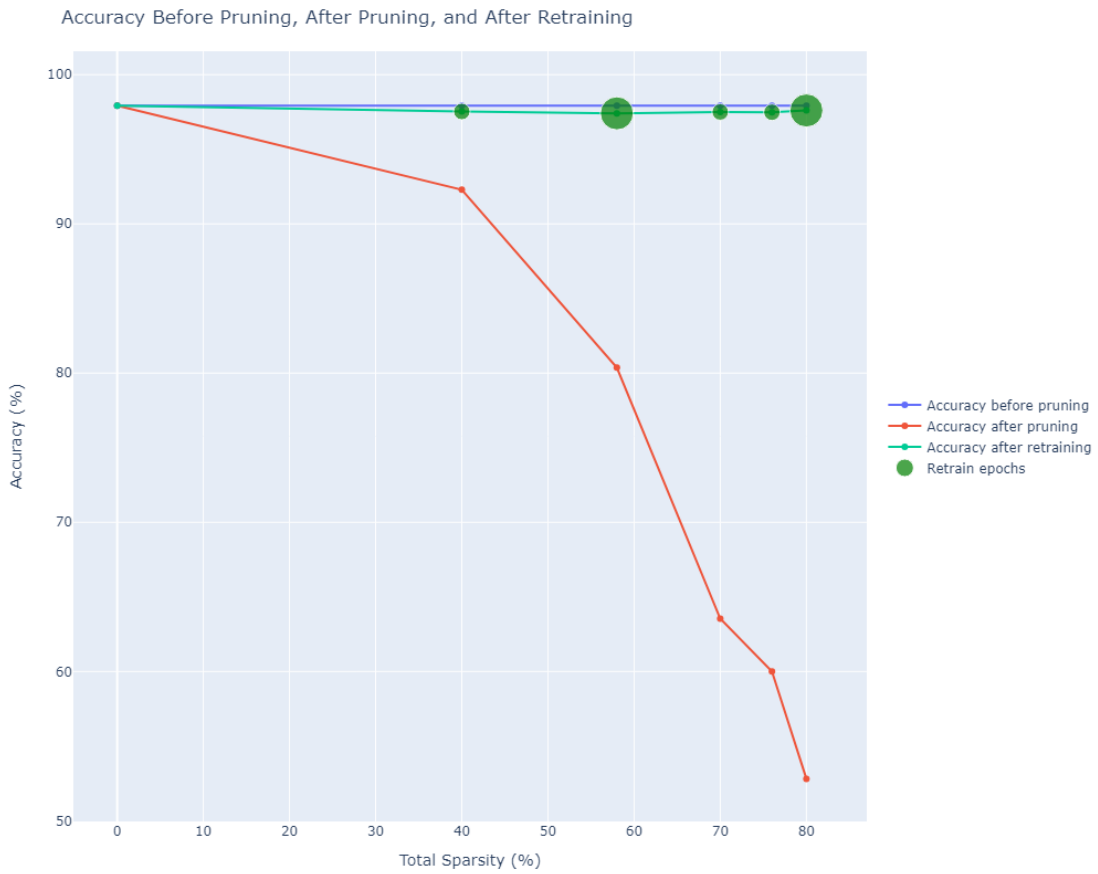


Figure 6-3: Pruning the parameters and recovering the accuracy

After iteratively pruning the architecture for five iterations, we were able to obtain an 81% sparse. We also tested structured pruning but as network architecture is very deep and pruning even a small percentage of the network resulted in huge dependencies and caused a great loss in accuracy.

Table 6-1: Sparsity and decrease in accuracy

Pruning Percentage	Sparsity	Decrease in Accuracy
40%	40%	7.71%
30%	58%	17.16%
30%	70%	33.87%
20%	76%	44.92%
20%	81%	40.00%

Parameter distribution of 81% sparse EfficientNetV2

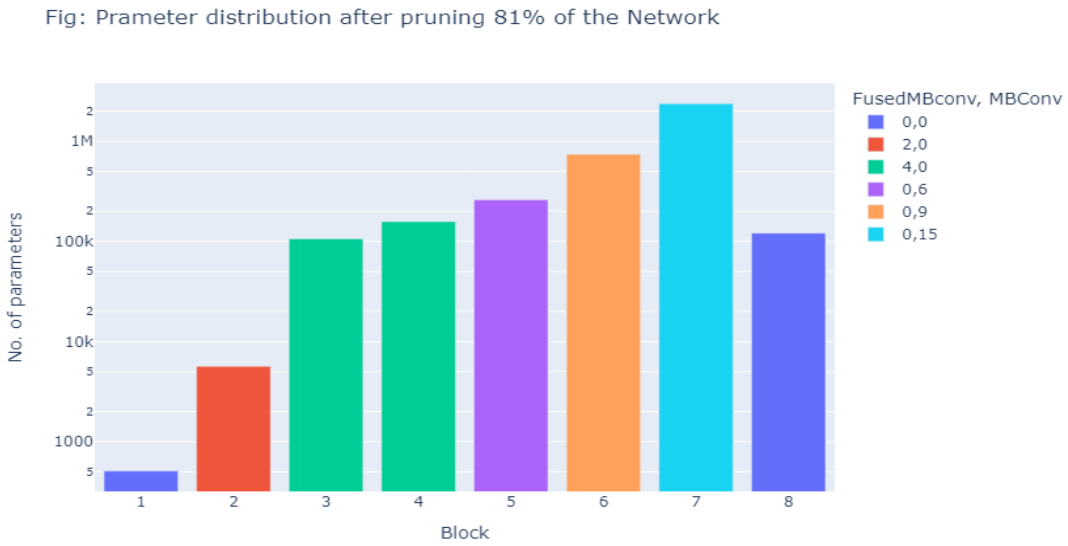


Figure 6-4: Parameter distribution of 81% sparse model

We can notice a significant difference in parameter reduction between the first and last blocks compared to the middle blocks. This indicates that the model had a substantial amount of redundancy in these blocks. To provide an example, let's consider Block No. 7, which initially consisted of 15 MB Conv units and had over 14 million parameters. Through pruning, it was possible to reduce this number by almost 85%, resulting in approximately 2 million parameters. In contrast, the first block had around 500 parameters, and despite the pruning process, the number of parameters remained relatively consistent.

Table 6-2: Parameters pruned in different superblocks of the network

Block No.	No. of Parameters Before Pruning	No. of Parameters After Pruning	Pruned Percentage
1	648	511	21.14%
2	10368	5667	45.34%
3	301824	106139	64.83%
4	586752	157641	73.13%
5	901888	260844	71.08%
6	3418368	739906	78.35%
7	14440896	2369347	83.59%
8	327680	120967	63.08%
Total	19988424	3761022	81.18%

6.3 Channel Pruning

The reduction in channels per layer is shown in the diagram below. After pruning all the layers have similar number of channels.

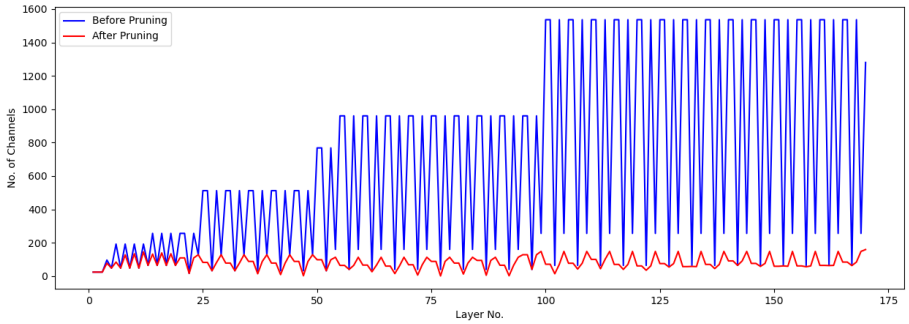


Figure 6-5: Illustration of reduction in channels after pruning

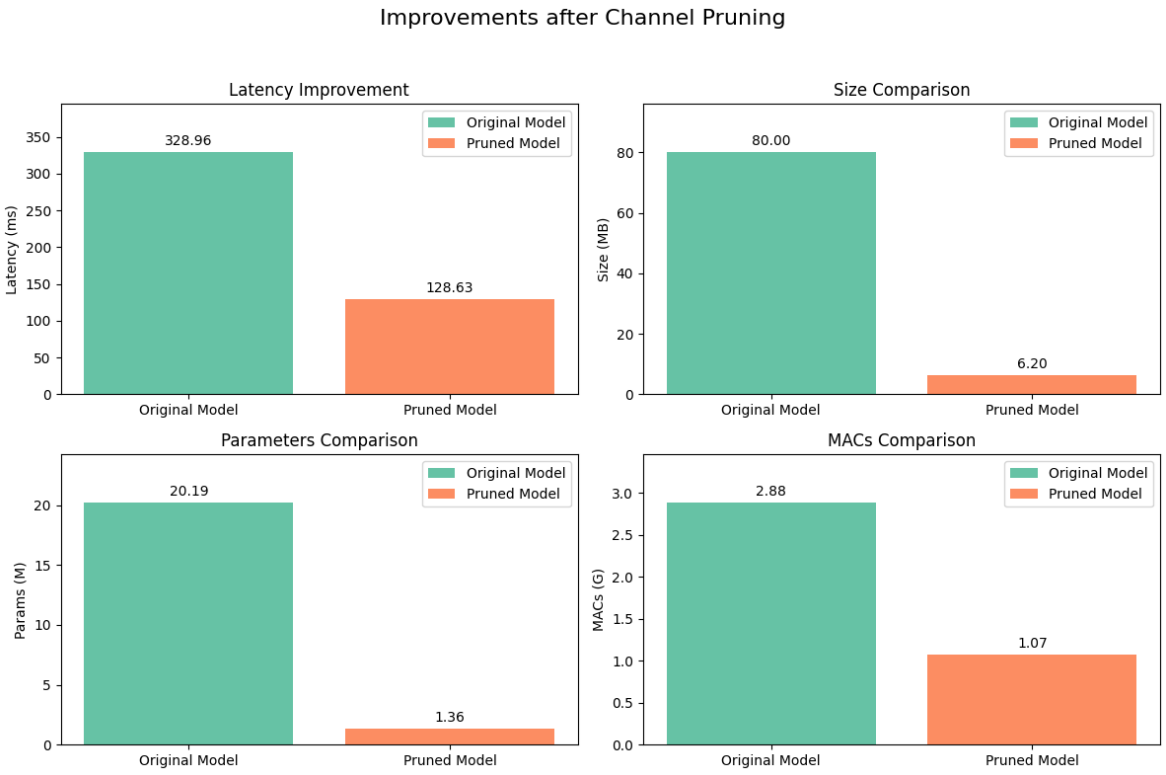


Figure 6-6: Efficiency Improvements after channel pruning

Whereas there was slight reduction in accuracy of the pruned model as illustrated in figure below.

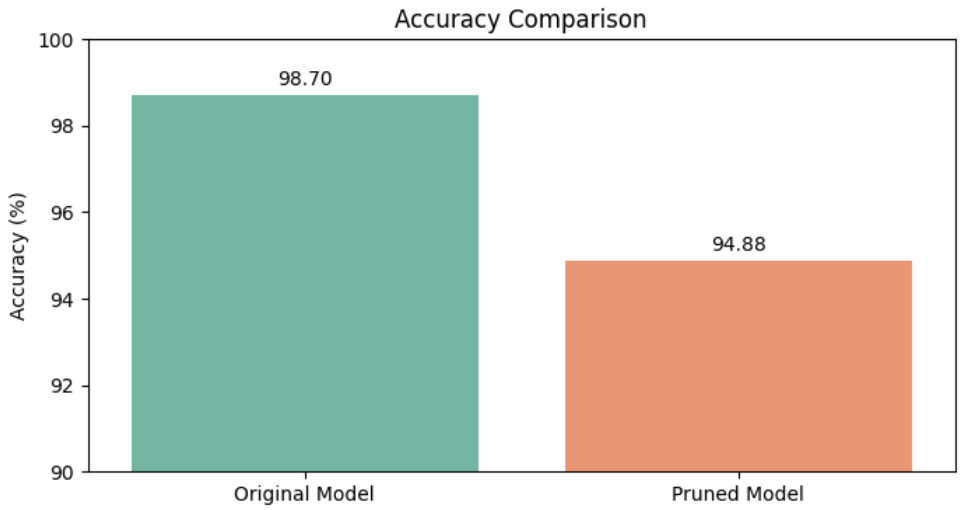


Figure 6-7: Accuracy comparison of original and pruned model

6.4 Quantization

The resulting model obtained from unstructured pruning after the pruning process with 81% sparsity and 97.7 % accuracy was quantized in different approaches. The results can be summarized in the table below.

Table 6-3: Quantization result

S.N	Quantization approach	Accuracy	Size (Mb)	MACs (B)
1	Non-Quantized Model	0.977	79.2	3.2247
2	Float32 to int8 (weights only)	0.922	20.3	-
3	Post training static quantization(diff)	0.72	20.4	3.4181
4	Post training static quantization(minmax)	0.64	20.4	3.4181

Initially the pruned model had a size of 79.2 megabytes and accuracy of 97.7%. After converting the weights into int8 accuracy dropped a little while size was reduced by around 4 times. The activations were still in fp32 format. Since the activations are stored in temporary memory, only the weights need to be considered while calculating the size on disk. After post training quantization the activations also got quantized due to which accuracy dropped significantly in both cases. QAT was able to recover some accuracy since it considers the error contributed by quantization process. The MAC operation of all the quantization operation is increased because extra calculation needs to be done for quantizing and dequantizing the weights and activations. Although MAC operation is increased, the inference time doesn't increase because calculation of int8 is faster than floating points.

Quantization of channel pruned Network

The channel pruned EfficientNet model was quantized using the calibration dataset of 2000 images. The resulting quantized model had drop in accuracy of 2%. The final accuracy was 92.2%. There were slight increase in number of MAC operations required and size of model. The final size of the model was 6.3MB.

6.5 Implementation of DPU on FPGA

6.5.1. Resource Utilization on B1024 DPU

The B1024 architecture of the DPUCZDX8G was chosen for implementation on the Ultra96-v2 board after an assessment of the available resources, including the number of LUTs, DSPs, and BRAM. We conducted a comprehensive comparison of different variants of the DPUCZDX8G and determined that the resource requirements of our project could be met by this particular variant. By selecting the B1024 architecture, the design's resource utilization aligns with the capabilities offered by the Ultra96-v2 board.

Table 6-4: Implemented design Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	35236	70560	49.94
LUTRAM	3182	28800	11.05
FF	51538	141120	36.52
BRAM	104	216	48.15
DSP	166	369	46.11
BUFG	3	196	1.53
PLL	1	6	16.67

Resource utilization provides detailed information about all the elements utilized in the design. In the case of our Ultra96v2 platform, we implemented a single core DPU with a particular emphasis on achieving low RAM usage and low DSP usage. The resource utilization can be observed in the table below.

6.5.2. Power analysis

Vectorless power analysis relies on assumptions regarding the switching probability and the percentage of time a signal is held high. During the initial power-up phase of a device, the current usage gradually ramps up until it reaches the static power level.

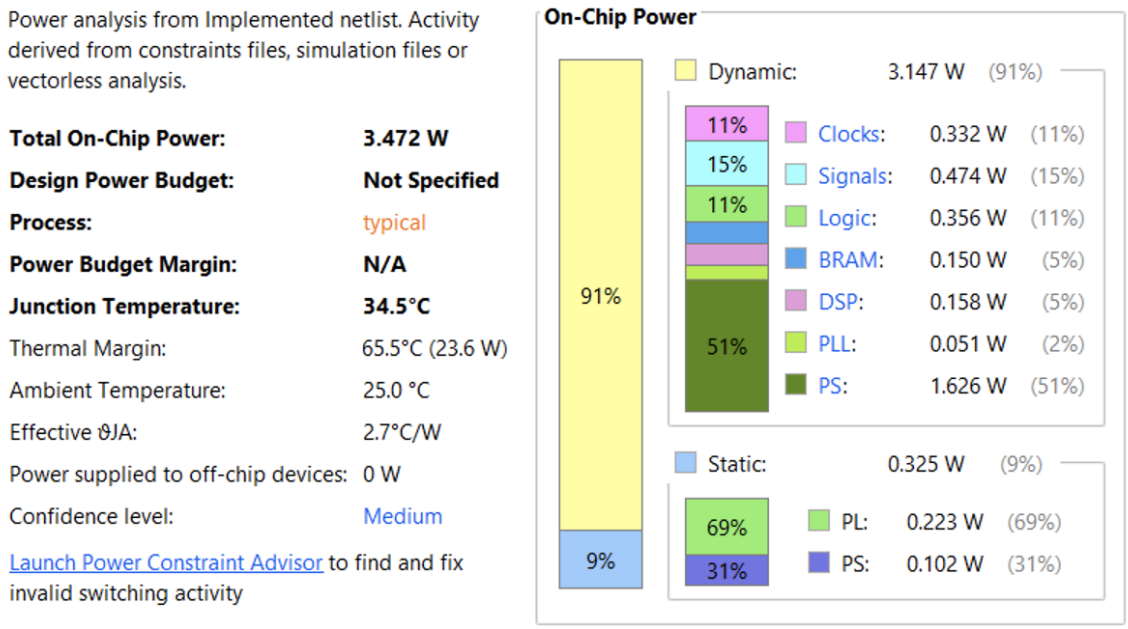


Figure 6-8: Power analysis of implemented design

Static power refers to the current drawn by device resources that are not clocked by the user's design. This includes leakage from all FPGA functional block resources and is influenced by the operating temperature environment. For our design, at a junction temperature of 34.5°C, the observed static power was observed to be 0.325W.

On the other hand, dynamic power depends upon voltage levels, logic, and routing resources utilized in the design. It is an instantaneous power that fluctuates at each clock cycle. The dynamic power (Pdynamic) encompasses the capacitive charging of transistors in the device, the momentary power expended during transistor switching, and the number of flip-flop switches occurring in each clock cycle.

A graphical representation of the power breakdown can be seen in the figure below, with PS accounting for 51% of the dynamic power.

6.5.3. Timing report

Slack, in the context of digital design, is the difference between the actual or achieved time and the desired time for that specific timing path. It helps measure the timing margin available in the design. Setup slack is calculated as follows:

$$\text{Setup slack} = \text{Data Required Time} - \text{Data Arrival Time}. \quad (8)$$

A positive setup slack indicates that the design is operating at the specified frequency and still has some additional timing margin. For the implemented design, the worst negative slack is positive at 0.512 ns for Setup and 0.010 ns for Hold. This means that the path meets the timing constraint with some margin to spare.

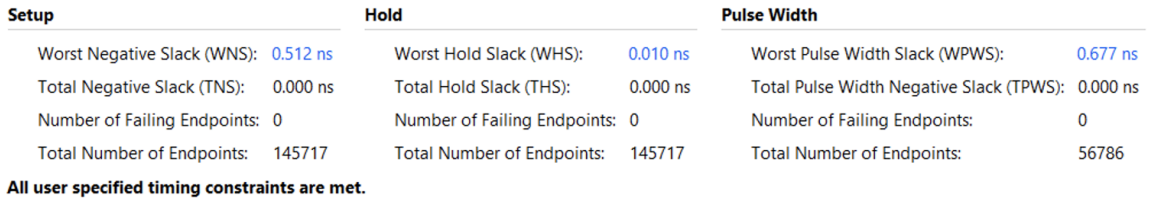


Figure 6-9: Timing report of implemented design

Total Negative Slack represents the cumulative sum of negative slack values in the design. A value of zero implies that there is no negative slack in the implemented design, which indicates the design has sufficient timing margins.

6.6 Deploying CNN models

We loaded the compiled model (.xmodel) of various CNN models and meticulously calculated their inference times. The determination of frames inferred per second (FPS) for each model was achieved by measuring the time span between sending input tensors and receiving output tensors. Several CNN models, including Resnet-50, Efficientnet-b0, Mobilenet-v1, and Mobilenet-v2, were performance tested, and the obtained results have been organized in the tabulated format below:

Table 6-5: Inference time of different Models

Model Name	FPS	Total frames	Time
Resnet-50	26.76	2880	107.6086
Efficientnet-B0	25.72	2880	111.993197
Mobilenet-v1	143.70	2880	20.041864
Mobilenet-v2	83.25	2880	34.595196

From the table, it is evident that CNN models like Mobilenet, specifically designed for mobile and embedded applications, exhibit superior latency performance compared to models such as Resnet-50 or Efficientnet-B0. This is particularly noteworthy as Mobilenet's design prioritizes efficiency and responsiveness. Additionally, the Once-For-All network leverages Mobilenet blocks, further substantiating the efficiency of such architectural choices in the context of latency-sensitive applications.

6.7 Latency Profiling of Model

To construct a comprehensive latency table, it is necessary to determine the latency of each constituent block within the Once-for-All Super-net. The latency of a CNN layer depends on various factors, including input shape, the number of input channels, output channels, and the dimensions of the kernel. To obtain latency information for all possible parameter combinations, inference must be executed on each block.

To achieve this, we designed a CNN composed of multiple CNN blocks. Subsequently, we used the Vitis Profiler to profile the CNN model to result a layer-wise profile. 10 images from the dataset were run for total of 8 times to get a total of 80 inference runs. The obtained results consists of metrics, including minimum time, maximum time, and average time for each layer during execution.

The latency information obtained for an example CNN model is presented below:

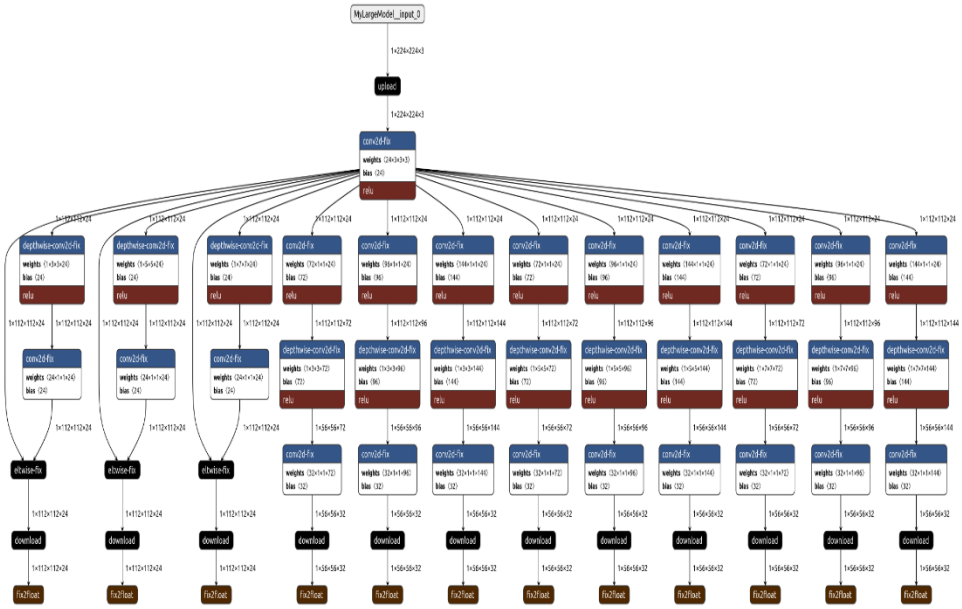


Figure 6-10: CNN model for layer-wise latency profiling

The sample latency information obtained for the aforementioned CNN model is presented below:

Table 6-6: Layer-wise Profiling of Blocks

Kernel Name	No. of Runs	Min Time (ms)	Average Time (ms)	Max Time (ms)
subgraph_MyLargeModel__MyLargeModel_myModel_block1__ConvLayer_blocks__ModuleList_0__Conv2d_conv__input_2	80	0.52	16.626	489.876
subgraph_MyLargeModel__MyLargeModel_myModel_block2__ResidualBlock_blocks__ModuleList_9__MBConvLayer_conv__Sequential_depth_conv__Conv2d_conv__input_62	80	1.364	1.47	5.938
subgraph_MyLargeModel__MyLargeModel_myModel_block2__ResidualBlock_blocks__ModuleList_9__MBConvLayer_conv__Sequential_point_linear__Conv2d_conv__input_65	80	0.245	0.328	4.659
subgraph_MyLargeModel__MyLargeModel_myModel_block2__ResidualBlock_blocks__ModuleList_8__MBConvLayer_conv__Sequential_depth_conv__Conv2d_conv__input_55	80	1.54	1.686	4.886

Following the profiling process, we systematically searched for each block's name within the model to extract information about the latency parameters.

Table 6-7: Extracted Information Summarization

Layer	Input size	Kernel size	Output dim	Average Time (ms)
Conv2d	1x224x224x3	3	24	16.626
Depthwise conv2d	1x112x112x24	7	24	1.47
Conv2d	1x56x56x72	1	32	0.328
Depthwise conv2d	1x112x112x144	5	144	1.686
Conv2d	1x56x56x144	7	144	0.408
Depthwise conv2d	1x112x112x96	5	96	4.349
Conv2d	1x56x56x96	1	32	0.315
Depthwise conv2d	1x112x112x72	5	72	0.844
Conv2d	1x56x56x72	1	32	0.283
Depthwise conv2d	1x112x112x144	3	144	1.071
Conv2d	1x56x56x144	1	32	0.4
Depthwise conv2d	1x112x112x96	3	96	0.73
Conv2d	1x56x56x96	1	32	0.296
Depthwise conv2d	1x112x112x72	3	72	0.604
Conv2d	1x56x56x72	1	32	0.309
Depthwise conv2d	1x112x112x24	7	24	14.126
Eltwise operation				0.643
Depthwise conv2d	1x112x112x24	5	24	12.706
Eltwise operation				0.66
Depthwise conv2d	1x112x112x114	7	144	2.748
Conv2d	1x56x56x144	1	32	0.402
Depthwise conv2d	1x112x112x96	7	96	1.838
Conv2d	1x56x56x96	1	32	0.311
Eltwise operation				0.617

6.8 Search Results of Evolutionary Algorithm

As the search process was carried with two different efficiency constraints resulting in different variations of architectures. Initially we performed NAS that was aware of arithmetic intensity and later on extended to our work to carry out search process with latency constraint of our target FPGA board. The results and configurations for both search process are presented below.

6.8.1. Searching with Arithmetic Intensity

Setup: For our experiment, the population used in each generation was 2500 and 1000 generations were searched. The ratio of networks used as parents for the next generation was 0.25. The constraints for search were arithmetic intensity and accuracy. The mutation probability was 0.1 and mutation ratio, which denotes number of networks generated through mutation in subsequent generation.

Size of population (P) = 2500,

No. of generations (N) = 1000,

Mutation ratio (r) = 0.25,

Constraint type: arithmetic_intensity

Efficiency constraint: 15, 18, 20 MAC ops/byte

Mutate Probability: 0.1,

Mutation Ratio: 0.5,

Evolutionary algorithm with constraints on arithmetic intensity resulted in different architectures. The visualization of architectures is presented below. There was no latency constraint, therefore all the resulting architectures have maximum depth at each stage. Search with latency constraint also will reduce the depth of the network.

Results: After the search process the obtained three network were evaluated on subset of ImageNet validation set of 10k. The results didn't match predicted accuracies. In order to check if the accuracy were recoverable or not one model from the searched models were retrained for one epoch on subset of 100k images of ImageNet train set.

The accuracy of model was recovered to 79% in just one epochs. As it is very time consuming and expensive to train on ImageNet further training was stopped.

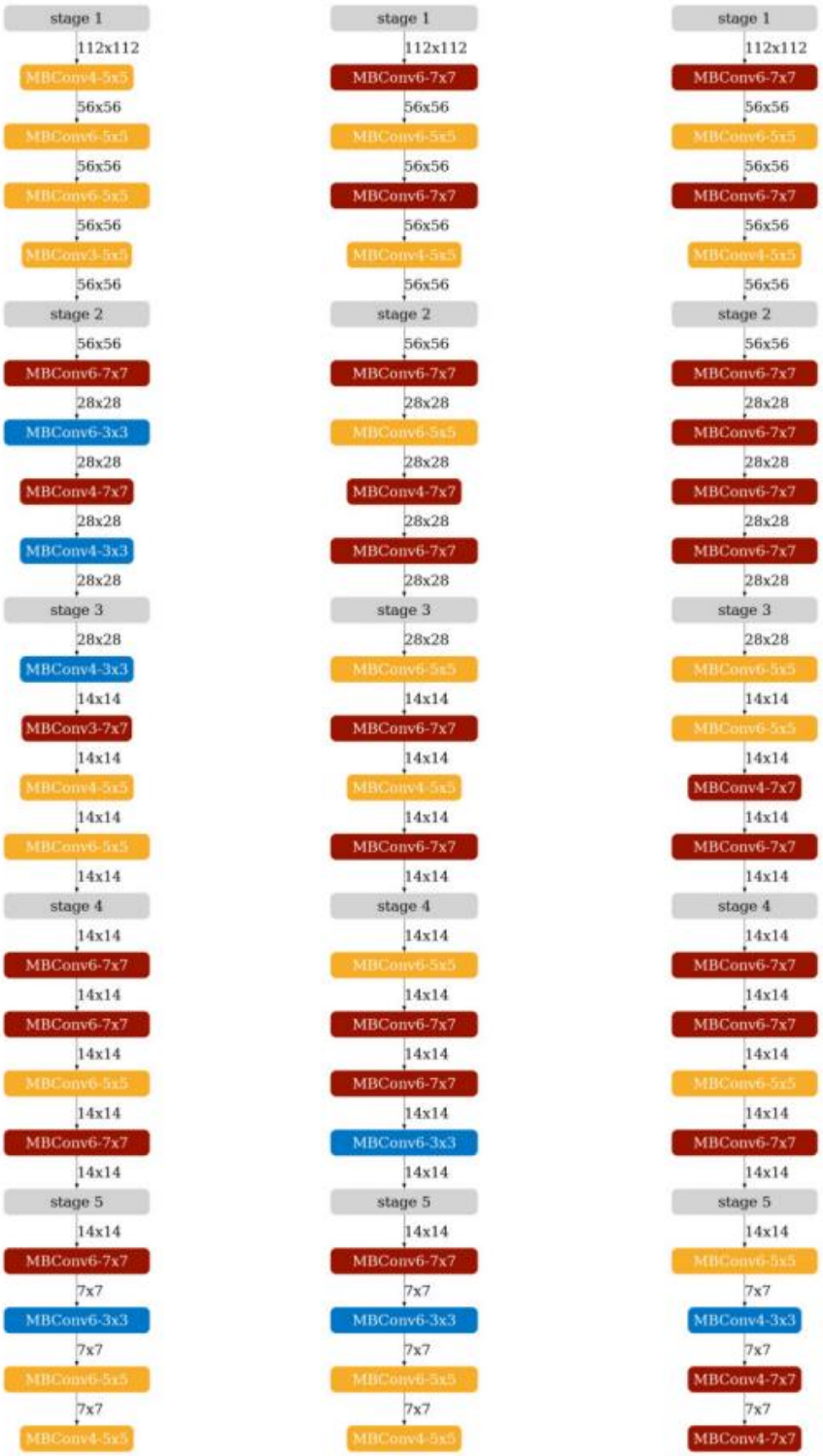


Figure 6-11: Searching with arithmetic intensity constraint (15, 18, 20 MAC ops/byte)

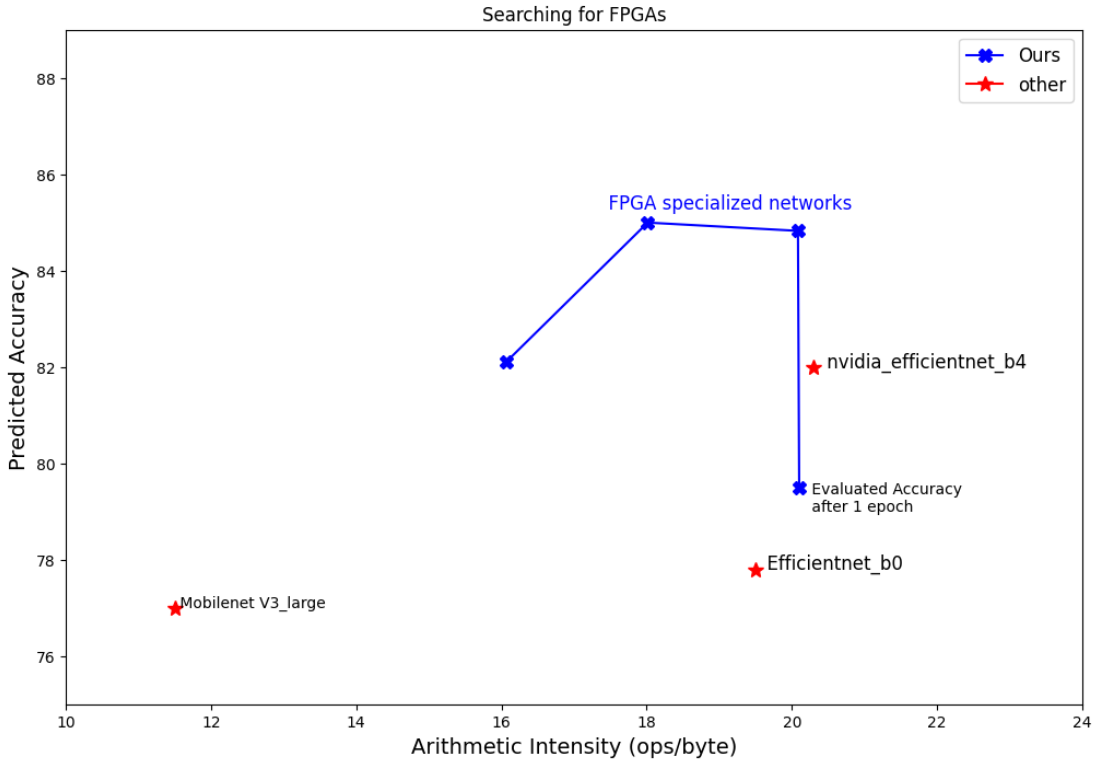


Figure 6-12: Our Search architectures vs existing models

The graph shows the accuracy comparison between different networks searched with different efficiency constraints which is arithmetic intensity. The plot shows predicted accuracy for our networks along with actual accuracy marked by red stars for other different networks. Our network was trained for single epoch only and obtained better accuracy than mobilenetv3 and efficientnetb0. On our FPGA we want the arithmetic intensity and accuracy to be higher, while latency to be slower.

6.8.2. Searching with Latency Constraint

We explored nearly 4*10^7 architectures and extracted 7 best performing networks for different latency requirements ranging from 10ms to 25ms.

Setup: An initial population of 5000 architectures were generated and was evolved for 1000 generation. Following hyper-parameters tuning was used for the evolutionary search process.

Size of population: P = 5000

No. of generations: N = 1000

Mutation ratio: r = 0.3

Constraint type: latency

Efficiency constraint: [25, 20, 15, 12, 11] ms

Mutate Probability: 0.3

Mutation Ratio: 0.4

This resulted in different architectures with different configuration with varying latencies and accuracies. The visualization of architectures are presented below.

Figure 6-13: Networks with (25ms, 20ms, 15ms, 13ms, 12ms, 11ms, 10ms)

Extracted Network Configuration:

This architecture is one illustrative example among searched network that was obtained though evolution with constraint as latency of 10ms. The search process took nearly 3.42 hours to find this architecture with predicted top-1 accuracy of 80.03% and estimated latency of just 9.96ms on our FPGA board.

Configuration: {'arch': [(0.803392231464386, {'wid': None, 'ks': [3, 3, 7, 3, 3, 3, 7, 5, 5, 3, 5, 7, 3, 3, 3, 3, 3, 3, 7, 5], 'e': [3, 4, 3, 3, 4, 3, 3, 4, 3, 3, 4, 3, 4, 3, 3, 6, 4, 4, 3, 6], 'd': [2, 2, 3, 3, 2], 'r': [224]}, 9.964999999999998)], 'info': [12356.262031793594]}

Retraining on Imagenet1k

All these networks were initialized with weights from the OFA super-network and then trained for 5 epochs on Imagenet1k utilizing GPU P100 in Kaggle. We used SDG with momentum =0.9 to train each network with learning rate of 0.00001. The learning rate was decayed by (gamma = 0.5) after every 2 epochs. No augmentation was used and the version with best validation was saved.

Table 6-8: FPGA aware NAS search results with varying parameters

Model	Latency (ms)	Params (M)	Top-1 Accuracy (%)	Top-5 Accuracy (%)	MACs (M)	Model Size (MB)
FPGANet_L10	10	5.27	75.6	92.38	278.8	21.2
FPGANet_L11	11	5.4	76.4	92.64	307.49	21.83
FPGANet_L12	12	6.28	76.98	93.11	318.58	25.35
FPGANet_L13	13	6.28	77.36	93.3	359.79	25.34
FPGANet_L14	15	7.44	77.9	93.72	410.01	30.08
FPGANet_L15	20	8.42	78.68	94.01	521.03	34.03
FPGANet_L25	25	9.79	79.12	94.16	608.75	39.55

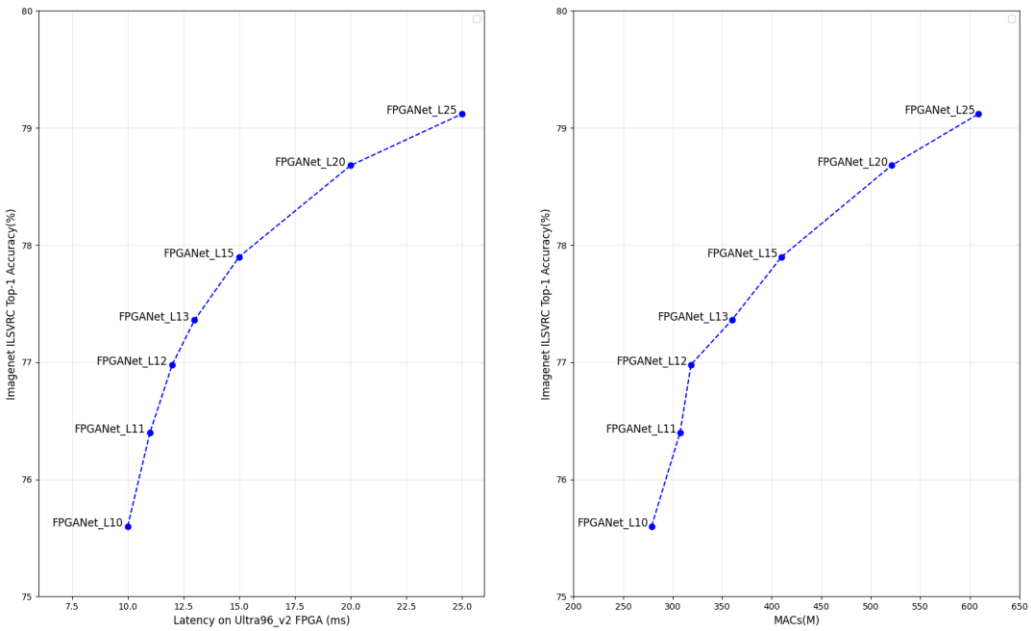


Figure 6-14: Accuracy vs MACs and latency of different models discovered through search process

6.9 Comparison with other Architectures

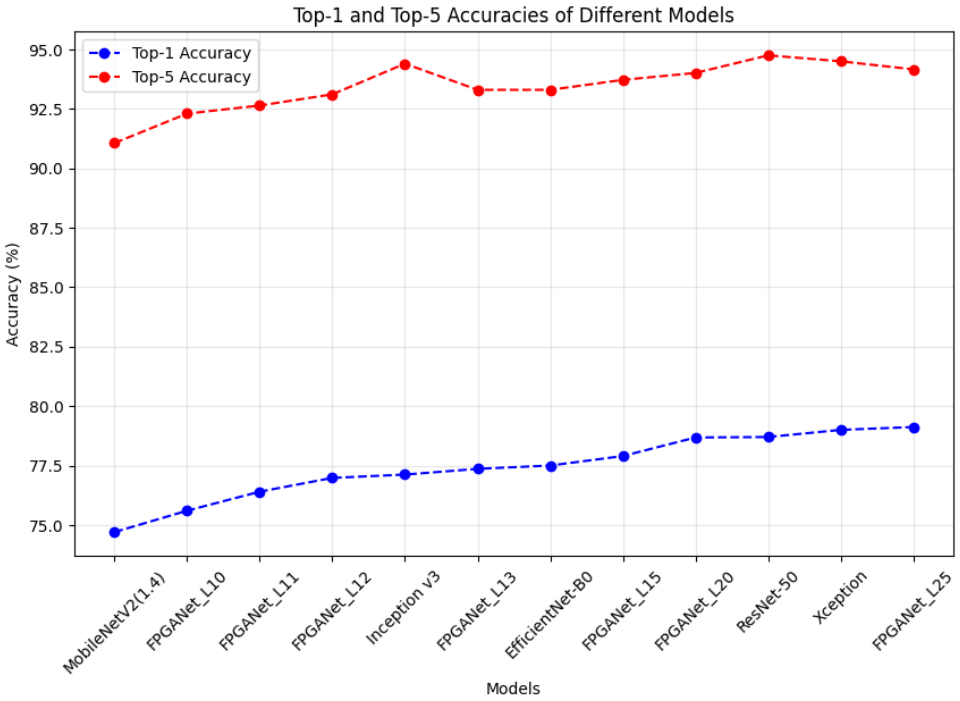


Figure 6-15: Top-1 and top-5 accuracy of our searched models

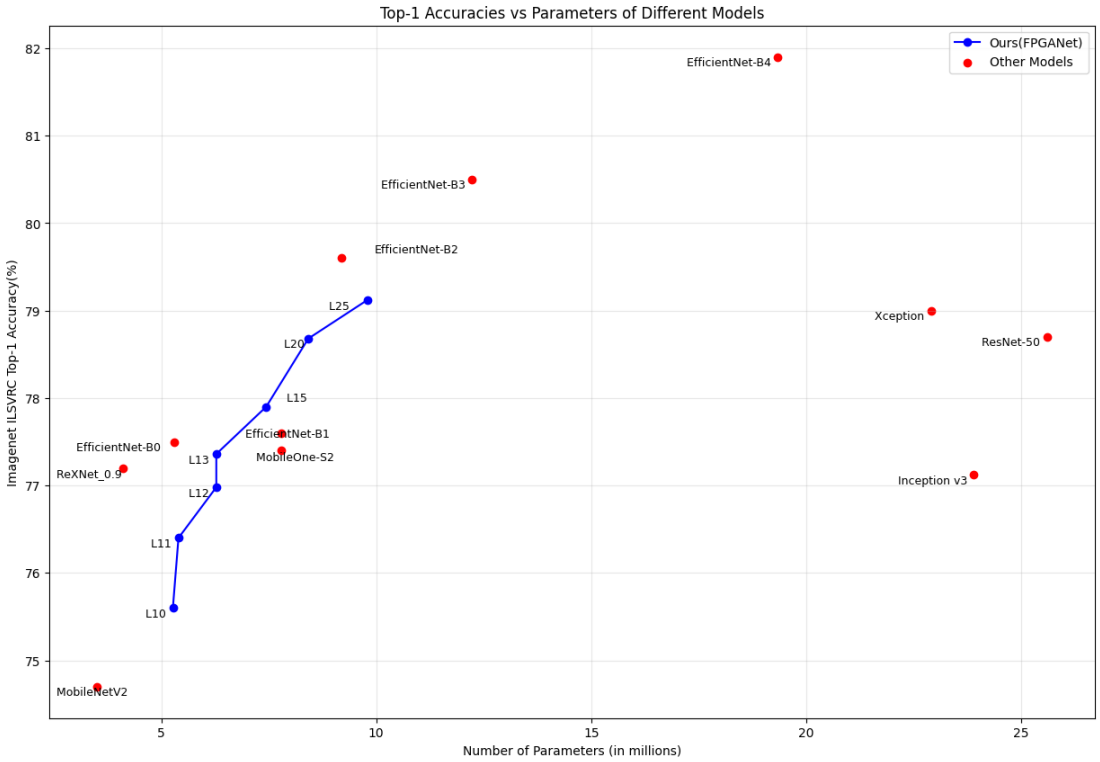


Figure 6-16: Comparison of our searched model with exiting work in terms of accuracy and number of parameters.

As presented in figure 5-16,.The models are labeled in the format L<Latency in milliseconds>. They were tested against established models like EfficientNet, ResNet, and Inception. While EfficientNet performed well with a top-1 accuracy in the low 80s, the model with the highest accuracy among our model was L25, with an accuracy of ~79%. Notably, L25 has fewer parameters, around 10 million, compared to the approximately 20 million of EfficientNet-B0.

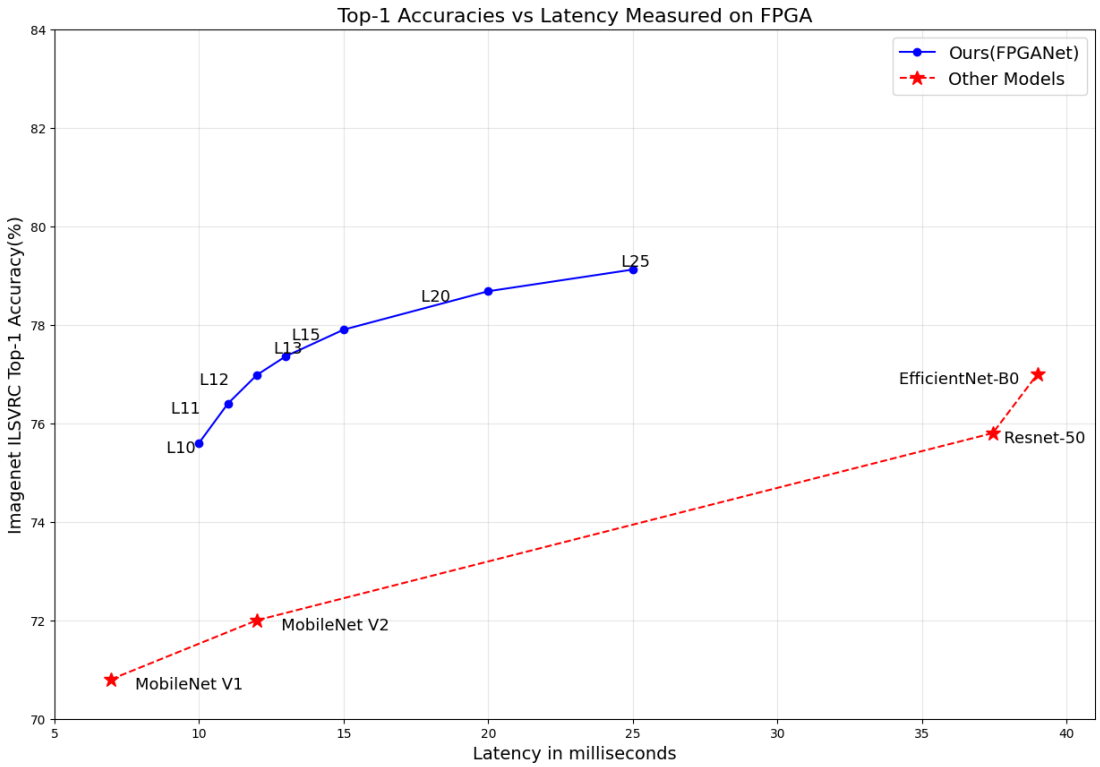


Figure 6-17: Top-1 Accuracy vs Latency measured on FPGA

This is an accuracy vs latency graph, models in top-left position are considered the better ones as they have lower latency/Parameters and higher accuracy. As shown in following graphs, comparatively our models have lower latency and number of parameters for given accuracy than most of the architectures.

6.10 Error Analysis on Latency

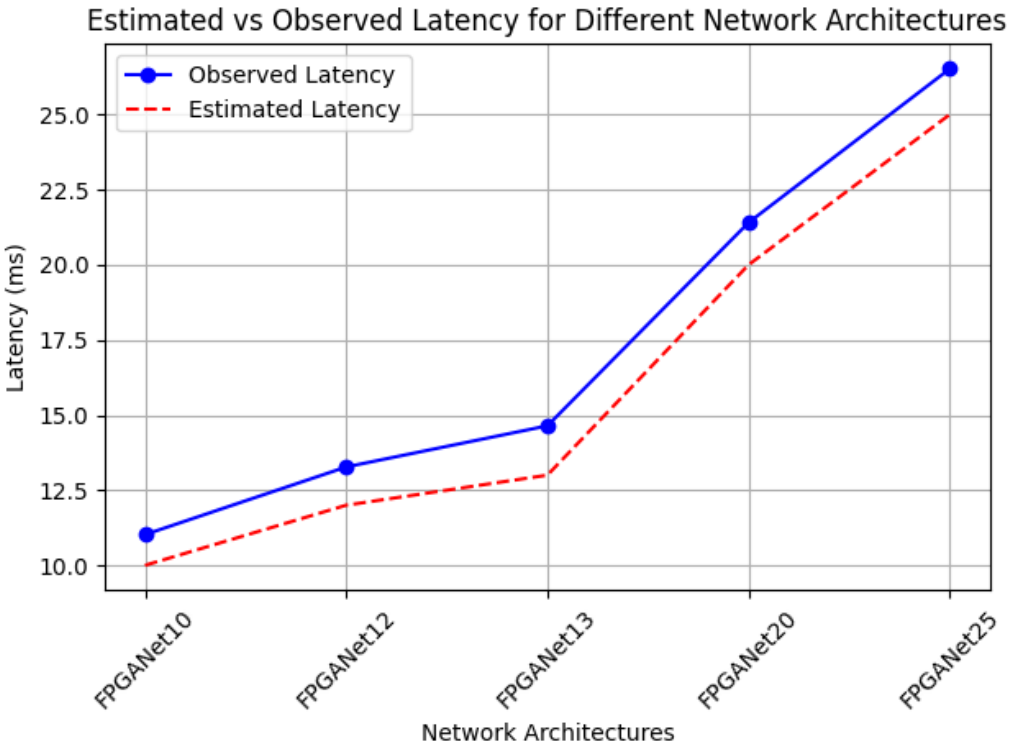


Figure 6-18: Estimated Latency against Actual Latency on Board

After measuring the latency of all the searched models (FPGANets), we observed that there was a minor discrepancy between actual latency and estimated latency. The actual latency was about 1ms more than the estimated latency. During the latency block construction and latency prediction we had excluded the Global Average Pooling layer in the estimation as every model included such with same feature. In actual testing, this block too contributed to actual latency.

FPGA designs involve additional overhead due to configuration, routing, and interconnects. These overheads contribute to latency beyond the computation time of individual blocks. Furthermore, as the latency discrepancy was almost similar for models of all sizes it adds to the fact that there is somewhat a constant overhead when deploying the full model. The communication between different blocks (inter-block communication) can cause delays due to data movement and synchronization.

7. FUTURE ENHANCEMENTS

1. Explore and extend different hardware configurations as defined in hardware search space and implement different networks on them to perform testing of co-optimization.
2. Extend the DNN search space to include object detection and image segmentation.

8. CONCLUSION

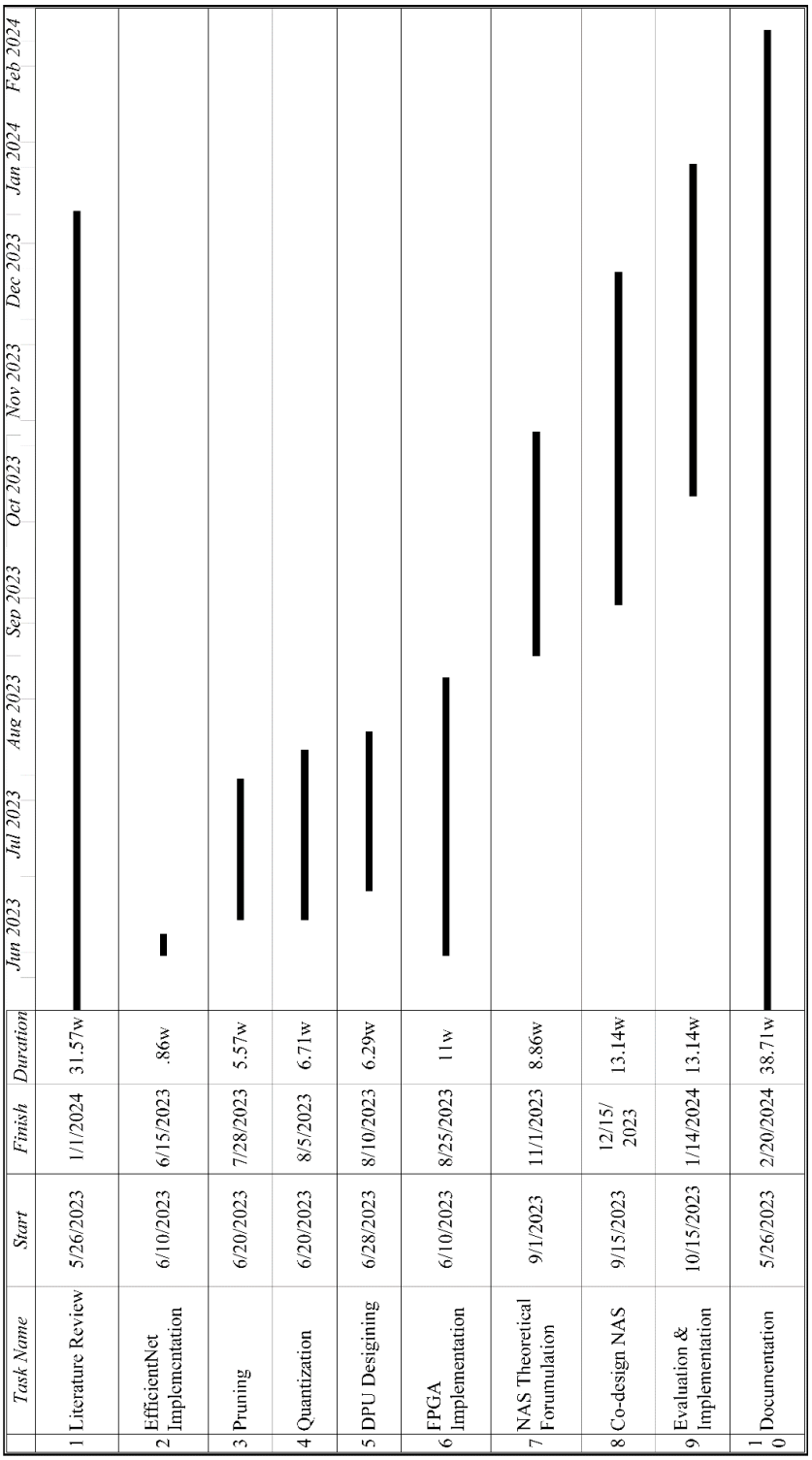
During the first part, we successfully compressed, quantized and compiled EfficientV2 for Ultra96V2 which enabled us to get familiarize with the workflow of development and runtime of neural networks in FPGA .We were able to reduce the model by 14 times the original size.

For searching architectures it was necessary to define a comprehensive search space. Evolutionary search algorithm was used due to its relevancy, simplicity and lesser computation time. For Further reducing the searching time Heuristic based accuracy and efficiency predictors were built. A latency dataset was also constructed specifically for that purpose.

The search process generated several architectures with reasonable tradeoff .Since, we are using DPU which is general IP developed to run most of CNN networks, we manually tuned the DPU for the ultra96v2 board and also figured out that among different IPs (B512,B1024,B2048,…) B2304 was the best in term of resource utilization. As, DPU is generalized inference engine, hardware search space was created by configuring systolic array size, internal memory size and accumulator size which allowed us to explore different choices and move on to different workflow.

9. APPENDICES

Appendix A: Project Timeline



	Task Name	Start	Finish	Duration
1	Literature Review	5/26/2023	1/1/2024	31.57w
2	EfficientNet Implementation	6/10/2023	6/15/2023	.86w
3	Pruning	6/20/2023	7/28/2023	5.57w
4	Quantization	6/20/2023	8/5/2023	6.71w
5	DPU Designing	6/28/2023	8/10/2023	6.29w
6	FPGA Implementation	6/10/2023	8/25/2023	11w
7	NAS Theoretical Forumulation	9/1/2023	11/1/2023	8.86w
8	Co-design NAS	9/15/2023	12/15/2023	13.14w
9	Evaluation & Implementation	10/15/2023	1/14/2024	13.14w
10	Documentation	5/26/2023	2/20/2024	38.71w

Appendix B: Project Budget

This project can be built using freely available software and library files. So, we can easily complete this project with minimum expenditure if we have a computer with a good processor. The FPGA was obtained from the college, so there was no expenditure in acquiring it.

Table: Project Budget

S.N	Particulars	Price (Rs.)	Quantity	Total Price (Rs.)
1	Cloud GPU	-	Subscription	-
2	Ultra96-V2 Board	35,000	1	35,000
3	USB Camera	1,000	1	1,000
4	Cables	1,000		1,000
5	SD Card	1,000	1	1,000
6	Miscellaneous	-	-	5,000
Total				38,000

Appendix C: Implementation of Design using PetaLinux

There are two ways to build the project in the PetaLinux, one way to build the project is from BSP (Board Support Package) which is provided to give reference design by the manufacturer.

petalinux-create -t project -s ultra96v2_2020_02.bsp --name <project_name>

This command is used to add the bsp on the project creation process. It will add the in build design parameters mentioned in the bsp file of the board.

Another way is to design our project using binary file constructed from Vivado according to our custom board (Ultra96v2). Regardless of how the hardware platform is created and configured, a small number of hardware IP and software platform configuration settings are required to make the hardware platform Linux ready.

```
// Go to Vivado project directory
$ cd /home/username/workspace/petalinux1

// Create a Petalinux project named peta_project (name optional)
$ petalinux-create --type project --template zynq --name peta_project
NFO: Create project: peta_project
INFO: New project successfully created in /home/username/workspace/petalinux1/peta_project
```

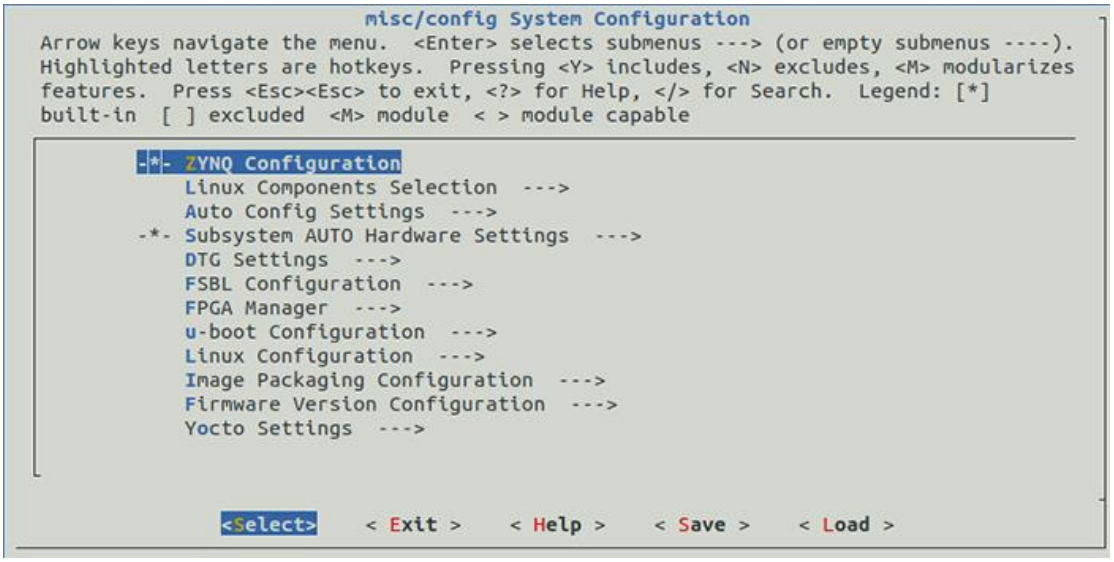
As shown above the command petalinux-create is used to create the project and in –template, since Ultra96v2 lie under AMD Zynq UltraScale MPSOC so we use --zynqMP in --template. The directory structure should include the Vivado deliverable design_1_wrapper.xsa shown below.

```
$ ls
design_1_wrapper.xsa  petalinux1.cache  petalinux1.hw            petalinux1.runs  petalinux1.srcs
peta_project          petalinux1.gen    petalinux1.ip_user_files  petalinux1.sim   petalinux1.xpr
```

We need to make PetaLinux tools software platform ready for building a Linux system customized to our new hardware platform. We use *petalinux-config –get-hw-description= command* to add design_1_wrapper.xsa file. The configuration wizard will run due to this command as shown in figure.

```
// Go to the Petalinux project directory you created
$ cd peta_project

// Perform system configuration by specifying the XSA file created in Vivado
$ petalinux-config --get-hw-description=../design_1_wrapper.xsa
[INFO] Sourcing buildtools
INFO: Getting hardware description...
INFO: Renaming design_1_wrapper.xsa to system.xsa
[INFO] Generating Kconfig for project
[INFO] Menuconfig project
```



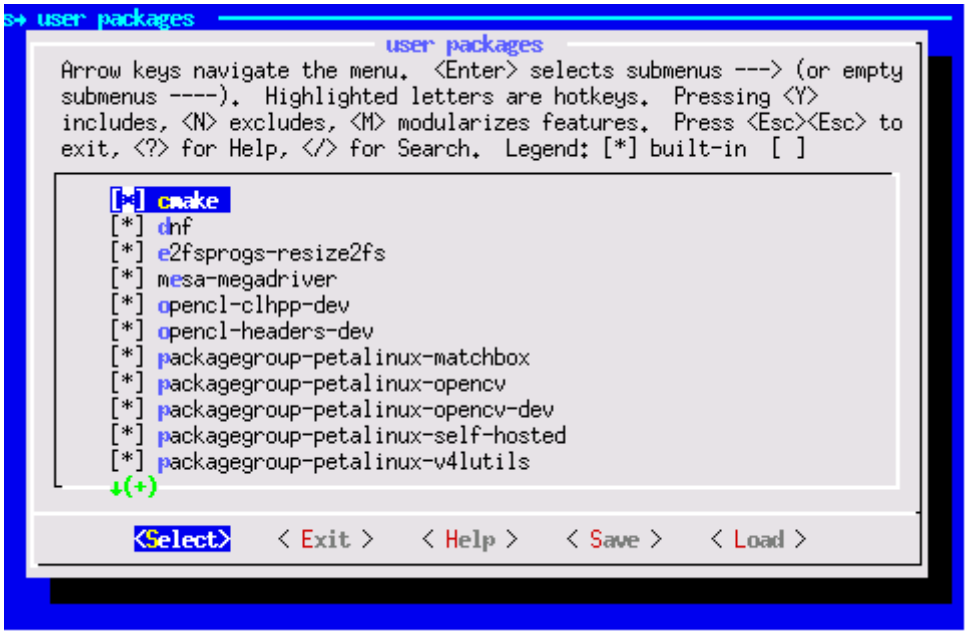
Customization of Root File System and Kernel

Since we are using Vitis ai for the quantization and compilation of our CNN models so we need to add XRT (Xilinx Runtime Library) as it enables developers to deploy on AMD adaptable platforms, while continuing to use familiar programming languages like C/C++, Python and high-level domain-specific frameworks like TensorFlow and PyTorch. For that, we need to customize Root File System and Kernel.

CONFIG_xrt is used for base XRT support where dependency packages such as ZOCL driver module will be added automatically.

CONFIG_packagegroup-petalinux-vitisai is a command which will install all the packages for Vitis-AI dependencies and Optional Packages for natively building Vitis AI applications on Ulta96v2 following commands are used.

```
CONFIG_packagegroup-petalinux-self-hosted
CONFIG_cmake
CONFIG_packagegroup-petalinux-vitisai-dev
CONFIG_xrt-dev
CONFIG_opencl-clhpp-dev
CONFIG_opencl-headers-dev
CONFIG_packagegroup-petalinux-opencv
CONFIG_packagegroup-petalinux-opencv-dev
```



Then we run petalinux-config –c rootfs and select all the user packages will list mentioned and save the settings. After manually checking all the settings and features, finally build the project using command petalinux-build as shown in figure. Host need to be connected to the network, as multiple ftp, git servers, etc. will be accessed during the build process.

```
// Build again
$ petalinux-build
[INFO] Sourcing buildtools
[INFO] Building project
[INFO] Sourcing build environment
[INFO] Generating workspace directory
INFO: bitbake petalinux-image-minimal
NOTE: Started PRServer with DBfile: /home/username/workspace/petalinux1/peta_project/build/cache
/prserv.sqlite3, Address: 127.0.0.1:36647, PID: 136488
Loading cache: 100%
|##################################################################################################
Time: 0:00:00
Loaded 5392 entries from dependency cache.
Parsing recipes: 100%
|##################################################################################################
Time: 0:00:00
Parsing of 3592 .bb files complete (3590 cached, 2 parsed). 5394 targets, 554 skipped, 0 masked, 0
errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100%
|##################################################################################################
Time: 0:00:03
Checking sstate mirror object availability: 100%
|##################################################################################################
Time: 0:00:06
Sstate summary: Wanted 466 Local 17 Network 304 Missed 145 Current 1111 (68% match, 90% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 4216 tasks of which 4212 didn't need to be rerun and all succeeded.
INFO: Failed to copy built images to tftp dir: /tftpboot
[INFO] Successfully built project
```

Build results are generated in *images/linux*. The img would be bootable for Ultra96v2 using software like balena Etcher into a SD-Card.

```
$ ls images/linux/
boot.scr      rootfs.cpio.gz         rootfs.tar.gz   u-boot.bin  zynq_fsbl.elf
config        rootfs.cpio.gz.u-boot  system.bit      u-boot.elf
image.ub      rootfs.ext4            system.dtb      uImage
pxelinux.cfg  rootfs.jffs2           u-boot-dtb.bin  vmlinux
rootfs.cpio   rootfs.manifest        u-boot-dtb.elf  zImage
```

Deploying the Model in the board

We need to insert the SD-Card to the Ultra96v2. The Ultra96v2 should be set to SD boot. The connection of the board with Host (computer) is done through USB UART cable and power on the board. It should take a minute to boot linux properly.

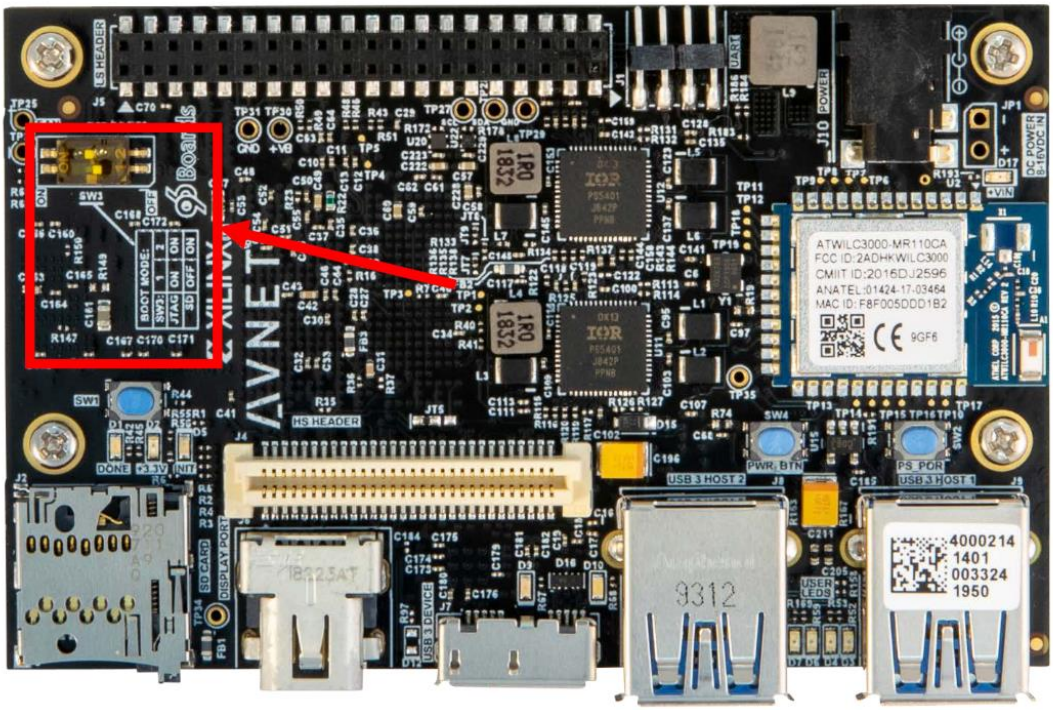
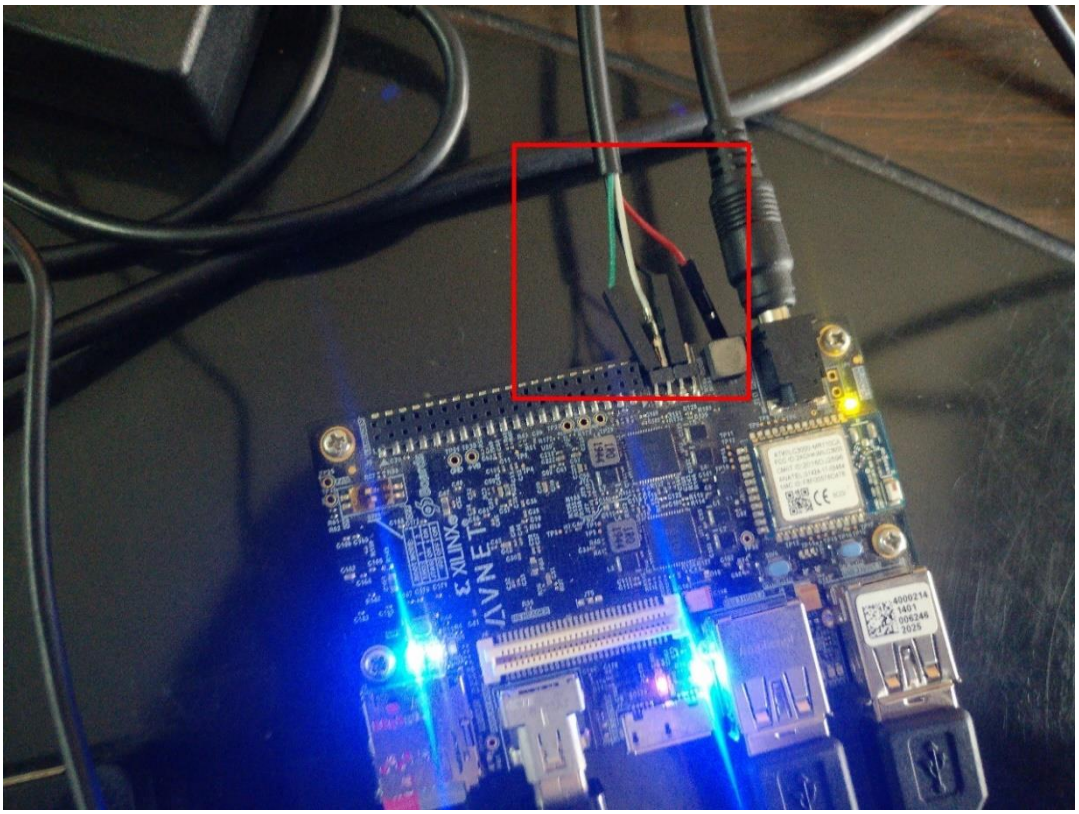


Figure 9-1: Boot Mode Location in Ultra96v2

The software like gtkterm and mobaterm are used to control console of the linux on the host through serial port.



```
root@petalinux:~# df .
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/root               564048    398340    122364  77% /
```

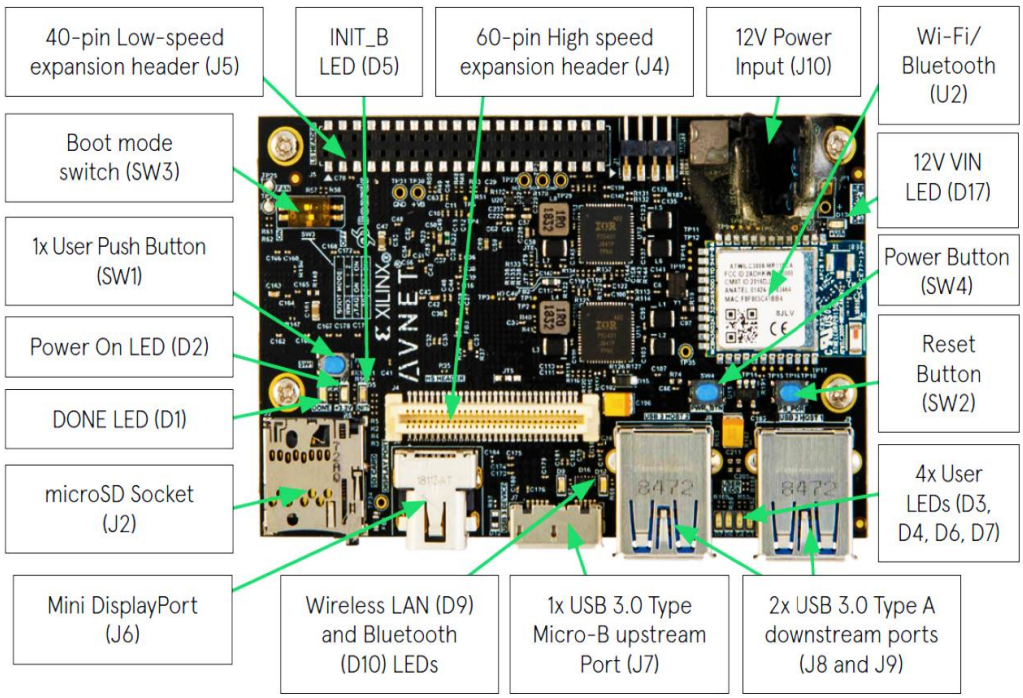
Figure 9-2: UART connection in Ultra96v2

Now, we can run the xmodel compiled for the defined DPU and test the model to obtain latency as well as other parameters. We have copied the model and run using following commands as shown:

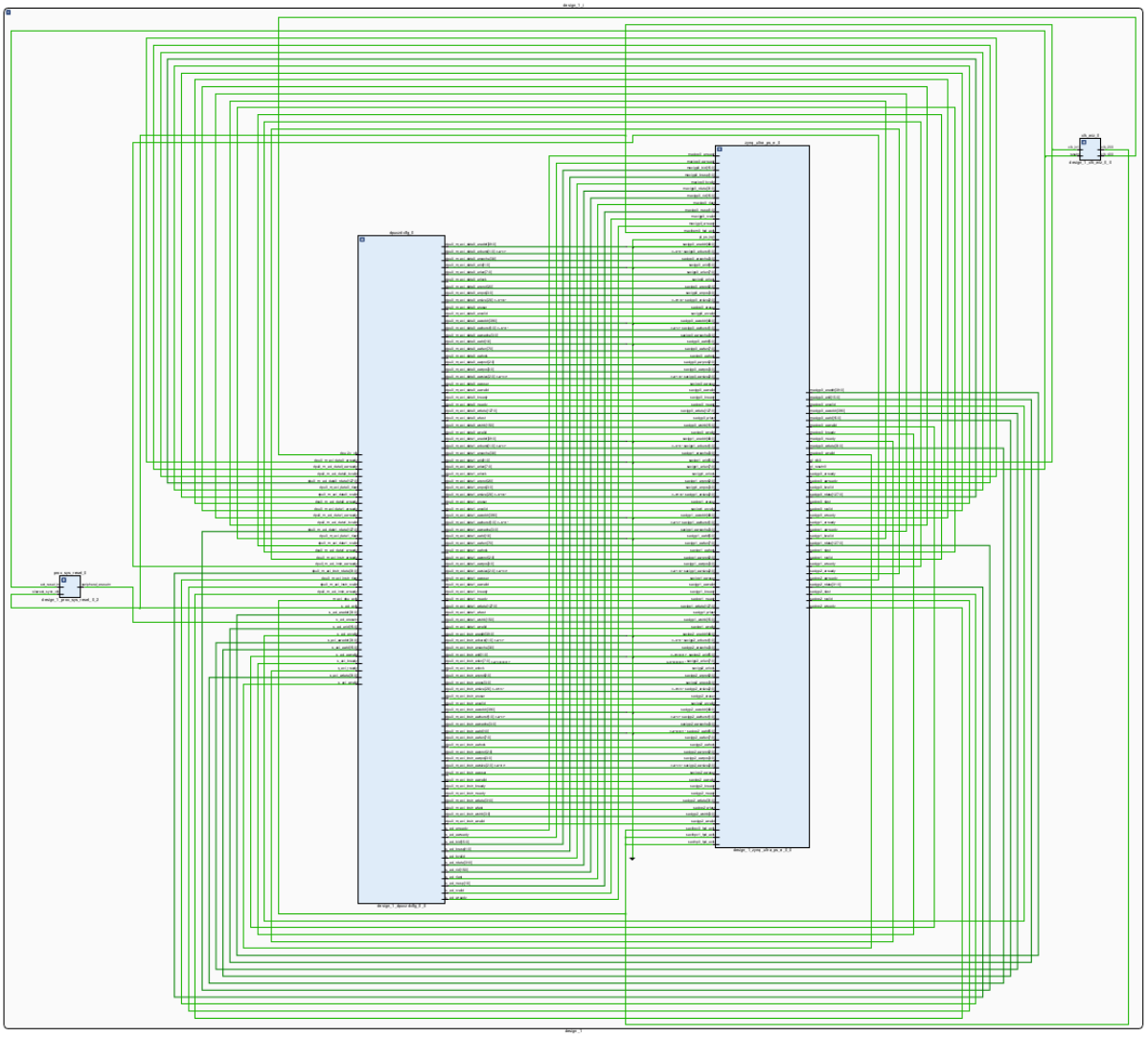
```
# Libraries
root@petalinux:~# cp -r /mnt/sd-mmcblk0p1/app/samples/ ~
# Model
root@petalinux:~# cp /mnt/sd-mmcblk0p1/app/model/resnet50.xmodel ~
# Host app
root@petalinux:~# cp /mnt/sd-mmcblk0p1/dpu_trd ~
# Image to test
root@petalinux:~# cp /mnt/sd-mmcblk0p1/app/img/bellpeppe-994958.JPEG ~
```

root@petalinux:~# env LD_LIBRARY_PATH=samples/lib XLNX_VART_FIRMWARE=/mnt/sd-mmcblk0p1/dpu.xclbin ./dpu_trd bellpeppe-994958.JPEG

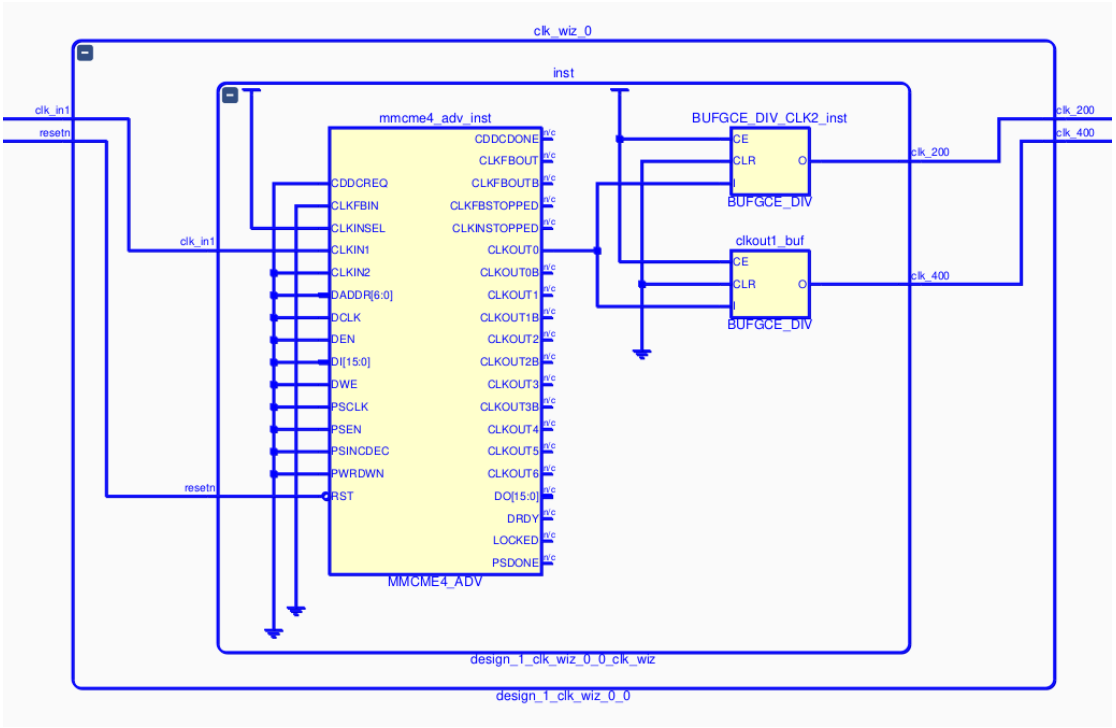
Appendix D: Ultra96-v2 board



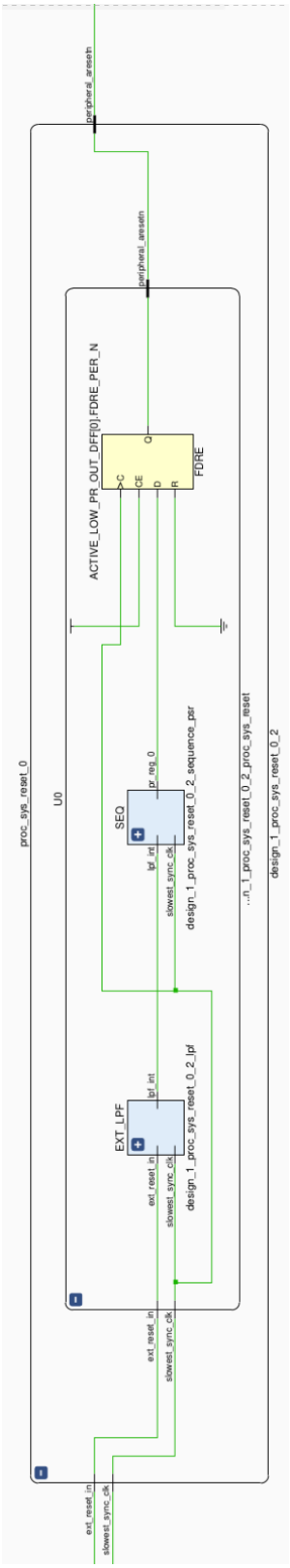
Appendix E: Overall Schematic diagram



Appendix F: Clock management tile (CMT)



Appendix G: Processor system reset



Appendix H: Used Linux Commands

- *cp*: Copies files or directories from one location to another.
- *mv*: Moves or renames files or directories.
- *cd*: Changes the current working directory.
- *mkdir*: Creates a new directory.
- *rm*: Removes (deletes) files or directories.
- *apt install*: Installs software packages using the Advanced Packaging Tool (APT) package manager.
- *pip install*: Installs Python packages using the pip package manager.
- *pip uninstall*: Uninstalls Python packages using the pip package manager.
- *git clone*: Creates a copy of a remote Git repository on the local machine.
- *git push:* Uploads local Git commits to a remote repository.
- *git commit*: Records changes to the local Git repository.
- *git pull*: Updates the local Git repository with the latest changes from the remote repository.
- *git push*: Uploads local Git commits to a remote repository.

Docker Commands

- *docker inspect*: Retrieves low-level information about a Docker container or image.
- *docker start*: Starts a stopped Docker container.
- *docker stop*: Stops a running Docker container.
- *docker exec -it vitis bash*: Executes an interactive shell (bash) inside a running Docker container named "vitis".
- *docker run -it image:* Runs a Docker container using a specified image and opens an interactive session with it.
- *docker cp source destination:* Copies files or directories between a Docker container and the local file system.

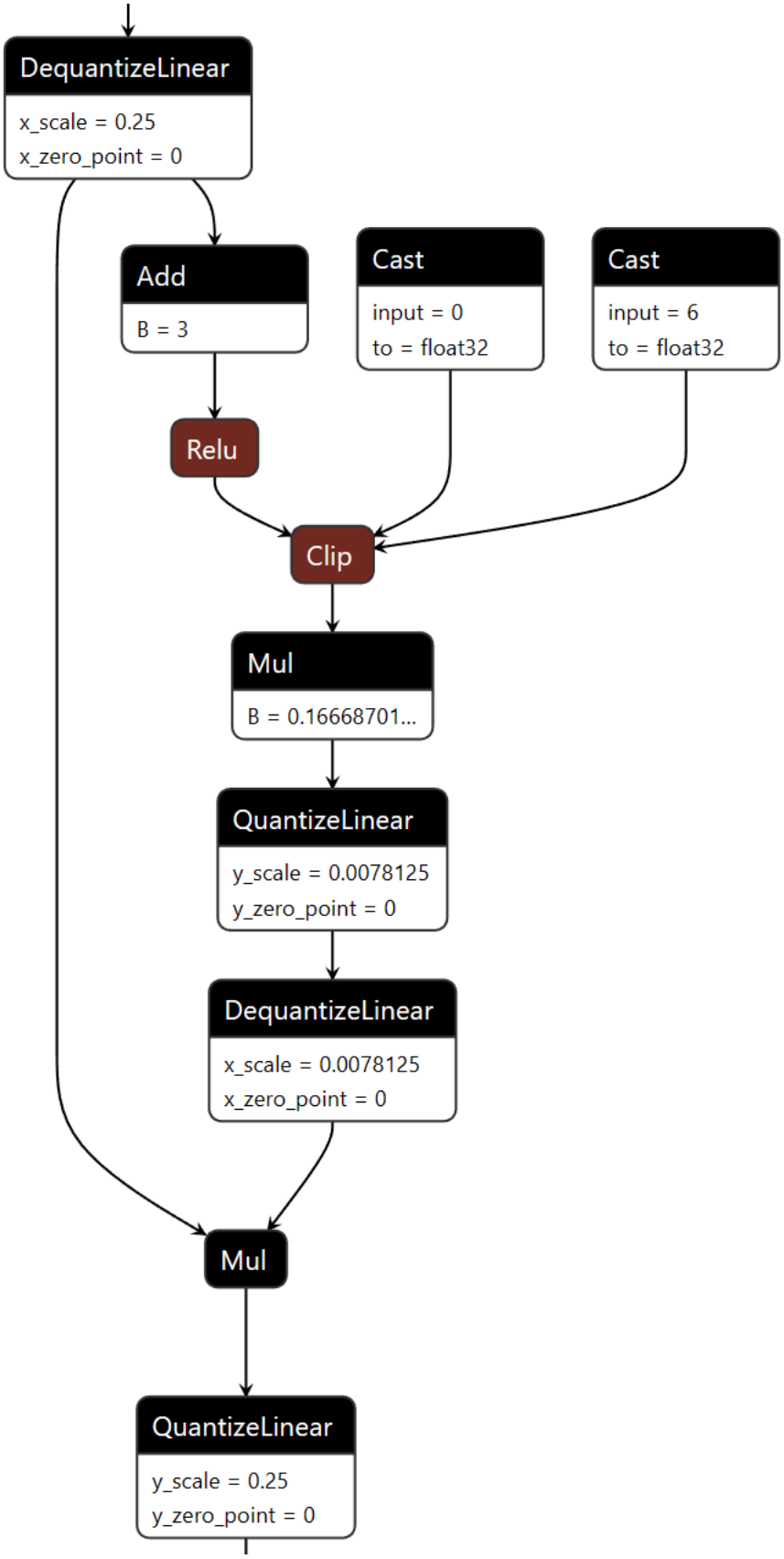
Vitis AI Commands

- *conda activate vitis-ai-pytorch:* Activates the "vitis-ai-pytorch" Conda

environment.

- *export W_QUANT=1*: Sets the environment variable "W_QUANT" to 1.
- *Python ./code/test/efficientnetv2_quant.py --device "cpu" --quant_mode calib --subset_len 200*: Quantizes a model using the "efficientnetv2_quant.py" script with the specified options.
- *sudo /opt/vitis_ai/conda/envs/vitis-ai-wego-torch/bin/python ./code/test/efficientnetv2_quant.py --quant_mode test --subset_len 5 --batch_size=10 --deploy*: Deploys and exports a quantized model using the "efficientnetv2_quant.py" script with the specified options, running as sudo and using the Vitis-AI environment.
- *Python code/test/efficientnetv2_QAT.py --device "cpu" --quant_mode calib --fast_finetune --subset_len 50*: Performs Quantization-Aware Training (QAT) on a model using the "efficientnetv2_QAT.py" script with the specified options.

Appendix I: Snapshot of Quantized Efficientnet-V2



Appendix J: Student Supervisor Consultation Form

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING, THAPATHALI CAMPUS
Department of Electronics and Computer Engineering
Student & Supervisor Consultation Form
(BCT Major Project, Batch 2076)

Project Title	FPGA based accelerator co-design using Neural Architecture Search
Group Members (Name & Roll Number)	Amit Raj Pant (THA076 BCT005) Bishal Rijal (THA076 BCT014) Ishika Poudel (THA076 BCT018) Pilot Khadka (THA076 BCT025)
Name of Supervisor	Assoc. Prof Shanta Maharjan

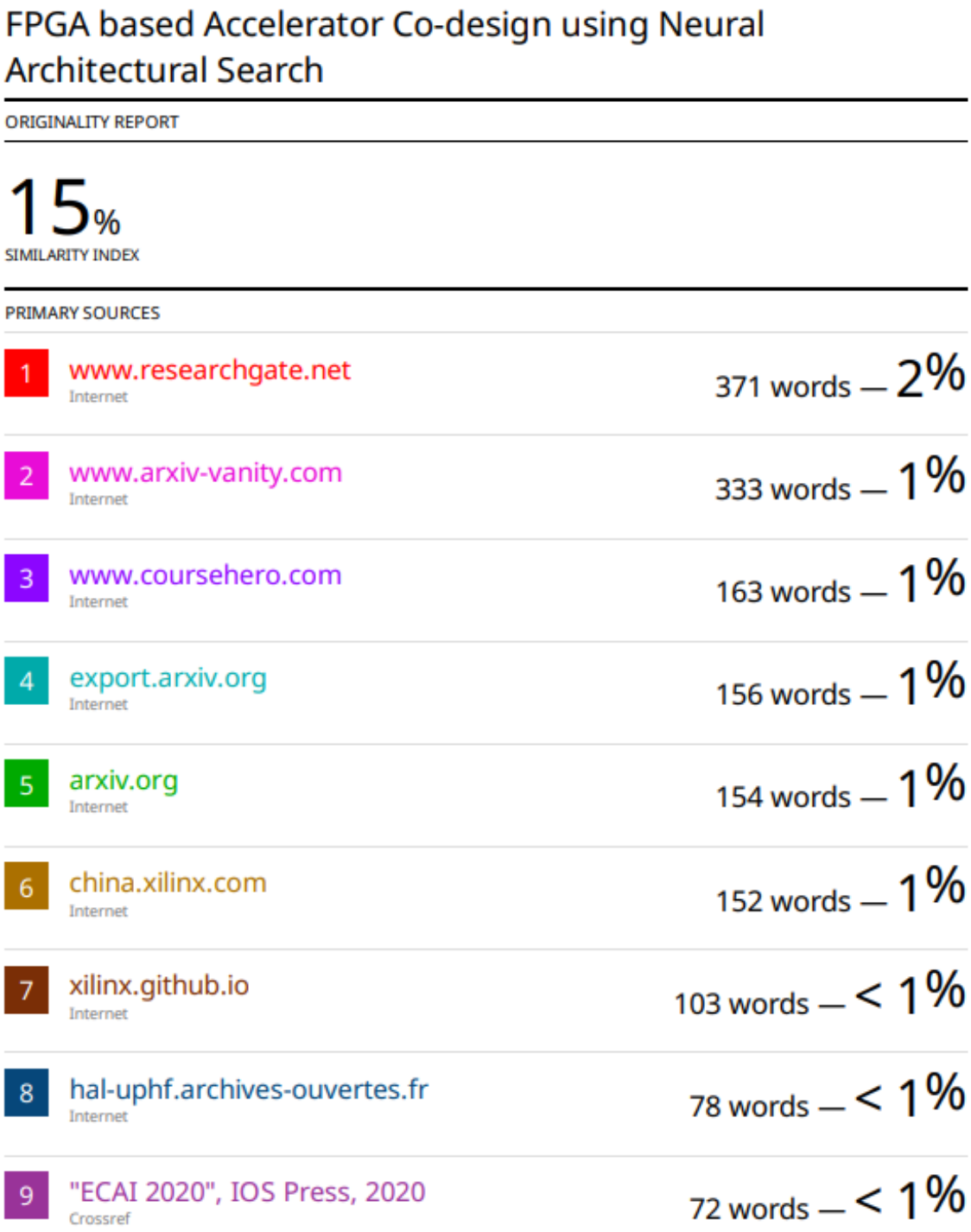
S.N.	Brief Summary of Discussion Agenda	Date	Supervisor Signature
1	Discussion on Optimization techniques	2080-9-18	
2	Discussion on OFA network	2080-9-20	
3	Discussion on progress, Project report	2080/10/2	
4	Discussion on Slide	2080/10/11	
5	Discussion on FPGA constraints	2080/10/26	
6	Discussion on Implementation on FPGA	2080/11/01	
7	Discussion on progress, report	2080/10/10	
8	Discussion on slides, presentation	2080/11/12	
9	Discussion on Report	2080/11/14	
10	Discussion on Final defense	2080/11/16	

2080/11/16
Date of Approval

Signature of Supervisor

At least TWO consultation is required before 1st Checkpoint Defense
At least FIVE consultations are required BEFORE 2nd Checkpoint and TEN consultations for FINAL

Appendix K: Plagiarism Check



FPGA based Accelerator Co-design using Neural Architectural Search

ORIGINALITY REPORT

15%
SIMILARITY INDEX

PRIMARY SOURCES

1	www.researchgate.net Internet	371 words — 2%
2	www.arxiv-vanity.com Internet	333 words — 1%
3	www.coursehero.com Internet	163 words — 1%
4	export.arxiv.org Internet	156 words — 1%
5	arxiv.org Internet	154 words — 1%
6	china.xilinx.com Internet	152 words — 1%
7	xilinx.github.io Internet	103 words — < 1%
8	hal-uphf.archives-ouvertes.fr Internet	78 words — < 1%
9	"ECAI 2020", IOS Press, 2020 Crossref	72 words — < 1%

10	elibrary.tucl.edu.np Internet	71 words — < 1%
11	www.xilinx.com Internet	66 words — < 1%
12	github.com Internet	62 words — < 1%
13	www.mdpi.com Internet	51 words — < 1%
14	www.slideshare.net Internet	48 words — < 1%
15	Krishna Teja Chitty-Venkata, Arun K. Somani. "Neural Architecture Search Survey: A Hardware Perspective", ACM Computing Surveys, 2022 Crossref	43 words — < 1%
16	bella.cc Internet	43 words — < 1%
17	huggingface.co Internet	39 words — < 1%
18	www.doc.ic.ac.uk Internet	39 words — < 1%
19	docplayer.net Internet	36 words — < 1%
20	developer.xilinx.com Internet	35 words — < 1%
21	hdl.handle.net Internet	34 words — < 1%

22	dokumen.pub Internet	33 words — < 1%
23	pdfslide.net Internet	32 words — < 1%
24	edoc.ub.uni-muenchen.de Internet	31 words — < 1%
25	wrap.warwick.ac.uk Internet	28 words — < 1%
26	web.archive.org Internet	27 words — < 1%
27	abis-files.marmara.edu.tr Internet	25 words — < 1%
28	Zhang, Jinjie. "Quantization for High-Dimensional Data and Neural Networks: Theory and Algorithms", University of California, San Diego, 2023 ProQuest	24 words — < 1%
29	www.ri.cmu.edu Internet	22 words — < 1%
30	"Computer Vision – ECCV 2022", Springer Science and Business Media LLC, 2022 Crossref	21 words — < 1%
31	www.docstoc.com Internet	21 words — < 1%
32	blog.csdn.net Internet	20 words — < 1%
	vivado1.rssing.com	

33	Internet	20 words — < 1%
34	kyutech.repo.nii.ac.jp Internet	19 words — < 1%
35	opus.lib.uts.edu.au Internet	19 words — < 1%
36	Shuanglong Liu, Hongxiang Fan, Xinyu Niu, Ho-cheung Ng, Yang Chu, Wayne LUK. "Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA", ACM Transactions on Reconfigurable Technology and Systems, 2018 Crossref	18 words — < 1%
37	S V Kushal Kumar, A. Anita Angeline. "Design of Arithmetic Logic Unit using Pseudo Dynamic Buffer based Domino Logic", Journal of Physics: Conference Series, 2020 Crossref	17 words — < 1%
38	addi.ehu.es Internet	17 words — < 1%
39	eprints.utar.edu.my Internet	17 words — < 1%
40	"Final", Design of Circuits and Integrated Systems, 2014. Crossref	16 words — < 1%
41	Abdelilah Haijoub, Anas Hatim, Mounir Arioua, Slama Hammia, Ahmed Eloualkadi, Antonio Guerrero-González. "Implementing Convolutional Neural	16 words — < 1%

Networks on FPGA: A Survey and Research", ITM Web of Conferences, 2023
Crossref

42 Amin Kalantar, Zachary Zimmerman, Philip Brisk. "FA-LAMP: FPGA-Accelerated Learned Approximate Matrix Profile for Time Series Similarity Prediction", 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2021 — 16 words — < 1%
Crossref

43 J. Flich, R. Tornero, D. Rodriguez, D. Russo, J.M. Martinez, C. Hernandez. "From a FPGA Prototyping Platform to a Computing Platform: The MANGO Experience", 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021 — 16 words — < 1%
Crossref

44 futurwiser.com — 16 words — < 1%
Internet

45 link.springer.com — 16 words — < 1%
Internet

46 www.catalyzex.com — 16 words — < 1%
Internet

47 www.kdnuggets.com — 16 words — < 1%
Internet

48 www.tdx.cat — 16 words — < 1%
Internet

49 Akihiko Ushiroyama, Minoru Watanabe, Nobuya Watanabe, Akira Nagoya. "Convolutional neural network implementations using Vitis AI", 2022 IEEE 12th Annual — 15 words — < 1%

Computing and Communication Workshop and Conference (CCWC), 2022
Crossref

50 Duarte, Luís Diogo Medina. "DLL Architecture for OFDM Based VLC Transceivers in FPGA", Universidade de Aveiro (Portugal), 2024 — 15 words — < 1%
ProQuest

51 Nana Sutisna, Zulfikar N. Arifuzzaki, Infall Syafalni, Rahmat Mulyawan, Trio Adiono. "Architecture Design of Q-Learning Accelerator for Intelligent Traffic Control System", 2022 International Symposium on Electronics and Smart Devices (ISESD), 2022 — 14 words — < 1%
Crossref

52 ce-publications.et.tudelft.nl — 14 words — < 1%
Internet

53 www.coursera.org — 14 words — < 1%
Internet

54 Dong, Zhen. "Hardware-Aware Efficient Deep Learning", University of California, Berkeley, 2023 — 13 words — < 1%
ProQuest

55 Ekaba Bisong. "Building Machine Learning and Deep Learning Models on Google Cloud Platform", Springer Science and Business Media LLC, 2019 — 13 words — < 1%
Crossref

56 Faustino Aguilar, Samuel Grayson, Darko Marinov. "Reproducing and Improving the BugsInPy Dataset", 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), 2023 — 13 words — < 1%
Crossref

57	Francesc Wilhelmi, Nima Afraz, Elia Guerra, Paolo Dini. "The implications of decentralization in blockchained federated learning: Evaluating the impact of model staleness and inconsistencies", Computer Networks, 2024 Crossref	13 words — < 1%
58	Foley, Samantha S. "More effective use of high performance systems using sub-batch allocation resource management within multiple component multiple data applications", Proquest, 20111108 ProQuest	12 words — < 1%
59	Georgios Bousdras, Francois Quitin, Dragomir Milojevic. "Template architectures for highly scalable, many-core Heterogeneous SoC: Could-of-Chips", 2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2018 Crossref	12 words — < 1%
60	Kasmani, Saeed Amirgholipour. "Convolutional Neural Network for Accurate Crowd Counting and Destiny Estimation", University of Technology Sydney (Australia), 2023 ProQuest	12 words — < 1%
61	liu.diva-portal.org Internet	12 words — < 1%
62	scholarsarchive.byu.edu Internet	12 words — < 1%
63	www.hackster.io Internet	12 words — < 1%
64	Sean Scott. "Chapter 4 Oracle Database QuickStart", Springer Science and Business Media	11 words — < 1%

LLC, 2023
Crossref

65	Shizuma Namekawa, Taro Tezuka. "Evolutionary Neural Architecture Search by Mutual Information Analysis", 2021 IEEE Congress on Evolutionary Computation (CEC), 2021 Crossref	11 words — < 1%
66	technodocbox.com Internet	11 words — < 1%
67	www.computeroptics.ru Internet	11 words — < 1%
68	www.wevolver.com Internet	11 words — < 1%
69	"Artificial Intelligence and Sustainable Computing", Springer Science and Business Media LLC, 2022 Crossref	10 words — < 1%
70	Yutong Li. "Forest fire smoke recognition and detection based on EfficientNet", 2022 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS), 2022 Crossref	10 words — < 1%
71	acp.copernicus.org Internet	10 words — < 1%
72	fdocuments.in Internet	10 words — < 1%
73	itvidya.com Internet	10 words — < 1%

74	m.moam.info Internet	10 words — < 1%
75	shodhbhagirathi.iitr.ac.in:8081 Internet	10 words — < 1%
76	speakerdeck.com Internet	10 words — < 1%
77	vdocuments.mx Internet	10 words — < 1%
78	vdocuments.site Internet	10 words — < 1%
79	Arkadiusz Kwasigroch, Michal Grochowski, Mateusz Mikolajczyk. "Deep neural network architecture search using network morphism", 2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR), 2019 Crossref	9 words — < 1%
80	Bin Wang, Yanan Sun, Bing Xue, Mengjie Zhang. "Evolving deep neural networks by multi-objective particle swarm optimization for image classification", Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '19, 2019 Crossref	9 words — < 1%
81	Jurgen Vandendriessche, Nick Wouters, Bruno da Silva, Mimoun Lamrini, Mohamed Yassin Chkouri, Abdellah Touhafi. "Environmental Sound Recognition on Embedded Systems: From FPGAs to TPUs", Electronics, 2021 Crossref	9 words — < 1%
82	Ke Han, Yingqi Luo. "Feasibility Analysis and Implementation of Adaptive Dynamic	9 words — < 1%

Reconfiguration of CNN Accelerators", Electronics, 2022
Crossref

83	Vafaii, Hadi. "Unveiling Secrets of Brain Function With Generative Modeling: Motion Perception in Primates & Cortical Network Organization in Mice", University of Maryland, College Park, 2024 ProQuest	9 words — < 1%
84	Zhu, Bingzhao. "Towards Efficient and Scalable Machine Learning for Future Neural Interfaces", Cornell University, 2023 ProQuest	9 words — < 1%
85	c.coek.info Internet	9 words — < 1%
86	oaktrust.library.tamu.edu Internet	9 words — < 1%
87	riunet.upv.es Internet	9 words — < 1%
88	uis.brage.unit.no Internet	9 words — < 1%
89	"Advances in Computational Intelligence", Springer Science and Business Media LLC, 2021 Crossref	8 words — < 1%
90	"Computer Vision – ECCV 2020", Springer Science and Business Media LLC, 2020 Crossref	8 words — < 1%
91	"New Trends in Intelligent Software Methodologies, Tools and Techniques", IOS Press, 2023 Crossref	8 words — < 1%

92 Aqing Yang, Huasheng Huang, Chan Zheng, Xunmu Zhu, Xiaofan Yang, Pengfei Chen, Yueju Xue. "High-accuracy image segmentation for lactating sows using a fully convolutional network", Biosystems Engineering, 2018
Crossref
8 words — < 1%

93 Cuji Dutan, Diego Andres. "OFDM Underwater Acoustic Communication System Implementation on FPGA.", Northeastern University, 2019
ProQuest
8 words — < 1%

94 Daniel Cummings, Sharath Nittur Sridhar, Anthony Sarah, Maciej Szankin. "Accelerating neural architecture exploration across modalities using genetic algorithms", Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2022
Crossref
8 words — < 1%

95 Filipe Morais. "ACTIVE ACOUSTIC NOISE CONTROL IN DUCTS", INFORMATICS IN CONTROL AUTOMATION AND ROBOTICS I, 2006
Crossref
8 words — < 1%

96 Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, El-Ghazali Talbi, Abdessamad Ait El Cadi. "Co-Optimization of DNN and Hardware Configurations on Edge GPUs", 2022 25th Euromicro Conference on Digital System Design (DSD), 2022
Crossref
8 words — < 1%

97 Han Sung Lee, Jae Wook Jeon. "Accelerating Image Processing on FPGAs using HLS and PYNQ", 2020 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia), 2020
Crossref
8 words — < 1%

98 Huang, Min, Zhaoqing Liu, and Liyan Qiao. "Asymmetric Programming: A Highly Reliable Metadata Allocation Strategy for MLC NAND Flash Memory-Based Sensor Systems", Sensors, 2014. — 8 words — < 1%
Crossref

99 Jose A. Belloch, Raul Coronado, Oscar Valls, Rocío del Amor et al. "Urban sound classification using neural networks on embedded FPGAs", The Journal of Supercomputing, 2024 — 8 words — < 1%
Crossref

100 K. Tatas. "A Survey of Existing Fine-Grain Reconfigurable Architectures and CAD tools", Fine- and Coarse-Grain Reconfigurable Computing, 2007 — 8 words — < 1%
Crossref

101 Kiruki Cosmas, Asami Kenichi. "Utilization of FPGA for Onboard Inference of Landmark Localization in CNN-Based Spacecraft Pose Estimation", Aerospace, 2020 — 8 words — < 1%
Crossref

102 Krishna Teja Chitty-Venkata, Murali Emani, Venkatram Vishwanath, Arun K. Somani. "Neural Architecture Search Benchmarks: Insights and Survey", IEEE Access, 2023 — 8 words — < 1%
Crossref

103 Lei Deng, Guoqi Li, Song Han, Luping Shi, Yuan Xie. "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey", Proceedings of the IEEE, 2020 — 8 words — < 1%
Crossref

104 Md Imtiaz Hossain, Sharmen Akhter, Choong Seon Hong, Eui-Nam Huh. "PURF: Improving teacher — 8 words — < 1%

representations by imposing smoothness constraints for knowledge distillation", Applied Soft Computing, 2024
Crossref

105 Ruben Nieto, Edel Diaz, Raul Mateos, Alvaro Hernandez. "Evaluation of Software Inter-Processor Synchronization Methods for the Zynq-UltraScale+ Architecture", 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), 2020 — 8 words — < 1%
Crossref

106 Xinwei Guo, Yong Wu, Jingjing Miao, Yang Chen. "LiteGaze: Neural architecture search for efficient gaze estimation", PLOS ONE, 2023 — 8 words — < 1%
Crossref

107 Yash Agrawal, Bhavesh Jain, Saurabh Kumar Mishra, S. Indu. "Android Application for Vegetable and Fruit Classification", 2021 International Conference on Intelligent Technologies (CONIT), 2021 — 8 words — < 1%
Crossref

108 Zhang, Kaiqi. "Overparameterization in Neural Networks: From Application to Theory", University of California, Santa Barbara, 2023 — 8 words — < 1%
ProQuest

109 amsdottorato.unibo.it — 8 words — < 1%
Internet

110 d197for5662m48.cloudfront.net — 8 words — < 1%
Internet

111 de Carvalho, Rafaela Garrido Ribeiro. "Multi-Modal Tasking for Skin Lesion Classification Using Deep Neural Networks", Universidade do Porto (Portugal), 2024 — 8 words — < 1%
ProQuest

REFERENCES

[1] Y. Lecun, Y. Benjio and G. Hinton, "Deep learning," *Nature 521,* pp. 436-444, 27 May 2015.

[2] S. Dong, P. Wang and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review,* vol. 40, 2021.

[3] C. White, M. Safari, Rhea, Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey and F. Hutter, "Neural Architecture Search: Insights from 1000 Papers," Cornell University, 20 January 2023. [Online]. Available: arXiv:2301.08727.

[4] X. D. a. Y. Yang, "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search," in *ICLR*, 2020.

[5] R. Tu, N. Roberts, M. Khodak, J. Shen, F. Sala and A. Talwalkar, "NAS-Bench-360: Benchmarking Neural Architecture Search on Diverse Tasks," Cornell University, 12 October 2021. [Online]. Available: https://arxiv.org/abs/2110.05668.

[6] H. Kitano, "Neurogenetic learning: an integrated method of designing and training neural networks using genetic algorithms," *Physica D: Nonlinear Phenomena,* vol. 75, no. 1-3, pp. 225-238, 1994.

[7] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," Cornell University, 5 November 2016. [Online]. Available: https://arxiv.org/abs/1611.01578.

[8] E. Real, A. Aggarwal, Y. Huang and Q. V. Le, "Aging evolution for image classifier architecture search," *IEEE,* 2018.

[9] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen and K. C. Tan, "A Survey on Evolutionary Neural Architecture," *IEEE,* 2021.

[10] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le and J. Dean, "Efficient Neural Architecture Search via Parameter Sharing," Cornell University, 9 February 2018. [Online]. Available: https://arxiv.org/abs/1802.03268.

[11] P. Dong, X. Niu, L. Li, L. Xie, W. Zou, T. Ye, Z. Wei and H. Pan, "Prior-Guided One-shot Neural Architecture Search," Cornell University, 27 June 2022. [Online]. Available: https://arxiv.org/abs/2206.13329.

[12] M. Tan and M. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *Proceedings of the 36th International Conference on Machine Learning*, pp. 6105-6114, 2019.

[13] M. Tan and Q. V. Le, "EfficientNetV2: Smaller Models and Faster Training," Cornell University, 1 April 2021. [Online]. Available: https://arxiv.org/abs/2104.00298.

[14] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba and N. Wang, "A Comprehensive Survey on Hardware-Aware Neural Architecture Search," Cornell University, 22 January 2021. [Online]. Available: https://arxiv.org/abs/2101.09336.

[15] H. Cai, C. Gan, T. Wang, Z. Zhang and S. Han, "Once-for-All: Train One Network and Specialize it for Efficient Deployment," Cornell University, 29 April 2020. [Online]. Available: arXiv:1908.09791v5.

[16] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei and M. Sun, "DPP-Net: Device-aware Progressive Search for Pareto-optimal Neural Architectures," Cornell University, 25 July 2018. [Online]. Available: https://arxiv.org/abs/1806.08198.

[17] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi and J. Hu, "Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search," Cornell University, 31 January 2019. [Online]. Available: https://arxiv.org/abs/1901.11211.

[18] A. Jahanshahi, "TinyCNN: A Tiny Modular CNN Accelerator for Embedded FPGA," Cornell University, 15 Nov 2019. [Online]. Available: https://arxiv.org/abs/1911.06777.

[19] S. Han, H. Mao and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," Cornell University, 1 october 2015. [Online]. Available: https://arxiv.org/abs/1510.00149.

[20] J.-H. Luo, J. Wu and W. Lin, "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression," Cornell University, 20 July 2017. [Online]. Available: https://arxiv.org/abs/1707.06342.

[21] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. v. Baalen and T. Blankevoort, "A White Paper on Neural Network Quantization," Cornell University, 15 June 2021. [Online]. Available: https://arxiv.org/abs/2106.08295.

[22] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson and S. R. W., "Exploring Zynq MPSoC with Pynq and Machine learning applications," in *Exploring Zynq MPSoC with Pynq and Machine learning applications*, 2019.

[23] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficiency of pruning for model compression," 10 2017. [Online].

[24] P.-E. Novac, G. Boukli Hacene, A. Pegatoquet, B. Miramond and V. Gripon, "Quantization and Deployment of Deep Neural Networks on Microcontrollers," 4 2021. [Online].

[25] A. Gholami, S. Kim, D. Zhen, Z. Yao, M. Mahoney and K. Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," 2022.

[26] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish and S. Orgrenci, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," Cornell University, 23 March 2021. [Online]. Available: https://arxiv.org/abs/2103.05579.

[27] "DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide," [Online]. Available: https://docs.xilinx.com/r/en-US/pg338-dpu/Number-of-DPU-Cores.

[28] H. Cai, C. Gan, T. Wang, Z. Zhang and S. Han, "Once-for-All: Train One Network and Specialize it for Efficient Deployment," *Cornell University*, vol. 1, 2019.